Programming with

lcc-win32

by

Jacob Navia and Q Software Solutions GmbH

Acknowledgements

Thanks to all people that have contributed to this work.

Thanks to the many people that sent me those bug reports that allowed me to improve the software. To all that sent me messages of encouragement.

Thanks specially to Friedrich Dominicus, John Finlay and Mike Caetano among many other people, that collaborated to make lcc-win32 what it is today.

Chapter 1 Introduction to C 1

Chapter

1

Introduction to C

This tutorial to the C language supposes you have the lcc-win32 compiler system installed. You will need a compiler anyway, and lcc-win32 is free for you to use, so please (if you haven't done that yet) download it and install it before continuing. **http://www.q-software-solutions.com**

What the C language concerns, this is not a full-fledged introduction to all of C. There are other, better books that do that (see the bibliography at the end of this book). Even if I try to explain things from ground up, there isn't here a description of all the features of the language.

Note too, that this is not just documentation or a reference manual. Functions in the standard library are explained, of course, but no exhaustive documentation of any of them is provided in this tutorial.[1]

But before we start, just a quick answer to the question: why learn C?

C has been widely criticized, and many people are quick to show its problems and drawbacks. But as languages come and go, C stands untouched. The code of lcc-win32 has software that was written many years ago, by many people, among others by Dennis Ritchie, the creator of the language itself[2]. The answer to this question is very simple: if you write software that is going to stay for some time, do not learn "the language of the day"; learn C.

C doesn't impose you any point of view. It is not object oriented, but you can do object oriented programming in C if you wish.[3] It is not a functional language but you can do functional programming[4] with it if you feel like. Most LISP interpreters and Scheme interpreters/compilers are written in C. You can do list processing in C, surely not so easily like in lisp, but you can do it. It has all essential features of a general purpose programming language like recursion, procedures as first class data types, and many others that this tutorial will show you.

Many people feel that C lacks the simplicity of Java, or the sophistication of C++ with its templates and other goodies. True. C is a simple language, without any frills. But it is precisely this lack of features that makes C adapted as a first time introduction into a complex high-level language that allows you fine control over what your program is doing without any hidden

1. For an overview of the lcc-win32 documentation see "How to find more information"

2. Dennis Ritchie wrote the pre-processor of the lcc-win32 system.

3. Objective C generates C, as does Eiffel and several other object-oriented languages. C, precisely because of this lack of a programming model is adapted to express all of them. Even C++ started as a pre-processor for the C compiler.

4. See the "Illinois FP" language implementations in C, and many other functional programming languages that are coded in C.

features. The compiler will not do anything else than what you told it to do. The language remains transparent, even if some features from Java like the garbage collection are incorporated into the implementation of C you are going to use.[5]

As languages come and go, C remains. It was at the heart of the UNIX operating system development in the seventies[6], it was at the heart of the microcomputer revolution in the eighties, and as C++, Delphi, Java, and many others came and faded, C remained, true to its own nature.

1.1 Organization of C programs

A program in C is written in one or several text files called source modules. Each of those modules is composed of *functions*, i.e. smaller pieces of code that accomplish some task[7], and *data*, i.e. variables or tables that are initialized before the program starts. There is a special function called *main* that is where the execution of the program begins.[8] In C, the organization of code in files has semantic meaning. The main source file given as an argument to the compiler defines a compilation unit.[9]

A unit can import common definitions using the #include preprocessor directive, or just by declaring some identifier as extern.[10]

C supports the separate compilation model, i.e. you can split the program in several independent units that are compiled separately, and then linked with the link editor to build the final program. Normally each module is written in a separate text file that contains functions or data declarations. Interfaces between modules are written in "header files" that describe types or functions visible to several modules of the program. Those files have a ".h" extension, and they come in two flavors: system-wide, furnished with lcc-win32, and private, specific to the application you are building.

A function has a parameter list, a body, and possibly a return value.[11] The body can contain declarations for local variables, i.e. variables activated when execution reaches the function body.

---

5. Lisp and scheme, two list oriented languages featured automatic garbage collection since several decades. APL and other interpreters offered this feature too. Lcc-win32 offers you the garbage collector developed by Hans Boehm.

6. And today, the linux kernel is written entirely in C as most operating systems.

7. There is no distinction between functions and procedures in C. A procedure is a function of return type void.

8. Actually, the startup code calls main. When main returns, the startup code regains control and ends the program. This is explained in more detail in the technical documentation.

9. Any program, in any computer in any language has two main types of memory at the start:

   The code of the program, i.e. the sequence of machine instructions that the program will execute. This section has an "entry point", the above mentioned "main" procedure in C, or other procedure that is used as the entry point

   The static data of the program, i.e. the string literals, the numbers or tables that are known when the program starts. This data area con be further divided into an initialized data section, or just empty, reserved space that is initialized by the operating system to zero when the program is loaded.

10. There is no way to import selectively some identifiers from another included file. Either you import all of it or none.

The body is a series of expressions separated by semicolons. Each statement can be an arithmetic operation, an assignment, a function call, or a compound statement, i.e. a statement that contains another set of statements.

1.2 Hello

To give you an idea of the flavor of C we use the famous example given already by the authors of the language[12]. We build here a program that when invoked will put in the screen the message “hello”.

```
#include <stdio.h>      (1)
int main(void)          (2)
{                       (3)
     printf("Hello\n"); (4)
     return 0;          (5)
}                       (6)
```

1) Using a feature of the compiler called ‘pre-processor’, you can textually include a whole file of C source with the “#include” directive. In this example we include from the standard includes of the compiler the “stdio.h” header file.[13]

2) We define a function called “main” that returns an integer as its result, and receives no arguments (void). [14]

3) The body of the function is a list of statements enclosed by curly braces.

4) We call the standard function “printf” that formats its arguments in a character string that is displayed in the screen. A function call in C is written like this:`function-name ‘(‘ argument-list ‘)’`. In this case the function name is “`printf`”, and its argument list is the character string “`Hello\n`”[15]. Character strings are enclosed in double quotes. They are represented in C as an array of characters finished by a zero byte.

5) The return statement indicates that control should be returned (hence its name) to the calling function. Optionally, it is possible to specify a return result, in this case the integer zero.

6) The closing brace finishes the function scope.

---

11. In C, only one return value is possible. A function, however can return several return values if it modifies its environment.

12. This example is a classic, and appears already in the tutorial of the C language published by B. W. Kernighan in 1974, four years before the book “The C programming language” was published. Their example would still compile today, albeit with some warnings:

```
main() { printf(“Hello world\n”); }
```

13. The name of the include file is enclosed within a <> pair. This indicates the compiler that it should look for this include file in the standard include directory, and not in the current directory. If you want to include a header file in another directory or in the compilation directory, use the double quotes to enclose the name of the file, like #include “myfile.h”

14. This is one of the two possible definitions of the “main” function. Later we will see the other one.

15. Character strings can contain sequences of characters that denote graphical characters like new line (\n) tab (\t), backspace (\b), or others. In this example, the character string is finished by the new line character \n.

Programs in C are defined in text files that normally have the.c extension. You can create those text files with any editor that you want, but lcc-win32 proposes a specialized editor for this task called “Wedit”. This program allows you to enter the program text easily, since it is adapted to the task of displaying C source text.

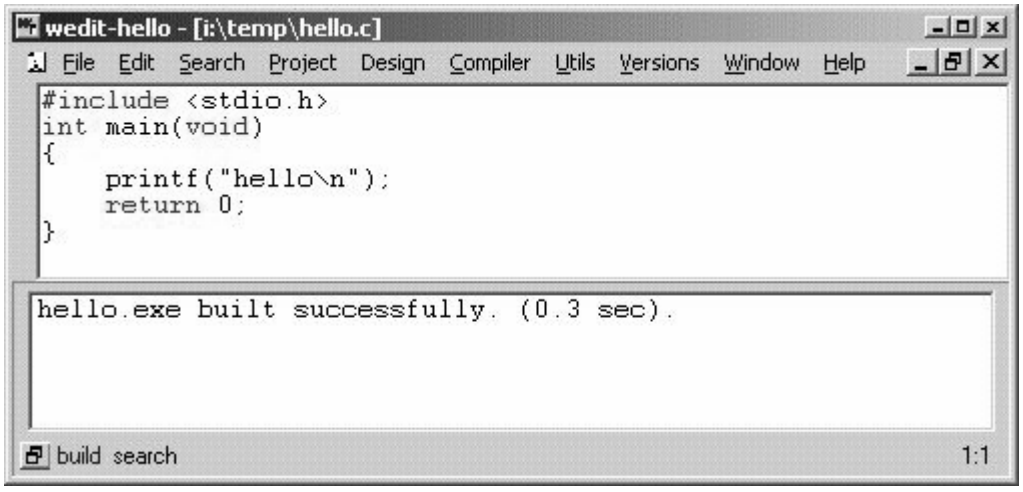
To make this program then, we start Wedit and enter the text of that program above.[16]

Once this is done, you can compile, and link-edit your program by just clicking in the compile menu or pressing F9.[17]

To run the program, you use the “execute” option in the compile menu (Ctrl+F5), or you open a command shell and type the program’s name. Let’s do it the hard way first.

The first thing we need to know is the name of the program we want to start. This is easy; we ask Wedit about it using the “Executable stats” option in the “Utils” menu. We get the following display.

We see at the first line of the bottom panel, that the program executable is called: `h:\lcc\projects\hello.exe`.[18]



16. You start wedit by double clicking in its icon, or, if you haven’t an icon for it by going to the “Start” menu, run, and then type the whole path to the executable. For instance, if you installed lcc-win32 in c:\lcc, wedit will be in c:\lcc\bin\wedit.exe

17. If this doesn’t work or you receive warnings, you have an installation problem (unless you made a typing mistake). Or maybe I have a bug. When writing mail to me do not send messages like: “It doesn’t work”. Those messages are a nuisance since I can’t possibly know what is wrong if you do not tell me **exactly** what is happening. Wedit doesn’t start? Wedit crashes? The computer freezes? The sky has a black color?

Keep in mind that in order to help you I have to reproduce the problem in my setup. This is impossible without a detailed report that allows me to see what goes wrong.

Wedit will make a default project for you, when you click the “compile” button. This can go wrong if there is not enough space in the disk to compile, or the installation of lcc-win32 went wrong and Wedit can’t find the compiler executable, or many other reasons. If you see an error message please do not panic, and try to correct the error the message is pointing you to.

A common failure happens when you install an older version of Wedit in a directory that has spaces in it. Even if there is an explicit warning that you should NOT install it there, most people are used to just press return at those warnings without reading them. Then, lcc-win32 doesn’t work and they complain to me. I have improved this in later versions, but still problems can arise.

18. For understanding the rest of the output see the technical notes below.

We open a command shell window, and type the command:

```
C:\>h:\lcc\projects\lcc1\hello.exe
Hello
C:\>
```

Our program displays the character string “Hello” and then a new line, as we wanted. If we erase the \n of the character string, press F9 again to recompile and link, the display will be:

```
C:\>h:\lcc\projects\lcc1\hello.exe
Hello
C:\>
```

But how did we know that we have to call “printf” to display a string?

Because the documentation of the library told us so… The first thing a beginner to C must do is to get an overview of the libraries provided already with the system so that he/she doesn’t waste time rewriting programs that can be already used without any extra effort. Printf is one of those, but are several thousands of pre-built functions of all types and for all tastes. We present an overview of them in the next section.

1.2.1 Console mode programs and windows programs

Windows makes a difference between text mode programs and windows programs. In the first part of this book we will use console programs, i.e. programs that run in a text mode window receiving only textual input and producing text output. Those are simpler to build than the more complicated GUI (Graphical User Interface) programs.

Windows knows how to differentiate between console/windows programs by looking at certain fields in the executable file itself. If the program has been marked by the compiler as a console mode program, windows opens a window with a black background by default, and initializes the standard input and standard output of the program before it starts. If the program is marked as a windows program, nothing is done, and you can’t use the text output or input library functions.

For historical reasons this window is called sometimes a “DOS” window, even if there is no MSDOS since more than a decade. The programs that run in this console window are 32 bit programs and they can open a window if they wish. They can use all of the graphical features of windows. The only problem is that an ugly black window will be always visible, even if you open a new window.

You can change the type of program lcc-win32 will generate by checking the corresponding boxes in the “Linker” tab of the configuration wizard, accessible from the main menu with “Project” then “Configuration”.

Under other operating systems the situation is pretty much the same. Linux offers a console, and even the Macintosh has one too. In many situations typing a simple command sequence is much faster than clicking dozens of menus/options till you get where you want to go. Besides, an additional advantage is that console programs are easier to automate and make them part of bigger applications as independent components that receive command-line arguments and produce their output without any human intervention.

1.2.2 An overview of the compilation process

When you press F9 in the editor, a complex sequence of events, all of them invisible to you, produce an executable file. Here is a short description of this, so that at least you know what’s happening behind the scene.

Wedit calls the C compiler proper. This program is called lcc.exe and is in the installation directory of lcc, in the bin directory. For instance, if you installed lcc in `c:\lcc`, the compiler will be in `c:\lcc\bin`.

This program will read your source file, and produce another file called object file,[19] that has the same name as the source file but a..obj extension. C supports the separate compilation model, i.e. you can compile several source modules producing several object files, and rely in the link-editor lcclnk.exe to build the executable.

Lcclnk.exe is the link-editor, or linker for short. This program reads different object files, library files and maybe other files, and produces either an executable file or a dynamically loaded library, a DLL.

When compiling your hello.c file then, the compiler produced a “hello.obj” file, and from that, the linker produced a hello.exe executable file. The linker uses several files that are stored in the `\lcc\lib` directory to bind the executable to the system DLLs, used by all programs: kernel32.dll, crtdll.dll, and many others.

The workings of the lcc compiler are described in more detail in the technical documentation. Here we just tell you the main steps.

- The source file is first pre-processed. The #include directives are resolved, and the text of the included files is inserted into the source file.[20]
- The front end of the compiler proper processes the resulting text. Its task is to generate a series of intermediate code statements.[21] The code generator that emits assembler instructions from it processes these.[22]
- Eventually the compiler produces an object file with the .obj extension. This file is passed then (possibly with other object files) to the linker lcclnk that builds the executable.

Organizing all those steps and typing all those command lines can be boring. To easy this, the IDE will do all of this with the F9 function key.

1.2.3 Technical notes

The output shown in the wedit window above means the following:

1 Size of code. This is the number of bytes that the instructions of your program will use. This includes the startup, and the code for the statically linked c-runtime functions like printf, if necessary. Lcc-win32 uses the standard windows library CRTDLL.DLL, a C run time library provided by windows itself. Most of the functions in that library are not usable if you want to support the latest ANSI standard however, so they have been rewritten in a static library libc.lib, that is used by the linker. That is why the size of your code can grow

19. This has nothing to do with object oriented programming of course!

20. The result of this process can be seen if you call the compiler with the –E flag. For instance, to see what is the result of pre-processing the hello.c file you call the compiler in a command shell window with the command line: `lcc -E hello.c`. The resulting file is called `hello.i`.

21. Again, you can see the intermediate code of lcc by calling the compiler with `lcc -z hello.c`. This will produce an intermediate language file called `hello.lil` that contains the intermediate language statements.

22. Assembly code can be generated with the `lcc -S hello.c` command, and the generated assembly file will be called `hello.asm`. The generated file contains a listing of the C source and the corresponding translation into assembly language.

suddenly by several kilobytes when you add just one single line to your program source.

2 Size of initialized data. This is the number of bytes that the tables, constants and other initial data use. In this case we see that besides the character string “hello” we have many other constants added to the program by the C library.

3 Uninitialized data. This is the part of your program that reserves space for variables that will be zeroed by the system at the start.

4 Size of image. This the place the whole will take in memory when loaded, including all the above items. For alignment reasons this is greater than a simple sum of the above parts. This size must be a multiple of 4096 bytes, a page.

What does this mean?

Program size has lately become a “non-issue”. Machines have grown so enormously, that most people think that if the program makes 2MB or 20MB it doesn’t matter. This has some justification, of course, but it is not the philosophy of lcc-win32 or C in general. Wedit has several tools to get the size of each function, and it reports summaries for size information about your program. Remember that smaller programs fit better in the cache of the CPU, and execute faster since they require less resources.

1.2.4 The run time environment

The program starts in your machine. A specific operating system is running, a certain file and hard disk configuration is present, you have so many RAM chips installed, etc. This is the run-time environment.

The file built by the linker lcclnk is started through a user action (you double click in its icon) or by giving its name at a command shell prompt, or by the action of another program that requests to the operating system to start it.

The operating system accesses the hard disk at the specified location, and reads all the data in the file into RAM. Then, it determines where the program starts, and sets the program counter of the printed circuit in your computer to that memory location.

The piece of code that starts is the “startup” stub, a small program that does some initialization and calls the “main” procedure. It pushes the arguments to main in the same way as for any other procedure.

The main function starts by calling another function in the C library called “printf”. This function writes characters using a “console” emulation, where the window is just text. This environment is simpler conceptually, and it is better suited to many things for people that do not like to click around a lot.

The printf function deposits characters in the input buffer of the terminal emulation program, that makes the necessary bits change color using the current font, and at the exact position needed to display each glyph.

Windows calls the graphic drivers in your graphic card, that control the video output of the machine with those bits to change. The bits change before your hand has had the time to move a millimeter. Graphic drivers are fast today, and in no time they return to windows that returns control to the printf function.

The printf function exits, then control returns to main, that exits to the startup, that calls ExitProcess, and the program is finished by the operating system

Your hand is still near the return key.

.

We have the following phases in this process:

1: Design-time. We wrote the program first.

2: Compile-time. We compiled our design.

3: Run-time. The compiled instructions are started and the machine executes what we told it to do.

1.2.4.1 We wrote the program first

The central point in communicating with a printed circuit is the programming language you use to define the sequence of operations to be performed. The sequence is prepared using that language, first in your own circuit, your brain, then written down with another (the keyboard controller), then stored and processed by yet another, a personal computer (PC).

1.2.4.2 We compiled our design

Compiled languages rely on piece of software to read a textual representation first, translating it directly into a sequence of numbers that the printed circuit understands. This is optionally done by assembling several pieces of the program together as a unit.

1.2.4.3 Run time

The operating system loads the prepared sequence of instructions from the disk into main memory, and passes control to the entry point.

This is done in several steps. First the main executable file is loaded, then all the libraries the program needs.

When everything has been mapped in memory, and all the references in each part have been resolved, the OS calls the initialization procedures of each loaded library. If everything goes well, the OS gives control to the program entry point.

1.3 An overview of the standard libraries

This headers and the associated library functions are found in all ANSI compliant compilers.[23]

Header	Purpose
assert.h	Diagnostics for debugging help.
complex.h	Complex numbers definitions. See page 167.
ctype.h	Character classification (isalpha, islower, isdigit)
errno.h	Error codes set by the library functions
fenv.h	Floating point environment. Functions concerning the precision of the calculations, exception handling, and related items. See page 156.
float.h	Characteristics of floating types (float, double, long double, qfloat). See page 156.
inttypes.h	Characteristics of integer types
iso646.h	Alternative spellings for some keywords. If you prefer writing the operator “&&” as “and”, use this header.
limits.h	Size of integer types.
locale.h	Definitions for the formatting of currency values using local conventions.
math.h	Mathematical functions.
setjmp.h	Non local jumps, i.e. jumps that can go past function boundaries. See page 59.
signal.h	Signal handling. See page 153.
stdarg.h	Definitions concerning functions with variable number of arguments.
stdbool.h	Boolean type and values
stddef.h	Standard definitions for the types of a pointer difference, or others.
stdint.h	Integer types
stdio.h	Standard input and output.
stdlib.h	Standard library functions.
stddef.h	This file defines macros and types that are of general use in a program. NULL, offsetof, ptrdiff_t, size_t, and several others.
string.h	String handling. Here are defined all functions that deal with the standard representation of strings as used in C. See “Traditional string representation in C” on page 108.
stdarg.h	Functions with variable number of arguments are described here. See page 42.
time.h	Time related functions.See page 124.
wchar.h	Extended multibyte/wide character utilities
wctype.h	Wide character classification and mapping utilities

1.3.1 The “stdheaders.h” include file

Normally, it is up to you to remember which header contains the declaration of which function. This can be a pain, and it is easy to confuse some header with another. To avoid this overloading of the brain memory cells, lcc-win32 proposes a “stdheaders.h” file, that consists of :

23. In the user’s manual there is an exhaustive list of the entire set of header files distributed with lcc-win32. Please look there for an in-depth view.

```
#include <assert.h>
#include <complex.h>
...
etc
```

Instead of including the standard headers in several include statements, you just include the "stdheaders.h" file and you are done with it. True, there is a very slight performance lost in compilation time, but it is not really significant.

1.3.2 Windows specific headers

There are several megabytes of windows header files, and we will not explain them all here.

windows.h	All windows definitions. Creating a window, opening a window, this is an extensive header file, makes approx half a megabyte of definitions. Note that under lcc-win32, several headers like winbase.h of other distributions are concentrated in a single file.
winsock.h	Network (tcpip)
shellapi.h	Windows Shell

1.4 Passing arguments to a program

We can't modify the behavior of our hello program with arguments. We have no way to pass it another character string for instance, that it should use instead of the hard-wired "hello\n". We can't even tell it to stop putting a trailing new line character.

Programs normally receive arguments from their environment. A very old but still quite effective method is to pass a command line to the program, i.e. a series of character strings that the program can use.

Let's see how arguments are passed to a program.[24]

```
#include <stdio.h>                          (1)
int main(int argc,char *argv[])             (2)
{
     int count;                             (3)

     for (count=0;count < argc;count++) {(4)
          printf(                           (5)
                  "Argument %d = %s\n",
                  count,
                  argv[count]);
     }                                      (6)
     return 0;
}
```

1) We include again stdio.h

2) We use a longer definition of the "main" function as before. This one is as standard as the previous one, but allows us to pass parameters to the program. There are two arguments:

24. Here we will only describe the standard way of passing arguments as specified by the ANSI C standard, the one lcc-win32 uses. Under the Windows operating system, there is an alternative entry point, called WinMain, and its arguments are different than those described here. See the Windows programming section later in this tutorial.

int argc This is an integer that in C is known as “int”. It contains the number of arguments passed to the program plus one.

char *argv[] This is an array of pointers to characters[25] containing the actual arguments given. For example, if we call our program from the command line with the arguments “foo” and “bar”, the argv[ ] array will contain:

argv[0] The name of the program that is running.

argv[1] The first argument, i.e. “foo”.

argv[2] The second argument, i.e. “bar”.

We use a memory location for an integer variable that will hold the current argument to be printed. This is a local variable, i.e. a variable that can only be used within the enclosing scope, in this case, the scope of the function “main”.[26]

3) We use the “for” construct, i.e. an iteration. The “for” statement has the following structure:
   - Initialization. Things to be done before the loop starts. In this example, we set the counter to zero. We do this using the assign statement of C: the “=” sign. The general form of this statement is
   - variable “=” value
   - Test. Things to be tested at each iteration, to determine when the loop will end. In this case we test if the count is still smaller than the number of arguments passed to the program, the integer argc.
   - Increment. Things to be updated at each iteration. In this case we add 1 to the counter with the post-increment instruction: counter++. This is just a shorthand for writing counter = counter + 1.
   - Note that we start at zero, and we stop when the counter is equal to the upper value of the loop. Remember that in C, array indexes for an array of size n elements always start at zero and run until n-1.[27]

4) We use again printf to print something in the screen. This time, we pass to printf the following arguments:

---

25. This means that you receive the machine address of the start of an integer array where are stored the start addresses of character strings containing the actual arguments. In the first position, for example, we will find an integer that contains the start position in RAM of a sequence of characters containing the name of the program. We will see this in more detail when we handle pointers later on.

26. Local variables are declared (as any other variables) with:

```
<type> identifier;
```

For instance

```
int a;
double b;
char c;
```

Arrays are declared in the same fashion, but followed by their size in square brackets:

```
int a[23];
double b[45];
char c[890];
```

```
"Argument %d = ‘%s’\n"
count
argv[count]
```

Printf will scan its first argument. It distinguishes directives (introduced with a per-cent sign %), from normal text that is outputted without any modification. We have in the character string passed two directives a %d and a %s.

The first one, a **%d** means that printf will introduce at this position, the character representation of a number that should also be passed as an argument. Since the next argument after the string is the integer “count”, its value will be displayed at this point.

The second one, a **%s** means that a character string should be introduced at this point. Since the next argument is argv[count], the character string at the position “count” in the argv[ ] array will be passed to printf that will display it at this point.

5) We finish the scope of the for statement with a closing brace. This means, the iteration definition ends here.

Now we are ready to run this program. Suppose that we have entered the text of the program in the file “args.c”. We do the following:

```
h:\lcc\projects\args> lcc args.c
h:\lcc\projects\args> lcclnk args.obj
```

We first compile the text file to an object file using the lcc compiler. Then, we link the resulting object file to obtain an executable using the linker lcclnk. Now, we can invoke the program just by typing its name:[28]

```
h:\lcc\projects\args> args
Argument 0 = args
```

We have given no arguments, so only argv[0] is displayed, the name of the program, in this case “args”. Note that if we write:

```
h:\lcc\projects\args> args.exe
Argument 0 = args.exe
```

We can even write:

```
h:\lcc\projects\args> h:\lcc\projects\args.exe
Argument 0 = h:\lcc\projects\args.exe
```

But that wasn’t the objective of the program. More interesting is to write:

```
h:\lcc\projects\args> args foo bar zzz
Argument 0 = args
Argument 1 = foo
Argument 2 = bar
Argument 3 = zzz
```

The program receives 3 arguments, so argc will have a value of 4. Since our variable count will run from 0 to argc-1, we will display 4 arguments: the zeroth, the first, the second, etc.

---

27. An error that happens very often to beginners is to start the loop at 1 and run it until its value is smaller or equal to the upper value. If you do NOT use the loop variable for indexing an array this will work, of course, since the number of iterations is the same, but any access to arrays using the loop index (a common case) will make the program access invalid memory at the end of the loop.

28. The detailed description of what happens when we start a program, what happens when we compile, how the compiler works, etc., are in the technical documentation of lcc-win32. With newer versions you can use the compilation driver ‘lc.exe’ that will call the linker automatically.

1.4.1 Iteration constructs

We introduced informally the “for” construct above, but a more general introduction to loops is necessary to understand the code that will follow.

There are three iteration constructs in C: “for”, “do”, and “while”.

1.4.1.1 for

The **“for”** construct has

1: An initialization part, i.e. code that will be always executed before the loop begins,
2: A test part, i.e. code that will be executed at the start of each iteration to determine if the loop has reached the end or not, and
3: An increment part, i.e. code that will be executed at the end of each iteration. Normally, the loop counters are incremented (or decremented) here.

The general form is then:

```
for(init ; test ; increment) {
     statement block
}
```

Within a for statement, you can declare variables local to the “for” loop. The scope of these variables is finished when the for statement ends.

```
#include <stdio.h>
int main(void)
{
        for (int i = 0; i< 2;i++) {
                printf("outer i is %d\n",i);
                for (int i = 0;i<2;i++) {
                        printf("i=%d\n",i);
                }
        }
        return 0;
}
The output of this program is:
outer i is 0
i=0
i=1
outer i is 1
i=0
i=1
```

Note that the scope of the identifiers declared within a ‘for’ scope ends just when the for statement ends, and that the ‘for’ statement scope is a new scope. Modify the above example as follows to demonstrate this:

```
#include <stdio.h>
int main(void)
{
     for (int i = 0; i< 2;i++) {1
          printf("outer i is %d\n",i);2
          int i = 87;

          for (int i = 0;i<2;i++) {4
               printf("i=%d\n",i);5
          }    6
     }           7
     return 0; 8
}
```

At the innermost loop, there are three identifiers called ‘i’.

- The first i is the outer i. Its scope goes from line 1 to 7 — the scope of the for statement.
- The second i (87) is a local identifier of the compound statement that begins in line 1 and ends in line 7. Compound statements can always declare local variables.
- The third i is declared at the innermost for statement. Its scope starts in line 4 and goes up to line 6. It belongs to the scope created by the second for statement.

Note that for each new scope, the identifiers of the same name are shadowed by the new ones, as you would normally expect in C.

1.4.1.2 while

The **“while”** construct is much more simple. It consists of a single test that determines if the loop body should be executed or not. There is no initialization part, nor increment part.

The general form is:

```
while (test) {
     statement block
}
```

Any “for” loop can be transformed into a “while” loop by just doing:

```
init
while (test) {
     statement block
     increment
}
```

1.4.1.3 do

The **“do”** construct is a kind of inverted while. The body of the loop will always be executed at least once. At the end of each iteration the test is performed. The general form is:

```
do {
     statement block
} while (test);
```

Using the “break” keyword can stop any loop. This keyword provokes an exit of the block of the loop and execution continues right afterwards.

The “**continue**” keyword can be used within any loop construct to provoke a jump to the end of the statement block. The loop continues normally, only the statements between the continue keyword and the end of the loop are ignored.

1.4.2 Basic types

The implementation of the C language by the lcc-win32 compiler has the following types built in:[29]All this types are part of the standard ANSI C language. With the exception of the _Complex type they should appear in most C implementations and they do appear in all windows compilers.[30]

---

29. In most compilers the char/short/int/long types are present but their sizes can change from machine to machine. Some embedded systems compilers do not support floating point. Many compilers do not implement the recent types _Bool, long long, or long double. Within the windows environment however, the char/short/int/long/float/double types are identical to this ones in all 32 bit windows compilers I know of.
30. Microsoft Visual C implements "long double" as double, and calls the long long type "__int64".To remain compatible with this compiler, lcc-win32 accepts __int64 as an equivalent of long long.

Type	Size (bytes)	Description
`_Bool`[1]	1	Logical type, can be either zero or one.
`char`	1	Character or small integer type. Comes in two flavors: signed or unsigned.
`short`	2	Integer or unicode character stored in 16 bits. Signed or unsigned.
`int`	4	Integer stored in 32 bits. Signed or unsigned.
`long`	4	Identical to int
`long long`	8	Integer stored in 64 bits. Signed or unsigned.
`float`	4	Floating-point single precision. (Approx 7 digits)
`double`	8	Floating-point double precision. (Approx. 15 digits)
`long double`	12	Floating point extended precision (Approx 19 digits)

`float _Complex`
`double _Complex`
`long double _Complex`
	32
	32
	32
		Complex numbers. Each _Complex is composed of two parts: real and imaginary part. Each of those parts is a floating point number. Include <complex.h> when using them.

1. The actual type of the Boolean type should be “bool”, but in the standard it was specified that this type wouldn't be made the standard name for now, for compatibility reasons with already running code. If you want to use bool, you should include the header “stdbool.h”.

These are the basic types of ANSI-C. Lcc-win32 offers you other types of numbers. To use them you should include the corresponding header file, they are not “built in” into the compiler. They are built using a property of this compiler that allows you to define your own kind

Type	**Header**	**Size (bytes)**	**Description**
qfloat	qfloat.h	56	352 bits floating point
bignum	bignum.h	variable	Extended precision number

of numbers and their operations. This is called operator overloading and will be explained further down.

1.5 Declarations and definitions

It is very important to understand exactly the difference between a declaration and a definition in C.

A *declaration* introduces an identifier to the compiler. It says in essence: this identifier is a xxx and its definition will come later. An example of a declaration is

```
extern double sqrt(double);
```

With this declaration, we introduce to the compiler the identifier sqrt, telling it that it is a function that takes a double precision argument and returns a double precision result. Nothing more. No storage is allocated for this declaration, besides the storage allocated within the compiler internal tables.[31]

A *definition* tells the compiler to allocate storage for the identifier. For instance, when we defined the function main above, storage for the code generated by the compiler was created, and an entry in the program's symbol table was done. In the same way, when we wrote:

```
int count;
```

above, the compiler made space in the local variables area of the function to hold an integer.

And now the central point: You can declare a variable many times in your program, but there must be only one place where you *define* it. Note that a definition is also a declaration, because when you define some variable, automatically the compiler knows what it is, of course. For instance if you write:

```
double balance;
```

even if the compiler has never seen the identifier balance before, after this definition it knows it is a double precision number.[32]

1.5.1 Variable declaration

A variable is declared with

```
<type> <identifier> ;
```

like

```
int a;
double d;
long long h;
```

All those are definitions of variables. If you just want to declare a variable, without allocating any storage, because that variable is defined elsewhere you add the keyword extern:

```
extern int a;
extern double d;
extern long long d;
```

Optionally, you can define an identifier, and assign it a value that is the result of some calculation:

```
double fn(double f) {
double d = sqrt(f);
     // more statements
}
```

Note that initializing a value with a value unknown at compile time is only possible within a function scope. Outside a function you can still write:

```
int a = 7;
```

or

```
int a = (1024*1024)/16;
```

but the values you assign must be compile time constants, i.e. values that the compiler can figure out when doing its job.

Pointers are declared using an asterisk:

31. Note that if the function so declared is never used, absolutely no storage will be used. A declaration doesn't use any space in the compiled program, unless what is declared is effectively used. If that is the case, the compiler emits a record for the linker telling it that this object is defined elsewhere.

32. Note that when you do not provide for a declaration, and use this feature: definition is a declaration; you can only use the defined object after it is defined. A declaration placed at the beginning of the program module or in a header file frees you from this constraint. You can start using the identifier immediately, even if its definition comes much later, or even in another module.

```
int *a;
```

This means that a will contain the machine address of some unspecified integer. [33]

You can save some typing by declaring several identifiers of the same type in the same declaration like this:

```
int a,b=7,*c,h;
```

Note that c is a pointer to an integer, since it has an asterisk at its left side. This notation is somehow confusing, and forgetting an asterisk is quite common. Use this multiple declarations when all declared identifiers are of the same type and put pointers in separate lines.

The syntax of C declarations has been criticized for being quite obscure. This is true; there is no point in negating an evident weakness. In his book “Deep C secrets”[34] Peter van der Linden writes a simple algorithm to read them. He proposes (chapter 3) the following:

The Precedence Rule for Understanding C Declarations.

Rule **1**: Declarations are read by starting with the name and then reading in precedence order.

Rule **2**: The precedence, from high to low, is:

2.A : Parentheses grouping together parts of a declaration

2.B: The postfix operators:

2.B.1: Parentheses ( ) indicating a function prototype, and

2.B.2: Square brackets [ ] indicating an array.

2.B.3: The prefix operator: the asterisk denoting "pointer to".

Rule **3**: If a const and/or volatile keyword is next to a type specifier e.g. int, long, etc.) it applies to the type specifier. Otherwise the const and/or volatile keyword applies to the pointer asterisk on its immediate left.

Using those rules, we can even understand a thing like:

```
char * const *(*next)(int a, int b);
```

We start with the variable name, in this case “next”. This is the name of the thing being declared. We see it is in a parenthesized expression with an asterisk, so we conclude that “next is a pointer to…” well, something. We go outside the parentheses and we see an asterisk at the left, and a function prototype at the right. Using rule 2.B.1 we continue with the prototype. “next is a pointer to a function with two arguments”. We then process the asterisk: “next is a pointer to a function with two arguments returning a pointer to…” Finally we add the `char * const`, to get

“next” is a pointer to a function with two arguments returning a pointer to a constant pointer to char.

Now let’s see this:

```
char (*j)[20];
```

Again, we start with “j is a pointer to”. At the right is an expression in brackets, so we apply 2.B.2 to get “j is a pointer to an array of 20”. Yes what? We continue at the left and see ”char”. Done. “j” is a pointer to an array of 20 chars. Note that we use the declaration in the same form without the identifier when making a cast:

---

33. Machine addresses are just integers, of course. For instance, if you have a machine with 128MB of memory, you have 134 217 728 memory locations. They could be numbered from zero up, but Windows uses a more sophisticated numbering schema called “Virtual memory”.

34. Deep C secrets. Peter van der Linden ISBN 0-13-177429-8

```
j = (char (*)[20]) malloc(sizeof(*j));
```

We see in bold and enclosed in parentheses (a cast) the same as in the declaration but without the identifier j.

1.5.2 Function declaration

A declaration of a function specifies:

- The return type of the function, i.e. the the kind of result value it produces, if any.
- Its name.
- The types of each argument, if any.

The general form is:

```
<type> <Name>(<type of arg 1>, ... <type of arg N> ) ;
double sqrt(double) ;
```

Note that an identifier can be added to the declaration but its presence is optional. We can write:

```
double sqrt(double x);
```

if we want to, but the "x" is not required and will be ignored by the compiler.

Functions can have a variable number of arguments. The function "printf" is an example of a function that takes several arguments. We declare those functions like this:

```
int printf(char *, ...);
```

The ellipsis means "some more arguments".[35]

Why are function declarations important?

When I started programming in C, prototypes for functions didn't exist. So you could define a function like this:

```
int fn(int a)
{
     return a+8;
}
```

and in another module write:

```
     fn(7,9);
```

without any problems.

Well, without any problems at compile time of course. The program crashed or returned nonsense results. When you had a big system of many modules written by several people, the probability that an error like this existed in the program was almost 100%. It is impossible to avoid mistakes like this. You can avoid them most of the time, but it is impossible to avoid them always.

Function prototypes introduced compile time checking of all function calls. There wasn't anymore this dreaded problem that took us so many debugging hours with the primitive debugger of that time. In the C++ language, the compiler will abort compilation if a function is used without prototypes. I have thought many times to introduce that into lcc-win32, because ignoring the function prototype is always an error. But, for compatibility reasons I haven't done it yet.[36]

---

35. The interface for using functions with a variable number of arguments is described in the standard header file "stdarg.h". See "Functions with variable number of arguments." on page 42.

1.5.3 Function definitions

Function definitions look very similar to function declarations, with the difference that instead of just a semi colon, we have a block of statements enclosed in curly braces, as we saw in the function “main” above. Another difference is that here we have to specify the name of each argument given, these identifiers aren’t optional any more: they are needed to be able to refer to them within the body of the function. Here is a rather trivial example:

```
int addOne(int input)
{
     return input+1;
}
```

1.5.4 Variable definition

A variable is defined when the compiler allocates space for it. For instance, at the global level, space will be allocated by the compiler when it sees a line like this:

```
int a;
```

or

```
int a = 67;
```

In the first case the compiler allocates sizeof(int) bytes in the non-initialized variables section of the program. In the second case, it allocates the same amount of space but writes 67 into it, and adds it to the initialized variables section.

1.5.5 Statement syntax

In C, the enclosing expressions of control statements like if, or while, must be enclosed in parentheses. In many languages that is not necessary and people write:

```
if a < b run(); // Not in C...
```

in C, the if statement requires a parentheses

```
if (a<b) run();
```

The assignment in C is an expression, i.e. it can appear within a more complicated expression:

```
if ( (x =z) > 13) z = 0;
```

This means that the compiler generates code for assigning the value of z to x, then it compares this value with 13, and if the relationship holds, the program will set z to zero.

1.6 Errors and warnings

It is very rare that we type in a program and that it works at the first try. What happens, for instance, if we forget to close the main function with the corresponding curly brace? We erase the curly brace above and we try:

```
h:\lcc\examples>lcc args.c
Error args.c: 15  syntax error; found `end of input' expecting `}'
1 errors, 0 warnings
```

36. There is a strong commitment, from the part of the compiler writers, to maintain the code that was written in the language, and to avoid destroying programs that are working. When the standards committee proposed the prototypes, all C code wasn’t using them yet, so a transition period was set up. Compilers would accept the old declarations without prototypes and just emit a warning. Some people say that this period should be over by now (it is more than 10 years that we have prototypes already), but still, new compilers like lcc-win32 are supporting old style declarations.

Well, this is at least a clear error message. More difficult is the case of forgetting to put the semi-colon after the declaration of count, in the line 3 in the program above:

```
D:\lcc\examples>lcc args.c
Error args.c: 6  syntax error; found `for' expecting `;'
Error args.c: 6  skipping `for'
Error args.c: 6  syntax error; found `;' expecting `)'
Warning args.c: 6  Statement has no effect
Error args.c: 6  syntax error; found `)' expecting `;'
Error args.c: 6  illegal statement termination
Error args.c: 6  skipping `)'
6 errors, 1 warnings

D:\lcc\examples>
```

We see here a chain of errors, provoked by the first. The compiler tries to arrange things by skipping text, but this produces more errors since the whole "for" construct is not understood. Error recovering is quite a difficult undertaking, and lcc-win32 isn't very good at it. So the best thing is to look at the first error, and in many cases, the rest of the error messages are just consequences of it.[37]

Another type of errors can appear when we forget to include the corresponding header file. If we erase the `#include <stdio.h>` line in the args program, the display looks like this:

```
D:\lcc\examples>lcc args.c
Warning args.c: 7  missing prototype for printf
0 errors, 1 warnings
```

This is a warning. The printf function will be assumed to return an integer, what, in this case, is a good assumption. We can link the program and the program works. It is surely NOT a good practice to do this, however, since all argument checking is not done for unknown functions; an error in argument passing will pass undetected and will provoke a much harder type of error: a run time error.

In general, it is better to get the error as soon as possible. The later it is discovered, the more difficult it is to find it, and to track its consequences. Do as much as you can to put the C compiler in your side, by using always the corresponding header files, to allow it to check every function call for correctness.

The compiler gives two types of errors, classified according to their severity: a **warning**, when the error isn't so serious that doesn't allow the compiler to finish its task, and the hard **errors**, where the compiler doesn't generate an executable file and returns an error code to the calling environment.

We should keep in mind however that warnings are errors too, and try to get rid from them.

The compiler uses a two level "warning level" variable. In the default state, many warnings aren't displayed to avoid cluttering the output. They will be displayed however, if you ask explicitly to raise the warning level, with the option –A. This compiler option will make the compiler emit all the warnings it would normally suppress. You call the compiler with `lcc -A <filename>`, or set the corresponding button in the IDE, in the compiler configuration tab.

Errors can appear in later stages of course. The linker can discover that you have used a procedure without giving any definition for it in the program, and will stop with an error. Or it can

37. You will probably see another display in your computer if you are using a recent version of lcc-win32. I improved error handling when I was writing this tutorial…

discover that you have given two different definitions, maybe contradictory to the same identifier. This will provoke a link time error too.

But the most dreaded form of errors are the errors that happen at execution time, i.e. when the program is running. Most of these errors are difficult to detect (they pass through the compilation and link phases without any warnings…) and provoke the total failure of the software.

The C language is not very "forgiving" what programmer errors concerns. Most of them will provoke the immediate stop of the program with an exception, or return completely nonsense results. In this case you need a special tool, a debugger, to find them. Lcc-win32 offers you such a tool, and you can debug your program by just pressing F5 in the IDE.

Summary:

- Syntax errors (missing semi-colons, or similar) are the easiest of all errors to correct.
- The compiler emits two kinds of diagnostic messages: warnings and errors.
- You can rise the compiler error reporting with the –A option.
- The linker can report errors when an identifier is defined twice or when an identifier is missing a definition.
- The most difficult errors to catch are run time errors, in the form of traps or incorrect results.

1.7 Reading from a file

For a beginner, it is very important that the basic libraries for reading and writing to a stream, and the mathematical functions are well known. Here is an example of a function that will read a text file, counting the number of characters that appear in the file.

A program is defined by its specifications. In this case, we have a general goal that can be expressed quickly in one sentence: "Count the number of characters in a file". Many times, the specifications aren't in a written form, and can be even completely ambiguous. What is important is that before you embark in a software construction project, at least for you, the specifications are clear.

```
#include <stdio.h>                (1)
int main(int argc,char *argv[])   (2)
{
     int count=0;// chars read    (3)
     FILE *infile;                (4)
     int c;                       (5)

     infile = fopen(argv[1],"r");(6)
     c = fgetc(infile);           (7)
     while (c != EOF) {           (8)
          count++;                (9)
          c = fgetc(infile);      (10)
     }
     printf("%d\n",count);        (11)
     return 0;
}
```

1) We include the standard header "stdio.h" again. Here is the definition of a FILE structure.

2) The same convention as for the "args" program is used here.

3) We set at the start, the count of the characters read to zero. Note that we do this in the declaration of the variable. C allows you to define an expression that will be used to initialize a variable.[38]

4) We use the variable "infile" to hold a FILE pointer. Note the declaration for a pointer: `<type> * identifier;` the type in this case, is a complex structure (composite type) called `FILE` and defined in stdio.h. We do not use any fields of this structure, we just assign to it, using the functions of the standard library, and so we are not concerned about the specific layout of it. Note that a pointer is just the machine address of the start of that structure, not the structure itself. We will discuss pointers extensively later.

5) We use an integer to hold the currently read character.

6) We start the process of reading characters from a file first by opening it. This operation establishes a link between the data area of your hard disk, and the FILE variable. We pass to the function fopen an argument list, separated by commas, containing two things: the name of the file we wish to open, and the mode that we want to open this file, in our example in read mode. Note that the mode is passed as a character string, i.e. enclosed in double quotes.

7) Once opened, we can use the fgetc function to get a character from a file. This function receives as argument the file we want to read from, in this case the variable "infile", and returns an integer containing the character read.

8) We use the while statement to loop reading characters from a file. This statement has the general form: while (condition) { … statements… }. The loop body will be executed for so long as the condition holds. We test at each iteration of the loop if our character is not the special constant EOF (End Of File), defined in stdio.h.

9) We increment the counter of the characters. If we arrive here, it means that the character wasn't the last one, so we increase the counter.

10) After counting the character we are done with it, and we read into the same variable a new character again, using the fgetc function.

11) If we arrive here, it means that we have hit EOF, the end of the file. We print our count in the screen and exit the program returning zero, i.e. all is OK. By convention, a program returns zero when no errors happened, and an error code, when something happened that needs to be reported to the calling environment.

Now we are ready to start our program. We compile it, link it, and we call it with:

```
h:\lcc\examples> countchars countchars.c
288
```

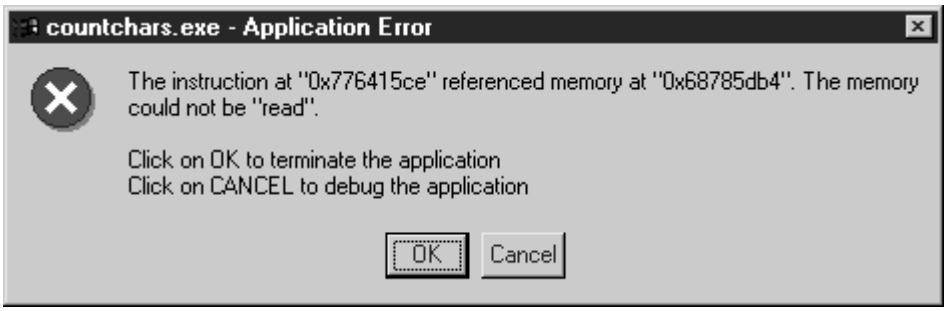
We have achieved the first step in the development of a program. We have a version of it that in some circumstances can fulfill the specifications that we received.

But what happens if we just write

```
h:\lcc\examples> countchars
```

38. There is another construct in this line, a comment. Commentaries are textual remarks left by the programmer for the benefit of other human readers, and are ignored by the compiler. We will come back to commentaries in a more formal manner later.

We get the following box that many of you have already seen several times:[39]



Why?

Well, let's look at the logic of our program. We assumed (without any test) that argv[1] will contain the name of the file that we should count the characters of. But if the user doesn't supply this parameter, our program will pass a nonsense argument to fopen, with the obvious result that the program will fail miserably, making a trap, or exception that the system reports.

We return to the editor, and correct the faulty logic. Added code is in bold.

```
#include <stdio.h>
#include <stdlib.h>      (1)
int main(int argc,char *argv[])
{
     int count=0;// chars read
     FILE *infile;
     int c;

     if (argc < 2) {     (2)
          printf("Usage: countchars <file name>\n");
          exit(1);       (3)
     }
     infile = fopen(argv[1],"r");
     c = fgetc(infile);
     while (c != EOF) {
          count++;
          c = fgetc(infile);
     }
     printf("%d\n",count);
     return 0;
}
```

1) We need to include <stdlib.h> to get the prototype declaration of the exit() function that ends the program immediately.

2) We use the conditional statement "if" to test for a given condition. The general form of it is: if (condition) { … statements… } else { … statements… }.

3) We use the exit function to stop the program immediately. This function receives an integer argument that will be the result of the program. In our case we return the error code 1. The result of our program will be then, the integer 1.

Now, when we call countchars without passing it an argument, we obtain a nice message:

```
h:\lcc\examples> countchars
Usage: countchars <file name>
```

39. This is the display under Windows NT. In other systems like Linux for instance, you will get a "Bus error" message.

This is MUCH clearer than the incomprehensible message box from the system isn't it?

Now let's try the following:

```
h:\lcc\examples> countchars zzzssqqqqq
```

And we obtain the dreaded message box again.

Why?

Well, it is very unlikely that a file called "zzzssqqqqq" exists in the current directory. We have used the function fopen, but we didn't bother to test if the result of fopen didn't tell us that the operation failed, because, for instance, the file doesn't exist at all!

A quick look at the documentation of fopen (that you can obtain by pressing F1 with the cursor over the "fopen" word in Wedit) will tell us that when fopen returns a NULL pointer (a zero), it means the open operation failed. We modify again our program, to take into account this possibility:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[])
{
     int count=0;// chars read
     FILE *infile;
     int c;

     if (argc < 2) {
          printf("Usage: countchars <file name>\n");
          exit(1);
     }
     infile = fopen(argv[1],"r");
     if (infile == NULL) {
          printf("File %s doesn't exist\n",argv[1]);
          exit(1);
     }
     c = fgetc(infile);
     while (c != EOF) {
          count++;
          c = fgetc(infile);
     }
     printf("%d\n",count);
     return 0;
}
```

We try again:

```
H:\lcc\examples> lcc countchars.c
H:\lcc\examples> lcclnk countchars.obj
H:\lcc\examples> countchars sfsfsfsfs
File sfsfsfsfs doesn't exist
H:\lcc\examples>
```

Well this error checking works. But let's look again at the logic of this program.

Suppose we have an empty file. Will our program work?

If we have an empty file, the first fgetc will return EOF. This means the whole `while` loop will never be executed and control will pass to our printf statement. Since we took care of initializing our counter to zero at the start of the program, the program will report correctly the number of characters in an empty file: zero.

Still it would be interesting to verify that we are getting the right count for a given file. Well that's easy. We count the characters with our program, and then we use the DIR directive of windows to verify that we get the right count.

```
H:\lcc\examples>countchars countchars.c
466
H:\lcc\examples>dir countchars.c

07/01/00  11:31p                   492 countchars.c
               1 File(s)            492 bytes
```

Wow, we are missing 492-466 = 26 chars!

Why?

We read again the specifications of the fopen function. It says that we should use it in read mode with "r" or in binary mode with "rb". This means that when we open a file in read mode, it will translate the sequences of characters \r (return) and \n (new line) into ONE character. When we open a file to count all characters in it, we should count the return characters too.

This has historical reasons. The C language originated in a system called UNIX, actually, the whole language was developed to be able to write the UNIX system in a convenient way. In that system, lines are separated by only ONE character, the new line character.

When the MSDOS system was developed, dozens of years later than UNIX, people decided to separate the text lines with two characters, the carriage return, and the new line character. This provoked many problems with software that expected only ONE char as line separator. To avoid this problem the MSDOS people decided to provide a compatibility option for that case: `fopen` would by default open text files in text mode, i.e. would translate sequences of \r\n into \n, skipping the \r.

Conclusion:

Instead of opening the file with `fopen(argv[1], "r");` we use `fopen(argv[1], "rb");`, i.e. we force NO translation. We recompile, relink and we obtain:

```
H:\lcc\examples> countchars countchars.c
493

H:\lcc\examples> dir countchars.c

07/01/00  11:50p                   493 countchars.c
               1 File(s)            493 bytes
```

Yes, 493 bytes instead of 492 before, since we have added a "b" to the arguments of fopen!

Still, we read the docs about file handling, and we try to see if there are no hidden bugs in our program. After a while, an obvious fact appears: we have opened a file, but we never closed it, i.e. we never break the connection between the program, and the file it is reading. We correct this, and at the same time add some commentaries to make the purpose of the program clear.

```
/*------------------------------------------------------------
 Module:         H:\LCC\EXAMPLES\countchars.c
 Author:         Jacob
 Project:        Tutorial examples
 State:          Finished
 Creation Date:  July 2000
 Description:    This program opens the given file, and
                 prints the number of characters in it.
------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[])
```

```
{
     int count=0;
     FILE *infile;
     int c;

     if (argc < 2) {
          printf("Usage: countchars <file name>\n");
          exit(1);
     }
     infile = fopen(argv[1],"rb");
     if (infile == NULL) {
          printf("File %s doesn't exist\n",argv[1]);
          exit(1);
     }
     c = fgetc(infile);
     while (c != EOF) {
          count++;
          c = fgetc(infile);
     }
     fclose(infile);
     printf("%d\n",count);
     return 0;
}
```

The skeleton of the commentary above is generated automatically by the IDE. Just right-click somewhere in your file, and choose “edit description”.

Summary:

- A program is defined by its specifications. In this example, counting the number of characters in a file.
- A first working version of the specification is developed. Essential parts like error checking are missing, but the program “works” for its essential function.
- Error checking is added, and test cases are built.
- The program is examined for correctness, and the possibility of memory leaks, unclosed files, etc., is reviewed. Comments are added to make the purpose of the program clear, and to allow other people know what it does without being forced to read the program text.

1.8 Commentaries

The writing of commentaries, apparently simple, is, when you want to do it right, quite a difficult task. Let’s start with the basics.

Commentaries are introduced in two forms:

Two slashes `//` introduce a commentary that will last until the end of the line. No space should be present between the first slash and the second one.

A slash and an asterisk `/*` introduce a commentary that can span several lines and is only terminated by an asterisk and a slash, `*/`. The same rule as above is valid here too: no space should appear between the slash and the asterisk, and between the asterisk and the slash to be valid comment delimiters.

Examples:

```
// This is a one-line commentary. Here /* are ignored anyway.

/* This is a commentary that can span several lines. Note that here the
```

```
two slashes // are ignored too */
```

This is very simple, but the difficulty is not in the syntax of commentaries, of course, but in their content. There are several rules to keep in mind:

Always keep the commentaries current with the code that they are supposed to comment. There is nothing more frustrating than to discover that the commentary was actually misleading you, because it wasn't updated when the code below changed, and actually instead of helping you to understand the code it contributes further to make it more obscure.

Do not comment **what** are you doing but **why**. For instance:

```
record++; // increment record by one
     This comment doesn't tell anything the C code doesn't tell us
anyway.
     record++; //Pass to next record.
// The boundary tests are done at
// the beginning of the loop above
     This comment brings useful information to the reader.
```

At the beginning of each procedure, try to add a standard comment describing the purpose of the procedure, inputs/outputs, error handling etc.[40]

At the beginning of each module try to put a general comment describing what this module does, the main functions etc.

Note that you yourself will be the first guy to debug the code you write. Commentaries will help you understand again that hairy stuff you did several months ago, when in a hurry.

1.8.1 Standard comments

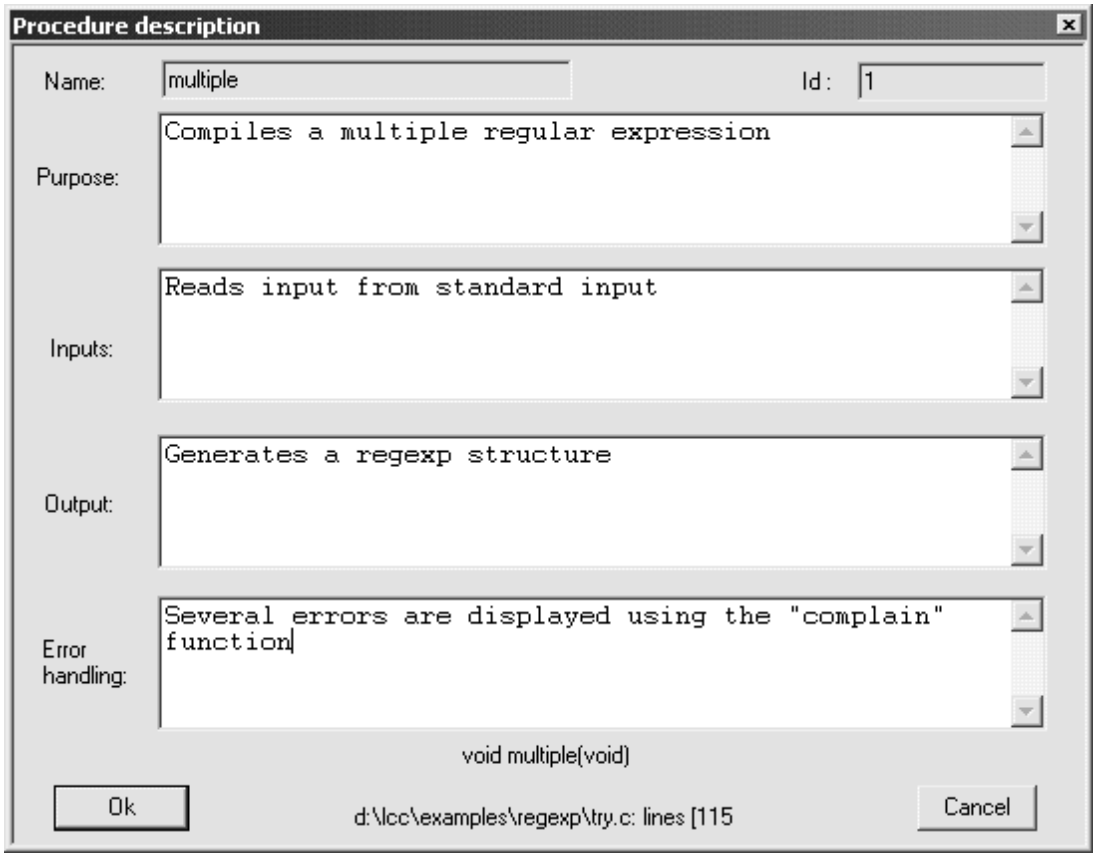
The editor of lcc-win32 provides a «Standard comments» feature. There are two types of comments supported: comments that describe a function, and comments that apply to a whole file. This comments are maintained by the editor that displays a simple interface for editing them.

1.8.1.1 Describing a function

You place the mouse anywhere within the body of a function and you click the right mouse button. A context menu appears that offers you to edit the description of the current function. The interface that appears by choosing this option looks like this:

40. The IDE of lcc-win32 helps you by automatic the construction of those comments. Just press, "edit description" in the right mouse button menu.

There are several fields that you should fill:



1) Purpose. This should explain what this function does, and how it does it.

2) Inputs: Here you should explain how the interface of this function is designed: the arguments of the function and global variables used if any.

3) Outputs. Here you should explain the return value of the function, and any globals that are left modified.

4) Error handling. Here you should explain the error return, and the behavior of the function in case of an error occurring within its body.

For the description provided in the screen shot above, the editor produces the following output:

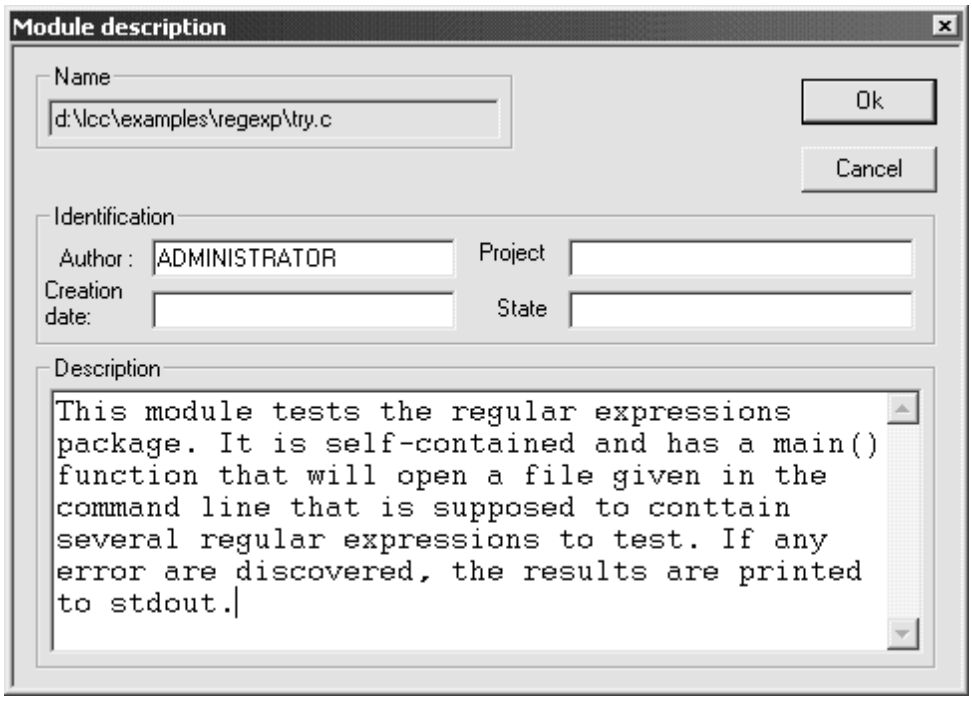
```
/*---------------------------------------------------------------
 Procedure:     multiple ID:1
 Purpose:       Compiles a multiple regular expression
 Input:         Reads input from standard input
 Output:        Generates a regexp structure
 Errors:        Several errors are displayed using the "complain"
                function
---------------------------------------------------------------*/
void multiple(void)
{
```

This comment will be inserted in the interface the next time you ask for the description of the function.

1.8.1.2 Describing a file

In the same context menu that appears with a right click, you have another menu item that says «description of file.c», where «file.c» is the name of the current file.

This allows you to describe what the file does. The editor will add automatically the name of the currently logged on user, most of the time the famous «administrator». The output of the interface looks like this:




```
/*-------------------------------------------------------------------
 Module:         d:\lcc\examples\regexp\try.c
 Author:         ADMINISTRATOR
 Project:
 State:
 Creation Date:
 Description:    This module tests the regular expressions
                 package. It is self-contained and has a main()
                 function that will open a file given in the
                 command line that is supposed to conttain
                 several regular expressions to test. If any
                 error are discovered, the results are printed
                 to stdout.
-------------------------------------------------------------------*/
```

As with the other standard comment, the editor will re-read this comment into the interface.

This features are just an aid to easy the writing of comments, and making them uniform and structured. As any other feature, you could use another format in another environment. You could make a simple text file that would be inserted where necessary and the fields would be tailored to the application you are developing. Such a solution would work in most systems too, since most editors allow you to insert a file at the insertion point.

1.9 An overview of the whole language

Let’s formalize a bit what we are discussing. Here are some tables that you can use as reference tables. We have first the words of the language, the statements. Then we have a dictio-

nary of some sentences you can write with those statements, the different declarations and control-flow constructs. And in the end is the summary of the pre-processor instructions. I have tried to put everything hoping that I didn't forget something.

You will find in the left column a more or less formal description of the construct, a short explanation in the second column, and an example in the third. In the first column, this words have a special meaning: "id", meaning an identifier, "type" meaning some arbitrary type and "expr" meaning some arbitrary C expression.

I have forced a page break here so that you can print these pages separately, when you are using the system.

1.9.1 Statements

Expression	*Meaning and value of result*	*Example*
`identifier`	The value associated with that identifier. (see “A closer view” on page 40.)	`id`
`constant`	The value defined with this constant (see “Constants.” on page 40.).	
	Integer constant.	`45 45L 45LL`
	Floating constant	`45.9 45.9f 45.9L`
	character constant	`‘A’ L’A’`
	String literal	`"Hello" L"Hello"`
`{ constants }`	Define tables or structure data	`{1,67}`
`integer constants`	Integer constants.	`45`
	long integer constants	`45L`
	long long (64 bits) integer constant	`45LL`
	octal constant (base 8) introduced with a leading zero	`055` (This is 45 in base 8)
	Hexadecimal constant introduced with 0x	`0x2d` (this is 45 in hexa)
	Binary constant introduced with 0b. This is an lcc-win32 extension.	`0b101101` (this is 45 in decimal)
`floating constants`	double precision constant	`45.9 or 4.59e2`
	Float (single precision) constant	`45.9f or 4.59e2f`
	long double constant	`45.9L or 4.59e2L`
`character constant`	char enclosed in simple quotes	`’a’ or ’8’`
`string literals`	enclosed in double quotes	`"a string"`
`Array [index ]`	Access the position “index” of the given array. Indexes start at zero (see “Within the string, the following abbreviations are recognized:” on page 41.)	`Table[45]`
`Array[i1][i2]`	Access the n dimensional array using the indexes i1, i2, … i*n*..See “Arrays.” on page 42.	`Table[34][23]` This access the 35^{th} line, 24^{th} position of `Table`
`fn ( args )`	Call the function “fn” and pass it the comma separated argument list «args». see “Function call syntax” on page 42.	`printf(“%d“,5)`
`fn (arg, ...)`	See “Functions with variable number of arguments.” on page 42.	

`(*fn)(args)`	Call the function whose machine address is in the pointer fn.	
`struct.field`	Access the member of the structure	`Customer.Name`
`struct->field`	Access the member of the structure through a pointer	`Customer->Name`
`var = value`	Assign to the variable[1] the value of the right hand side of the equals sign. See "Assignment." on page 43.	`a = 45`
`expression++`	Equivalent to expression = expression + 1. Increment expression after using its value. See "Postfix" on page 43..	`a = i++`
`expression--`	Equivalent to expression = expression – 1. Decrement expression after using its value. see "Postfix" on page 43.	`a = i--`
`++expression`	Equivalent to expression = expression+1. Increment expression before using its value.	`a = ++I`
`--expression`	Equivalent to Expression = expression – 1. Decrement expression before using it.	`a = --i`
`& object`	Return the machine address of object. The type of the result is a pointer to object.	`&i`
`* pointer`	Access the contents at the machine address stored in the pointer. .See "Indirection" on page 52.	`*pData`
`- expression`	Subtract expression from zero, i.e. change the sign.	`-a`
`~ expression`	Bitwise complement expression. Change all 1 bits to 0 and all 0 bits to 1.	`~a`
`! expression`	Negate expression: if expression is zero, !expression becomes one, if expression is different than zero, it becomes zero.	`!a`
`sizeof(expr)`	Return the size in bytes of expr. .see "sizeof." on page 45.	`sizeof(a)`
`(type) expr`	Change the type of expression to the given type. This is called a "cast". The expression can be a literal expression enclosed in braces, as in a structure initialization.	`(int *)a`
`expr * expr`	Multiply	`a*b`
`expr / expr`	Divide	`a/b`
`expr % expr`	Divide first by second and return the remainder	`a%b`
`expr + expr`	Add	`a+b`
`expr1 - expr2`	Subtract expr2 from expr1. .see "Subtraction." on page 43.	`a-b`
`expr1 << expr2`	Shift left expr1 expr2 bits.	`a << b`
`expr1 >> expr2`	Shift right expr1 expr2 bits.	`a >> b`
`expr1 < expr2`	1 if expr1 is smaller than expr2, zero otherwise	`a < b`
`expr1 <= expr2`	1 if expr1 is smaller or equal than expr2, zero otherwise	`a <= b`
`expr1 >= expr2`	1 if expr1 is greater or equal than expr2, zero otherwise	`a >= b`

`expr1 > expr2`	1 if expr2 is greater than expr2, zero otherwise	`a > b`
`expr1 == expr2`	1 if expr1 is equal to expr2, zero otherwise	`a == b`
`expr1 != expr2`	1 if expr1 is different from expr2, zero otherwise	`a != b`
`expr1 & expr2`	Bitwise AND expr1 with expr2. See “Bitwise operators” on page 50.	`a&8`
`expr1 ^ expr2`	Bitwise XOR expr1 with expr2. See “Bitwise operators” on page 50.	`a^b`
`expr1 | expr2`	Bitwise OR expr1 with expr2. See “Bitwise operators” on page 50.	`a|16`
`expr1 && expr2`	Evaluate expr1. If its result is zero, stop evaluating the whole expression and set the result of the whole expression to zero. If not, continue evaluating expr2. The result of the expression is the logical AND of the results of evaluating each expression. See “Logical operators” on page 49.	`a < 5 && a > 0` This will be 1 if “a” is between 1 to 4. If a >= 5 the second test is not performed.
`expr1 || expr2`	Evaluate expr1. If the result is one, stop evaluating the whole expression and set the result of the expression to 1. If not, continue evaluating expr2. The result of the expression is the logical OR of the results of each expression. See “Logical operators” on page 49.	`a == 5 ||a == 3` This will be 1 if either a is 5 or 3
`expr ? val1:val2`	If expr evaluates to non-zero (true), return val1, otherwise return val2. see “Conditional operator.” on page 44.	`a= b ? 2 : 3` a will be 2 if b is true, 3 otherwise
`expr *= expr1`	Multiply expr by expr1 and store the result in expr	`a *= 7`
`expr /= expr1`	Divide expr by expr1 and store the result in expr	`a /= 78`
`expr %= expr1`	Calculate the remainder of expr % expr1 and store the result in expr	`a %= 6`
`expr += expr1`	Add expr1 with expr and store the result in expr	`a += 6`
`expr -= expr1`	Subtract expr1 from expr and store the result in expr	`a -= 76`
`expr <<= expr1`	Shift left expr by expr1 bits and store the result in expr	`a <<= 6`
`expr >>= expr1`	Shift right expr by expr1 bits and store the result in expr	`a >>= 7`
`expr &= expr1`	Bitwise and expr with expr1 and store the result in expr	`a &= 32`
`expr ^= expr1`	Bitwise xor expr with expr1 and store the result in expr	`a ^= 64`
`expr |= expr1`	Bitwise or expr with expr1 and store the result in expr.	`a |= 128`
`expr , expr1`	Evaluate expr, then expr1 and return the result of evaluating the last expression, in this case expr1. .See page 51	`a=7,b=8` `The result of this is 8`
`;`	Null statement	`;`

1. Variable can be any value that can be assigned to: an array element or other constructs like *ptr = 5. In technical language this is called an “lvalue”.

1.9.2 Declarations[41]

Declaration	*Meaning*	*Example*
type id;	Identifier will have the specified type within this scope. In a local scope its value is undertemined. In a global scope, its initial value is zero, at program start.	`int a;`
type * id;	Identifier will be a pointer to objects of the given type. You add an asterisk for each level of indirection. A pointer to a pointer needs two asterisks, etc.	`int *pa;` pa will be a pointer to integers
type id[expr]	Identifier will be an array of expr elements of the given type. The expression must evaluate to a compile time constant or to a constant expression that will be evaluated at run time. In the later case this is a variable length array.	`int *ptrArray[56];` `Array of 56 int pointers.`
typedef old new	Define a new type-name for the old type. see “typedef.” on page 44.	`typedef unsigned int uint;`
register id;	Try to store the identifier in a machine register. The type of identifier will be equivalent to signed integer if not explicitly specified. see “register.” on page 44.	`register int f;`
extern type id;	The definition of the identifier is in another module. No space is reserved.	`extern int frequency;`
static type id	Make the definition of identifier not accessible from other modules.	`static int f;`

struct id {
declarations
}
	Define a compound type composed by the enumeration of fields enclosed within curly braces.	`struct coord {`
		`int x;`
		`int y;`
		`};`

type id:n	Within a structure field declaration, declare “id” as a sequence of n bits of type “type”. See “bit fields” on page 46.	`unsigned n:4` `n is an unsigned int of 4 bits`

union id {
declarations
};
	Reserve storage for the biggest of the declared types and store all of them in the same place. see “union.” on page 44.	`union dd {`
		`double d;`
		`int id[2];`
		`};`

enum identifier {
enum list
}
	Define an enumeration of comma-separated identifiers assigning them some integer value. see “enum.” on page 45.	`enum color {`
		`red,green,blue`
		`};`

const type identifier;	Declare that the given identifier can’t be changed (assigned to) within this scope. see “const.” on page 46.	`const int a;`
unsigned int-type	When applied to integer types do not use the sign bit. see “unsigned.” on page 46.	`unsigned char a = 178;`
volatile type identifier	Declare that the given object changes in ways unknown to the implementation.The compiler will not store this variable in a register, even if optimizations are turned on.	`volatile int hardware_clock;`

type id (args);	Declare the prototype for the given function. The arguments are a comma separated list. see “Prototypes.” on page 45.	`double sqrt(double x);`
type (*id)(args);	Declare a function pointer called “id” with the given return type and arguments list	`void (*fn)(int)`
id :	Declare a label.	`lab1:`

type fn(args) {
...
statements
...
}
	Definition of a function with return type <type> and arguments <args> .	`int add1(int x)`
		`{ return x+1;}`

inline	This is a qualifier that applies to functions. If present, it can be understood by the compiler as a specification to generate the fastest function call possible, generally by means of replicating the function body at each call site.	`double inline overPi(double a)`
		`{`
		`return a/3.14159;`
		`}`

41. Lcc-win32 doesn’t yet implement the keyword restrict.

1.9.3 Pre-processor

// commentary	Double slashes introduce comments up to the end of the line.see “Comments” on page 48.	`// comment`
/* commentary ***/**	Slash star introduces a commentary until the sequence star slash */ is seen. see “Comments” on page 48.	`/* comment */`
#**define** id text	Replace all appearances of the given identifier by the corresponding expression. See “Preprocessor commands” on page 134.	`#define TAX 6`
#**define** macro(a,b)	Define a macro with n arguments. When used, the arguments are lexically replaced within the macro. See page 135	`#define max(a,b) ((a)<(b)? (b):(a))`[1]
#**undef** id	Erase from the pre-processor tables the given identifier.	`#undef TAX`
#**include** <header.h>	Insert the contents of the given file from the standard include directory into the program text at this position.	`#include <stdio.h>`
#**include** "header.h"	Insert the contents of the given file from the current directory.	`#include “foo.h“`
#**ifdef** id	If the given identifier is defined (using #define) include the following lines. Else skip them. See page 136.	`#ifdef TAX`
#**ifndef** id	The contrary of the above	`#ifnef TAX`
#**if** (expr)	Evaluate expression and if the result is TRUE, include the following lines. Else skip all lines until finding an #else or #endif	`#if (TAX==6)`
#**else**	the else branch of an #if or #ifdef	`#else`
#**elif**	Abbreviation of #else #if	`#elif`
#**endif**	End an #if or #ifdef preprocessor directive statement	`#endif`
defined (id)	If the given identifier is #defined, return 1, else return 0.	`#if defined(max)`
##	Token concatenation	`a##b ab`
#token	Make a string with a token. Only valid within macro declarations	`#foo --> "foo"`
#**line** nn	Set the line number to nn	`#line 56`
#**file** "foo.c"	Set the file name	`#file “ff.c”`
#**error** errmsg	Show the indicated error to the user	`#error "undefined cpu"`
#**pragma** instructions	Special compiler directives[2]	`#pragma optimize(on)`
_Pragma(string)	Special compiler directives. This is a C99 feature.	`_Pragma("optimize (on)");`

****	If a \ appears at the end of a line just before the newline character, the line and the following line will be joined by the preprocessor and the \ character will be eliminated.	`/\`
		`* this is a comment`
		`*/`

__LINE__	Replace this token by the current line number	`printf("error line %d\n", __LINE__);`
__FILE__	Replace this token by the current file name	`printf("error file %s\n", __FILE__);`
__ func__	Replace this token by the name of the current function being compiled.	`printf("fn %s\n",__func__ );`
__STDC__	Defined as 1	`#if __STDC__`
__LCC__	Defined as 1 This allows you to conditionally include or not code for lcc-win32.	`#if __LCC__`

1. The parentheses ensure the correct evaluation of the macro.
2. The different pragma directives of lcc-win32 are explained in the user's manual.

1.9.4 Windows specific defined symbols

WIN32	#defined as 1	`#if WIN32`
_WIN32	#defined as 1	`#if _WIN32`
WINVER	Evaluates to the version of windows you are running	`#if WINVER == 5`
_WIN32_IE	Evaluates to the version of the internet explorer software installed in your system	`#if _WIN32_IE > 0x500`

1.9.5 Structured exception handling

__try { protected block }	Introduces a protected block of code.
__except (integer expression) { exception handler block }	If the integer expression evaluates to 1 the associated code block will be executed in case of an exception.
__leave;	Provokes an exit to the end of the current __try block
__retry;	Provokes a jump to the start of the try/except block

1.9.6 Control-flow

if (expression) { block}
else {
block
}
	If the given expression evaluates to something different than zero execute the statements of the following block. Else, execute the statements of the block following the else keyword. The else statement is optional. Note that a single statement can replace blocks.

while (expression) {
... statements ...
}
	If the given expression evaluates to something different than zero, execute the statements in the block, and return to evaluate the controlling expression again. Else continue after the block. See “while” on page 14.

do { ... statements ... } **while** (condition);	Execute the statements in the block, and afterwards test if condition is true. If that is the case, execute the statements again.See “do” on page 14.

for(init;test;incr) {
... statements ...
}
	Execute unconditionally the expressions in the init statement. Then evaluate the test expression, and if evaluates to true, execute the statements in the block following the for. At the end of each iteration execute the incr statements and evaluate the test code again. See “for” on page 13.

switch (expression) {
case int-expr:
statements ...
break;
default:
statements
}
	Evaluate the given expression. Use the resulting value to test if it matches any of the integer expressions defined in each of the ‘case’ constructs. If the comparison succeeds, execute the statements in sequence beginning with that case statement.
	If the evaluation of expression produces a value that doesn’t match any of the cases and a “default” case was specified, execute the default case statements in sequence. See “Switch statement.” on page 48.

goto label	Transfer control unconditionally to the given label.
continue	Within the scope of a for/do/while loop statement, continue with the next iteration of the loop, skipping all statements until the end of the loop.See “break and continue statements” on page 47.
break	Stop the execution of the current do/for/while loop statement.
return expression	End the current function and return control to the calling one. The return value of the function (if any) can be specified in the expression following the return keyword.

1.9.7 Windows specific syntax

`_stdcall`	Use the stdcall calling convention for this function or function pointer: the called function cleans up the stack. See “stdcall.” on page 47.	`int _stdcall fn(void);`
`__declspec (dllexport)`	Export this identifier in a DLL to make it visible from outside.	`int __declspec (dllexport) fn(int);`
`__declspec (naked)`	Do not add the standard prologue or epilogue to the function. The return instruction and any other code should be added as _asm() statements.	`int __declspec(naked) fn(int);`
`__int64`	#defined as long long for compatibility reasons with Microsoft’s compiler.	`__int64 big;`

1.10 Extensions of lcc-win32

operator opname (args)
{
}
	Redefine one of the operators like +, * or others so that instead of doing the normal operation, this function is called instead.	`T operator +(T a, T b) {`
		`... statements`
		`}`

type & id = expr;	Identifier will be a reference to a single object of the given type. References must be initialized immediately after their declaration.	`int &ra = a;` pa will be a reference to an integer.
int fn(int a,int b=0)	Default function arguments. If the argument is not given in a call, the compiler will fill it with the specified compile time constant	
int overloaded fn(int); int overloaded fn(double);	Generic functions. This functions have several forms of invocation, but the same name.	

1.11 A closer view

Let's explain a bit the terms above. The table gives a compressed view of C. Now let's see some of the details.

1.11.1 Identifiers.

An identifier is a sequence of non digit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits. Lowercase and uppercase letters are distinct. An identifier never starts with a digit. There is no specific limit on the maximum length of an identifier but lcc-win32 will give up at 255 chars.

Identifiers are the vocabulary of your software. When you create them, give a mnemonic that speaks about the data stored at that location.

Anonymous identifiers (or counters) are usually the one letters 'i', or "c" for char, etc. I think the habit of using i, j, k is quite ancient, maybe inherited from fortran and physics.

1.11.2 Constants.

1.11.2.1 Evaluation of constants

The expressions that can appear in the definition of a constant will be evaluated in the same way as the expressions during the execution of the program. For instance, this will put 1 into the integer constant d:

```
static int d = 1;
```

This will also put one in the variable d:

```
static int d = 60 || 1 +1/0;
```

1.11.2.2 Integer constants

An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type. A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 through 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and the letters a (or A) through f (or F) with values 10 through 15 respectively. Here are various examples of integer constants:

```
12345          (integer constant, decimal)
0777           (octal for 511 decimal)
0xF98A         (hexa for 63882 decimal)
12345L         (long integer constant)
2634455LL      (long long integer constant)
5488UL         (unsigned long constant)
548ULL         (unsigned long long constant)
```

1.11.2.3 Floating constants

For floating constants, the convention is either to use a decimal point (1230.0) or scientific notation (in the form of 1.23e3). They can have the suffix 'F' (or 'f') to mean that they are float constants, and not double constants as it is implicitly assumed when they have no suffix. A suffix of "l" or "L" means long double constant. A suffix of "q" or "Q" means a qfloat.[42]

42. Qfloats are an extension of lcc-win32.

1.11.2.4 Character string constants

For character string constants, they are enclosed in double quotes. If immediately before the double quote there is an "L" it means that they are double byte strings. Example:

```
L"abc"
```

This means that the compiler will convert this character string into a wide character string and store the values as double byte character string instead of just ASCII characters.

To include a double quote within a string it must be preceded with a backslash. Example:

```
"The string \"the string\" is enclosed in quotes"
```

Note that strings and numbers are completely different data types. Even if a string contains only digits, it will never be recognized as a number by the compiler: "162" is a string, and to convert it to a number you must explicitly write code to do the transformation.

Within the string, the following abbreviations are recognized:

Abbreviation	**Meaning**	**Value**
\n	New line	10
\r	carriage return	12
\b	backspace	8
\v	vertical tab	11
\t	tab	9
\f	form feed	12
\e	escape	27
\a	bell	7
\x<hex digits>	Insert at the current position the character with the integer value of the hexadecimal digits.	Any, since any digit can be entered. Example: "ABC\xA" is equivalent to "ABC\n"
\<octal number>	The same as the \x case above, but with values entered as 3 octal digits, i.e. numbers in base 8. Note that no special character is needed after the backslash. The octal digits start immediately after it.	Any. Example: The string "ABC\012" is equivalent to "ABC\n"

Character string constants that are too long to write in a single line can be entered in two ways:

```
char *a = "This is a long string that at the end has a backslash \
that allows it to go on in the next line";
```

Another way, introduced with C99 is:

```
char *a = "This is a long string written"
     "in two lines";
```

Note too that character string constants should not be modified by the program. Lcc-win32 stores all character string constants once, even if they appear several times in the program text. For instance if you write:

```
char *a = “abc”;
char *b = “abc”;
```

Both a and b will point to the SAME string, and if either is modified the other will not retain the original value.

1.11.3 Arrays.

Here are various examples for using arrays.

```
int a[45];   // Array of 45 elements
a[0] = 23;  // Sets first element to 23;
a[a[0]] = 56; // Sets the 24th element to 56
a[23] += 56; // Adds 56 to the 24th element
```

Multidimensional arrays are indexed like this:

```
int tab[2][3];
...
tab[1][2] = 7;
```

A table of 2 rows and three columns is declared. Then we assign 7 to the second row, third column. (Remember: arrays indexes start with zero).

Note that when you index a two dimensional array with only one index you obtain a pointer to the start of the indicated row.

```
int *p = tab[1];
```

Now p contains the address of the start of the second column.

1.11.4 Function call syntax

```
sqrt( hypo(6.0,9.0) ); // Calls the function hypo with
                       // two arguments and then calls
                       // the function sqrt with the
                       // result of hypo
```

An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.

A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to.

A parameter declared to have array or function type is converted to a parameter with a pointer type.

The order of evaluation of the actual arguments, and sub expressions within the actual arguments is unspecified. For instance:

```
fn( g(), h(), m());
```

Here the order of the calls to the functions g(), h() and m() is unspecified.

1.11.5 Functions with variable number of arguments.

To access the unnamed, extra arguments you should include <stdarg.h>. To access the additional arguments, you should execute the `va_start`, then, for each argument, you execute a `va_arg`. Note that if you have executed the macro `va_start`, you should always execute the `va_end` macro before the function exits. Here is an example that will add any number of integers passed to it. The first integer passed is the number of integers that follow.

```
#include <stdarg.h>

int va_add(int numberOfArgs, ...)
{
     va_list ap;
     int n = numberOfArgs;
```

```
        int sum = 0;

        va_start(ap,numberOfArgs);
        while (n--) {
             sum += va_arg(ap,int);
        }
        va_end(ap);
        return sum;
    }
```

We would call this function with

```
va_add(3,987,876,567);
```

or

```
va_add(2,456,789);
```

1.11.6 Assignment.

An assignment has the left hand side of the equal's sign that must be a value that can be assigned to, and the right hand side that can be any expression other than void.

```
int a = 789; // "a" is assigned 789
array[345] = array{123]+897; //An element of an array is assigned
Struct.field = sqrt(b+9.0); // A field of a structure is assigned
p->field = sqrt(b+9.0);
     /* A field of a structure is assigned through  a pointer. */
```

Within an assignment there is the concept of "L-value", i.e. any assignable object. You can't, for instance, write:

```
5 = 8;
```

The constant 5 can't be assigned to. It is not an "L-value", the "L" comes from the left hand side of the equals sign of course. In the same vein we speak of LHS and RHS as abbreviations for left hand side and right hand side of the equals sign in an assignment.

1.11.7 Postfix

This expressions increment or decrement the expression at their left side returning the old value. For instance:

```
array[234] = 678;
a = array[234]++;
```

In this code fragment, the variable a will get assigned 678 and the array element 234 will have a value of 679 after the expression is executed. In the code fragment:

```
array[234] = 678;
a = ++array[234];
```

The integer a and the array element at the 235^{th} position will both have the value 679.

When applied to pointers, these operators increment or decrement the pointer to point to the next or previous *element*. Note that if the size of the object those pointers point to is different than one, the pointer will be incremented or decremented by a constant different than one too.

1.11.8 Subtraction.

When two pointers are subtracted they have to have the same type, and the result is the difference of the subscripts of the two array elements or, in other words, the number of elements

between both pointers. The size of the result is implementation-defined, and its type (a signed integer type) is ptrdiff_t defined in the <stddef.h> header.[43]

When an integer expression is subtracted (or added) to a pointer, it means to increase the pointer by that number of elements. For instance if the pointer is pointing to the 3rd element of an array of structures, adding it 2 will provoke to advance the pointer to point to the 5th element.

1.11.9 Conditional operator.

The first operand of the conditional expression is evaluated first. The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result of the whole expression is the value of the second or third operand (whichever is evaluated), converted to the type described below.

If both the second and the third operand have an arithmetic type, the result of the expression has that type. If both are structures, the result is a structure. If both are void, the result is void. This expressions can be nested.

```
int a = (c == 66) ? 534 : 698;
```

the integer a will be assigned 534 if c is equal to 66, 698 otherwise.

```
struct b *bb = (bstruct == NULL) ? NULL : b->next;
```

If bstruct is different than NULL, the pointer bb will receive the “next” field of the structure, otherwise bb will be set to NULL.

1.11.10 struct.

A structure or a union can’t contain another structure that hasn’t been fully specified, but they can contain a pointer to such a structure since the size of any pointer is always fixed. To build recursive structures like list you should specify a pointer to the structure, see “Lists” on page 128.. For a detailed description of this keyword see “Structures” on page 82.

1.11.11 union.

You can store several values in a single memory location or a group of memory locations with the proviso that they can’t be accessed at the same time of course. This allows you to reduce the memory requirements of a structure, or to interpret a sequence of bits in a different fashion. For a detailed discussion see “Unions” on page 87.

1.11.12 typedef.

The typedef keyword defines a name that can be used as a synonym for a type or derived type. In contrast to the struct, union, and enum declarations, typedef declarations doesn’t introduce new types — it introduces new names for *existing* types.

1.11.13 register.

This keyword is a recommendation to the compiler to use a machine register for storing the values of this type. The compiler is free to follow or not this directive. The type must be either an integer type or a pointer. If you use this declaration, note that you aren’t allowed to use the address-of operator since registers do not have addresses. Lcc-win32 tries to honor your rec-

43. typedef int ptrdiff_t;

ommendations, but it is better not to use this declaration and leave the register usage to the compiler.

Registers are the highest part of your machine memory hierarchy. They are the fastest storage available to the program by the circuit, and in a PC x86 architecture there are just a few of them available at a time.

After registers there is the level 1 cache, level 2 cache, main memory, then the disk, in order of decreasing access speed.

1.11.14 sizeof.

The result of sizeof is normally a constant integer known when the compiler is running. For instance `sizeof(int)` will yield under lcc-win32 the constant 4. In the case of a variable length array however, the compiler can't know its size on advance, and it will be forced to generate code that will evaluate the size of the array when the program is running.

1.11.15 enum.

An enumeration is a sequence of symbols that are assigned integer values by the compiler. The symbols so defined are equivalent to integers, and can be used for instance in switch statements. The compiler starts assigning values at zero, but you can change the values using the equals sign. An enumeration like `enum { a,b,c};` will provoke that a will be zero, b will be 1, and c will be 2. You can change this with `enum {a=10,b=25,c=76};`

1.11.16 Prototypes.

A prototype is a description of the return value and the types of the arguments of a function. The general form specifies the return value, then the name of the function. Then, enclosed by parentheses, come a comma-separated list of arguments with their respective types. If the function doesn't have any arguments, you should write 'void', instead of the argument list. If the function doesn't return any value you should specify void as the return type. At each call, the compiler will check that the type of the actual arguments to the function is a correct one.

The compiler cannot guarantee, however, that the prototypes are consistent across different compilation units. For instance if in file1.c you declare:

```
int fn(void);
```

then, the call

```
fn();
```

will be accepted. If you then in file2.c you declare another prototype

```
void fn(int);
```

and then you use:

```
fn(6);
```

the compiler cannot see this, and the program will be in error, crashing mysteriously at run time. This kind of errors can be avoided if you always declare the prototypes in a header file that will be included by all files that use that function. Do not declare prototypes in a source file if the function is an external one.

1.11.17 variable length array.

This arrays are based on the evaluation of an expression that is computed when the program is running, and not when the program is being compiled. Here is an example of this construct:

```
int Function(int n)
{
```

```
        int table[n];
        …
}
```

The array of integers called “table” has n elements. This “n” is passed to the function as an argument, so its value can’t be known in advance. The compiler generates code to allocate space for this array in the stack when this function is entered. The storage used by the array will be freed automatically when the function exits.

1.11.18 const.

Constant values can’t be modified. The following pair of declarations demonstrates the difference between a ‘‘variable pointer to a constant value’’ and a ‘‘constant pointer to a variable value’’.

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of any object pointed to by `ptr_to_constant` shall not be modified through that pointer, but `ptr_to_constant` itself may be changed to point to another object. Similarly, the contents of the int pointed to by `constant_ptr` may be modified, but `constant_ptr` itself shall always point to the same location.

1.11.19 unsigned.

Integer types (long long, long, int, short and char) have the most significant bit reserved for the sign bit. This declaration tells the compiler to ignore the sign bit and use the values from zero the 2^n for the values of that type. For instance, a signed short goes from –32767 to 32767, an unsigned short goes from zero to 65535 (2^{16}). See the standard include file <stdint.h> for the ranges of signed and unsigned integer types.

1.11.20 bit fields

A "bit field" is an unsigned or signed integer composed of some number of bits. Lcc-win32 will accept some other type than int for a bit field, but the real type of a bit field will be always either "int" or "unsigned int".

For example, in the following structure, we have 3 bit fields, with 1, 5, and 7 bits:

```
struct S {
     int a:1;
     int b:5;
     int c:7;
};
```

With lcc-win32 the size of this structure will be 4 with no special options. With maximum packing (-Zp1 option) the size will be two.

When you need to leave some space between adjacent bit fields you can use the notation:

```
     unsigned : n;
```

For example

```
struct S {
     int a:1;
     int b:5;
     unsigned:10;
     int c:7;
};
```

Between the bit fields *a* and *b* we leave 10 unused bits so that *c* starts at a 16 bit word boundary.

1.11.21 stdcall.

Normally, the compiler generates assembly code that pushes each argument to the stack, executes the "call" instruction, and then adds to the stack the size of the pushed arguments to return the stack pointer to its previous position. The stdcall functions however, return the stack pointer to its previous position before executing their final return, so this stack adjustment is not necessary.

The reason for this is a smaller code size, since the many instructions that adjust the stack after the function call are not needed and replaced by a single instruction at the end of the called function.

Functions with this type of calling convention will be internally "decorated" by the compiler by adding the stack size to their name after an "@" sign. For instance a function called fn with an integer argument will get called fn@4. The purpose of this "decorations" is to force the previous declaration of a stdcall function so that always we are sure that the correct declarations was seen, if not, the program doesn't link.

1.11.22 break and continue statements

The break and continue statements are used to break out of a loop or switch, or to continue a loop at the test site. They can be explained in terms of the goto statement:

```
while (condition != 0) {
          doSomething();
          if (condition == 0)
               break;
          doSomethingElse();
}
```

is equivalent to:

```
while (condition != 0) {
          doSomething();
          if (condition == 0)
               goto lab1;
          doSomethingElse();
}
lab1:
```

The continue statement can be represented in a similar way:

```
while (condition != 0) {
          doSomething();
          if (condition == 25)
               continue;
          doSomethingElse();
}
```

is equivalent to:

```
restart:
while (condition != 0) {
          doSomething();
          if (condition == 25)
               goto restart;
          doSomethingElse();
}
```

The advantage of avoiding the goto statement is the absence of a label. Note that in the case of the "for" statement, execution continues with the increment part.

Remember that the continue statement within a switch statement doesn't mean that execution will continue the switch but continue the next enclosing for, while, or do statement.

1.11.23 Null statements

A null statement is just a semicolon. This is used in two contexts:

1) An empty body of an iterative statement (while, do, or for). For instance you can do:

```
while (*p++)
          ;  /* search the end of the string */
```

2) A label should appear just before a closing brace. Since labels must be attached to a statement, the empty statement does that just fine.

1.11.24 Comments

Multi-line comments are introduced with the characters "/" and "*" and finished with the opposite sequence: "*" followed by "/". This commentaries can't be nested. Single line comments are introduced by the sequence "//" and go up to the end of the line. Here are some examples:

```
"a//b"    Four character string literal
// */     Single line comment, not syntax error
f = g/**//h;Equivalent to f = g/h;
//\
fn();Part of a comment since the last line ended with a "\"
```

1.11.25 Switch statement.

The purpose of this statement is to dispatch to several code portions according to the value in an integer expression. A simple example is:

```
enum animal {CAT,DOG,MOUSE};

enum animal pet = GetAnimalFromUser();
switch (pet) {
     case CAT:
          printf("This is a cat");
          break;
     case DOG:
          printf("This is a dog");
          break;
     case MOUSE:
          printf("This is a mouse");
          break;
     default:
          printf("Unknown animal");
          break;
}
```

We define an enumeration of symbols, and call another function that asks for an animal type to the user and returns its code. We dispatch then upon the value of the In this case the integer expression that controls the switch is just an integer, but it could be any expression. Note that the parentheses around the switch expression are mandatory. The compiler generates code that evaluates the expression, and a series of jumps (gotos) to go to the corresponding portions of the switch. Each of those portions is introduced with a "case" keyword that is followed by an

integer constant. Note that no expressions are allowed in cases, only constants that can be evaluated by the compiler during compilation.

Cases end normally with the keyword “break” that indicates that this portion of the switch is finished. Execution continues after the switch. A very important point here is that if you do not explicitly write the break keyword, execution will continue into the next case. Sometimes this is what you want, but most often it is not. Beware.

There is a reserved word “default” that contains the case for all other values that do not appear explicitly in the switch. It is a good practice to always add this keyword to all switch statements and figure out what to do when the input doesn’t match any of the expected values.

If the input value doesn’t match any of the enumerated cases and there is no default statement, no code will be executed and execution continues after the switch.

Conceptually, the switch statement above is equivalent to:

```
if (pet == CAT) {
     printf("This is a cat");
}
else if (pet == DOG) {
     printf("This is a dog");
}
else if (pet == MOUSE) {
     printf("This is a mouse");
} else printf("Unknown animal");
```

Both forms are exactly equivalent, but there are subtle differences:

Switch expressions must be of integer type. The “if” form doesn’t have this limitation.

In the case of a sizeable number of cases, the compiler will optimize the search in a switch statement to avoid comparisons. This can be quite difficult to do manually with “if”s.

Cases of type other than int, or ranges of values can’t be specified with the switch statement, contrary to other languages like Pascal that allows a range here. Switch statements can be nested to any level (i.e. you can write a whole switch within a case statement), but this makes the code unreadable and is not recommended.

1.11.26 inline

This instructs the compiler to replicate the body of a function at each call site. For instance:

```
int inline f(int a) { return a+1;}
```

Then:

```
int a = f(b)+f(c);
```

will be equivalent to writing:

```
int a = (b+1)+(c+1);
```

Note that this expansion is realized in the lcc-win32 compiler only when optimizations are ON. In a normal (debug) setting, the “inline” keyword is ignored. You can control this behavior also, by using the command line option “`-fno-inline`”.

1.11.27 Logical operators

A logical expression consists of two boolean expressions (i.e. expressions that are either true or false) separated by one of the logical operators && (AND) or || (or).

The AND operator evaluates from left to right. If any of the expressions is zero, the evaluation stops with a FALSE result and the rest of the expressions is not evaluated. The result of several AND expressions is true if and only if all the expressions evaluate to TRUE.

Example:

```
1 && 1 && 0 && 1 && 1
```

Here evaluation stops after the third expression yields false (zero). The fourth and fifth expressions are not evaluated. The result of all the AND expressions is zero.

The OR operator evaluates from left to right. If any of the expressions yields TRUE, evaluation stops with a TRUE result and the rest of the expressions is not evaluated. The result of several OR expressions is true if and only if one of the expressions evaluates to TRUE.

If we have the expression:

```
result = expr1 && expr2;
```

this is equivalent to the following C code:

```
if (expr1 == 0)
     result = 0;
else {
     if (expr2 == 0)
          result = 0;
     else result = 1;
}
```

In a similar way, we can say that the expression

```
result = expr1 || expr2;
```

is equivalent to:

```
if (expr1 != 0)
     result = 1;
else {
     if (expr2 != 0)
          result = 1;
     else result = 0;
}
```

1.11.28 Bitwise operators

The operators & (bitwise AND), ^ (bitwise exclusive or), and | (bitwise or) perform boolean operations between the bits of their arguments that must be integers: long long, long, int, short, or char.

The operation of each of them is as follows:

1) The & (AND) operator yields a 1 bit if both arguments are 1. Otherwise it yields a 0.

2) The ^ (exclusive or) operator yields 1 if one argument is 1 and the other is zero, i.e. it yields 1 if their arguments are different. Otherwise it yields zero

3) The | (or) operator yields 1 if either of its arguments is a 1. Otherwise it yields a zero.

We can use for those operators the following truth table:

a	b	a&b	a^b	a|b
0	0	0	0	0
0	1	0	1	1

1	0	0	1	1
1	1	1	0	1

Note that this operators are normal operators, i.e. they evaluate always their operands, unlike && or || that use short-circuit evaluation. If we write:

```
a = 0 && fn(67);
```

the function call will never be executed. If we write

```
a = 0&fn(67);
```

the function call will be executed even if the result is fixed from the start.

1.11.29 Address-of operator

The unary operator & yields the machine address of its argument that must be obviously an addressable object. For instance if you declare a variable as a "register" variable, you can't use this operator to get its address because registers do not live in main memory. In a similar way, you can't take the address of a constant like &45 because the number 45 has no address.

The result of the operator & is a pointer with the same type as the type of its argument. If you take the address of a short variable, the result is of type "pointer to short". If you take the address of a double, the result is a pointer to double, etc.

If you take the address of a local variable, the pointer you obtain is valid only until the function where you did this exits. Afterward, the pointer points to an invalid address and will produce a machine fault when used, if you are lucky. If you are unlucky the pointer will point to some random data and you will get strange results, what is much more difficult to find.

In general, the pointer you obtain with this operator is valid only if the storage of the object is pointing to is not released. If you obtain the address of an object that was allocated using the standard memory allocator malloc, this pointer will be valid until there is a "free" call that releases that storage. Obviously if you take the address of a static or global variable the pointer will be always valid since the storage for those objects is never released.

Note that if you are using the memory manager (gc), making a reference to an object will provoke that the object is not garbage collected until at least the reference goes out of scope.

1.11.30 Sequential expressions

A comma expression consists of two expressions separated by a comma. The left operand is fully evaluated first, and if it produces any value, that value will be discarded. Then, the right operand is evaluated, and its result is the result of the expression.

For instance:

```
p = (fn(2,3),6);
```

The "p" variable will always receive the value 6, and the result of the function call will be discarded.

Do not confuse this usage of the comma with other usages, for example within a function call. The expression:

```
fn(c,d=6,78);
```

is always treated as a function call with three arguments, and not as a function call with a comma expression. Note too that in the case of a function call the order of evaluation of the different expressions separated by the comma is undefined, but with the comma operator it is well defined: always from left to right.

1.11.31 Casts

A cast expression is the transformation of an object from one type to another. For instance, a common need is to transform double precision numbers into integers. This is specified like this:

```
double d;
...
(int)d
```

In this case, the cast needs to invoke run time code to make the actual transformation. In other cases there is no code emitted at all. For instance in:

```
void *p;
...
(char *)p;
```

Transforming one type of pointer into another needs no code at all at run-time.

You can use a cast expression in another context, to indicate the type of a composite constant literal. For instance:

```
typedef struct tagPerson {
     char Name[75];
     int age;
} Person;

void process(Person *);
...
     process(&(Person){“Mary Smith” , 38});
```

This is one of the new features of C99. The literal should be enclosed in braces, and it should match the expected structure. This is just “syntactic sugar” for the following:

```
Person __998815544ss = { “Mary Smith”, 38};
          process(&__998815544ss);
```

The advantage is that now you are spared that task of figuring out a name for the structure since the compiler does that for you. Internally however, that code represents exactly what happens inside lcc-win32.

Casts, as any other of the constructs above, can be misused. In general, they make almost impossible to follow the type hierarchy automatically. C is weakly typed, and most of the “weakness” comes from casts expressions.

1.11.32 Indirection

The * operator is the contrary of the address-of operator above. It expects a pointer and returns the object the pointer is pointing to. For instance if you have a pointer pint that points to an integer, the operation *pint will yield the integer value the pointer pint is pointing to.

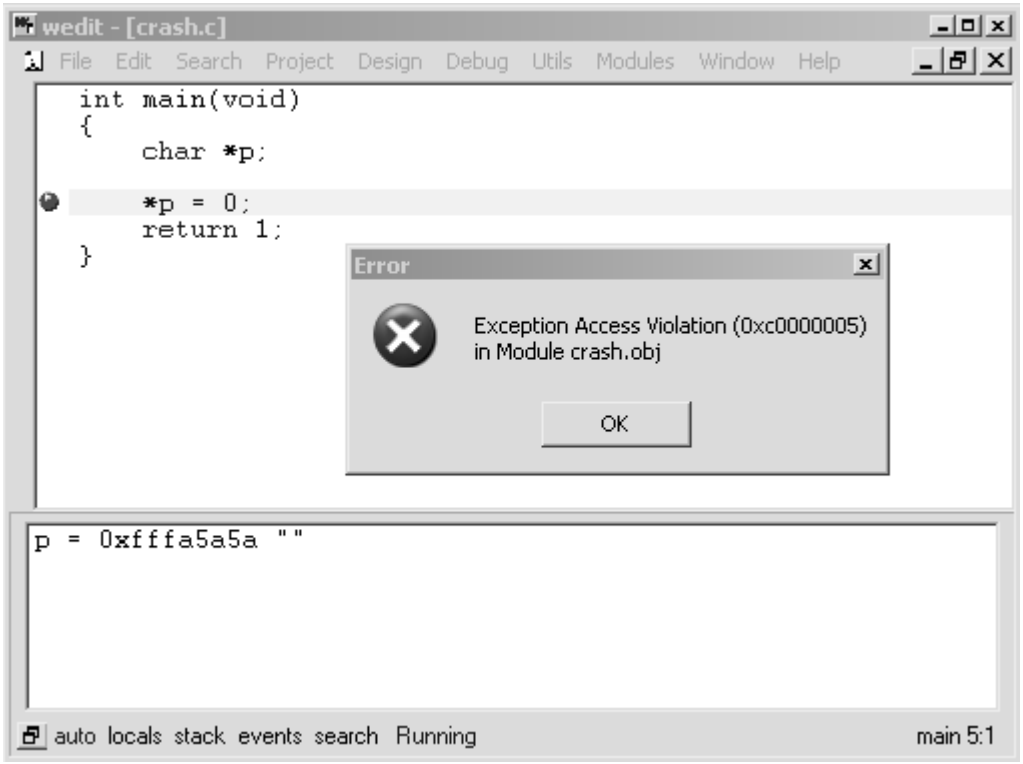
The result of this operator is invalid if the pointer it is de referencing is not valid. In some cases, de referencing an invalid pointer will provoke the dreaded window “This program has performed an invalid operation and will terminate” that windows shows up when a machine fault is detected. In other cases, you will be unlucky and the de referencing will yield a non-sense result.

For instance, this program will crash:

```
int main(void)
{
     char *p;

     *p = 0;
```

```
        return 1;
}
```

We have never assigned to p an object where it should point to. We are using a dangling pointer. When we follow this program we obtain:



The debugger tells us that a machine exception has occurred, with the code 0xc0000005. This means that the CPU has detected an invalid memory reference and has called the exception mechanism of windows, that notified the debugger of the problem. Note the value of the pointer in the output window: `0xfffa5a5a`.

Lcc-win32 follows the philosophy that the sooner you see an error, the better. When it allocates the stack frame of the function, it will write this value to all memory that has not been explicitly initialized by the program. When you see this value in the debugger you can be highly confident that this is an unitialized pointer or variable. This will not be done if you turn on optimizations. In that case the pointer will contain whatever was in there when the compiler allocated the stack frame.

Note that many other compilers do not do this, and some programs run without crashing out of sheer luck. Since lcc-win32 catches this error, it looks to the users as if the compiler was buggy. I have received a lot of complaints because of this.

This kind of problem is one of the most common bugs in C. Forgetting to initialize a pointer is something that you can never afford to do.

Another error is initializing a pointer within a conditional expression:

```
char *BuggyFunction(int a)
{
     char *result;

     if (a > 34) {
          result = malloc(a+34);
     }
     return result;
```

```
}
```

If the argument of this function is less than 35, the pointer returned will be a dangling pointer since it was never initialized.

1.11.33 Precedence of the different operators.

In their book «C, a reference manual», Harbison and Steele propose the following table.

Tokens	Operator	Class	Precedence	Associates
`names, literals`	simple tokens	primary	16	n/a
`a[k]`	subscripting	postfix	16	left-to-right
`f(...)`	function call	postfix	16	left-to-right
`. (point)`	direct selection	postfix	16	left-to-right
`->`	indirect selection	postfix	16	left-to-right
`++ --`	increment, decrement	postfix	16	left-to-right
`(type name){init}`	compound literal	postfix	16	left-to-right
`++ --`	increment, decrement	prefix	15	right-to-left
`sizeof`	size	unary	15	right-to-left
`~`	bitwise not	unary	15	right-to-left
`!`	logical not	unary	15	right-to-left
`- +`	arithmetic negation, plus	unary	15	right-to-left
`&`	address of	unary	15	right-to-left
`*`	indirection	unary	15	right-to-left
`( type name )`	casts	unary	14	right-to-left
`* / %`	multiplicative	binary	13	left-to-right
`+ -`	additive	binary	12	left-to-right
`<<  >>`	left/right shift	binary	11	left-to-right
`< > <= >=`	relational	binary	10	left-to-right
`==  !=`	equal/not equal	binary	9	left-to-right
`&`	bitwise and	binary	8	left-to-right
`^`	bitwise xor	binary	7	left-to-right
`|`	bitwise or	binary	6	left-to-right
`&&`	logical and	binary	5	left-to-right
`||`	logical or	binary	4	left-to-right
`? :`	conditional	binary	2	right-to-left
`= += -= *= /= %= <<= >>= &= ^= |=`	assignment	binary	2	right-to-left
`,`	sequential evaluation	binary	1	left-to-right

1.12 The printf family

The functions fprintf, printf, sprintf and snprintf are a group of functions to output formatted text into a file (fprintf, printf) or a character string (sprintf). The snprintf is like sprintf function, but accepts a count of how many characters should be put as a maximum in the output string.

Function	Prototype
fprintf	`int fprintf(FILE * stream,const char *fmt, ...);`
printf	`int printf(const char *fmt,...);`
sprintf	`char *outputstring, const char *fmt,...);`
snprintf	`int snprintf(char *out,size_t maxchars,const char *fmt, ...);`

The printf function is the same as fprintf, with an implicit argument “stdout”, i.e. the standard output of the program, that in most cases is the console window.

```
fprintf(stdout,"hello\n"); <---> printf("hello\n");
```

The value returned by all this functions is EOF (End Of File, usually -1) if an error occurred during the output operation. Otherwise, all went OK and they return the number of characters written. For sprintf, the returned count does not include the terminating zero appended to the output string.

The “fmt” argument is a character string or “control string”. It contains two types of characters: *normal characters* that are copied to the output without any change, and *conversion specifications*, that instruct the function how to format the next argument. In the example above, we have just the string “hello\n”, without any conversion specification so that character string is simply copied to the destination.

There should be always at least as many arguments to this functions as there are conversion specifications. If you fail to do this with lcc-win32, you will get a warning from the compiler. Other compilers can be less user friendly, so do not rely on that.

1.12.1 Conversions

A conversion specification begins with a percent sign (%) and is made of the following elements

1) Zero or more flag characters (-, +, 0, #, ‘, or space), which modify the meaning of the operation.[44]

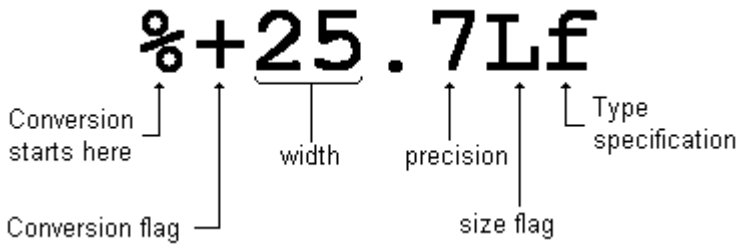
2) An optional **minimum** field with.Note well this. The printf function will not truncate a field. The specified width is just that: a minimum.

3) An optional precision field made of a period followed by a number.

4) An optional size flag, expressed as one of the letters **`ll`**, **`l`**, **`L`**, **`h`**, **`hh`**, **`j`**, **`q`**, **`t`**, or **`z`**.

5) The type specification, a single character from the set a,A, c, d, e, E, f, g, G, i, n, o, p, s, u, x, X, and %.

44. the ‘ (quote) char is specific to lcc-win32.



1.12.2 The conversion flags

- (minus)	Value will be left justified. The pad character is space.
0	Use zero as pad character instead of the default space. This is relevant only if a minimum field width is specified, otherwise there is no padding. If the data requires it, the minimum width is not honored. Note that the padding character will be always space when padding is introduced right of the data.
+	Always add a sign, either + or -. Obviously, a minus flag is always written, even if this flag is not specified.
' (single quote)	Separate the digits of the formatted numbers in groups of three. For instance 123456 becomes 123,456. This is an lcc-win32 extension.
space	Use either space or minus, i.e. the + is replaced by a space.
#	use a variant of the main version algorithm

1.12.2.1 The minimum field width

This specifies that if the data doesn't fit the given width, the pad character is inserted to increase the field. If the data exceeds the field width the field will be bigger than the width setting. Numbers are written in decimal without any leading zeroes, that could be misunderstood with the 0 flag.

1.12.2.2 The precision

In floating point numbers (formats g G f e E) this indicates the number of digits to be printed after the decimal point. Used with the s format (strings) it indicates the *maximum* number of characters that should be used from the input string. If the precision is zero, the floating point number will not be followed by a period and rounded to the nearest integer.

1.12.3 The size specification

This table shows the different size specifications, together with the formats they can be used

l	d,i,o,u,x, X	The letter **l** with this formats means `long` or `unsigned long`.
l	n	The letter **l** with the n format means `long *`.
l	c	Used with the c (character) format, it means the character string is in wide character format.
l	all others	No effect, is ignored
l l	d,i,o,u,x,X	The letters **ll** mean `long long` or `unsigned long long.`
l l	n	With this format, **ll** means `long long *`.
h	d,i,o,u,x,X	With this formats, **h** indicates `short` or `unsigned short.`
h	n	Means `short *`.
h h	d,i,o,u,x,X	Means `char` or `unsigned char`.
h h	n	Means `char *` or `unsigned char *`.
L	A, a, E, e, F, f, G, g,	Means that the argument is a `long double`. Notice that the `l` modifier has no effect with those formats. It is uppercase **L**.
j	d, i, o, u, x, X	Means the argument is of type `intmax_t`, i.e. the biggest integer type that an implementation offers. In the case of lcc-win32 this is `long long`.
q	f,g,e	Means the argument is of type `qfloat` (350 bits precision). This is an extension of lcc-win32.
t	d, i, o, u, x, X	Means the argument is `ptrdiff_t`, under lcc-win32 `int`.
z	d, i, o, u, x, X	Means the argument is `size_t`, in lcc-win32 `unsigned int`.

with.

Well now we can pass to the final part.

1.12.4 The conversions

d,i	Signed integer conversion is used and the argument is by default of type `int`. If the **h** modifier is present, the argument should be a `short`, if the **ll** modifier is present, the argument is a `long long.`
u	Unsigned decimal conversion. Argument should be of type `unsigned int` (default), `unsigned short` (`h` modifier) or `unsigned long long` (`ll` modifier).
o	Unsigned octal conversion is done. Same argument as the `u` format.
x,X	Unsigned hexadecimal conversion. If `x` is used, the letters will be in lower case, if `X` is used, they will be in uppercase. If the # modifier is present, the number will be prefixed with `0x`.
c	The argument is printed as a character or a wide character if the l modifier is present.
s	The argument is printed as a string, and should be a pointer to byte sized characters (default) or wide characters if the l modifier is present.If no precision is given all characters are used until the zero byte is found. Otherwise, the maximum number of characters used is the given precision.
p	The argument is a pointer and is printed in pointer format. Under lcc-win32 this is the same as the unsigned format (`#u`).
n	The argument is a pointer to int (default), pointer to short (`h` modifier) or pointer to long long (`ll` modifier). Contrary to all other conversions, this conversion writes the number of characters written so far in the address pointed by its argument.
e, E	Signed decimal floating point conversion..Argument is of type `double` (default), or `long double` (with the `L` modifier) or `qfloat` (with the `q` modifier). The result will be displayed in scientific notation with a floating point part, the letter 'e' (for the e format) or the letter E (for the E format), then the exponent. If the precision is zero, no digits appear after the decimal point, and no point is shown. If the # flag is given, the point will be printed.
f, F	Signed decimal floating point conversion. Argument is of type double (default), or long double (with the `L` modifier). If the argument is the special value infinite, `inf` will be printed. If the argument is the special value NAN the letters `nan` are written.
g, G	This is the same as the above but with a more compact representation. Arguments should be floating point. Trailing zeroes after the decimal point are removed, and if the number is an integer the point disappears. If the # flag is present, this stripping of zeroes is not performed. The scientific notation (as in format `e`) is used if the exponent falls below -4 or is greater than the precision, that defaults to 6.
%	How do you insert a % sign in your output? Well, by using this conversion: %%.

1.13 setjmp and longjmp

This two functions implement a jump across function calls to a defined place in your program. You define a place where it would be wise to come back to, if an error appears in any of the procedures below this one.

For instance you will engage in the preparation of a buffer to send to the database., or some other lengthy operation that can fail. Memory can be exhausted, the disk can be full (yes, that can still arrive, specially when you get a program stuck in an infinite write loop...), or the user can become fed up with the waiting and closes the window, etc. etc.

For all those cases, you devise an exit with longjmp, into a previously saved context. The classical example is given by Harbison and Steele:

```
#include <setjmp.h>
jmp_buf ErrorEnv;

int guard(void)
/* Return 0 if successful; else lonjmp code */
{
     int status = setjmp(ErrorEnv);
     if (status != 0)
          return status; /* error */
     process();
     return 0;
}

int process(void)
{
     int error_code;
     ...
     if (error_happened) longjmp(ErrorEnv,error_code);
     ...
}
```

With all respect I have for Harbison and Steele and their excellent book, this example shows how NOT to use setjmp/longjmp.

The ErrorEnv global variable is left in an undefined state after the function exits with zero. When you use this facility utmost care must be exercised to avoid executing a longjmp to a function that has already exited. This will always lead to catastrophic consequences. After this function exists with zero, the contents of the global ErrorEnv variable are a bomb that will explode your program if used. Now, the process() function is entirely tied to that variable and its validity. You can’t call process() from any other place. A better way could be:

```
#include <setjmp.h>
jmp_buf ErrorEnv;

int guard(void)
/* Return 0 if successful; else longjmp code */
{
     jmp_buf pushed_env;
     memcpy(push_env,ErrorEnv,sizeof(jmp_buf));
     int status = setjmp(ErrorEnv);
     if (status == 0)
          process();
     memcpy(ErrorEnv, pushed_env, sizeof(jmp_buf));
     return status;
}

int process(void)
```

```
        {
             int error_code=0;
             ...
             if (error_code) longjmp(ErrorEnv,error_code);
             ...
}
```

This way, the contents ErrorEnv are left as they were before, and if you setup in the first lines of the main() function:

```
        int main(void)
        {
             if (setjmp(ErrorEnv))      // Do not pass any other code.
                  return ERROR_FAILURE; // Just a general failure code
             ...
        }
```

This way the ErrorEnv can be always used without fearing a crash. Note that I used memcpy and not just the assignment:

```
        pushed_env = ErrorEnv; /* wrong! */
```

since jmp_buf is declared as an array as the standard states. Arrays can only be copied with memcpy or a loop assigning each member individually.

Note that this style of programming is sensitive to global variables. Globals will not be restored to their former values, and, if any of the procedures in the process() function modified global variables, their contents will be unchanged after the longjmp.

```
        #include <setjmp.h>
        jmp_buf ErrorEnv;
        double global;

        int guard(void)
        /* Return 0 if successful; else longjmp code */
        {
             jmp_buf pushed_env;
             memcpy(push_env,ErrorEnv,sizeof(jmp_buf));
             int status = setjmp(ErrorEnv);
             global = 78.9776;
             if (status == 0)
                  process();
             memcpy(ErrorEnv, pushed_env, sizeof(jmp_buf));
             // Here the contents of “global” will be either 78.9776
             // or 23.87 if the longjmp was taken.
             return status;
        }

        int process(void)
        {
             int error_code=0;
             ...
             global = 23.87;
             if (error_code) longjmp(ErrorEnv,error_code);
             ...
        }
```

And if you erase a file longjmp will not undelete it. Do not think that longjmp is a time machine that will go to the past.

Yet another problem to watch is the fact that if any of the global pointers pointed to an address that was later released, after the longjmp their contents will be wrong.

Any pointers that were allocated with malloc will not be released, and setjmp/longjmp could be the source of a memory leak. Within lcc-win32 there is an easy way out, since you can use the garbage collector instead of malloc/free. The garbage collector will detect any unused memory and will released when doing the gc.

1.13.1 Register variables and longjmp()

When you compile with optimizations *on*, the use of setjmp and longjmp can produce quite a few surprises. Consider this code:

```
#include <setjmp.h>
#include <stdio.h>
int main(void)
{
     jmp_buf jumper;
     int localVariable = 1;              (1)

     printf("1: %d\n",localVariable);
     if (setjmp(jumper) == 0) {          // return from longjmp
          localVariable++;               (2)
          printf("2: %d\n",localVariable);
          longjmp(jumper,1);
     }
     localVariable++;                    (3)
     printf("3: %d\n",localVariable);
     return 0;
}
```

Our “localVariable” starts with the value 1. Then, before calling longjmp, it is incremented. Its value should be two. At exit, “localVariable” is incremented again at should be three. We would expect the output:

1: 1

2: 2

3: 3

And this is indeed the output we get if we compile without any optimizations. When we turn optimizations on however, we get the output:

1: 1

2: 2

3: 2

Why?

Because “localVariable” will be stored in a register. When longjmp returns, it will restore all registers to the values they had when the setjmp was called, and if localVariable lives in a register it will return to the value 1, even if we incremented it before calling longjmp.

The only way to avoid this problem is to force the compiler to allocate localVariable in memory, using the “volatile” keyword. The declaration should look like this:

```
int volatile localVariable;
```

This instructs the compiler to avoid any optimizations with this variable, i.e. it forces allocating in the stack, and not in a register. This is required by the ANSI C standard. You can’t assume that local variables behave normally when using longjmp/setjmp.

The setjmp/longjmp functions have been used to implement larger exception handling frameworks. For an example of such a usage see for example “Exceptions and assertions” in “C Interfaces and implementations” of David Hanson, Chapter 4.

1.14 Simple programs

To give you a more concrete example of how C works, here are a few examples of very simple programs. The idea is to find a self-contained solution for a problem that is simple to state and understand.

1.14.1 strchr

Find the first occurrence of a given character in a character string. Return a pointer to the character if found, NULL otherwise.

This problem is solved in the standard library by the `strchr` function. Let's write it. The algorithm is very simple: We examine each character. If it is zero, this is the end of the string, we are done and we return NULL to indicate that the character is not there. If we find it, we stop searching and return the pointer to the character position.

```
char *FindCharInString(char *str, int ch)
{
     while (*str != 0 && *str != ch) {
          str++;
     }
     if (*str == ch)
          return str;
     return NULL;
}
```

We loop through the characters in the string. We use a while condition requiring that the character pointed to by our pointer "str" is different than zero and it is different than the character given. In that case we continue with the next character by incrementing our pointer, i.e. making it point to the next char. When the while loop ends, we have either found a character, or we have arrived at the end of the string. We discriminate between these two cases after the loop.

How can this program fail?

We do not test for NULL. Any NULL pointer passed to this program will provoke a trap. A way of making this more robust would be to return NULL if we receive a NULL pointer. This would indicate to the calling function that the character wasn't found, what is always true if our pointer doesn't point anywhere.

A more serious problem happens when our string is missing the zero byte… In that case the program will blindly loop through memory, until it either finds the byte is looking for, or a zero byte somewhere. This is a much more serious problem, since if the search ends by finding a random character somewhere, it will return an invalid pointer to the calling program!

This is really bad news, since the calling program may not use the result immediately. It could be that the result is stored in a variable, for instance, and then used in another, completely unrelated section of the program. The program would crash without any hints of what is wrong and where was the failure.

Note that this implementation of strchr will correctly accept a zero as the character to be searched. In this case it will return a pointer to the zero byte.

1.14.2 strlen

Return the length of a given string not including the terminating zero.

This is solved by the `strlen` function. We just count the chars in the string, stopping when we find a zero byte.

```
int strlen(char *str)
```

```
{
     char *p = str;

     while (*p != 0) {
          p++;
     }
     return p – str;
}
```

We copy our pointer into a new one that will loop through the string. We test for a zero byte in the while condition. Note the expression `*p != 0`. This means "Fetch the value this pointer is pointing to (`*p`), and compare it to zero". If the comparison is true, then we increment the pointer to the next byte.[45]

We return the number of characters between our pointer p and the saved pointer to the start of the string. This pointer arithmetic is quite handy.

How can this program fail? The same problems apply that we discussed in the previous example, but in an attenuated form: only a wrong answer is returned, not an outright wrong pointer. The program will only stop at a zero byte.

1.14.3 ispowerOfTwo

Given a positive number, find out if it is a power of two.

Algorithm: A power of two has only one bit set, in binary representation. We count the bits. If we find a bit count different than one we return 0, if there is only one bit set we return 1.

Implementation: We test the rightmost bit, and we use the shift operator to shift the bits right, shifting out the bit that we have tested. For instance, if we have the bit pattern 1 0 0 1, shifting it right by one gives 0 1 0 0: the rightmost bit has disappeared, and at the left we have a new bit shifted in, that is always zero.

```
int ispowerOfTwo(unsigned int n)
{
     unsigned int bitcount = 0;

     while (n != 0) {
          if (n & 1) {
               bitcount++;
          }
          n = n >> 1;
     }
     if (bitcount == 1)
          return 1;
     return 0;
}
```

Our condition here is that n must be different[46] than zero, i.e. there must be still some bits to count to go on. We test the rightmost bit with the binary `and` operation. The number one has only one bit set, the rightmost one. By the way, one is a power of two[47].

---

45. The expression `(*p != 0)` could have been written in the form `while (*p)`, using the implicit test for a non-zero result in any logical expression. Any expression will be considered true if its value is anything but zero. It is better, however, to make comparisons explicit.

46. Different than is written in C != instead of ≠ . The symbol ≠ wasn't included in the primitive typewriters in use when the C language was designed, and we have kept that approximation. It is consistent with the usage of ! as logical not, i.e. != would mean not equal.

Note that the return expression could have also been written like this:

```
return bitcount == 1;
```

The intention of the program is clearer with the “if” expression.

How can this program fail? The while loop has only one condition: that n is different than zero, i.e. that n has some bits set. Since we are shifting out the bits, and shifting in always zero bits since bitcount is unsigned, in a 32 bit machine like a PC this program will stop after at most 32 iterations. Running mentally some cases (a good exercise) we see that for an input of zero, we will never enter the loop, `bitcount` will be zero, and we will return 0, the correct answer. For an input of 1 we will make only one iteration of the loop. Since 1 & 1 is 1, `bitcount` will be incremented, and the test will make the routine return 1, the correct answer. If n is three, we make two passes, and `bitcount` will be two. This will be different than 1, and we return zero, the correct answer.

Anh Vu Tran anhvu.tran@ifrance.com made me discover an important bug. If you change the declaration of “n” from unsigned `int` to `int`, without qualification, the above function will enter an infinite loop if `n` is negative.

Why?

When shifting signed numbers sign is preserved, so the sign bit will be carried through, provoking that n will become eventually a string of 1 bits, never equal to zero, hence an infinite loop.

1.14.4 Write ispowerOfTwo without any loops

After working hard to debug the above program, it is disappointing to find out that it isn’t the best way of doing the required calculation. Here is an idea I got from reading the discussions in comp.lang.c.

```
isPow2 = x && !( (x-1) & x );
```

How does this work?

Algorithm:
If x is a power of two, it doesn't have any bits in common with x-1, since it consists of a single bit on. Any positive power of two is a single bit, using binary integer representation.

For instance 32 is a power of two. It is represented in binary as:

```
100000
```

32-1, 31, is represented as:

```
011111
```

32&31 is:

```
100000 & 011111 ==> 0
```

This means that we test if x-1 and x doesn't share bits with the and operator. If they share some bits, the AND of their bits will yield some non-zero bits. The only case where this will not happen is when x is a power of two.

Of course, if x is zero (not a power of two) this doesn't hold, so we add an explicit test for zero with the logical AND operator:

47. For a more detailed discussion, see the section News groups at the end of this tutorial.

```
xx && expression.
```

Negative powers of two (0.5, 0.25, 0.125, etc) could share this same property in a suitable fraction representation. 0.5 would be 0.1, 0.250 would be 0.01, 0.125 would be 0.001 etc. This snippet and several others are neatly explained in:

http://www.caam.rice.edu/~dougm/twiddle.

1.14.5 strlwr

Given a string containing upper case and lower case characters, transform it in a string with only lower case characters. Return a pointer to the start of the given string.[48]

This is the library function `strlwr`. In general is not a good idea to replace library functions, even if they are not part of the standard library (as defined in the C standard) like this one.

We make the transformation in-place, i.e. we transform all the characters of the given string. This supposes that the user of this program has copied the original string elsewhere if the original is needed again.

```
#include <ctype.h> /* needed for using isupper and tolower */
#include <stdio.h> /* needed for the NULL definition */
char *strTolower(char *str)
{
     /* iterates through str */
     unsigned char *p = (unsigned char *)str;

     if (str == NULL)
          return NULL;
     while (*p) {
          *str = tolower(*p);
          p++;
     }
     return str;
}
```

We include the standard header ctype.h, which contains the definition of several character classification functions (or macros) like "`isupper`" that determines if a given character is upper case, and many others like "`isspace`", or "`isdigit`". We need to include the stdio.h header file too, since it contains the definition for NULL.[49]

The first thing we do is to test if the given pointer is NULL. If it is, we return NULL. Then, we start our loop that will span the entire string. The construction `while(*p)` tests if the contents of the character pointer `p` is different than zero. If this is the case, we transform it into a lower case one. We increment our pointer to point to the next character, and we restart the

48. This convention is used in the library function. Actually, it is a quite bad interface, since the return value doesn't give any new information to the user, besides the expected side effect of transforming the given string. A better return value would be the number of changed characters, for instance, that would allow the caller to know if a transformation was done at all, or the length of the string, or several others. But let's implement this function as it is specified in the standard library. Many times, you will see that even if it is obvious that software must be changed, the consequences of a change are so vast that nobody wants to assume it, and we get stuck with software "for compatibility reasons". Here is yet another example.

49. If remembering which headers contain which definitions bothers you (as it bothers me) just use the <stdheaders.h> header file included with lcc-win32. That file is just an include of all standard header files.

loop. When the loop finishes because we hit the zero byte that terminates the string, we stop and return the saved position of the start of the string.

Note the cast that transforms str from a char * into an unsigned char *. The reason is that it could exist a bad implementation of the toupper() function, that would index some table using a signed char. Characters above 128 would be considered negative integers, what would result in a table being indexed by a negative offset, with bad consequences, as you may imagine.

How can this program fail?

Since we test for NULL, a NULL pointer can't provoke a trap. Is this a good idea?

Well this depends. This function will not trap with NULL pointers, but then the error will be detected later when other operations are done with that pointer anyway. Maybe making a trap when a NULL pointer is passed to us is not that bad, since it will uncover the error sooner rather than later. There is a big probability that if the user of our function is calling us to transform the string to lower case, is because he/she wants to use it later in a display, or otherwise. Avoiding a trap here only means that the trap will appear later, probably making error finding more difficult.

Writing software means making this type of decisions over and over again.

Obviously this program will fail with any incorrect string, i.e. a string that is missing the final zero byte. The failure behavior of our program is quite awful: in this case, this program will start destroying all bytes that happen to be in the range of uppercase characters until it hits a random zero byte. This means that if you pass a non-zero terminated string to this apparently harmless routine, you activate a randomly firing machine gun that will start destroying your program's data in a random fashion. The absence of a zero byte in a string is fatal for any C program. In a tutorial this can't be too strongly emphasized!

1.14.6 paste

You have got two text files, and you want to merge them in a single file, separated by tabulations. For instance if you have a file1 with this contents:

```
line 1
line2
```

and you got another file2 with this contents

```
line 10
line 11
```

you want to obtain a file

```
line1         line 10
line 2        line 11
```

Note that both file can be the same.

A solution for this could be the following program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
/*
We decide arbitrarily that lines longer than 32767 chars will make this program fail.
*/
#define MAXLINELEN 32767
int main(int argc,char *argv[])
{
/*
```

```
We need two FILE pointers, and two line buffers to hold each line from each file. We receive
in argc the number of arguments passed + 1, and in the character array argv[] the names of
the two files
*/
   FILE *f1,*f2;
   char buf1[MAXLINELEN],buf2[MAXLINELEN];

/*
We test immediately if the correct number of arguments has been given. If not, we exit with a
clear error message.
*/
   if (argc < 3) {
      fprintf(stderr,"Usage: paste file1 file2\n");
      exit(1);
   }
/*
We open both files, taking care not to open the same file twice. We test with strcmp if they are
equal.
*/
   f1 = fopen(argv[1],"r");
   if (strcmp(argv[1],argv[2]))
      f2 = fopen(argv[2],"r");
   else
      f2 = f1;
/*
We read line after line of the first file until we reach the end of the first file.
*/
   while(fgets(buf1,MAXLINELEN,f1)) {
      char *p = strchr(buf1,'\n');
/*
the fgets function leaves a \n in the input. We erase it if it is there. We use for this the strchr
function, that returns the first occurrence of a character in a string and returns a pointer to it.
If it doesn't it returns NULL, so we test below before using that pointer
*/
      if (p)
         *p = 0;
/*
We output the first file line, separated from the next with a single tabulation char.
*/
      printf("%s\t",buf1);
/*
If there are still lines to be read from file 2, we read them and we print them after doing the
same treatment as above.
*/
      if (f2 != f1 && fgets(buf2,MAXLINELEN,f2)) {
         p = strchr(buf2,'\n');
         if (p)
            *p = 0;
         printf("%s\n",buf2);
      }
/*
If we are duplicating the same file just print the same line again.
*/
      else printf("%s\n",buf1);
   }
/*
End of the while loop. When we arrive here the first file has been completely scanned. We
close and shut down.
*/
   fclose(f1);
```

```
    if (f1 != f2)
      fclose(f2);
    return 0;
}
```

How can this program fail?.

Well, there are obvious bugs in this program. Before reading the answer, try to see if you can see them. What is important here is that you learn how to spot bugs and that is a matter of logical thinking and a bit of effort.

Solution will be in the next page. But just try to find those bugs yourself.

Before that bug however we see this lines in there:

```
if (f2 != f1 && fgets(buf2,MAXLINELEN,f2)) {
}
else printf("%s\n",buf1);
```

If f1 is different from f2 (we have two different files) and file two is shorter than file one, that if statement will fail after n2 lines, and the else portion will be executed, provoking the duplication of the contents of the corresponding line of file one.

To test this, we create two test files, file1 and file2. their contents are:

File1:

```
File 1: line 1
File 1: line 2
File 1: line 3
File 1: line 4
File 1: line 5
File 1: line 6
File 1: line 7
File 1: line 8
```

File2:

```
File 2: line 1
File 2: line 2
File 2: line 3
File 2: line 4
```

We call our paste program with that data and we obtain:

```
D:\lcc\test>paste file1 file2
File 1: Line 1  File 2: Line 1
File 1: Line 2  File 2: Line 2
File 1: Line 3  File 2: Line 3
File 1: Line 4  File 2: Line 4
File 1: Line 5  File 1: Line 5
File 1: Line 6  File 1: Line 6
File 1: Line 7  File 1: Line 7
File 1: Line 8  File 1: Line 8

D:\lcc\test>
```

The line five of file one was read and since file two is already finished, we repeat it.

Is this a bug or a feature?

We received vague specifications. Nothing was said about what the program should do with files of different sizes. This can be declared a feature, but of course is better to be aware of it.

We see that to test hypothesis about the behavior of a program, there is nothing better than test data, i.e. data that is designed to exercise one part of the program logic.

In many real cases, the logic is surely not so easy to follow as in this example. Building test data can be done automatically. To build file one and two, this small program will do:

```
#include <stdio.h>
int main(void)
{
  FILE *f = fopen("file1","w");
  for (int i =0; i<8;i++)
    fprintf(f,"File 1: Line %d\n",i);
  fclose(f);
  f = fopen("file2","w");
  for (int i = 0; i < 5;i++)
    fprintf(f,"File 2: Line %d\n",i);
  fclose(f);
  return 0;
}
```

This a good example of throw away software, software you write to be executed once. No error checking, small and simple, so that there is less chance for mistakes.

And now the answer to the other bug above.

One of the first things to notice is that the program tests with strcmp to see if two files are the same. This means that when the user passes the command line:

```
paste File1 filE1
```

our program will believe that they are different when in fact they are not. Windows is not case sensitive for file names. The right thing to do there is to compare the file names with **stricmp**, that ignores the differences between uppercase and lowercase.

But an even greater problem is that we do not test for NULL when opening the files. If any of the files given in the command line doesn't exist, the program will crash. Add the necessary tests before you use it.

Another problem is that we test if we have the right number of arguments (i.e. at least two file names) but if we have *more* arguments we simply ignore them. What is the right behavior?

Obviously we could process (and paste) several files at once. Write the necessary changes in the code above. Note that if you want to do the program really general, you should take into account the fact that a file could be repeated several times in the input, i.e.

```
paste file1 file2 file1 file3
```

Besides, the separator char in our program is now hardwired to the tab character in the code of the program. Making this an option would allow to replace the tab with a vertical bar, for instance.

But the problem with such an option is that it supposes that the output will be padded with blanks for making the vertical bars align. Explain why that option needs a complete rewrite of our program. What is the hidden assumption above that makes such a change impossible?

Another feature that paste.exe could have, is that column headers are automatically underlined. Explain why adding such an option is falling into the featurism that pervades all modern software. Learn when to stop!

1.15 Using arrays and sorting

Suppose we want to display the frequencies of each letter in a given file. We want to know the number of 'a's, of 'b', and so on.

One way to do this is to make an array of 256 integers (one integer for each of the 256 possible character values) and increment the array using each character as an index into it. When we see a 'b', we get the value of the letter and use that value to increment the corresponding position in the array. We can use the same skeleton of the program that we have just built for counting characters, modifying it slightly.[50]

```
#include <stdio.h>
#include <stdlib.h>

int Frequencies[256]; // Array of frequencies

int main(int argc,char *argv[])
{
     // Local variables declarations
     int count=0;
     FILE *infile;
     int c;

     if (argc < 2) {
```

50. Yes, code reuse is not only possible in object-oriented programming.

```
            printf("Usage: countchars <file name>\n");
            exit(1);
        }
        infile = fopen(argv[1],"rb");
        if (infile == NULL) {
            printf("File %s doesn't exist\n",argv[1]);
            exit(1);
        }
        c = fgetc(infile);
        while (c != EOF) {
            count++;
            Frequencies[c]++;
            c = fgetc(infile);
        }
        fclose(infile);
        printf("%d chars in file\n",count);
        for (count=0; count<256;count++) {
            if (Frequencies[count] != 0) {
                printf("'%3c' (%4d) = %d\n", count, count,
                  Frequencies[count]);
            }
        }
        return 0;
}
```

We declare an array of 256 integers, numbered from zero to 255. Note that in C the index origin is always zero.

This array is not enclosed in any scope. Its scope then, is global, i.e. this identifier will be associated to the integer array for the current translation unit (the current file and its includes) from the point of its declaration on.

Since we haven't specified otherwise, this identifier will be exported from the current module and will be visible from other modules. In another compilation unit we can then declare:

```
extern int Frequencies[];
```

and we can access this array. This can be good (it allow us to share data between modules), or it can be bad (it allows other modules to tamper with private data), it depends on the point of view and the application. [51]

If we wanted to keep this array local to the current compilation unit we would have written:

```
static int Frequencies[256];
```

The "static" keyword indicates to the compiler that this identifier should not be made visible in another module.

The first thing our program does, is to open the file with the name passed as a parameter. This is done using the fopen library function. If the file exists, and we are able to read from it, the library function will return a pointer to a FILE structure, defined in stdio.h. If the file can't be opened, it returns NULL. We test for this condition right after the fopen call.

We can read characters from a file using the fgetc function. That function updates the current position, i.e. the position where the next character will be read.

---

51. Global variables provoke an undocumented coupling between several, apparently unrelated procedures or modules. Overuse of them is dangerous, and provokes errors that can be difficult to understand and get rid of. I learned this by experience in long debugging sessions, and now I use global variables more sparingly.

But let's come back to our task. We update the array at each character, within the while loop. We just use the value of the character (that must be an integer from zero to 256 anyway) to index the array, incrementing the corresponding position. Note that the expression:

```
Frequencies[count]++
```

means

```
Frequencies[count] = Frequencies[count]+1;
```

i.e.; the integer at that array position is incremented, and not the count variable!

Then at the end of the while loop we display the results. We only display frequencies when they are different than zero, i.e. at least one character was read at that position. We test this with the statement:

```
if (Frequencies[count] != 0) { ... statements ... }
```

The printf statement is quite complicated. It uses a new directive %c, meaning character, and then a width argument, i.e. %3c, meaning a width of three output chars. We knew the %d directive to print a number, but now it is augmented with a width directive too. Width directives are quite handy when building tables to get the items of the table aligned with each other in the output.

The first thing we do is to build a test file, to see if our program is working correctly. We build a test file containing

ABCDEFGHIJK

And we call:

```
lcc frequencies.c
lcclnk frequencies.obj
frequencies fexample
```

and we obtain:

```
D:\lcc\examples>frequencies fexample
13 chars in file

 (  10) = 1
 (  13) = 1
  A (  65) = 1
  B (  66) = 1
  C (  67) = 1
  D (  68) = 1
  E (  69) = 1
  F (  70) = 1
  G (  71) = 1
  H (  72) = 1
  I (  73) = 1
  J (  74) = 1
  K (  75) = 1
```

We see that the characters \r (13) and new line (10) disturb our output. We aren't interested in those frequencies anyway, so we could just eliminate them when we update our Frequencies table. We add the test:

```
if (c >= ' ')
          Frequencies[c]++;
```

i.e. we ignore all characters with value less than space: \r, \n or whatever. Note that we ignore tabulations too, since their value is 8.

The output is now more readable:

```
H:\lcc\examples>frequencies fexample
```

```
13 chars in file
  A (  65) = 1
  B (  66) = 1
  C (  67) = 1
  D (  68) = 1
  E (  69) = 1
  F (  70) = 1
  G (  71) = 1
  H (  72) = 1
  I (  73) = 1
  J (  74) = 1
  K (  75) = 1
```

We test now our program with itself. We call:

```
frequencies frequencies.c
758 chars in file
```

I have organized the data in a table to easy the display.

(32) = 57	! (33) = 2	" (34) = 10
# (35) = 2	% (37) = 5	' (39) = 3
((40) = 18	) (41) = 18	* (42) = 2
+ (43) = 6	, (44) = 7	. (46) = 2
/ (47) = 2	0 (48) = 4	1 (49) = 4
2 (50) = 3	3 (51) = 1	4 (52) = 1
5 (53) = 2	6 (54) = 2	: (58) = 1
; (59) = 19	< (60) = 5	= (61) = 11
> (62) = 4	A (65) = 1	E (69) = 2
F (70) = 7	I (73) = 1	L (76) = 3
N (78) = 1	O (79) = 1	U (85) = 2
[(91) = 7	\ (92) = 4	] (93) = 7
a (97) = 12	b (98) = 2	c (99) = 33
d (100) = 8	e (101) = 38	f (102) = 23
g (103) = 8	h (104) = 6	i (105) = 43
l (108) = 14	m (109) = 2	n (110) = 43
o (111) = 17	p (112) = 5	q (113) = 5
r (114) = 23	s (115) = 14	t (116) = 29
u (117) = 19	v (118) = 3	w (119) = 1
x (120) = 3	y (121) = 1	{ (123) = 6
} (125) = 6		

What is missing obviously, is to print the table in a sorted way, so that the most frequent characters would be printed first. This would make inspecting the table for the most frequent character easier. How can we do that in C?

We have in the standard library the function “qsort”, that sorts an array. We study its prototype first, to see how we should use it:[52]

```
void qsort(void *b,size_t n,size_t s,int(*f)(const void *));
```

Well, this is quite an impressing prototype really. But if we want to learn C, we will have to read this, as it was normal prose. So let’s begin, from left to right.

The function qsort doesn’t return an explicit result. It is a void function. Its argument list, is the following:

52. The prototype is in the header file stdlib.h

Argument 1: is a `void *`. Void *??? What is that? Well, in C you have void, that means none, and void *, that means this is a pointer that can point to anything, i.e. a pointer to an untyped value. We still haven't really introduced pointers, but for the time being just be happy with this explanation: qsort needs the start of the array that will sort. This array can be composed of anything, integers, user defined structures, double precision numbers, whatever. This "whatever" is precisely the "void *".

Argument 2 is a `size_t`. This isn't a known type, so it must be a type defined before in stdlib.h. By looking at the headers, and following the embedded include directives, we find:

"stdlib.h" includes "stddef.h", that defines a "typedef" like this:[53]

typedef unsigned int size_t;

This means that we define here a new type called "size_t", that will be actually an unsigned integer. Typedefs allow us to augment the basic type system with our own types. Mmmm interesting. We will keep this for later use.

In this example, it means that the size_t n, is the number of elements that will be in the array.

Argument 3 is also a `size_t.` This argument contains the size of each element of the array, i.e. the number of bytes that each element has. This tells qsort the number of bytes to skip at each increment or decrement of a position. If we pass to qsort an array of 56 double precision numbers, this argument will be 8, i.e. the size of a double precision number, and the preceding argument will be 56, i.e. the number of elements in the array.

Argument 4 is a function: `int (*f)(const void *));` Well this is quite hard really. We are in the first pages of this introduction and we already have to cope with gibberish like this?

We have to use recursion now. We have again to start reading this from left to right, more or less. We have a function pointer (f) that points to a function that returns an int, and that takes as arguments a void *, i.e. a pointer to some unspecified object, that can't be changed within that function (const).

This is maybe quite difficult to write, but quite a powerful feature. Functions can be passed as arguments to other functions in C. They are first class objects that can be used to specify a function to call.

Why does qsort need this?

Well, since the qsort function is completely general, it needs a helper function, that will tell it when an element of the array is smaller than the other. Since qsort doesn't have any *a priori* knowledge of the types of the elements of the passed array, it needs a helper function that returns an integer smaller than zero if the first element is smaller than the next one, zero if the elements are equal, or bigger than zero if the elements are bigger.

Let's apply this to a smaller example, so that the usage of qsort is clear before we apply it to our frequencies problem.

```
#include <stdlib.h>
#include <string.h>      (1)
#include <stdio.h>
```

53. Finding out where is something defined can be quite a difficult task. The easiest way is to use the IDE of lcc-win32, right click in the identifier, and choose "goto definition". If that doesn't work, you can use "grep" to search in a set of files.

```
int compare(const void *arg1,const void *arg2)(2)
{
  /* Compare all of both strings: */  (3)
    return stricmp( *( char** ) arg1, * ( char** ) arg2 );
}

int main( int argc, char **argv )
{
  /* Eliminate argv[0] from sort: */     (4)
    argv++;
    argc--;

  /* Sort remaining args using qsort */(5)
    qsort((void*)argv,(size_t)argc,sizeof(char *),compare);

  /* Output sorted list: */
    for(int i = 0; i < argc; ++i )(6)
       printf( "%s ", argv[i] );
    printf( "\n" );       (7)
    return 0;
}
```

The structure of this example is as follows:

We build a program that will sort its arguments and output the sorted result.

To use qsort we define a comparison function that returns an integer, which encodes the relative lexical ordering of the two arguments passed to it. We use a library function for doing that, the stricmp[54] function, that compares two character strings without caring about case differences.

But there is quite a lot of new material in this example, and it is worth going through it in detail.

We include the standard header string.h, to get the definitions of string handling functions like stricmp.

We define our comparison function with:

```
int compare(const void *arg1,const void *arg2) { ... }
```

This means that our compare function will return an int, and that takes two arguments, named arg1 and arg2, that are pointers to any object (void *). The objects pointed to by arg1, and arg2 will not be changed within this function, i.e. they are “const”.

We need to get rid of the void * within our compare function. We know we are going to pass to this function actually pointers to characters, i.e. machine addresses to the start of character strings, so we have to transform the arguments into a type we can work with. For doing this we use a **cast**. A cast is a transformation of one type to another type at compile time. Its syntax is like this: `(newtype)(expression);`. In this example we cast a void * to a char **, a pointer to a pointer of characters. The whole expression needs quite a lot of reflection to be analyzed fully. Return here after reading the section about pointers.

Note that our array argv, can be used as a pointer and incremented to skip over the first element. This is one of the great weaknesses of the array concept of the C language. Actually, arrays and pointers to the first member are equivalent. This means that in many situations, arrays “decay” into pointers to the first element, and loose their “array”ness. That is why you can do in C things with arrays that would never be allowed in another languages. At the end of

54. Most compilers do not have the C99 standard implemented. In those compilers you can’t do this and you will have to declare the loop counter as a normal local variable. Another reason to stick to lcc-win32.

this tutorial we will se how we can overcome this problem, and have arrays that are always normal arrays that can be passed to functions without losing their soul.

At last we are ready to call our famous qsort function. We use the following call expression:

```
qsort((void*)argv,(size_t)argc,sizeof(char *),compare);
```

The first argument of qsort is a void *. Since our array argv is a char **, we transform it into the required type by using a cast expression: (void *)argv.

The second argument is the number of elements in our array. Since we need a *size_t* and we have argc, that is an integer variable, we use again a cast expression to transform our int into a *size_t*. Note that typedefs are accepted as casts.

The third argument should be the size of each element of our array. We use the built-in pseudo function sizeof, which returns the size in bytes of its argument. This is a pseudo function, because there is no such a function actually. The compiler will replace this expression with an integer that it calculates from its internal tables. We have here an array of char *, so we just tell the compiler to write that number in there.

The fourth argument is our comparison function. We just write it like that. No casts are needed, since we were careful to define our comparison function exactly as qsort expects.

To output the already sorted array we use again a “for” loop. Note that the index of the loop is declared at the initialization of the “for” construct. This is one of the new specifications of the C99 language standard, that lcc-win32 follows. You can declare variables at any statement, and within “for” constructs too. Note that the scope of this integer will be only the scope of the enclosing “for” block. It can’t be used outside this scope.[55]

Note that we have written the “for” construct without curly braces. This is allowed, and means that the “for” construct applies only to the next statement, nothing more. The `...printf("\n");...` is NOT part of the for construct.

Ok, now let’s compile this example and make a few tests to see if we got that right.

```
h:\lcc\examples> sortargs aaa bbb hhh sss ccc nnn
aaa bbb ccc hhh nnn sss
```

OK, it seems to work. Now we have acquired some experience with qsort, we can apply our knowledge to our frequencies example. We use cut and paste in the editor to define a new compare function that will accept integers instead of char **. We build our new comparison function like this:

```
int compare( const void *arg1, const void *arg2 )
{
     return ( * ( int * ) arg1 -  * ( int * ) arg2 );
}
```

We just return the difference between both numbers. If arg1 is bigger than arg2, this will be a positive number, if they are equal it will be zero, and if arg1 is smaller than arg2 it will be a negative number, just as qsort expects.

Right before we display the results then, we add the famous call we have been working so hard to get to:

```
qsort(Frequencies,256,sizeof(int),compare);
```

We pass the Frequencies array, its size, the size of each element, and our comparison function.

Here is the new version of our program, for your convenience. New code is in bold:

55. The compiler emits a record for the linker, telling it to put there the address of the global, if the argument is a global variable, or will emit the right instructions to access the address of a local using the frame pointer. This has been working for a while now.

```
#include <stdio.h>
#include <stdlib.h>

int Frequencies[256]; // Array of frequencies

int compare( const void *arg1, const void *arg2 )
{
  /* Compare both integers */
    return ( * ( int * ) arg1 -  * ( int * ) arg2 );
}

int main(int argc,char *argv[])
{
        int count=0;
        FILE *infile;
        int c;

        if (argc < 2) {
                ...
        }
        infile = fopen(argv[1],"rb");
        if (infile == NULL) {
                ...
        }
        c = fgetc(infile);
        while (c != EOF) {
                ...
        }
        fclose(infile);
        printf("%d chars in file\n",count);
        qsort(Frequencies,256,sizeof(int),compare);
        for (count=0; count<256;count++) {
                if (Frequencies[count] != 0) {
                        printf("%3c (%4d) = %d\n",
                                count,
                                count,
                                Frequencies[count]);
                }
        }
        return 0;
}
```

We compile, link, and then we write

```
frequencies frequencies.c
957 chars in file
```

Well, sorting definitely works (you read this display line by line), but we note with dismay that

À (192) = 1	Á (193) = 1	Â (194) = 1
Ã (195) = 1	Ä (196) = 1	Å (197) = 1
Æ (198) = 1	Ç (199) = 1	È (200) = 1
É (201) = 1	Ê (202) = 1	Ë (203) = 2
Ì (204) = 2	Í (205) = 2	Î (206) = 2
Ï (207) = 2	Ð (208) = 2	Ñ (209) = 3
Ò (210) = 3	Ó (211) = 3	Ô (212) = 3
Õ (213) = 3	Ö (214) = 3	× (215) = 4
Ø (216) = 4	Ù (217) = 4	Ú (218) = 4
Û (219) = 5	Ü (220) = 5	Ý (221) = 5
Þ (222) = 5	ß (223) = 6	à (224) = 6
á (225) = 6	â (226) = 7	ã (227) = 7
ä (228) = 7	å (229) = 7	æ (230) = 7
ç (231) = 7	è (232) = 8	é (233) = 8
ê (234) = 10	ë (235) = 10	ì (236) = 10
í (237) = 11	î (238) = 11	ï (239) = 13
ð (240) = 16	ñ (241) = 20	ò (242) = 21
ó (243) = 21	ô (244) = 21	õ (245) = 24
ö (246) = 24	÷ (247) = 25	ø (248) = 28
ù (249) = 35	ú (250) = 38	û (251) = 39
ü (252) = 46	ý (253) = 52	þ (254) = 52
ÿ (255) = 93		

All the character names are wrong!

Why?

Well we have never explicitly stored the name of a character in our integer array; it was implicitly stored. The sequence of elements in the array corresponded to a character value. But once we sort the array, this ordering is gone, and we have lost the correspondence between each array element and the character it was representing.

C offers us many solutions to this problem, but this is taking us too far away from array handling, the subject of this section. We will have to wait until we introduce structures and user types before we can solve this problem.

1.15.1 Summary of Arrays and sorting

- Arrays are declared by indicating their size in square brackets, after the identifier declaration: `<type> identifier[SIZE];`
- Arrays are equivalent to pointers to their first element.
- Arrays “decay”, i.e. are transformed into pointers, when passed to other functions.
- You can sort an array using the qsort function.

1.16 Pointers and references

Pointers are one of the “hard” subjects of the C language. They are somehow mysterious, quite difficult for beginners to grasp, and their extensive use within C makes them unavoidable.

Pointers are machine addresses, i.e. they point to data. It is important to have clear this distinction: pointers are NOT the data they point to, they contain just a machine address where the data will be found. When you declare a pointer like this:

```
FILE *infile;
```

you are declaring: reserve storage for a machine address and not a FILE structure. This machine address will contain the location where that structure starts.

This allows us to pass a small machine address around, instead of a big FILE structure that has dozens of fields. The big advantage of pointers is that they are efficient. But, as anything, they are error prone. There is no free lunch.

The contents of the pointer are undefined until you initialize it. Before you initialize a pointer, its contents can be anything; it is not possible to know what is in there, until you make an assignment. A pointer before is initialized is a dangling pointer, i.e. a pointer that points to nowhere.

A pointer can be initialized by:

1) Assign it a special pointer value called NULL, i.e. empty.

2) Assignment from a function or expression that returns a pointer of the same type. In the frequencies example we initialize our infile pointer with the function fopen, that returns a pointer to a FILE.

3) Assignment to a specific address. This happens in programs that need to access certain machine addresses for instance to use them as input/output for special devices. In those cases you can initialize a pointer to a specific address. Note that this is not possible under windows, or Linux, or many operating systems where addresses are virtual addresses. More of this later.

4) You can assign a pointer to point to some object by taking the address of that object. For instance:

   ```
   int integer;
   int *pinteger = &integer;
   ```

   Here we make the pointer “pinteger” point to the int “integer” by taking the address of that integer, using the “&” operator. This operator yields the machine address of its argument.

5) You can access the data the pointer is pointing to by using the “*” operator. When we want to access the integer “pinteger” is pointing to, we write:

   ```
   *pinteger = 7;
   ```

   This assigns to the “integer” variable indirectly the value 7.

In lcc-win32 pointers can be of two types. We have normal pointers, as we have described above, and “references”, i.e. compiler maintained pointers, that are very similar to the objects themselves. [56]

References are declared in a similar way as pointers are declared:[57]

```
int a = 5;    // declares an integer a
int * pa = &a;// declares a pointer to the integer a
int &ra = a;// declares a reference to the integer a
```

Here we have an integer, that within this scope will be called “a”. Its machine address will be stored in a pointer to this integer, called “pa”. This pointer will be able to access the data of

56. This has nothing to do with object oriented programming of course. The word object is used here with its generic meaning.

57. References aren’t part of the C language standard, and are in this sense an extension of lcc-win32. They are wildly used in another related language (C++), and the implementation of lcc-win32 is compatible with the implementation of references of that language.

"a", i.e. the value stored at that machine address by using the "*" operator. When we want to access that data we write:

```
*pa = 8944;
```

This means: "store at the address contained in this pointer pa, the value 8944".

We can also write:

```
int m = 698 + *pa;
```

This means: "add to 698 the contents of the integer whose machine address is contained in the pointer pa and store the result of the addition in the integer m"

We have a "reference" to a, that in this scope will be called "ra". Any access to this compiler maintained pointer is done as we would access the object itself, no special syntax is needed. For instance we can write:

```
ra = (ra+78) / 79;
```

Note that with references the "*" operator is not needed. The compiler will do automatically this for you.

It is obvious that a question arises now: why do we need references? Why can't we just use the objects themselves? Why is all this pointer stuff necessary?

Well this is a very good question. Many languages seem to do quite well without ever using pointers the way C does.

The main reason for these constructs is *efficiency.* Imagine you have a huge database table, and you want to pass it to a routine that will extract some information from it. The best way to pass that data is just to pass the address where it starts, without having to move or make a copy of the data itself. Passing an address is just passing a 32-bit number, a very small amount of data. If we would pass the table itself, we would be forced to copy a huge amount of data into the called function, what would waste machine resources.

The best of all worlds are references. They must always point to some object, there is no such a thing as an uninitialized reference. Once initialized, they can't point to anything else but to the object they were initialized to, i.e. they can't be made to point to another object, as normal pointers can. For instance, in the above expressions, the pointer pa is initialized to point to the integer "a", but later in the program, you are allowed to make the "pa" pointer point to another, completely unrelated integer. This is not possible with the reference "ra". It will always point to the integer "a".

When passing an argument to a function, if that function expects a reference and you pass it a reference, the compiler will arrange for you passing only the address of the data pointed to by the reference.

1.17 Structures and unions

1.17.1 Structures

Structures are a contiguous piece of storage that contains several simple types, grouped as a single object.[58] For instance, if we want to handle the two integer positions defined for each pixel in the screen we could define the following structure:

```
struct coordinates {
     int x;
     int y;
};
```

Structures are introduced with the keyword “struct” followed by their name. Then we open a scope with the curly braces, and enumerate the fields that form the structure. Fields are declared as all other declarations are done. Note that a structure declaration is just that, a declaration, and it reserves no actual storage anywhere.

After declaring a structure, we can use this new type to declare variables or other objects of this type:

```
struct coordinate Coords = { 23,78};
```

Here we have declared a variable called Coords, that is a structure of type coordinate, i.e. having two fields of integer type called “x” and “y”. In the same statement we initialize the structure to a concrete point, the point (23,78). The compiler, when processing this declaration, will assign to the first field the first number, i.e. to the field “x” will be assigned the value 23, and to the field “y” will be assigned the number 78.

Note that the data that will initialize the structure is enclosed in curly braces.

Structures can be recursive, i.e. they can contain pointers to themselves. This comes handy to define structures like lists for instance:

```
struct list {
     struct list *Next;
     int Data;
};
```

Here we have defined a structure that in its first field contains a pointer to the same structure, and in its second field contains an integer. Please note that we are defining a pointer to an identical structure, not the structure itself, what is impossible. A structure can’t contain itself.

Double linked list can be defined as follows:

```
struct dl_list {
     struct dl_list *Next;
     struct dl_list *Previous;
     int Data;
};
```

This list features two pointers: one forward, to the following element in the list, and one backward, to the previous element of the list.

A special declaration that can only be used in structures is the bit-field declaration. You can specify in a structure a field with a certain number of bits. That number is given as follows:

```
struct flags {
     unsigned HasBeenProcessed:1;
```

58. The usage of the #pragma pack construct is explained in lcc-win32 user’s manual. Those explanations will not be repeated here.

```
        unsigned HasBeenPrinted:1;
        unsigned Pages:5;
};
```

This structure has three fields. The first, is a bit-field of length 1, i.e. a Boolean value, the second is also a bit-field of type Boolean, and the third is an integer of 5 bits. In that integer you can only store integers from zero to 31, i.e. from zero to 2 to the 5th power, minus one. In this case, the programmer decides that the number of pages will never exceed 31, so it can be safely stored in this small amount of memory.

We access the data stored in a structure with the following notation:

```
<structure-name> '.' field-name
```

or

```
<structure-name '->' field-name
```

We use the second notation when we have a pointer to a structure, not the structure itself. When we have the structure itself, or a reference variable, we use the point.

Here are some examples of this notation:

```
void fn(void)
{
     coordinate c;
     coordinate *pc;
     coordinate &rc = c;

     c.x = 67; // Assigns the field x
     c.y = 78;// Assigns the field y
     pc = &c; // We make pc point to c
     pc->x = 67; // We change the field x to 67
     pc->y = 33;// We change the field y to 33
     rc.x = 88;// References use the point notation
}
```

Structures can contain other structures or types. After we have defined the structure coordinate above, we can use that structure within the definition of a new one.

```
struct DataPoint {
     struct coordinate coords;
     int Data;
};
```

This structure contains a "coordinate" structure. To access the "x" field of our coordinate in a DataPoint structure we would write:

```
struct DataPoint dp;
dp.coords.x = 78;
```

Structures can be contained in arrays. Here, we declare an array of 25 coordinates:

```
struct coordinate coordArray[25];
```

To access the x coordinate from the 4th member of the array we would write:

```
coordArray[3].x = 89;
```

Note (again) that in C array indexes start at zero. The fourth element is numbered 3.

Many other structures are possible their number is infinite:

```
struct customer {
     int ID;
     char *Name;
     char *Address;
     double balance;
```

```
        time_t lastTransaction;
        unsigned hasACar:1;
        unsigned mailedAlready:1;
};
```

This is a consecutive amount of storage where:

- an integer contains the ID of the customer,
- a machine address pointing to the start of the character string with the customer name,
- another address pointing to the start of the name of the place where this customer lives,
- a double precision number containing the current balance,
- a time_t (time type) date of last transaction,
- and other bit fields for storing some flags.

```
struct mailMessage {
        MessageID ID;
        time_t date;
        char *Sender;
        char *Subject;
        char *Text;
        char *Attachements;
};
```

This one starts with another type containing the message ID, again a time_t to store the date, then the addresses of some character strings.

The set of functions that use a certain type are the methods that you use for that type, maybe in combination with other types. There is no implicit "this" in C. Each argument to a function is explicit, and there is no predominance of anyone.

A `customer` can send a `mailMessage` to the company, and certain functions are possible, that handle `mailMessages` from customers. Other `mailMessages` aren't from customers, and are handled differently, depending on the concrete application.

Because that's the point here: an application is a coherent set of types that performs a certain task with the computer, for instance, sending automated mailings, or invoices, or sensing the temperature of the system and acting accordingly in a multi-processing robot, or whatever. It is up to you actually.

Note that in C there is no provision or compiler support for associating methods in the structure definitions. You can, of course, make structures like this:

```
struct customer {
        int ID;
        char *Name;
        char *Address;
        double balance;
        time_t lastTransaction;
        unsigned hasACar:1;
        unsigned mailedAlready:1;
        bool (*UpdateBalance)(struct customer *Customer,
                              double newBalance);
};
```

The new field, is a function pointer that contains the address of a function that returns a Boolean result, and takes a customer and a new balance, and should (eventually) update the balance field, that isn't directly accessed by the software, other than trough this procedure pointer.

When the program starts, you assign to each structure in the creation procedure for it, the function DefaultGetBalance() that takes the right arguments and does hopefully the right thing.

This allows you the flexibility of assigning different functions to a customer for calculating his/her balance according to data that is known only at runtime. Customers with a long history of overdraws could be handled differently by the software after all. But this is no longer C, is the heart of the application.

True, there are other languages that let you specify with greater richness of rules what and how can be sub classed and inherited. C, allows you to do anything, there are no other rules here, other the ones you wish to enforce.

You can subclass a structure like this. You can store the current pointer to the procedure somewhere, and put your own procedure instead. When your procedure is called, it can either:

Do some processing before calling the original procedure

Do some processing after the original procedure returns

Do not call the original procedure at all and replace it entirely.

We will show a concrete example of this when we speak about windows sub classing later. Sub classing allows you to implement dynamic inheritance. This is just an example of the many ways you can program in C.

But is that flexibility really needed?

Won't just

```
bool UpdateBalance(struct customer *pCustomer, double newBalance);
```

do it too?

Well it depends. Actions of the general procedure could be easy if the algorithm is simple and not too many special cases are in there. But if not, the former method, even if more complicated at first sight, is essentially simpler because it allows you greater flexibility in small manageable chunks, instead of a monolithic procedure of several hundred lines full of special case code…

Mixed strategies are possible. You leave for most customers the UpdateBalance field empty (filled with a NULL pointer), and the global UpdateBalance procedure will use that field to calculate its results only if there is a procedure there to call. True, this wastes 4 bytes per customer in most cases, since the field is mostly empty, but this is a small price to pay, the structure is probably much bigger anyway.

1.17.2 Structure size

In principle, the size of a structure is the sum of the size of its members. This is, however, just a very general rule, since it depends a lot on the compilation options valid at the moment of the structure definition, or in the concrete settings of the structure packing as specified with the #pragma pack() construct.[59]

Normally, you should never make any assumptions about the specific size of a structure. Compilers, and lcc-win32 is no exception, try to optimize structure access by aligning members of the structure at predefined addresses. For instance, if you use the memory manager, pointers must be aligned at addresses multiples of four, if not, the memory manager doesn't detect them and that can have disastrous consequences.

---

59. Note that putting structure names in typedefs all uppercase is an old habit that somehow belongs to the way I learned C, but is in no way required by the language. Personally I find those all-uppercase names clearer as a way of indicating to the reader that a user defined type and not a variable is used, since I have never used an all-uppercase name for a variable name. Separating these names by upper/lower case improves the readability of the program, but this is a matter of personal taste.

The best thing to do is to always use the sizeof operator when the structure size needs to be used somewhere in the code. For instance, if you want to allocate a new piece of memory managed by the memory manager, you call it with the size of the structure.

```
GC_malloc(sizeof(struct DataPoint)*67);
```

This will allocate space for 67 structures of type "DataPoint" (as defined above). Note that we could have written

```
GC_malloc(804);
```

since we have:

```
struct DataPoint {
     struct coordinate coords;
     int Data;
};
```

We can add the sizes:

Two integers of 4 bytes for the coordinate member, makes 8 bytes, plus 4 bytes for the Data member, makes 12, that multiplies 67 to make 804 bytes.

But this is very risky because of two reasons:

Compiler alignment could change the size of the structure

If you add a new member to the structure, the sizeof() specification will continue to work, since the compiler will correctly recalculate it each time. If you write the 804 however, when you add a new member to the structure this number has to be recalculated again, making one more thing that can go wrong in your program.

In general, it is always better to use compiler-calculated constants like sizeof() instead of hard-wired numbers.

1.17.3 Defining new types

Structures are then, a way of augmenting the type system by defining new types using already defined ones. The C language allows you to go one step further in this direction by allowing you to specify a new type definition or typedef for short.

This syntax for doing this is like this:

```
typedef <already defined type> <new name>;
```

For instance, you can specify a new type of integer called "my integer" with:

```
typedef int my_integer;
```

Then, you can use this new type in any position where the "int" keyword would be expected. For instance you can declare:

```
my_integer i;
```

instead of:

```
int i;
```

This can be used with structures too. For instance, if you want to avoid typing at each time you use a coordinate `struct coordinate a;` you can define

typedef struct coordinate COORDINATE;

and now you can just write:

```
COORDINATE a;
```

what is shorter, and much clearer.[60]

This new name can be used with the `sizeof()` operator too, and we can write:

```
GC_malloc(sizeof(COORDINATE));
```

instead of the old notation. But please keep in mind the following: once you have defined a typedef, never use the "struct" keyword in front of the typedef keyword, if not, the compiler will get really confused.

1.17.4 Unions

Unions are similar to structures in that they contain fields. Contrary to structures, unions will store all their fields in the same place. They have the size of the biggest field in them. Here is an example:

```
union intfloat {
     int i;
     double d;
};
```

This union has two fields: an integer and a double precision number. The size of an integer is four in lcc-win32, and the size of a double is eight. The size of this union will be eight bytes, with the integer and the double precision number starting at the same memory location. The union can contain either an integer or a double precision number but not the two. If you store an integer in this union you should access only the integer part, if you store a double, you should access the double part. Field access syntax is the same as for structures: we use always the point.

Using the definition above we can write:

```
int main(void)
{
     union intfloat ifl;
     union intfloat *pIntfl = &ifl;

     pIntfl.i = 2;
     pintfl.d = 2.87;
}
```

First we assign to the integer part of the union an integer, then we assign to the double precision part a double.

Unions are useful for storing structures that can have several different memory layouts. In general we have an integer that tells us which kind of data follows, then a union of several types of data. Suppose the following data structures:

```
struct fileSource {
     char *FileName;
     int LastUse;
};

struct networkSource {
     int socket;
     char *ServerName;
     int LastUse;
};

struct windowSource {
     WINDOW window;
```

60. An identifier can also represent a macro or a macro argument, but here we will assume that the preprocessor already has done its work.

```
        int LastUse;
};
```

All of this data structures should represent a source of information. We add the following defines:

```
#define ISFILE 1
#define ISNETWORK 2
#define ISWINDOW 3
```

and now we can define a single information source structure:

```
struct Source {
     int type;
     union {
          struct fileSource file;
          struct networkSource network;
          struct windowSource window;
     } info;
};
```

We have an integer at the start of our generic "Source" structure that tells us, which of the following possible types is the correct one. Then, we have a union that describes all of our possible data sources.

We fill the union by first assigning to it the type of the information that follows, an integer that must be one of the defined constants above. Then we copy to the union the corresponding structure. Note that we save a lot of wasted space, since all three structures will be stored beginning at the same location. Since a data source must be one of the structure types we have defined, we save wasting memory in fields that would never get used.

Another usage of unions is to give a different interpretation of the same data. For instance, an MMX register in an x86 compatible processor can be viewed as two integers of 32 bits, 4 integers of 16 bits, or 8 integers of 8 bits. Lcc-win32 describes this fact with a union:

```
typedef struct _pW {
        char high;
        char low;
} _packedWord; // 16 bit integer

typedef struct _pDW {
        _packedWord high;
        _packedWord low;
} _packedDWord; // 32 bit integer of two 16 bit integers

typedef struct _pQW {
        _packedDWord high;
        _packedDWord low;
} _packedQWord; // 64 bits of two 32 bit structures

typedef union __Union {
        _packedQWord packed;
        int dwords[2];
        short words[4];
        char bytes[8];
} _mmxdata; // This is the union of all those types
```

Union usage is not checked by the compiler, i.e. if you make a mistake and access the wrong member of the union, this will provoke a trap or another failure at run time. One way of debugging this kind of problem is to define all unions as structures during development, and see where you access an invalid member. When the program is fully debugged, you can switch back to the union usage.

1.18 Using structures

Now that we know how we can define structures we can (at last) solve the problem we had with our character frequencies program.

We define a structure containing the name of the character like this:

```
typedef struct tagChars {
     int CharacterValue;
     int Frequency;
} CHARS;
```

Note that here we define two things in a single statement: we define a structure called “tagChars” with two fields, and we define a typedef CHARS that will be the name of this type.

Within the program, we have to change the following things:

We have to initialize the name field of the array that now will be an array of structures and not an array of integers.

When each character is read we have to update the frequency field of the corresponding structure.

When displaying the result, we use the name field instead of our count variable.

Here is the updated program:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct tagChars {
     int CharacterValue;
     int Frequency;
} CHARS;

CHARS Frequencies[256]; // Array of frequencies

int compare( const void *arg1, const void *arg2 )
{
     CHARS *Arg1 = (CHARS *)arg1;
     CHARS *Arg2 = (CHARS *)arg2;
     /* Compare both integers */
     return ( Arg2->Frequency -  Arg1->Frequency );
}

int main(int argc,char *argv[])
{
     int count=0;
     FILE *infile;
     int c;

     if (argc < 2) {
          printf("Usage: countchars <file name>\n");
          exit(1);
     }
     infile = fopen(argv[1],"rb");
     if (infile == NULL) {
          printf("File %s doesn't exist\n",argv[1]);
          exit(1);
     }
     for (int i = 0; i<256; i++) {
          Frequencies[i].CharacterValue = i;
     }
```

```
        c = fgetc(infile);
        while (c != EOF) {
             count++;
             if (c >= ' ')
                  Frequencies[c].Frequency++;
             c = fgetc(infile);
        }
        fclose(infile);
        printf("%d chars in file\n",count);
        qsort(Frequencies,256,sizeof(CHARS),compare);
        for (count=0; count<256;count++) {
             if (Frequencies[count].Frequency != 0) {
                  printf("%3c (%4d) = %d\n",
                    Frequencies[count].CharacterValue,
                    Frequencies[count].CharacterValue,
                    Frequencies[count].Frequency);
             }
        }
        return 0;
   }
```

We transformed our integer array Frequencies into a CHARS array with very few changes: just the declaration. Note that the array is still accessed as a normal array would. By the way, it **is** a normal array.

We changed our "compare" function too, obviously, since we are now comparing two CHARS structures, and not just two integers. We have to cast our arguments into pointers to CHARS, and I decided that using two temporary variables would be clearer than a complicated expression that would eliminate those.

The initialization of the CharacterValue field is trivially done in a loop, just before we start counting chars. We assign to each character an integer from 0 to 256 that's all.

When we print our results, we use that field to get to the name of the character, since our array that before qsort was neatly ordered by characters, is now ordered by frequency. As before, we write the character as a letter with the %c directive, and as a number, with the %d directive.

When we call this program with:

frequencies frequencies.c

we obtain at last:

1311 chars in file

(32) = 154	e (101) = 77	n (110) = 60
i (105) = 59	r (114) = 59	c (99) = 52
t (116) = 46	u (117) = 35	a (97) = 34
; (59) = 29	o (111) = 29	f (102) = 27
s (115) = 26	((40) = 25	) (41) = 25
l (108) = 20	g (103) = 18	F (70) = 17
q (113) = 16	= (61) = 15	C (67) = 13
h (104) = 12	A (65) = 12	d (100) = 11
, (44) = 11	[(91) = 10	] (93) = 10
* (42) = 10	" (34) = 10	{ (123) = 9
2 (50) = 9	p (112) = 9	} (125) = 9
1 (49) = 8	. (46) = 8	y (121) = 8
+ (43) = 8	S (83) = 7	R (82) = 7
H (72) = 7	> (62) = 6	< (60) = 6
% (37) = 5	m (109) = 5	v (118) = 5

0 (48) = 5	/ (47) = 4	5 (53) = 4
\ (92) = 4	V (86) = 4	6 (54) = 4
- (45) = 3	x (120) = 3	b (98) = 3
' (39) = 3	L (76) = 3	! (33) = 2
: (58) = 2	# (35) = 2	U (85) = 2
E (69) = 2	4 (52) = 1	I (73) = 1
w (119) = 1	O (79) = 1	z (122) = 1
3 (51) = 1	N (78) = 1	

We see immediately that the most frequent character is the space with a count of 154, followed by the letter ‘e’ with a count of 77, then ‘n’ with 60, etc.

Strange, where does “z” appear? Ah yes, in sizeof. And that I? Ah in FILE, ok, seems to be working.

1.18.1 Fine points of structure use

1) When you have a *pointer* to a structure and you want to access a member of it you should use the syntax:
```
pointer->field
```

2) When you have a structure *OBJECT*, not a pointer, you should use the syntax:
```
object.field
```

Beginners easily confuse this.

3) When you have an array of structures, you index it using the normal array notation syntax, then use the object or the pointer in the array. If you have an array of pointers to structures you use:
```
array[index]->field
```

4) If you have an array of structures you use:
```
array[index].field
```

5) If you are interested in the offset of the field, i.e. the distance in bytes from the beginning of the structure to the field in question you use the `offsetof` macro defined in stddef.h:
```
offsetof(structure or typedef name,member name)
```

For instance to know the offset of the Frequency field in the structure CHARS above we would write:

```
offsetof(CHARS,Frequency)
```

This would return an integer with the offset in bytes.

Summary: Files are a sequence of bytes. They are central to most programs. Here is a short overview of the functions that use files:

Name	*Purpose*
fopen	Opens a file
fclose	Closes a file
fprintf	Formatted output to a file
fputc	Puts a character in a file
putchar	Puts a character to stdout
fputs	Puts a string in a file.

fread	Reads from a file a specified amount of data into a buffer.
freopen	Reassigns a file pointer
fscanf	Reads a formatted string from a file
fsetpos	Assigns the file pointer (the current position)
fseek	Moves the current position relative to the start of the file, to the end of the file, or relative to the current position
ftell	returns the current position
fwrite	Writes a buffer into a file
tmpnam	Returns a temporary file name
unlink	Erases a file
remove	Erases a file
rename	Renames a file.
rewind	Repositions the file pointer to the beginning of a file.
setbuf	Controls file buffering.
ungetc	Pushes a character back into a file.

1.19 Identifier scope and linkage

Until now we have used identifiers and scopes without really caring to define precisely the details. This is unavoidable at the beginning, some things must be left unexplained at first, but it is better to fill the gaps now.

An identifier in C can denote:[61]

- an object.
- a function
- a tag or a member of a structure, union or enum
- a typedef
- a label

For each different entity that an identifier designates, the identifier can be used (is visible) only within a region of a program called its scope. There are four kinds of scopes in C.

The **file scope** is built from all identifiers declared outside any block or parameter declaration, it is the outermost scope, where global variables and functions are declared.

A **function scope** is given only to label identifiers.

The **block scope** is built from all identifiers that are defined within the block. A block scope can nest other blocks.

The **function prototype scope** is the list of parameters of a function. Identifiers declared within this scope are visible only within it.

Let's see a concrete example of this:

```
static int Counter = 780;// file scope
extern void fn(int Counter); // function prototype scope
void function(int newValue, int Counter) // Block scope
```

61. You see the infinite loop here? Tell me: why is this loop never ending? Look at the code again.

```
{
     double d = newValue;
label:
     for (int i = 0; i< 10;i++) {
          if (i < newValue) {
               char msg[45];
               int Counter = 78;

               sprintf(msg,"i=%d\n",i*Counter); ←
          }
          if (i == 4)
               goto label;[62]
     }
}
```

At the point indicated by the arrow, the poor “Counter” identifier has had a busy life:

- It was bound to an integer object with file scope
- Then it had another incarnation within the function prototype scope
- Then, it was bound to the variables of the function ‘setCounter’ as a parameter
- That definition was again “shadowed” by a new definition in an inner block, as a local variable.

The value of “Counter” at the arrow is 78. When that scope is finished its value will be the value of the parameter called Counter, within the function “function”.

When the function definition finishes, the file scope is again the current scope, and “Counter” reverts to its value of 780.

The “linkage” of an identifier refers to the visibility to other modules. Basically, all identifiers that appear at a global scope (file scope) and refer to some object are visible from other modules, unless you explicitly declare otherwise by using the “static” keyword.

Problems can appear if you first declare an identifier as static, and later on, you define it as external. For instance:

```
static void foo(void);
```

and several hundred lines below you declare:

```
void foo(void) {
     ...
}
```

Which one should the compiler use? static or not static? That is the question…

Lcc-win32 chooses always non-static, to the contrary of Microsoft’s compiler that chooses always static. Note that the behavior of the compiler is explicitly left undefined in the standard, so both behaviors are correct.

1.20 Top-down analysis

The goal of this introduction is not to show you a lot of code, but to tell you how that code is constructed. A central point in software construction is learning how you decompose a task in

62. Yes, but then all initializations are done out of their respective contexts. Some people say this is the wrong way to go, and that each data type should initialize in a separate init procedure. In this concrete example and in many situations, making a global init procedure is a correct way of building software. Other contexts may be different of course.

sub-tasks, so that the whole is more modular, and easier to manage and change. Let's go back to our frequencies example. We see that the "main" function accomplishes several tasks: it checks its arguments, opens a file, checks the result, initializes the Frequency array, etc.

This is an example of a monolithic, highly complex function. We could decompose it into smaller pieces easily, for example by assigning each task to a single procedure.

One of the first things we could do is to put the checking of the input arguments in a single procedure:

```
FILE *checkargs(int argc,char *argv[])
{
     FILE *infile = NULL;
     if (argc < 2) {
          printf("Usage: countchars <file name>\n");
     }
     else {
     infile = fopen(argv[1],"rb");
          if (infile == NULL) {
               printf("File %s doesn't exist\n",argv[1]);
     }
}
     return infile;
}
```

We pass the arguments of main to our check procedure, that writes the corresponding error message if appropriate, and returns either an open FILE *, or NULL, if an error was detected. The calling function just tests this value, and exits if something failed.

```
int main(int argc,char *argv[])
{
     int count=0;
     FILE *infile = checkargs(argc,argv);
     int c;

     if (infile == NULL)
          return 1;
     for (int i = 0; i<256; i++) {
          Frequencies[i].CharacterValue = i;
     }
     ... the rest of "main" ...
}
```

The details of how the checking of the arguments is done is now away from the main line of the «main» function. The whole function is now easier to read, and the title of the piece of code that we hid away tells the reader immediately what is going on behind that function call.

The next step, is the initializing of the Frequencies array. This is common pattern that we find very often when building software: most programs initialize tables, and do some setup before actually beginning the computation they want to perform. The best is to collect all those initializations into a single procedure, so that any new initializations aren't scattered all around but concentrated in a single function.[63]

63. We find very often the expression:

```
while ((c=fgetc(infile)) != EOF) { ... }
```

instead of the expression above. Both expressions are strictly equivalent, since we first execute the fgetc function, assigning the result to c, then we compare that result with EOF. In the second, slightly more complicated, we need a set of parentheses to force execution to execute the fgetc and the assignment first. There is the danger that c would get assigned the result of the comparison of the fgetc result with EOF instead of the character itself.

```
void Initialize(void)
{
     for (int i = 0; i<256; i++) {
          Frequencies[i].CharacterValue = i;
     }
}
```

Following our analysis of "main", we see that the next steps are the processing of the opened file. We read a character, and we update the Frequencies array. Well, this is a single task that can be delegated to a function, a function that would need the opened file handle to read from, and a Frequencies array pointer to be updated.

We develop a ProcessFile function as follows:

```
int ProcessFile(FILE *infile,CHARS *Frequencies)
{
     int count = 0;
     int c = fgetc(infile);[64]
     while (c != EOF) {
          count++;
          if (c >= ' ')
               Frequencies[c].Frequency++;
          c = fgetc(infile);
     }
     return count;
}
```

The interface with the rest of the software for this function looks like this:

```
count = ProcessFile(infile,Frequencies);
```

We could have avoided passing the Frequencies array to ProcessFile, since it is a global variable. Its scope is valid when we are defining ProcessFile, and we could use that array directly. But there are good reasons to avoid that. Our global array can become a bottleneck, if we decide later to process more than one file, and store the results of several files, maybe combining them and adding up their frequencies.

Another reason to explicitly pass the `Frequencies` array as a parameter is of course clarity. The `Frequencies` array **is** a parameter of this function, since this function modifies it. Passing it explicitly to a routine that modifies it makes the software clearer, and this is worth the few cycles the machine needs to push that address in the stack.

When we write software in today's powerful micro-processors, it is important to get rid of the frame of mind of twenty years ago, when saving every cycle of machine time was of utmost importance. Pushing an extra argument, in this case the address of the `Frequencies` array, takes 1 cycle. At a speed of 1400-2500 MHz, this cycle isn't a high price to pay.

Continuing our analysis of our "main" function, we notice that the next task, is displaying the output of the frequencies array. This is quite a well-defined task, since it takes the array as input, and should produce the desired display. We define then, a new function `DisplayOutput()` that will do that. Its single parameter is the same `Frequencies` array.

```
void DisplayOutput(CHARS *Frequencies)
{
     for (int count=0; count<256;count++) {
          if (Frequencies[count].Frequency != 0) {
               printf("%3c (%4d) = %d\n",
```

64. The standard files defined by the standard are: stdin, or standard input, to read from the current input device, the stdout or standard output, and the stderr stream, to show errors. Initially, stdin is bound to the keyboard, stdout and stderr to the screen.

```
                    Frequencies[count].CharacterValue,
                    Frequencies[count].CharacterValue,
                    Frequencies[count].Frequency);
            }
        }
    }
```

Let's look at our "main() function again:

```
int main(int argc,char *argv[])
{
    int count;
    FILE *infile = checkargs(argc,argv);

    if (infile == NULL)
        return 1;
    Initialize();
    count = ProcessFile(infile,Frequencies);
    fclose(infile);
    printf("%d chars in file\n",count);
    qsort(Frequencies,256,sizeof(CHARS),compare);
    DisplayOutput(Frequencies);
}
```

Note how much clearer our "main" function is now. Instead of a lot of code without any structure we find a much smaller procedure that is constructed from smaller and easily understandable parts.

Now, suppose that we want to handle several files. With this organization, it is straightforward to arrange for this in a loop. ProcessFile() receives an open FILE and a Frequencies array, both can be easily changed now. A modular program is easier to modify than a monolithic one!

We see too that the function that process the files leaves them open. We could streamline more «main» if we got rid of that in the ProcessFile() function, but I find it personally better that the same function that opens a file closes it too, so that the reader can see if the fopen/fclose calls match.

1.21 Extending a program

Let's suppose then, that we want to investigate all frequencies of characters in a directory, choosing all files that end in a specific extension, for instance *.c. In principle this is easy, we pass to ProcessFile a different open file each time, and the same Frequencies array.

We develop a GetNextFile function, that using our "*.c" character string, will find the first file that meets this name specifications ("foo.c" for example), and then will find all the other files in the same directory that end with a .c extension.

How do we do this?

Well, looking at the documentation, we find that we have two functions that could help us: findfirst, and findnext. The documentation tells us that findfirst has a prototype like this:

```
long findfirst( char *spec, struct _finddata_t *fileinfo);
```

This means that it receives a pointer to a character string, and will fill the fileinfo structure with information about the file it finds, if it finds it.

We can know if findfirst really found something by looking at the return result. It will be –1 if there was an error, or a "unique value" otherwise. The documentation tell us too that errno, the global error variable will be set to ENOENT if there wasn't any file, or to EINVAL if the file specification itself contained an error and findfirst couldn't use it.

Then, there is findnext that looks like this:

```
int findnext(long handle,struct _finddata_t *fileinfo);
```

It uses the "unique value" returned by findfirst, and fills the fileinfo structure if it finds another file that matches the original specification. If it doesn't find a file it will return –1, as findfirst. If it does find another file, it will return zero.

We see now that a FILE that was uniquely bound to a name is now a possible ambiguous file specification, that can contain a _finddata_t (whatever that is) that can be used to read several FILEs.

We could formalize this within our program like this:

```
typedef struct tagStream  {
     char *Name;
     struct _finddata_t FindData;
     long handle;
     FILE *file;
} STREAM;
```

Our function checkargs() returns a FILE pointer now. It could return a pointer to this STREAM structure we have just defined, or NULL, if there was an error. Our program then, would loop asking for the next file, adding to our Frequencies array the new character frequencies found.

The first function to be modified is checkargs. We keep the interface with the calling function (return NULL on error), but we change the inner workings of it, so that instead of calling fopen, it calls findfirst.

```
STREAM *checkargs(int argc,char *argv[])
{
     STREAM *infile = NULL;
     long findfirstResult;
     struct _finddata_t fd;

     if (argc < 2) {
          printf("Usage: countchars <file name>\n");
     }
     else {
          findfirstResult = findfirst(argv[1],&fd);
          if (findfirstResult < 0) {
               printf("File %s doesn't exist\n",argv[1]);
               return NULL;
          }
     }
     infile = malloc(sizeof(STREAM));
     infile->Name = argv[1];
     memcpy(&infile->FindData, &fd,
               sizeof( struct _finddata_t ));
     infile->File = fopen(fd.name,"rb");
     infile->handle = findfirstResult;
     return infile;
}
```

We store in the local variable findfirstResult the long returned by findfirst. We test then, if smaller than zero, i.e. if something went wrong. If findfirst failed, this is equivalent to our former program when it opened a file and tested for NULL.

But now comes an interesting part. If all went well, we ask the system using the built-in memory allocator "malloc" for a piece of fresh RAM at least of size STREAM. If this call fails, there is no more memory left. For the time being (see later) we ignore this possibility.

We want to store in there all the parameters we need to use the findfirst/findnext function pair with easy, and we want to copy the finddata_t into our own structure, and even put the name of the stream and a FILE pointer into it. To do that, we need memory, and we ask it to the "malloc" allocator.

Once that done, we fill the new STREAM with the data:

- we set its name using the same pointer as argv[1],
- we copy the fd variable into our newly allocated structure, and
- we set the file pointer of our new structure with fopen, so that we can use the stream to read characters from it.

Another alternative to using the built-in memory allocator would have been to declare a global variable, call it CurrentStream that would contain all our data. We could have declared somewhere in the global scope something like:

```
STREAM CurrentStream;
```

and use always that variable.

This has several drawbacks however, the bigger of it being that global variables make following the program quite difficult. They aren't documented in function calls, they are always "passed" implicitly, they can't be used in a multi-threaded context, etc.

Better is to allocate a new STREAM each time we need one. This implies some memory management, something we will discuss in-depth later on.

Now, we should modify our ProcessFile function, since we are passing to it a STREAM and not a FILE. This is easily done like this:

```
int ProcessFile(STREAM *infile,CHARS *Frequencies)
{
     int count = 0;
     int c = fgetc(infile->file);
     while (c != EOF) {
          count++;
          if (c >= ' ')
               Frequencies[c].Frequency++;
          c = fgetc(infile->file);
     }
     return count;
}
```

Instead of reading directly from the infile argument, we use the "file" member of it. That's all. Note that infile is a pointer, so we use the notation with the arrow, instead of a point to access the "file" member of the structure.

But there is something wrong with the name of the function. It wrongly implies that we are processing a FILE instead of a stream. Let's change it to ProcessStream, and change the name of the stream argument to instream, to make things clearer:

```
int ProcessStream(STREAM *instream,CHARS *Frequencies)
{
     int count = 0;
     int c = fgetc(instream->file);
     while (c != EOF) {
          count++;
          if (c >= ' ')
               Frequencies[c].Frequency++;
          c = fgetc(instream->file);
     }
     return count;
```

```
}
```

This looks cleaner.

Now we have to change our “main” function, to make it read all the files that match the given name.

Our new main procedure looks like this:

```
int main(int argc,char *argv[])
{
     int count=0;
     STREAM *infile=checkargs(argc,argv);

     if (infile == NULL) {
          return(1);
     }
     Initialize();
     do {
          count += ProcessStream(infile,Frequencies);
          fclose(infile->file);
          infile = GetNext(infile);
     } while (infile != 0);
     printf("%d chars in file\n",count);
     qsort(Frequencies,256,sizeof(CHARS),compare);
     DisplayOutput(Frequencies);
     return 0;
}
```

We didn’t have to change a lot, thanks to the fact that the complexities of reading and handling a stream are now hidden in a function, with well-defined parameters. We build a GetNext function that returns either a valid new stream or NULL, if it fails. It looks like this:

```
STREAM *GetNext(STREAM *stream)
{
     STREAM *result;
     struct _finddata_t fd;
     long findnextResult = _findnext(stream->handle,&fd);

     if (findnextResult < 0)
          return NULL;
     result = malloc(sizeof(STREAM));
     memcpy(result->FindData,
            &fd,
            sizeof(struct _finddata_t));
     result->handle = stream->handle;
     result->file = fopen(fd.name,"rb");
     return result;
}
```

In the same manner that we allocate RAM for our first STREAM, we allocate now a new one, and copy into it our “finddata” handle, and we open the file.

We compile, and we get a compiler warning:

```
D:\lcc\examples>lcc -g2 freq1.c
Warning freq1.c: 44  missing prototype for memcpy
Warning freq1.c: 94  missing prototype for memcpy
0 errors, 2 warnings
```

Yes, but where is memcpy defined? We look at the documentation using F1 in Wedit, and we find out that it needs the <string.h> header file. We recompile and we get:

```
H:\lcc\examples>lcc freq1.c
```

```
Error freq1.c: 95  type error in argument 1 to `memcpy'; found
`struct _finddata_t' expected `pointer to void'
1 errors, 0 warnings
```

Wow, an error. We look into the offending line, and we see:

```
memcpy(result->FindData,&fd,sizeof(struct _finddata_t));
```

Well, we are passing it a structure, and the poor function is expecting a pointer !

This is a serious error. We correct it like this:

```
memcpy(&result->FindData,&fd,sizeof(struct _finddata_t));
```

We take the address of the destination structure using the address-of operator "&". We see that we would have never known of this error until run-time when our program would have crashed with no apparent reason; a difficult error to find. Note: *always use the right header file to avoid this kind of errors!*

Our program now looks like this:

```
#include <stdio.h> // We need it for using the FILE structure
#include <stdlib.h>// We need it for using malloc
#include <io.h>// We need it for using findfirst/findnext
#include <string.h>// We need it for memcpy
typedef struct tagChars {
     int CharacterValue;// The ASCII value of the character
     int Frequency;// How many seen so far
} CHARS;
typedef struct tagStream  {
     char Name;// Input name with possible "*" or "?" chars in it
     struct _finddata_t FindData;
     long handle;
     FILE *file;// An open file
} STREAM;
CHARS Frequencies[256]; // Array of frequencies
int compare(){} // Skipped, it is the same as above
STREAM *checkargs(int argc,char *argv[])
{
     STREAM *infile = NULL;
     long findfirstResult;
     struct _finddata_t fd;

     if (argc < 2) { // Test if enough arguments were passed
          printf("Usage: countchars <file name>\n");
     }
     else {// Call the _findfirst function with the name and info
buffer
          findfirstResult = _findfirst(argv[1],&fd);
          // Test result of findfirst, and return immediately NULL if
wrong
          if (findfirstResult < 0) {
               printf("File %s doesn't exist\n",argv[1]);
               return NULL;
          }
          // Ask more memory to the allocator
          infile = malloc(sizeof(STREAM));
          // Set the name of the new stream
          infile->Name = argv[1];
// Note the first argument of this call: it's the address within the infile structure of the FindData
//  member. We take the address with the "&" operator. Since we are using a pointer, we have
// to dereference a pointer, i.e. with "->" and not with the ".". Note that the "&"' operator is used
// with the "fd" local variable to obtain a pointer from a structure member.
          memcpy(&infile->FindData,&fd,
```

```
            sizeof(struct _finddata_t));
          infile->file = fopen(fd.name,"rb");
          infile->handle = findfirstResult;
     }
     return infile;
}
void Initialize(void) { this is the same as the function before}
int ProcessStream(STREAM *instream,CHARS *Frequencies)
{
     int count = 0;
     int c = fgetc(instream->file);
     while (c != EOF) {
          count++;
          if (c >= ' ')
               Frequencies[c].Frequency++;
          c = fgetc(instream->file);
     }
     return count;
}
void DisplayOutput(CHARS *Frequencies){ this is the same function as before }
STREAM *GetNext(STREAM *stream)
{
     STREAM *result;
     struct _finddata_t fd;
     long findnextResult = _findnext(stream->handle,&fd);

     if (findnextResult < 0)
          return NULL;
     result = malloc(sizeof(STREAM));
     memcpy(&result->FindData,&fd,
            sizeof(struct _finddata_t));
     result->handle = stream->handle;
     result->file = fopen(fd.name,"rb");
     result->Name = stream->Name;
     return result;
}
int main(int argc,char *argv[])
{
     int count=0;
     STREAM *infile=checkargs(argc,argv);

     if (infile == NULL) {
          return(1);
     }
     Initialize();
     do {
          count += ProcessStream(infile,Frequencies);
          fclose(infile->file);
          infile = GetNext(infile);
     } while (infile != 0);
     printf("%d chars in file\n",count);
     qsort(Frequencies,256,sizeof(CHARS),compare);
     DisplayOutput(Frequencies);
     return 0;
}
```

1.22 Improving the design

There are several remarks that can be done about our program. The first one is that the memory allocator could very well fail, when there is no more memory available. When the allocator fails, it returns NULL. Since we never test for this possibility, our program would crash in low memory conditions. What a shame!

We arrange for this immediately. Instead of using the allocator, we will write a function that will call the allocator, test the result, and call the exit() routine if there is no more memory left. Continuing processing without more memory is impossible anyway.

```
void *xmalloc(unsigned int size)
{
     void *result = malloc(size);

     if (result == NULL) {
          fprintf(sdterr,
              No more memory left!\nProcessing stops\n);
          exit(1);
     }
     return result;
}
```

Note that we keep the same signature, i.e. the same type of result and the same type of arguments as the original function we want to replace. This is function sub classing.

Note too, that we use fprintf instead of printf. Fprintf takes an extra argument, a file where the output should go. We use the predefined file of standard error, instead of the normal output file stdout, that printf implicitly takes.

Why?

Because it is possible that the user redirects the output to a file instead of letting the output go directly to the screen. In that case we would write our error messages to that file, and the user would not see the error message.[65]

We change all occurrences of malloc by xmalloc, and this error is gone.

We change too, all other error-reporting functions, to take into account stderr.

But there are other issues. Take for instance our finddata_t structure that we carry around in each STREAM structure. What's its use? We do not use it anywhere; just copy it into our STREAM.

But why we introduced that in the first place?

Well, we didn't really know much about findfirst, etc, and we thought it could be useful.

So we are stuck with it?

No, not really. Actually, it is very easy to get rid of it. We just change the structure STREAM like this:

```
typedef struct tagStream  {
        char *Name;
        long handle;
        FILE *file;
} STREAM;
```

65. Some people would say that this is not "Standard C", since the standard doesn't explicitly allow for this. But I would like to point out that the standard explicitly states (page 96 of my edition) that: "An implementation may accept other forms of constant expressions.". The implementation lcc-win32 then, is free to accept the above declaration as a constant expression.

and we take care to erase any references to that member. We eliminate the memcpy calls, and that's all. Our program is smaller, uses less memory, and, what is essential, does the same thing quicker than the older version, since we spare the copying.

It is very important to learn from the beginning that software gains not only with the lines of code that you write, but also with the lines of code that you eliminate!

1.23 Path handling

But let's continue with our program. It looks solid, and running it with a few files in the current directory works.

Let's try then:

```
H:\lcc\examples>freq1 "..\src77\*.c" | more
```

CRASH!

What's happening?

Following the program in the debugger, we see that we do not test for NULL, when opening a file. We correct this both in checkargs and GetNext. We write a function Fopen, using the same model as xmalloc: if it can't open a file, it will show an error message in stderr, and exit the program.

```
FILE *Fopen(char *name,char *mode)
{
        FILE *result = fopen(name,mode);
        if (result == NULL) {
                fprintf(stderr,
                      "Impossible to open '%s'\n",name);
                exit(1);
        }
        return result;
}
```

Ok, we change all `fopen()` into `Fopen()` , recompile, and we test again:

```
H:\lcc\examples>freq1 "..\src77\*.c" | more
Impossible to open 'Alloc.c'
```

Well, this looks better, but why doesn't open Alloc.c?

Well, it seems that the path is not being passed to fopen, so that it tries to open the file in the current directory, instead of opening it in the directory we specify in the command line.

One way to solve this, would be to change our current directory to the directory specified in the command line, and then try to open the file. We could do this in checkargs, since it is there where we open a file for the first time. All other files will work, if we change the current directory there.

How we could do this?

If the argument contains backslashes, it means there is a path component in it. We could copy the string up to the last backslash, and then change our current directory to that. For instance, if we find an argument like "..\src77*.c", the path component would be "..\src77\".

Here is an updated version of checkargs:

```
STREAM *checkargs(int argc,char *argv[])
{
        STREAM *infile = NULL;
        long findfirstResult;
        struct _finddata_t fd;
```

```
            char *p;

            if (argc < 2) { ... error handling elided ...
            }
            else {
                 findfirstResult = _findfirst(argv[1],&fd);
                 if (findfirstResult < 0) {
                           fprintf(stderr,
                           "File %s doesn't exist\n",argv[1]);
                           return NULL;
                 }
                 infile = malloc(sizeof(STREAM));
                 infile->Name = argv[1];
                 p = strrchr(argv[1],'\\');
                 if (p) {
                    *p = 0;
                    chdir(argv[1]);
                    *p = '\\';
                }
                infile->file = fopen(fd.name,"rb");
                infile->handle = findfirstResult;
            }
            return infile;
    }
```

We use the library function strrchr. That function will return a pointer to the last position where the given character appears in the input string, or NULL, if the given character doesn't appear at all. Using that pointer, we replace the backslash with a NULL character. Since a zero terminates all strings in C, this will effectively cut the string at that position.

Using that path, we call another library function, chdir that does what is name indicates: changes the current directory to the given one. Its prototype is in <direct.h>.

After changing the current directory, we restore the argument argv[1] to its previous value, using the same pointer "p". Note too, that when we enter a backslash in a character constant (enclosed in *single* quotes), we have to double it. This is because the backslash is used, as within strings, for indicating characters like '\n', or others.

But this isn't a good solution. We change the current directory, instead of actually using the path information. Changing the current directory could have serious consequences in the working of other functions. If our program would be a part of a bigger software, this solution would surely provoke more headaches than it solves. So, let's use our "name" field, that up to now isn't being used at all. Instead of passing a name to Fopen, we will pass it a STREAM structure, and it will be Fopen that will take care of opening the right file. We change it like this:

```
FILE *Fopen(STREAM *stream,char *name,char *mode)
{
        FILE *result;
        char fullname[1024],*p;

        p = strrchr(stream->Name,'\\');
        if (p == NULL) {
                fullname[0] = 0;
        }
        else {
                *p = 0;
                strcpy(fullname,stream->Name);
                strcat(fullname,"\\");
                *p = '\\';
```

```
            }
            strcat(fullname,name);
            result = fopen(fullname,mode);
            if (result == NULL) {
                    fprintf(stderr,
                      "Impossible to open '%s'\n",fullname);
                    exit(1);
            }
            return result;
    }
```

We declare a array of characters, with enough characters inside to hold a maximum path, and a few more. Then, and in the same declaration, we declare a character pointer, p. This pointer will be set with strrchr. If there isn't any backslash in the path, we just set the start of our fullname[ ] to zero. If there is a path, we cut the path component as we did before, and copy the path component into the fullname variable. The library function strcpy will copy the second argument to the first one, including the null character for terminating correctly the string.

We add then a backslash using the strcat function that appends to its first argument the second string. It does this by copying starting at the terminator for the string, and copying all of its second argument, including the terminator.

We restore the string, and append to our full path the given name. In our example, we copy into fullpath the character string "..\src77", then we add the backslash, and then we add the rest of the name to build a name like "..\src77\alloc.c".

This done, we look again into our program. Yes, there are things that could be improved. For instance, we use the 256 to write the number of elements of the array Frequencies. We could improve the readability of we devised a macro `NELEMNTS`, that would make the right calculations for us.

That macro could be written as follows:

```
#define NELEMENTS(array) (sizeof(array)/sizeof(array[0]))
```

This means just that the number of elements in any array, is the size of that array, divided by the size of each element. Since all elements have the same size, we can take any element to make the division. Taking array[0] is the best one, since that element is always present.

Now, we can substitute the 256 by `NELEMENTS(Frequencies)`, and even if we change our program to use Unicode, with 65535 different characters, and each character over two bytes, our size will remain correct. This construct, like many others, points to one direction: making the program more flexible and more robust to change.

We still have our 256 in the definition of the array though. We can define the size of the array using the preprocessor like this:

```
#define FrequencyArraySize 256
```

This allows us later by changing this single line, to modify all places where the size of the array is needed.

Lcc-win32 allows you an alternative way of defining this:

```
static const int FrequencyArraySize = 256;
```

This will work just like the pre-processor definition.[66]

66. Memory allocation problems plague also other languages like C++ that use a similar schema than C.

1.23.1 Security considerations

Let's come back to the code snippet above:

```
else {
        *p = 0;
        strcpy(fullname,stream->Name);
        strcat(fullname,"\\");
        *p = '\\';
}
strcat(fullname,name);
```

If the length of `stream->Name` is bigger than the size of the buffer, a security hole appears: the program will write over the local variables of the function, and later over the return address, stored in the stack. when our function is active.

We have a stack layout like this:

local variables of the calling function

arguments to the current function

return address

saved frame pointer

local variables: `p` is at the lowest address, then `fullname`, then `result`.

saved registers

stack pointer is here

An overflow when copying the Name field would destroy the saved frame pointer, and maybe destroy the return address. The copy starts at the lowest address, the start of the fullname buffer, then goes on to higher addresses.

After the overflow occurs, the function would continue normally until it executes the return statement. Depending on the type of overflow, the return address would contain characters from the Name field, that in most cases would lead to a wrong return address and a program crash.

But that is not always the case. It could be that a user of our program would notice this style of programming, and give us a file name that, when overflowing in the copy to the temporary buffer would form a correct return address, that would pass then control to some other routine that the malicious user prepared for us.

This kinds of exploits can be avoided if we use other functions of the standard library:

```
else {
        *p = 0;
        strncpy(fullname,stream->Name,sizeof(fullname));
        strncat(fullname,"\\",sizeof(fullname));
        *p = '\\';
}
strncat(fullname,name,sizeof(fullname));
```

Those functions test for overflow conditions and are safer in case of unforeseen input patterns.

There is a problem with strncpy though. It does NOT terminate the resulting string with a zero. If we get a Name field with exactly sizeof(fullname) chars, the string will be missing the trailing zero since strncpy doesn't add it. One way to cover this possibility is to do:

```
else {
        *p = 0;
        fullname[sizeof(fullname)-1] = 0;
```

```
                strncpy(fullname,stream->Name,sizeof(fullname)-1);
                strncat(fullname,"\\",sizeof(fullname)-1);
                *p = '\\';
        }
        strncat(fullname,name,sizeof(fullname)-1);
```

We finish fullname with a zero, and we copy only up to `sizeof(fullname)-1` chars, leaving our terminating zero intact.

But you should have noted that there is something wrong here. We do initialize the terminating zero within an else statement. What happens if the execution of the function takes the other path?

We see the following:

```
FILE *Fopen(STREAM *stream,char *name,char *mode)
{
        FILE *result;
        char fullname[1024],*p;

        p = strrchr(stream->Name,'\\');
        if (p == NULL) {
                fullname[0] = 0;
        }
        else {
                *p = 0;
                fullname[sizeof(fullname)-1] = 0;
                strncpy(fullname,stream->Name,sizeof(fullname)-1);
                strncat(fullname,"\\",sizeof(fullname)-1);
                *p = '\\';
        }
        strncat(fullname,name,sizeof(fullname)-1);
        result = fopen(fullname,mode);
        if (result == NULL) {
                fprintf(stderr,
                  "Impossible to open '%s'\n",fullname);
                exit(1);
        }
        return result;
}
```

If p is NULL, we will initialize the first char of fullname to zero. Then, execution continues at the strncat call after the else statement, and we will copy at most `sizeof(fullname)-1` chars into it, overwriting the zero and maybe leaving the fullname character array without the terminating zero if the length of the passed buffer is bigger than the size of `fullname`. That could lead to a crash in fopen that surely expects a well formed string.

The solution is to finish the fullname buffer in ALL cases.

```
FILE *Fopen(STREAM *stream,char *name,char *mode)
{
        FILE *result;
        char fullname[1024],*p;
        p = strrchr(stream->Name,'\\');
        fullname[sizeof(fullname)-1] = 0;
        if (p == NULL) {
                fullname[0] = 0;
        }
        else {
                *p = 0;
                strncpy(fullname,stream->Name,sizeof(fullname)-1);
                strncat(fullname,"\\",sizeof(fullname)-1);
```

```
                *p = '\\';
        }
        strncat(fullname,name,sizeof(fullname)-1);
        result = fopen(fullname,mode);
        if (result == NULL) {
                fprintf(stderr,
                  "Impossible to open '%s'\n",fullname);
                exit(1);
        }
        return result;
}
```

Never forget to initialize a variable in BOTH cases of an if statement. Bugs such as this are very difficult to catch later on.

What will happen if the name of the file is bigger than our buffer? This function will fail. The fopen call will return NULL since the file name has been truncated, and we will show an error telling that a truncated file name doesn't exist. Is this a good behavior?

It depends. For a technical user, a long and truncated file name could be an indicator that the file name is just too long. Better error reporting would be appropriate if required, for instance at the start of the function a test could produce a clear message like "Name too long".

Summary: We examined some of the functions that the C library provides for strings and directories. Strings are ubiquitous in any serious program. We will examine this with more depth in the next section.

Working with directories is mandatory if you make any program, even the simplest one. Here is an overview of the path handling functions as defined in the standard include file <direct.h>

Function	**Purpose**
`getcwd`	Returns the current directory
`chdir`	Changes the current directory
`chdrive`	Changes the current drive
`mkdir`	Makes a new directory
`rmdir`	Erase a directory if its empty.
`diskfree`	Returns the amount of space available in a disk.

1.24 Traditional string representation in C

In C character strings are represented by a sequence of bytes finished by a trailing zero byte. For example, if you got:

```
char *Name = “lcc-win32”;
```

You will have in memory something like this:

l	c	c	-	w	i	n	3	2	0
108	99	99	45	119	105	110	51	50	0

We will have at each of the position of the string array a byte containing a number: the ASCII equivalent of a letter. The array will be followed by a zero byte. Zero is not an ASCII character, and can't appear in character strings, so it means that the string finishes there.

This design is quite ancient, and dates to the beginning of the C language. It has several flaws, as you can immediately see:

- There is no way to know the length of a string besides parsing the whole character array until a zero byte is found.
- Any error where you forget to assign the last terminated byte, or this byte gets overwritten will have catastrophic consequences.
- There is no way to enforce indexing checks.

The most frequently used function of this library are:

"`strlen`" that returns an integer containing the length of the string. Example:

```
int len = strlen("Some character string");
```

Note that the length of the string is the number of characters **without** counting the trailing zero. The physical length of the string includes this zero byte however, and this has been (and will be) the source of an infinite number of bugs!

"`strcmp`" that compares two strings. If the strings are equal it returns zero. If the first is greater (in the lexicographical sense) than the second it returns a value greater than zero. If the first string is less than the second it returns some value less than zero. The order for the strings is based in the ASCII character set.

a == b	strcmp(a,b) == 0
a < b	strcmp(a,b) < 0
a >= b	strcmp(a,b) >= 0

"`strcpy`" copies one string into another. `strcpy(dst,src)` copies the `src` string into the `dst` string. This means it will start copying characters from the beginning of the `src` location to the `dst` location until it finds a zero byte in the `src` string. No checks are ever done, and it is assumed that the `dst` string contains sufficient space to hold the `src` string. If not, the whole program will be destroyed. One of the most common errors in C programming is forgetting these facts.

"`strcat`" appends a character string to another. `strcat(src,app)` will add all the characters of "`app`" at the end of the "`src`" string. For instance, if we have the string pointer that has the characters "`lccwin32`" as above, and we call the library function `strcat(str," compiler")` we will obtain the following sequence:

l	c	c	w	i	n	3	2		c	o	m	p	i	l	e	r	0
108	99	99	119	105	110	51	50	32	99	111	109	112	105	108	101	114	0

The common operations for strings are defined in the header file <string.h>.

Function	*Purpose*	*Function*	*Purpose*
strcat	Appends strings.	strerror	Get a system error message (strerror) or prints a user-supplied error message (_strerror).
strchr	Find the first occurrence of a character in a string	strlen	Find the length of a string
strrchr	Find the last occurrence of a character in a string	strncat	Append characters of a string.

strcmp	Compares two strings	strncpy	Copy strings up to a maximum length
strncmp	Compare strings up to a maximum length	strpbrk	Scan strings for characters in specified character sets.
strnicmp	Compare strings up to a maximum length ignoring case	strspn	Find the first substring
strcol	Compare strings using locale-specific information.	strstr	Find a substring
strcpy	Copy a string into another	stristr	Find a string ignoring case.
strcspn	Find a substring in a string	strtok	Find the next token in a string
strupr	Convert string to upper case	strdup	Duplicate a string. Uses malloc.
strlwr	Convert string to lower case	strrev	Reverse characters in a string

You will find the details in the online documentation.

Besides these functions in the standard C library, the operating system itself provides quite a few other functions that relate to strings. Besides some relicts of the 16 bit past like lstrcat and others, we find really useful functions, especially for UNICODE handling.

CharLower	CharLowerBuff	CharNext	CharNextExA
CharPrev	CharPrevExA	CharToOem	CharToOemBuff
CharUpper	CharUpperBuff	CompareString	FoldString
GetStringTypeA	GetStringTypeEx	GetStringTypeW	IsCharAlpha
IsCharAlphaNu-meric	IsCharLower	IsCharUpper	LoadString
lstrcat	lstrcmp	lstrcmpi	lstrcpy
lstrcpyn	lstrlen	MultiByteToWide-Char	OemToChar
OemToCharBuff	WideCharToMultiByte	wsprintf	wvsprintf

1.25 Memory management and memory layout

We have until now ignored the problem of memory management. We ask for more memory from the system, but we never release it, we are permanently leaking memory. This isn't a big problem in these small example applications, but we would surely run into trouble in bigger undertakings.

Memory is organized in a program in different areas:

1) The *initial data area* of the program. Here are stored compile time constants like the character strings we use, the tables we input as immediate program data, the space we allocate in fixed size arrays, and other items. This area is further divided into initialized data, and uninitialized data, that the program loader sets to zero before the program starts.

   When you write a declaration like `int data = 78;` the `data` variable will be stored in the initialized data area. When you just write at the global level `int data;` the variable will be stored in the uninitialized data area, and its value will be zero at program start.

2) The *stack*. Here is stored the procedure frame, i.e. the arguments and local variables of each function. This storage is dynamic: it grows and shrinks when procedures are called and they return. At any moment we have a stack pointer, stored in a machine register, that contains the machine address of the topmost position of the stack.

3) The *heap*. Here is the space that we obtain with malloc or equivalent routines. This also a dynamic data area, it grows when we allocate memory using `malloc`, and shrinks when we release the allocated memory with the `free()` library function.

There is no action needed from your side to manage the initial data area or the stack. The compiler takes care of all that.

The program however, manages the heap, i.e. it expects that ***you*** keep book *exactly and without any errors* from each piece of memory you allocate using malloc. This is a very exhausting undertaking that takes a lot of time and effort to get right. Things can be easy if you always free the allocated memory before leaving the function where they were allocated, but this is impossible in general, since there are functions that precisely return newly allocated memory for other sections of the program to use.

There is no other solution than to keep book in your head of each piece of RAM. Several errors, *all of them fatal*, can appear here:

- You allocate memory and forget to free it. This is a memory leak.
- You allocate memory, and you free it, but because of a complicated control flow (many ifs, whiles and other constructs) you free a piece of memory twice. This corrupts the whole memory allocation system, and in a few milliseconds all the memory of your program can be a horrible mess.
- You allocate memory, you free it once, but you forget that you had assigned the memory pointer to another pointer, or left it in a structure, etc. This is the dangling pointer problem. A pointer that points to an invalid memory location.

Memory leaks provoke that the RAM space used by the program is always growing, eventually provoking a crash, if the program runs for enough time for this to become significant. In short-lived programs, this can have no consequences, and even be declared as a way of memory management. The lcc compiler for instance, always allocates memory without ever bothering to free it, relying upon the windows system to free the memory when the program exits.

Freeing a piece of RAM *twice* is much more serious than a simple memory leak. It can completely confuse the malloc() system, and provoke that the next allocated piece of RAM will be

the same as another random piece of memory, a catastrophe in most cases. You write to a variable and without you knowing it, you are writing to another variable at the same time, destroying all data stored there.

More easy to find, since more or less it always provokes a trap, the dangling pointer problem can at any moment become the dreaded show stopper bug that crashes the whole program and makes the user of your program loose all the data he/she was working with.

I would be delighted to tell you how to avoid those bugs, but after more than 10 years working with the C language, I must confess to you that memory management bugs still plague my programs, as they plague all other C programmers.[67]

The basic problem is that the human mind doesn't work like a machine, and here we are asking people (i.e. programmers) to be like machines and keep book exactly of all the many small pieces of RAM a program uses during its lifetime without ever making a mistake.

But there is a solution that I have implemented in lcc-win32. Lcc-win32 comes with an automatic memory manager (also called garbage collector in the literature) written by Hans Boehm. This automatic memory manager will do what you should do but do not want to do: take care of all the pieces of RAM for you.

Using the automatic memory manager you just allocate memory with GC_malloc instead of allocating it with malloc. The signature (i.e. the result type and type of arguments) is the same as malloc, so by just replacing all malloc by GC_malloc in your program you can benefit of the automatic memory manager without writing any new line of code.

The memory manager works by inspecting regularly your whole heap and stack address space, and checking if there is anywhere a reference to the memory it manages. If it doesn't find any references to a piece of memory it will mark that memory as free and recycle it. It is a very simple schema, taken to almost perfection by several years of work from the part of the authors.

To use the memory manager you should add the gc.lib library to your link statement or indicate that library in the IDE in the linker configuration tab.

1.25.1 Functions for memory allocation

malloc	Returns a pointer to a newly allocated memory block
free	Releases a memory block
calloc	Returns a pointer to a newly allocated zero-filled memory block.
realloc	Resizes a memory block preserving its contents.
alloca	Allocate a memory block in the stack that is automatically destroyed when the function where the allocation is requested exits.
_msize	Returns the size of a block
_expand	Increases the size of a block without moving it.
GC_malloc	Allocates a memory block managed by the memory manager.

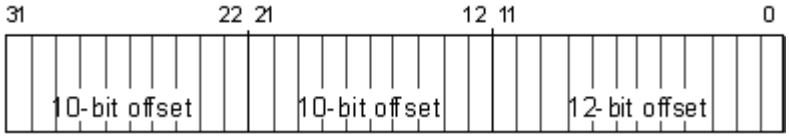
1.25.2 Memory layout under windows[68]

A 32 bit address can be used to address up to 4GB of RAM. From this potential address space, windows reserves for the system 2GB, leaving the other 2GB for each application. These

67. This discussion is based upon the article of Randy Kath, published in MSDN.

addresses, of course, are virtual, since not all PCs have 2GB of real RAM installed. To the windows memory manager, those numbers are just placeholders that are used to find the real memory address.

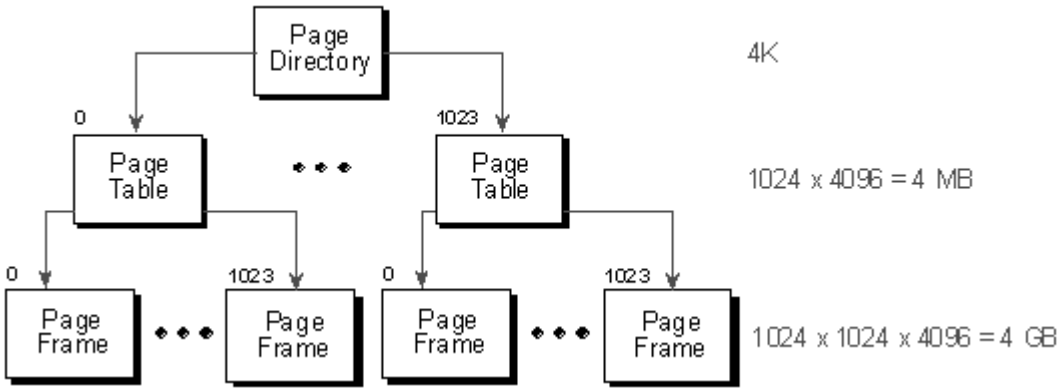
Each 32-bit address is divided in three groups, two containing 10 bits, and the third 12 bits.



The translation goes as follows:

The higher order bits (31-21) are used to index a page of memory called the page directory. Each process contains its own page directory, filled with 1024 numbers of 32 bits each, called page description entry or PDE for short.

The PDE is used to get the address of another special page, called page table. The second group of bits (21-12) is used to get the offset in that page table. Once the page frame found, the remaining 12 bits are used to address an individual byte within the page frame. Here is a figure that visualizes the structure:



We see that a considerable amount of memory is used to… manage memory. To realize the whole 4GB address space, we would use 4MB of RAM. But this is not as bad as it looks like, since Windows is smart enough to fill these pages as needed. And anyway, 4MB is not even 0.1% of the total 4GB address space offered by the system.[69]

Each process has its own page directory. This means that processes are protected from stray pointers in other programs. A bad pointer can't address anything outside the process address space. This is good news, compared to the horrible situation under windows 3.1 or even MSDOS, where a bad pointer would not only destroy the data of the application where it belonged, but destroyed data of other applications, making the whole system unstable. But this means too, that applications can't share data by sending just pointers around. A pointer is meaningful only in the application where it was created. Special mechanisms are needed (and provided by Windows) to allow sharing of data between applications.

---

68. Note that this is a logical view of this address translation process. The actual implementation is much more sophisticated, since Windows uses the memory manager of the CPU to speed up things. Please read the original article to get a more in-depth view, including the mechanism of page protection, the working set, and many other things.

69. Since this is stored in a 32 bit integer, the counter will overflow somewhere in year 2038. I hope I will be around to celebrate that event…

1.26 Memory management strategies

Each program needs some workspace to work in. How this space is managed (allocated, recycled, verified) makes a memory allocation strategy. Here is a short description of some of the most popular ones.

1.26.1 Static buffers

This is the simplest strategy. You reserve a fixed memory area (buffer) at compile time, and you use that space and not a byte more during the run time of the program.

Advantages:

1 It is the fastest possible memory management method since it as no run time overhead. There is no memory allocation, nor recycling that incurs in run time costs.

2 In memory starved systems (embedded systems, micro controller applications, etc) it is good to know that there is no possibility of memory fragmentation or other memory space costs associated with dynamic allocation.

Drawbacks:

1 Since the amount of memory allocated to the program is fixed, it is not possible to adapt memory consumption to the actual needs of the program. The static buffers could be either over-dimensioned, wasting memory space, or not enough to hold the data needed. Since the static buffers must be patterned after the biggest possible input, they will be over-dimensioned for the average case.

2 Unless programming is adapted to this strategy, it is difficult to reuse memory being used in different buffers to make space for a temporary surge in the space needs of the program.

1.26.2 Stack based allocation

The C standard allows for this when you write:

```
int fn(int a)
{
     char workspace[10000];
     ...
}
```

In this case, the compiler generates code that allocates 10000 bytes of storage from the stack. This is a refinement of the static buffers strategy. The stack is 1MB in normal programs but this can be increased with a special linker option.

A variant of this strategy allows for dynamic allocation. Instead of allocating a memory block of size “siz with malloc, we can write:

```
char workspace[siz];
```

and the compiler will generate code that allocates “siz” bytes from the program stack.

Advantages:

1 Very fast allocation and deallocation. To allocate a memory block only a few assembly instructions are needed. Deallocation is done without any extra cost when the function where the variables are located exits.

Drawbacks:

1 There is no way to know if the allocation fails. If the stack has reached its maximum size, the application will catastrophically fail with a stack overflow exception.

2 There is no way to pass this memory block to a calling function. Only functions called by the current function can see the memory allocated with this method.

3 Even if the C99 standard is already several years old, some compilers do not implement this. Microsoft compilers, for instance, do not allow this type of allocation. A work-around is to use the _alloca function. Instead of the code above you would write:

```
char *workspace = _alloca(siz);
```

1.26.3 "Arena" based allocation

This strategy is adapted when a lot of allocations are done in a particular sequence of the program, allocations that can be released in a single block after the phase of the program where they were done finishes. The program allocates a large amount of memory called "arena", and sub-allocates it to the consuming routines needing memory. When a certain phase of the program is reached the whole chunk of memory is either marked as free or released to the operating system.

The windows operating system provides support for this strategy with the APIs CreateHeap, HeapAlloc, and others.

Advantages:

1 Fewer calls to memory allocation/deallocation routines.

2 No global fragmentation of memory.

Drawbacks:

1 Since the size of the memory that will be needed is not known in advance, once an arena is full, the strategy fails or needs to be complemented with more sophisticated variations. A common solution is to make the arena a linked list of blocks, what needs a small processing overhead.

2 Determining when the moment has come to release all memory is tricky unless the data processed by the program has a logical structure that adapts itself to this strategy. Since there is no way of preserving data beyond the frontier where it is released, data that is to be preserved must be copied into another location.

1.26.4 The malloc / free strategy

This is the strategy that is most widely used in the C language. The standard provides the functions malloc, a function that returns a pointer to an available memory block, and free, a function that returns the block to the memory pool or to the operating system. The program allocates memory as needed, keeping track of each memory block, and freeing it when no longer needed. The free function needs a pointer to the same exact location that was returned by malloc. If the pointer was incremented or decremented, and it is passed to the free function havoc ensues.

Advantages:

1 It is very flexible, since the program can allocate as needed, without being imposed any other limit besides the normal limit of available memory.

2 It is economic since the program doesn't grab any more memory than it actually needs.

3 It is portable since it is based in functions required by the C language.

Drawbacks:

1 It is very error prone. Any error will provoke obscure and difficult to track bugs that

need advanced programming skills to find. And the possibilities of errors are numerous: freeing twice a memory block, passing a wrong pointer to free, forgetting to free a block, etc.

2 The time used by memory allocation functions can grow to an important percentage of the total run time of the application. The complexity of the application increases with all the code needed to keep track and free the memory blocks.

3 This strategy suffers from the memory fragmentation problem. After many malloc/free cycles, the memory space can be littered with many small blocks of memory, and when a request for a big block of memory arrives, the malloc system fails even if there is enough free memory to satisfy the request. Since it is impossible for the malloc system to move memory blocks around, no memory consolidation can be done.

4 Another problem is aliasing, i.e. when several pointers point to the same object. It is the responsibility of the programmer to invalidate all pointers to an object that has been freed, but this can be very difficult to do in practice. If any pointer to a freed object remains in some data structure, the next time it will be used the program can catastrophically fail or return invalid results, depending on whether the block was reallocated or not.

5 It can be slow. Malloc/free was a big bottleneck for performance using the Microsoft C runtime provided by the windows system for windows 95/98, for instance.

1.26.5 The malloc with no free strategy

This strategy uses only malloc, never freeing any memory. It is adapted to transient programs, i.e. programs that do a well defined task and then exit. It relies on the operating system to reclaim the memory used by the program.

Advantages:

1 Simplified programming, since all the code needed to keep track of memory blocks disappears.

2 It is fast since expensive calls to free are avoided.

Drawbacks:

1 The program could use more memory than strictly needed.

2 It is very difficult to incorporate software using this strategy into another program, i.e. to reuse it. This strategy can be easily converted into an arena based strategy though, since only a call to free the arena used by the program would be needed. It is even easier to convert it to a garbage collector based memory management. Just replace malloc by GC_malloc and you are done.

1.26.6 Automatic freeing (garbage collection).

This strategy relies upon a collector, i.e. a program that scans the stack and the global area of the application looking for pointers to its buffers. All the memory blocks that have a pointer to them, or to an inner portion of them, are marked as used, the others are considered free.

This strategy combines easy of use and reclaiming of memory in a winning combination for most applications, and it is the recommended strategy for people that do not feel like messing around in the debugger to track memory accounting bugs.

Advantages:

1 Program logic is simplified and freed from the chores of keeping track of memory blocks.

2 The program uses no more memory than needed since blocks no longer in use are recycled.

Drawbacks:

1 It requires strict alignment of pointers in addresses multiple of four. Normally, this is ensured by the compiler, but under certain packing conditions (compilation option -Zp1) the following layout could be disastrous:

```
#pragma pack(1)
struct {
     short a;
     char *ptr;
} s;
```

The pointer “ptr” will NOT be aligned in a memory address multiple of four, and it will not be seen by the collector because the alignment directive instructs the compiler to pack structure members.

2 You are supposed to store the pointers in memory accessible to the collector. If you store pointers to memory allocated by the collector in files, for instance, or in the “windows extra bytes” structure maintained by the OS, the collector will not see them and it will consider the memory they point to as free, releasing them again to the application when new requests are done.

3 Whenever a full gc is done (a full scan of the stack and the heap), a noticeable stop in program activity can be perceived by the user. In normal applications this can take a bit less than a second in large memory pools. The collector tries to improve this by doing small partial collections each time a call to its allocator function is done.

4 If you have only one reference to a block, the block will be retained. If you have stored somewhere a pointer to a block no longer needed, it can be very difficult indeed to find it.

5 The garbage collector of lcc-win32 is a conservative one, i.e. if something in the stack looks like a pointer, it will be assumed that this is a pointer (fail-safe) and the memory block referenced will be retained. This means that if by chance you are working with numeric data that contains numbers that can be interpreted as valid memory addresses more memory will be retained than strictly necessary. The collector provides special APIs for allocating tables that contain no pointers and whose contents will be ignored by the collector. Use them to avoid this problems.

1.26.7 Mixed strategies

Obviously you can use any combination of this methods in your programs. But some methods do not mix well. For instance combining malloc/free with automatic garbage collection exposes you to more errors than using only one strategy. If you pass to free a pointer allocated with GC_malloc chaos will reign in your memory areas. To the contrary, the stack allocation strategy can be combined very well with all other strategies since it is specially adapted to the allocation of small buffers that make for many of the calls to the allocator functions.

1.27 Counting words

There is no introduction to the C language without an example like this:

“Exercise 24.C: Write a program that counts the words in a given file, and reports its result sorted by word frequency.”

OK. Suppose you got one assignment like that. Suppose also, that we use the C language definition of a word, i.e. an identifier. A word is a sequence of letters that begins with an underscore or a letter and continues with the same set of characters or digits.

In principle, the solution could look like this:

1) Open the file and repeat for each character

2) If the character starts a word, scan the word and store it. Each word is stored once. If it is in the table already, the count is incremented for that word, otherwise it is entered in the table.

3) Sort the table of words by frequency

4) Print the report.

We start with an outline of the “main” procedure. The emphasis when developing a program is to avoid getting distracted by the details and keep the main line of the program in your head. We ignore all error checking for the time being.

```
int main(int argc,char *argv[])
{
     FILE *f;
     int c;

     f = fopen(argv[1],"r"); // open the input file
     c = fgetc(f);           // Read the first character
     while (c != EOF) {      // Until the end of file
          if (isWordStart(c)) { // Starts a word?
               ScanWord(c,f);     // Yes. Scan it
          }
          c = fgetc(f);      // Go on with the next character
     }
     fclose(f);          // Done with the file
     DoReports(argv[1]); // Print results
     return 0;           // Return with OK.
}
```

This would do nicely. We have now just to fill the gaps. Let’s start with the easy ones. A word, we said, is a sequence of `_ [A-Z] [a-z]` followed by `_ [A-Z][a-z][0-9]`. We write a function that returns 1 if a character is the start of an identifier (word).

```
int isWordStart(int c)
{
     if (c == '_')
          return 1;
     if (c >= 'a' && c <= 'z')
          return 1;
     if (c >= 'A' && c <= 'Z')
          return 1;
     return 0;
}
```

This function will do its job, but is not really optimal. We leave it like that for the time being. Remember: optimizations are done later, not when designing the program.

Now, we go to the more difficult task of scanning a word into the computer. The algorithm is simple: we just keep reading characters until we find a non-word char, that stops our loop. We use a local array in the stack that will hold until MAXIDLENGTH chars.

```
#define MAXIDLENGTH 512
int ScanWord(int firstchar,FILE *f)
{
     int i = 1, // index for the word buffer
          c=0;  // Character read
     char idbuf[MAXIDLENGTH+1]; // Buffer for the word

     idbuf[0] = firstchar; // We have at least one char
     c = fgetc(f);         // Read the next one
     while (isWordStart(c) || (c >= '0' && c <= '9')) {
          idbuf[i++] = c;  // Store it in the array
          if (i >= MAXIDLENGTH) { // Check for overflow!
               fprintf(stderr,
                 "Identifier too long\n");
               return 0;    // Returning zero will break the loop
                            // in the calling function
          }
          c = fgetc(f);    // Scan the next char
     }
     idbuf[i] = 0;         // Always zero terminate
     EnterWord(idbuf);     // Enter into the table
     return 1;             // OK to go on.
}
```

We hold the index into our array in the identifier “i”, for index. It starts at one since we receive already the first character of a word. Note that we test with this index if we are going to overflow our local table “idbuf”. We said before that error checking should be abstracted when designing the program but as any rule, that one has exceptions.

If we were going to leave a lot of obvious errors in many functions around, we would need a lot of work later on to fix all those errors. Fundamental error checking like a buffer overrun should always be in our minds from the beginning, so we do it immediately. Note that this test is a very simple one.

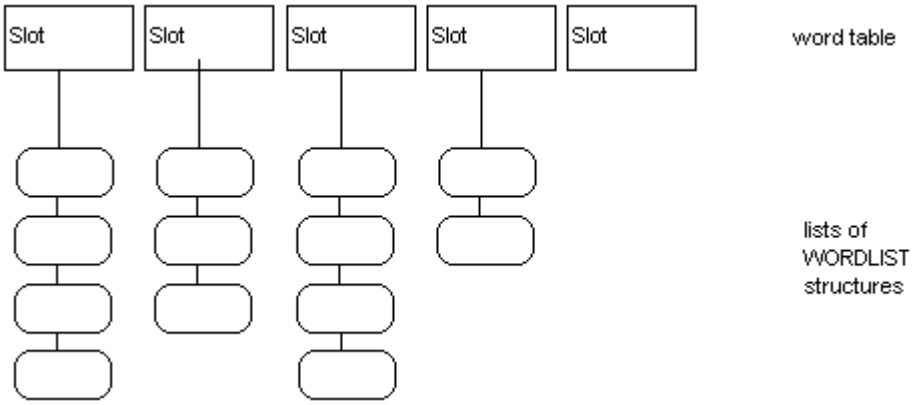
1.27.1 The organization of the table

Now, we have no choice but to start thinking about that “EnterWord” function. All the easy work is done, we have to figure out now, an efficient organization for our word table. We have the following requirements:

1) It should provide a fast access to a word to see if a given sequence of characters is there already.

2) It should not use a lot of memory and be simple to use.

The best choice is the hash table. We use a hash table to hold all the words, and before entering something into our hash table we look if is in there already. Conceptually, we use the following structure:



Our word table is a sequence of lists of words. Each list is longer or shorter, depending on the hash function that we use and how good our hash function randomizes the input. If we use a table of 65535 positions (slots) and a good hash algorithm we divide the access time by 65535, not bad.

To enter something into our table we hash the word into an integer, and we index the slot in the table. We then compare the word with each one of the words in the list of words at that slot. If we found it, we do nothing else than increment the count of the word. If we do not find it, we add the word at the start of that slot.

Note that this requires that we define a structure to hold each word and its associated count. Since all the words are in a linked list, we could use the following structure, borrowing from the linked list representation discussed above:

```
typedef struct _WordList {
     int Count;
     struct _WordList *Next;
     char Word[];
} WORDLIST;
```

We have an integer that holds the number of times this word appears in the text, a pointer to the next word in the list, and an unspecified number of characters just following that pointer. This is a variable sized structure, since each word can hold more or less characters. Note that variable sized structures must have only one “flexible” member and it must be at the end of the definition.

Our “EnterWord” function can look like this:

```
void EnterWord(char *word)
{
     int h = hash(word); // Get the hash code for this word
     WORDLIST *wl = WordTable[h]; // Index the list at that slot
     while (wl) { // Go through the list
          if (!strcmp(wl->Word,word)) {
               wl->Count++; // Word is already in the table.
               return;      // increment the count and return
          }
          wl = wl->Next;   // Go to the next item in the list
     }
     // Here we have a new word, since it wasn't in the table.
     // Add it to the table now
     wl = NewWordList(word);
     wl->Next = WordTable[h];
     WordTable[h] = wl;
}
```

What would be a good hash function for this application?

This is a tutorial, so we keep things simple. Here is a very simple hash function:

```
int hash(char *word)
{
     int h = 0;
     while (*word) {
          h += *word;
          word++;
     }
     return h & 0xffff;
}
```

We just add up our characters. If we get a hash value of more than 65535 (the size of our table), we just take the lower 16 bits of the hash value. Easy isn't it?

We declare our word table now, like this:

```
WORDLIST *WordTable[0xffff+1];
```

1.27.2 Memory organization

Now we write the constructor for our word list structure. It should get more memory from the system to hold the new structure, and initialize its fields.

```
WORDLIST *NewWordList(char *word)
{
     int len = strlen(word);
     WORDLIST *result = more_memory(sizeof(WORDLIST)+len+1);
     result->Count = 1;
     strcpy(result->Word,word);
     return result;
}
```

We allocate more memory to hold the structure, the characters in the word, and the terminating zero. Then we copy the characters from the buffer we got, set the count to 1 since we have seen this word at least once, and return the result. Note that we do not test for failure. We rely on "more_memory" to stop the program if there isn't any more memory left, since the program can't go on if we have exhausted the machine resources.

Under windows, the implementation of the standard "malloc" function is very slow. To avoid calling "malloc" too often, we devise an intermediate structure that will hold chunks of memory, calling malloc only when each chunk is exhausted.

```
typedef struct memory {
     int used;
     int size;
     char *memory;
} MEMORY;
```

Now, we write our memory allocator:

```
#define MEM_ALLOC_SIZE 0xffff
int memoryused = 0;
void *more_memory(int siz)
{
     static MEMORY *mem;
     void *result;
     if (mem == NULL || mem->used+siz >= mem->size) {
          mem = malloc(sizeof(mem)+MEM_ALLOC_SIZE);
          if (mem == NULL) {
               fprintf(stderr,"No more memory at line %d\n",line);
               exit(1);
```

```
            }
            mem->used = 0;
            memoryused += MEM_ALLOC_SIZE;
            mem->size = MEM_ALLOC_SIZE;
        }
        result = mem->memory+mem->used;
        mem->used += siz;
        memset(result,siz,0);
        memoryused += siz;
        return result;
    }
```

We use a static pointer to a MEMORY structure to hold the location of the current memory chunk being used. Since it is static it will be initialized to NULL automatically by the compiler and will keep its value from one call to the next. We test before using it, if the chunk has enough room for the given memory size we want to allocate or if it is NULL, i.e. this is the very first word we are entering. If either of those if true, we allocate a new chunk and initialize its fields.[70]

Otherwise we have some room in our current chunk. We increase our counters and return a pointer to the position within the "memory" field where this chunk starts. We clean the memory with zeroes before returning it to the calling function.

Note that we do not keep any trace of the memory we have allocated so it will be impossible to free it after we use it. This is not so bad because the operating system will free the memory after this program exists. The downside of this implementation is that we can't use this program within another one that would call our word counting routine. We have a memory leak "built-in" into our software.

A way out of this is very easy though. We could just convert our `mem` structures into a linked list, and free the memory at the end of the program.

1.27.3 Displaying the results

After all this work, we have a program that will compile when run, but is missing the essential part: showing the results to the user. Let's fix this.

We need to sort the words by frequency, and display the results. We build a table of pointers to word-list structures and sort it.

But... to know how big our table should be, we need to know how many words we have entered. This can be done in two ways: Either count the number of words in the table when building the report, or count the words as we enter them.

Obviously, the second solution is simpler and requires much less effort. We just declare a global integer variable that will hold the number of words entered into the table so far:

```
int words = 0;
```

We increment this counter when we enter a new word, i.e. in the function NewWordList. [71]

We will need a comparison function for the qsort library function too.

```
int comparewords(const void *w1,const void *w2)
{
    WORDLIST *pw1 = *(WORDLIST **)w1,*pw2 = *(WORDLIST **)w2;
```

70. Note that we allocate MEM_ALLOC_SIZE bytes. If we want to change to more or less bytes, we just change the #define line and we are done with the change.

```
        if (pw1->Count == pw2->Count)
             return strcmp(pw1->Word,pw2->Word);
        return pw1->Count - pw2->Count;
   }
```

Note that we have implemented secondary sort key. If the counts are the same, we sort by alphabetical order within a same count.

```
   void DoReports(char *filename)
   {
        int i;
        int idx = 0; // Index into the resulting table

        // Print file name and number of words
        printf("%s: %d different words.\n",filename,words);

        // allocate the word-list pointer table
        WORDLIST **tab = more_memory(words*sizeof(WORDLIST *));

        // Go through the entire hash table
        for (i=0; i< sizeof(WordTable)/sizeof(WordTable[0]);i++) {
             WORDLIST *wl = WordTable[i];

             while (wl) {
                  // look at the list at this slot
                  tab[idx] = wl;
                  wl = wl->Next;
                  idx++;
                  if (idx >= words && wl) {
                    fprintf(stderr,"program error\n");
                    exit(1);
                  }
             }
        }
        // Sort the table
        qsort(tab,words,sizeof(WORDLIST *),comparewords);
        // Print the results
        for (i=0; i< words;i++) {
             printf("%s %5d\n",tab[i]->Word,tab[i]->Count);
        }
   }
```

We start by printing the name of the file and the number of different words found. Then, we go through our hash table, adding a pointer to the word list structure at each non-empty slot.

Note that we test for overflow of the allocated table. Since we increment the counter each time that we add a word, it would be very surprising that the count didn't match with the number of items in the table. But it is better to verify this.

After filling our table for the qsort call, we call it, and then we just print the results.

---

71. Global variables like this should be used with care. Overuse of global variables leads to problems when the application grows, for instance in multi-threaded applications. When you got a lot of global variables accessed from many points of the program it becomes impossible to use threads because the danger that two threads access the same global variable at a time.
Another problem is that our global is not static, but visible through the whole program. If somewhere else somebody writes a function called "words" we are doomed. In this case and for this example the global variable solution is easier, but not as a general solution.

1.27.4 Code review

Now that we have a bare skeleton of our program up and running, let's come back to it with a critical eye.

For instance look at our "`isWordStart`" function. We have:

```
int isWordStart(int c)
{
     if (c == '_')
          return 1;
     if (c >= 'a' && c <= 'z')
          return 1;
     if (c >= 'A' && c <= 'Z')
          return 1;
     return 0;
}
```

A look in the "ctype.h" system header file tells us that for classifying characters we have a lot of efficient functions. We can reduce all this code to:

```
int isWordStart(int c)
{
     return c == '_' || isalpha(c);
}
```

The "isalpha" function will return 1 if the character is one of the uppercase or lowercase alphabetic characters. Always use library functions instead of writing your own. The "isalpha" function does not make any jumps like we do, but indexes a table of property bits. Much faster.

And what about error checking? Remember, we just open the file given in the command line without any test of validly. We have to fix this.

Another useful feature would be to be able to report a line number associated with our file, instead of just an error message that leaves to the user the huge task of finding where is the offending part of the input file that makes our program crash. This is not very complex. We just count the new line characters.

The output of our program is far from perfect. It would be better if we justify the columns. To do that, we have to just count the length of each word and keep a counter to the longest word we find. Another nice thing to have would be a count of how many words with 1 character we find, how many with two, etc.

In the Appendix you will find the complete source code containing the answers to this problem.

1.28 Time and Date functions

The C library offers a lot of functions for working with dates and time. The first of them is the *time* function that returns the number of seconds that have passed since January first 1970, at midnight. [72]

Several structures are defined that hold time information. The most important from them are the "tm" structure and the "timeb" structure.

```
struct tm {
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
```

72. This clock will overflow in something like 2.000 years so be prepared for windows 4.000!

```
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

The fields are self-explanatory. The structure “timeb” is defined in the directory include\sys, as follows:

```
struct timeb {
        time_t time;
        unsigned short pad0;
        unsigned long lpad0;
        unsigned short millitm; // Fraction of a second in ms
        unsigned short pad1;
        unsigned long lpad1;
          // Difference (minutes), moving westward, between UTC and local time
        short timezone;
        unsigned short pad2;
        unsigned long lpad2;
          // Nonzero if daylight savings time is currently in effect for the local time zone.
        short dstflag;
};
```

We show here a small program that displays the different time settings.

```
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include <string.h>

void main()
{
    char tmpbuf[128], ampm[] = "AM";
    time_t ltime;
    struct _timeb tstruct;
    struct tm *today, *gmt, xmas = { 0, 0, 12, 25, 11, 93 };

  /* Display operating system-style date and time. */
    strtime( tmpbuf );
    printf( "OS time:\t\t\t\t%s\n", tmpbuf );
    strdate( tmpbuf );
    printf( "OS date:\t\t\t\t%s\n", tmpbuf );

  /* Get UNIX-style time and display as number and string. */
    time( &ltime );
    printf( "Time in seconds since UTC 1/1/70:\t%ld\n", ltime );
    printf( "UNIX time and date:\t\t\t%s", ctime( &ltime ) );

  /* Display UTC. See note (1) in text */
    gmt = gmtime( &ltime );
    printf( "Coordinated universal time:\t\t%s", asctime( gmt ) );

    /* Convert to time structure and adjust for PM if necessary. */
    today = localtime( &ltime );
    if( today->tm_hour > 12 ) {
   strcpy( ampm, "PM" );
   today->tm_hour -= 12;
    }
```

```
        if( today->tm_hour == 0 )  /* Adjust if midnight hour. */
       today->tm_hour = 12;

        /* See note (2) in text */
        printf( "12-hour time:\t\t\t\t%.8s %s\n",
           asctime( today ) + 11, ampm );

        /* Print additional time information. */
        ftime( &tstruct );
        printf( "Plus milliseconds:\t\t\t%u\n", tstruct.millitm );
        printf( "Zone difference in seconds from UTC:\t%u\n",
                 tstruct.timezone );
        printf( "Time zone name:\t\t\t\t%s\n", _tzname[0] );
        printf( "Daylight savings:\t\t\t%s\n", // See note (3) in text
                 tstruct.dstflag ? "YES" : "NO" );

    /* Make time for noon on Christmas, 1993. */
        if( mktime( &xmas ) != (time_t)-1 )
      printf( "Christmas\t\t\t\t%s\n", asctime( &xmas ) );

    /* Use time structure to build a customized time string. */
        today = localtime( &ltime );

    /* Use strftime to build a customized time string. */
        strftime( tmpbuf, 128,
             "Today is %A, day %d of the month of %B in the year %Y.\n",
             today );
        printf( tmpbuf );
}
```

We use this opportunity for introducing new C constructs.

We see the function call gmtime(&ltime);. What does this mean? The function gmtime requires a pointer to a variable of type time_t. We do not have a pointer, so we make one “on the fly” by using the “address-of” operator.

The printf statement uses pointer addition to skip the first 11 characters of the result of asctime. That function returns a pointer to a character string. To skip characters we just add up a constant (11) to that pointer, effectively skipping those characters. Since we want to display the 8 following characters only, we pass a width argument to the %s directive of printf. As you know, “%s” is a directive that instructs printf to display a character string. Since we give it a maximum width of 8, only the first 8 chars will be displayed.

We see here the construct (expr) ? val1 : val2; This construct evaluates first the expression, in this case “tstruct.dstflag”. If the value is different than zero, the return value of the expression will be the first value, in this case the character string “YES”. If the expression evaluates to zero, the second value will be chosen, in this case the character string “NO”. The result of this is passed to printf as an argument.

The Windows system too has a lot of time-related functions. Here is a handy list of the most important. Note that file times are kept using 64 bits in modern versions of windows, i.e. the numbers represent the number of 100 nanosecond intervals since January first, 1601. [73]

73. I hope this happens to you too…

To convert from the ANSI C time format to the Windows time format you can use the follow-

Function	*Purpose*
CompareFileTime	Compares two 64-bit file times
DosDateTimeToFileTime	Converts MS-DOS date and time values to a 64-bit file time.
FileTimeToDosDateTime	Converts a 64-bit file time to MS-DOS date and time values.
FileTimeToLocalFileTime	Converts a file time based on the Coordinated Universal Time (UTC) to a local file time.
FileTimeToSystemTime	Converts a 64-bit file time to system time format
GetFileTime	Retrieves the date and time that a file was created, last accessed, and last modified.
GetLocalTime	Retrieves the current local date and time.
GetSystemTime	Retrieves the current system date and time.
GetSystemTimeAdjustment	Determines whether the system is applying periodic time adjustments to its time-of-day clock at each clock interrupt, along with the value and period of any such adjustments.
GetSystemTimeAsFileTime	Obtains the current system date and time. The information is in Coordinated Universal Time (UTC) format.
GetTickCount	Retrieves the number of milliseconds that have elapsed since the system was started. It is limited to the resolution of the system timer.
GetTimeZoneInformation	Retrieves the current time-zone parameters. These parameters control the translations between Coordinated Universal Time (UTC) and local time.
LocalFileTimeToFileTime	Converts a local file time to a file time based on the Coordinated Universal Time (UTC).
SetFileTime	Sets the date and time that a file was created, last accessed, or last modified.
SetLocalTime	Sets the current local time and date.
SetSystemTime	Sets the current system time and date. The system time is expressed in Coordinated Universal Time (UTC).
SetSystemTimeAdjustment	Tells the system to enable or disable periodic time adjustments to its time of day clock.
SetTimeZoneInformation	Sets the current time-zone parameters. These parameters control translations from Coordinated Universal Time (UTC) to local time.
SystemTimeToFileTime	Converts a system time to a file time.
SystemTimeToTzSpecificLocalTime	Converts a Coordinated Universal Time (UTC) to a specified time zone's corresponding local time.

ing function:

```
#include <winbase.h>
#include <winnt.h>
#include <time.h>

void UnixTimeToFileTime(time_t t, LPFILETIME pft)
{
  long long ll;

  ll = Int32x32To64(t, 10000000) + 116444736000000000;
  pft->dwLowDateTime = (DWORD)ll;
```

```
  pft->dwHighDateTime = ll >> 32;
}
```

Once the UNIX time is converted to a FILETIME structure, other Win32 time formats can be easily obtained by using Win32 functions such as `FileTimeToSystemTime()` and `FileTimeToDosDateTime()`.

```
void UnixTimeToSystemTime(time_t t, LPSYSTEMTIME pst)
{
  FILETIME ft;

  UnixTimeToFileTime(t, &ft);
  FileTimeToSystemTime(&ft, pst);
}
```

1.29 Using structures (continued)

C allows implementation of any type of structure. Here is a description of some simple ones so you get an idea of how they can be built and used.

1.29.1 Lists

Lists are members of a more general type of objects called sequences, i.e. objects that have a natural order. You can go from a given list member to the next element, or to the previous one.

We have several types of lists, the simplest being the single-linked list, where each member contains a pointer to the next element, or NULL, if there isn't any. We can implement this structure in C like this:

```
typedef struct _list {
     struct _list *Next; // Pointer to next element
     void *Data;// Pointer to the data element
} LIST;
```

We can use a fixed anchor as the head of the list, for instance a global variable containing a pointer to the list start.

```
LIST *Root;
```

We define the following function to add an element to the list:

```
LIST *Append(LIST **pListRoot, void *data)
{
     LIST *rvp = *pListRoot;

     if (rvp == NULL) { // is the list empty?
          // Yes. Allocate memory
          *pListRoot = rvp = GC_malloc(sizeof(LIST));
     }
     else { // find the last element
          while (rvp->Next)
               rvp = rvp->Next;
          // Add an element at the end of the list
          rvp->Next = GC_malloc(sizeof(LIST));
          rvp = rvp->Next;
     }
     // initialize the new element
     rvp->Next = NULL;
     rvp->Data = data;
     return rvp;
}
```

This function receives a pointer to a pointer to the start of the list.

Why?

If the list is empty, it needs to modify the pointer to the start of the list. We would normally call this function with:

```
newElement = Append(&Root,data);
```

Note that loop:

```
while (rvp->Next)
     rvp = rvp->Next;
```

This means that as long as the Next pointer is not NULL, we position our roving pointer (hence the name "rvp") to the next element and repeat the test. We suppose obviously that the last element of the list contains a NULL "Next" pointer. We ensure that this condition is met by initializing the `rvp->Next` field to NULL when we initialize the new element.

To access a list of n elements, we need in average to access n/2 elements.

Other functions are surely necessary. Let's see how a function that returns the nth member of a list would look like:

```
LIST *ListNth(LIST *list, int n)
{
     while (list && n-- > 0)
          list = list->Next;
     return list;
}
```

Note that this function finds the nth element beginning with the given element, which may or may not be equal to the root of the list. If there isn't any nth element, this function returns NULL.

If this function is given a negative n, it will return the same element that was passed to it. Given a NULL list pointer it will return NULL.

Other functions are necessary. Let's look at Insert.

```
LIST *Insert(LIST *list,LIST *element)
{
     LIST *tmp;

     if (list == NULL)
          return NULL;
     if (list == element)
          return list;
     tmp = list->Next;
     list->Next = element;
     if (element) {
          element->Next = tmp;
     }
     return list;
}
```

We test for different error conditions. The first and most obvious is that "list" is NULL. We just return NULL. If we are asked to insert the same element to itself, i.e. "list" and "element" are the same object, their addresses are identical, we refuse. This is an error in most cases, but maybe you would need a circular element list of one element. In that case just eliminate this test.

Note that `Insert(list, NULL);` will effectively cut the list at the given element, since all elements after the given one would be inaccessible.

Many other functions are possible and surely necessary. They are not very difficult to write, the data structure is quite simple.

Double linked lists have two pointers, hence their name: a Next pointer, and a Previous pointer, that points to the preceding list element.

Our data structure would look like this:

```
typedef struct _dlList {
     struct _dlList *Next;
     struct _dlList *Previous;
     void *data;
} DLLIST;
```

Our "Append" function above would look like: (new material in bold)

```
LIST *AppendDl(DLLIST **pListRoot, void *data)
{
     DLLIST *rvp = *pListRoot;

     // is the list empty?
     if (rvp == NULL) {
          // Yes. Allocate memory
          *pListRoot = rvp = GC_malloc(sizeof(DLLIST));
          rvp->Previous = NULL;
     }
     else {
          // find the last element
          while (rvp->Next)
               rvp = rvp->Next;
          // Add an element at the end of the list
          rvp->Next = GC_malloc(sizeof(DLLIST));
          rvp->Next->Previous = rvp;
          rvp = rvp->Next;
     }
     // initialize the new element
     rvp->Next = NULL;
     rvp->Data = data;
     return rvp;
}
```

The Insert function would need some changes too:

```
LIST *Insert(LIST *list,LIST *element)
{
     LIST *tmp;

     if (list == NULL)
          return NULL;
     if (list == element)
          return list;
     tmp = list->Next;
     list->Next = element;
     if (element) {
          element->Next = tmp;
          element->Previous = list;
          if (tmp)
               tmp->Previous = element;
     }
     return list;
}
```

Note that we can implement a Previous function with single linked lists too. Given a pointer to the start of the list and an element of it, we can write a Previous function like this:

```
LIST *Previous(LIST *root, LIST *element)
{
     if (root == NULL )
          return NULL;
     while (root && root->Next != element)
          root = root->Next;
     return root;
}
```

Circular lists are useful too. We keep a pointer to a special member of the list to avoid infinite loops. In general we stop when we arrive at the head of the list. Wedit uses this data structure to implement a circular double linked list of text lines. In an editor, reaching the previous line by starting at the first line and searching and searching would be too slow. Wedit needs a double linked list, and a circular list makes an operation like wrapping around easier when searching.

1.29.2 Hash tables

A hash table is a table of lists. Each element in a hash table is the head of a list of element that happen to have the same hash code, or key.

To add an element into a hash table we construct from the data stored in the element a number that is specific to the data. For instance we can construct a number from character strings by just adding the characters in the string.

This number is truncated module the number of elements in the table, and used to index the hash table. We find at that slot the head of a list of strings (or other data) that maps to the same key modulus the size of the table.

To make things more specific, let's say we want a hash table of 128 elements, which will store list of strings that have the same key.

Suppose then, we have the string "abc". We add the ASCII value of 'a' + 'b' + 'c' and we obtain 97+98+99 = 294. Since we have only 128 positions in our table, we divide by 128, giving 2 and a rest of 38. We use the rest, and use the 38^{th} position in our table.

This position should contain a list of character strings that all map to the 38^{th} position. For instance, the character string "aE": (97+69 = 166, mod 128 gives 38). Since we keep at each position a single linked list of strings, we have to search that list to find if the string that is being added or looked for exists.

A sketch of an implementation of hash tables looks like this:

```
#define HASHELEMENTS 128
typedef struct hashTable {
     int (*hashfn)(char *string);
     LIST *Table[HASHELEMENTS];
} HASH_TABLE;
```

We use a pointer to the hash function so that we can change the hash function easily. We build a hash table with a function.

```
HASH_TABLE newHashTable(int (*hashfn)(char *))
{
     HASH_TABLE *result = GC_malloc(sizeof(HASH_TABLE));
     result->hashfn = hashfn;
     return result;
}
```

To add an element we write:

```
LIST *HashTableInsert(HASH_TABLE *table, char *str)
```

```
{
     int h = (table->hashfn)(str);
     LIST *slotp = table->Table[h % HASHELEMENTS];

     while (slotp) {
          if (!strcmp(str,(char *)slotp->data)) {
               return slotp;
          }
          slotp = slotp->Next;
     }
     return Append(&table->Table[h % HASHELEMENTS],element);
}
```

All those casts are necessary because we use our generic list implementation with a void pointer. If we would modify our list definition to use a char * instead, they wouldn't be necessary.

We first call the hash function that returns an integer. We use that integer to index the table in our hash table structure, getting the head of a list of strings that have the same hash code. We go through the list, to ensure that there isn't already a string with the same contents. If we find the string we return it. If we do not find it, we append to that list our string

The great advantage of hash tables over lists is that if our hash function is a good one, i.e. one that returns a smooth spread for the string values, we will in average need only n/128 comparisons, n being the number of elements in the table. This is an improvement over two orders of magnitude over normal lists.

1.30 A closer look at the pre-processor

The first phase of the compilation process is the “pre-processing” phase. This consists of scanning in the program text all the *preprocessor directives*, i.e. lines that begin with a “#” character, and executing the instructions found in there before presenting the program text to the compiler.

We will interest us with just two of those instructions. The first one is the “#define” directive, that instructs the software to replace a macro by its equivalent. We have two types of macros:

Parameterless. For example:

```
#define PI 3.1415
```

Following this instruction, the preprocessor will replace all instances of the identifier PI with the text “3.11415”.

Macros with arguments. For instance:

```
#define s2(a,b) ( (a*a + b*b) /2.0)
```

When the preprocessor finds a sequence like:

```
s2(x,y)
```

It will replace it with:

```
(x*x + y*y)/2.0 )
```

The problem with that macro is that when the preprocessor finds a statement like:

```
s2(x+6.0,y-4.8);
```

It will produce :

```
(x+6.0*x+6.0 + y+6.0*y+6.0) /2.0 )
```

What will calculate completely another value:

```
(7.0*x + 7.0*y + 12.0)/2.0
```

To avoid this kind of bad surprises, it is better to enclose each argument within parentheses each time it is used:

```
#define s2(a,b) (((a)*(a) + (b)*(b))/2.0)
```

This corrects the above problem but we see immediately that the legibility of the macros suffers… quite complicated to grasp with all those redundant parentheses around.

Another problem arises when you want that the macro resembles exactly a function call and you have to include a block of statements within the body of the macro, for instance to declare a temporary variable.

```
#define s2(x,y) { int temp = x*x+y*y; x=temp+y *(temp+6);}
```

If you call it like this:

```
if (x < y) s2(x,y);
else
x = 0;
```

This will be expanded to:

```
if (x < y) { int temp = x*x+y*y; x=temp+y *(temp+6);} ;
else
x = 0;
```

This will provoke a syntax error.

To avoid this problem, you can use the do... while statement, that consumes the semicolon:

```
#define s2(x,y) do { int temp = x*x+y*y; x=temp+y *(temp+6);} \
```

```
while(0)
```

Note the \ that continues this long line, and the absence of a semicolon at the end of the macro.

An "#undef" statement can undo the definition of a symbol. For instance

```
#undef PI
```

will erase from the pre-processor tables the PI definition above. After that statement the identifier PI will be ignored by the preprocessor and passed through to the compiler.

The second form of pre-processor instructions that is important to know is the

```
#if (expression)
… program text …
#else
… program text …
#endif
or the pair
#ifdef (symbol)
#else
#endif
```

When the preprocessor encounters this kind of directives, it evaluates the expression or looks up in its tables to see if the symbol is defined. If it is, the "if" part evaluates to true, and the text until the #else or the #endif is copied to the output being prepared to the compiler. If it is NOT true, then the preprocessor ignores all text until it finds the #else or the #endif. This allows you to disable big portions of your program just with a simple expression like:

```
#if 0
…
#endif
```

This is useful for allowing/disabling portions of your program according to compile time parameters. For instance, lcc-win32 defines the macro `__LCC__`. If you want to code something only for this compiler, you write:

```
#ifdef __LCC__
… statements …
#endif
```

Note that there is no way to decide if the expression:

```
SomeFn(foo);
```

Is a function call to SomeFn, or is a macro call to SomeFn. The only way to know is to read the source code. This is widely used. For instance, when you decide to add a parameter to CreateWindow function, without breaking the millions of lines that call that API with an already fixed number of parameters you do:

```
#define CreateWindow(a,b, … ) CreateWindowEx(0,a,b,…)
```

This means that all calls to CreateWindow API are replaced with a call to another routine that receives a zero as the new argument's value.

It is quite instructive to see what the preprocessor produces. You can obtain the output of the preprocessor by invoking lcc with the –E option. This will create a file with the extension `.i` (intermediate file) in the compilation directory. That file contains the output of the preprocessor. For instance, if you compile hello.c you will obtain hello.i.

1.30.1 Preprocessor commands

The preprocessor receives its commands with lines beginning with the special character "#". This lines can contain:

1) Macros
2) Conditional compilation instructions
3) Pragma instructions
4) The "##" operator
5) Line instructions

1.30.1.1 Preprocessor macros

The **#define** command has two forms depending on whether a left parenthesis appears immediately after the name of the macro. The first form, without parenthesis is simply the substitution of text. An example can be:

```
#define MAXBUFFERSIZE   8192
```

This means that whenever the preprocessor find the identifier MAXBUFFERSIZE in the program text, it will replace it with the character string "8192"

The second form of the preprocessor macros is the following:

```
#define add(a,b) ((a)+(b))
```

This means that when the preprocessor finds the sequence:

```
int n = add(7,b);
```

it will replace this with:

```
int n = ((7)+(b));
```

Note that the left parenthesis MUST be immediately after the name of the macro without any white space (blanks, tabs) in between. It is often said that white space doesn't change the meaning of a C program but that is not always true, as you can see now. If there is white space between the macro name and the left parenthesis the preprocessor will consider that this is a macro with no arguments whose body starts with a left parentheses!

If you want to delete the macro from the preprocessor table you use the

```
#undef <macro name>
```

command. This is useful for avoiding name clashes. Remember that when you define a macro, the name of the macro will take precedence before everything else since the compiler will not even see the identifier that holds the macro. This can be the origin of strange bugs like:

```
int fn(int a)
{
// some code
}
```

If you have the idea of defining a macro like this

```
#define fn 7987
```

the definition above will be transformed in

```
int 7987(int a)
{
}
```

not exactly what you would expect. This can be avoided by **#undef**ining the macros that you fear could clash with other identifiers in the program.

1.30.1.2 Conditional compilation

Very often we need to write some kind of code in a special situation, and some other kind in another situation. For instance we would like to call the function "initUnix()" when we are in a UNIX environment, and do NOT call that function when in other environments. Besides we would like to erase all UNIX related instructions when compiling for a windows platform and all windows related stuff when compiling for Unix.

This is achieved with the preprocessor

```
#ifdef UNIX
lines for the Unix system
#else
lines for other (non-unix) systems.
#endif
```

This means:

If the preprocessor symbol "UNIX" is defined, include the first set of lines, else include the second set of lines.

There are more sophisticated usages with the #elif directive:

```
#ifdef UNIX
                Unix stuff
#elif MACINTOSH
                Mac stuff
#elif WIN32
                Windows stuff
#else
#error "Unknown system!"
#endif
```

Note the **#error** directive. This directive just prints an error message and compilation fails.

The lines that are in the inactive parts of the code will be completely ignored, except (of course) preprocessor directives that tell the system when to stop. Note too that the flow of the program becomes much difficult to follow. This feature of the preprocessor can be abused, to the point that is very difficult to see which code is being actually compiled and which is not. The IDE of lcc-win32 provides an option for preprocessing a file and showing all inactive lines in grey color. Go to "Utils" the choose "Show #ifdefs" in the main menu.

Note: You can easily comment out a series of lines of program text when you enclose them in pairs of

```
#if 0
// lines to be commented out
#endif
```

1.30.1.3 The pragma directive

This directive is compiler specific, and means that what follows the #pragma is dependent on which compiler is running. The pragmas that lcc-wi32 uses are defined in the documentation. In general pragmas are concerned with implementation specific details, and are an advanced topic.

1.30.1.4 The ## operator

This operator allows you to concatenate two tokens:

```
#define join(a,b) (a##b)
a = join(anne,bob)
```

When preprocessed this will produce:

```
a = (annebob)
```

This is useful for constructing variable names automatically and other more or less obscure hacks.

1.30.1.5 The # operator

This operator converts the given token into a character string. It can be used only within the body of a macro definition. After defining a macro like this:

```
#define toString(Token) #Token
```

an expression like

```
toString(MyToken)
```

will be translated after preprocessing into:

```
"MyToken"
```

An example of its use is the following situation. We have a structure of an integer error code and a character field containing the description.

```
static struct table {
        unsigned int code;
        unsigned char *desc;
} hresultTab;
```

Then, we have a lot of error codes, defined with the preprocessor: E_UNEXPECTED, E_NOTIMPL, E_INVALIDARG, etc. We can build a table with:

```
hresultTab Table[] = {
     {E_UNEXPECTED,"E_UNEXPECTED",}
     {E_NOTIMPL,"E_NOTIMPL",}
     ... etc
};
```

This is tedious, and there is a big probablity of making a typing mistake. A more intelligent way is:

```
#define CASE(a) {a,#a},
```

Now we can build our table like this:

```
hresultTab Table[] = {
     CASE(E_UNEXPECTED)
     CASE(E_NOTIMPL)
     ...
};
```

1.30.2 Things to watch when using the preprocessor

1) One of the most common errors is to add a semi colon after the macro:

```
#define add(a,b) a+b;
```

When expanded, this macro will add a semicolon into the text, with the consequence of a cascade of syntax errors with apparently no reason.

2) Watch for side effects within macros. A macro invocation is similar to a function call, with the big difference that the arguments of the function call is evaluated once but in the macro can be evaluated several times. For instance we have the macro "square"

```
#define square(a) (a*a)
```

If we use it like this:

```
b = square(a++);
```

After expansion this will be converted into:

```
b = (a++)*(a++);
```

and the variable *a* will be incremented twice.

1.31 Using function pointers

A very common programming problem is to recursively explore a certain part of the file system to find files that have a certain name, for instance you want to know all the files with the ".c" extension in a directory and in all subdirectories.

To build such a utility you can do:

1) Build a list or table containing each file found, and return those results to the user.

2) For each file that is found, you call a user provided function that will receive the name of the file found. The user decides what does he/she want to do with the file.

Note that solution 2 includes solution 1, since the user can write a function that builds the list or table in the format he wants, instead of in a predefined format. Besides, there are *many* options as to what information should be provided to the user. Is he interested in the size of the file? Or in the date? Who knows. You can't know it in advance, and the most flexible solution is the best.

We can implement this by using a *function pointer* that will be called by the scanning function each time a file is found. We define

```
typedef int (*callback)(char *);
```

This means, "a function pointer called callback that points to a function that returns an int and receives a char * as its argument". This function pointer will be passed to our scanning function and will return non-zero (scanning should be continued) or zero (scanning should stop).

Here is a possible implementation of this concept:

```
#include <stdio.h>
#include <windows.h>
#include <direct.h>
// Here is our callback definition
typedef int(*callback)(char *);
// This function has two phases. In the first, we scan for normal files and ignore any
// directories that we find. For each file that matches we call the given function pointer.
// The input, the char * "spec" argument should be a character string like "*.c" or "*.h".
// If several specifications are needed, they should be separated by the ';' semi colon char.
// For instance we can use "*.c;*.h;*.asm" to find all files that match any of those
// file types. The second argument should be a function that will be called at each file
// found.
int ScanFiles(char *spec,callback fn)
{
     char *p,*q; // Used to parse the specs
     char dir[MAX_PATH]; // Contains the starting directory
     char fullname[MAX_PATH];// will be passed to the function
     HANDLE hdir;
     HANDLE h;
     WIN32_FIND_DATA dirdata;
     WIN32_FIND_DATA data;

     // Get the current directory so that we can always come back to it after calling
     // recursively this function in another dir.
     memset(dir,0,sizeof(dir));
     getcwd(dir,sizeof(dir)-1);
     // This variable holds the current specification we are using
     q = spec;
     // First pass. We scan here only normal files, looping for each of the specifications
     // separated by ';'
     do {
          // Find the first specification
```

```
            p = strchr(q,';');
            // Cut the specification at the separator char.
            if (p)
                 *p = 0;
            h = FindFirstFile(q,&data);
            if (h != INVALID_HANDLE_VALUE) {
                 do {
                   if (!(data.dwFileAttributes &
                           FILE_ATTRIBUTE_DIRECTORY)) {
                      // We have found a matching file. Call the user's function.
                      sprintf(fullname,"%s\\%s",dir,data.cFileName);
                      if (!fn(fullname))
                           return 0;
                   }
                 } while (FindNextFile(h,&data));
                 FindClose(h);
            }
            // Restore the input specification. It would be surprising for the user of this
            // application that we destroyed the character string that was passed to
            // this function.
            if (p)
                 *p++ = ';';
            // Advance q to the next specification
            q = p;
       } while (q);
       // OK. We have done all the files in this directory. Now look if there are any
       // subdirectories in it, and if we found any, recurse.
       hdir = FindFirstFile("*.*",&dirdata);
       if (hdir != INVALID_HANDLE_VALUE) {
            do {
                 if (dirdata.dwFileAttributes &
                           FILE_ATTRIBUTE_DIRECTORY) {
                   // This is a directory entry. Ignore the "." and ".." entries.
                   if (! (dirdata.cFileName[0] == '.' &&
                        (dirdata.cFileName[1] == 0 ||
                           dirdata.cFileName[1] == '.'))) {
                      // We change the current dir to the subdir and recurse
                      chdir(dirdata.cFileName);
                      ScanFiles(spec,fn);
                      // Restore current directory to the former one
                      chdir(dir);
                   }
                 }
            } while (FindNextFile(hdir,&dirdata));
            FindClose(hdir);
       }
       return 1;
  }
```

This function above could be used in a program like this:

```
static int files; // used to count the number of files seen
// This is the callback function. It will print the name of the file and increment a counter.
int printfile(char *fname)
{
     printf("%s\n",fname);
     files++;
     return 1;
}
// main expects a specification, and possibly a starting directory. If no starting directory
// is given, it will default to the current directory.
```

```
// Note that many error checks are absent to simplify the code. No validation is done to the
// result of the chdir function, for instance.
int main(int argc,char *argv[])
{
     char spec[MAX_PATH];
     char startdir[MAX_PATH];

     if (argc == 1) {
          printf("scan files expects a file spec\n");
          return 1;
     }
     memset(startdir,0,sizeof(startdir));
     memset(spec,0,sizeof(spec));
     strncpy(spec,argv[1],sizeof(spec)-1);
     if (argc > 2) {
          strcpy(startdir,argv[2]);
     }
     if (startdir[0] == 0) {
          getcwd(startdir,sizeof(startdir)-1);
          chdir(startdir);
     }
     files = 0;
     ScanFiles(spec,printfile);
     printf("%d files\n",files);
     return 0;
}
```

What is interesting about this solution, is that we use no intermediate memory to hold the results. If we have a lot of files, the size of the resulting list or table would be significant. If an application doesn't need this, it doesn't have to pay for the extra overhead.

Using the name of the file, the callback function can query any property of the file like the date of creation, the size, the owner, etc. Since the user writes that function there is no need to give several options to filter that information.

One of the big advantages of C is its ability of using function pointers as first class objects that can be passed around, and used as input for other procedures. Without this feature, this application would have been difficult to write and would be a lot less flexible.

Note too that any error in the function pointer argument will provoke a crash, since we do not test for the validity of the received function pointer.

But, let's be clear: the biggest drawback is that the user has to write a function in C. Imagine telling your friends that before they use your program they just write a function in C, compile it, link it with your stuff, etc.

1.31.1 Function pointers as decision tables

Problem:

Write a program to print all numbers from 1 to n where n integer > 0 without using any control flow statements (switch, if, goto, while, for, etc). Numbers can be written in any order.

Solution:

The basic idea is to use a table of function pointers as a decision table. This can be done like this:

```
#include <stdio.h>
#include <stdlib.h>

typedef void (*callback)(int);
```

```
void zero(int);
void greaterZero(int);
// We define a table of two functions, that represent a boolean  decision
// indexed by either zero or one.
callback callbackTable[2] = {
     zero,
     greaterZero
};

void zero(int a)
{
     exit(0); // This terminates recursion
}

void greaterZero(int a)
{
     printf("%d\n",a--);
     callbackTable[a>0](a); // recurse with a-1
}

int main(int argc,char *argv[])
{
     // assume correct arguments, n > 0
     greaterZero(atoi(argv[1]));
     return 0;
}
```

Error checking can be added to the above program in a similar way. This is left as an exercise for the interested reader.

Of course the utility of function tables is not exhausted by this rather artificial example; Decision tables can be used as a replacement of dense switch statements, for instance. When you are very interested in execution speed, this is the fastest way of making multi-way branches.

Instead of writing:

```
switch(a) {
     case 1:
          // some code
     break;
     case 2:
     break;
     ...
}
```

you could index a table of functions, each function containing the code of each case. You would construct a function table similar to the one above, and you would go to the correct "case" function using:

```
Table[a]();
```

The best is, of course, to use the switch syntax and avoid the overhead of a function call. Lcc-win32 allows you to do this with:

```
#pragma density(0)
```

This will modify the threshold parameter in the switch construction in the compiler so that a dense table of pointers is automatically generated for each case and all of the cases that aren't represented. This is dangerous if your code contains things like:

```
     case 1:
     ...
     case 934088:
     ...
```

In this case the table could be a very bad idea. This is a good technique for very dense switch tables only.

There are many other uses of function tables, for instance in object oriented programming each object has a function table with the methods it supports.

Since function pointers can be assigned during the run time, this allows to change the code to be run dynamically, another very useful application.

1.31.1.1 An even shorter solution

There is always someone cleverer than you. Now look at this:

```
#include <stdio.h>
void print(int n)
{
     n && (print(n-1), 1) && printf("%d\n", n);
}

int main(int argc,char*argv[]) {
    print(atoi(argv[1]);
    return 0;
}
```

How does this work?

By using short circuit evaluation. The statement:

```
n && (print(n-1), 1) && printf("%d\n", n);
```

can be read (from left to right) like this:

If n is zero do nothing and return.

The second expression (print(n-1),1) calls print, then yields 1 as its result. This means that the third expression is evaluated, that prints the number.

1.32 Advanced C programming with lcc-win32

Lcc-win32 offers several extensions to the C language described below. This extensions are not part of the C standard, but they are very much like similar features of the C++ language so they should be able to run in other environments as well.

1.32.1 Operator overloading

When you write:

```
int a=6,b=8;
int c = a+b;
```

you are actually calling a specific intrinsic routine of the compiler to perform the addition of two integers. Conceptually, it is like if you were doing:

```
int a=6,b=8;
int c = operator+(a,b);
```

This "operator+" function is inlined by the compiler. The compiler knows about this operation (and several others), and generates the necessary assembly instructions to perform it at run time.

Lcc-win32 allows you to define functions written by you, to take the place of the built-in operators. For instance you can define a structure *complex*, to store complex numbers. Lcc-win32 allows you to write:

```
COMPLEX operator+(COMPLEX A, COMPLEX B)
{
     … Code for complex number addition goes here
}
```

This means that whenever the compiler sees "a+b" and "a" is a `COMPLEX` and "b" is a `COMPLEX`, it will generate a call to the previously defined overloaded operator, instead of complaining about a "syntax error".

This is called in "tech-speak" operator overloading. There are several rules for writing those functions and using this feature. All of them explained in-depth in the user's manual. This short notice is just a pointer, to show you what is possible.

The implementation of this feature is compatible with the C++ language that offers a similar facility.

1.32.1.1 How to use this facility

Operator overloading is a powerful feature but that doesn't mean that you are exempted from thinking before you write your code. Please. OK?

For instance, even it is very tempting to do:

```
typedef struct _date {
     int year, month,day, hour,sec;
} DATE;

DATE operator+(DATE d1,DATE d2) { ... }
DATE operator-(DATE d1, DATE d2) { ... }
...
```

The operator "+" can't be used for dates. There is nothing that can be assigned to 16/July/1970 +23/December/1980. *The operation "+" has no sense for dates*. The same applies to multiplication and division.

Subtraction is a legal operation for dates, but the result type is *not* a date but a dimensionless number that represents a time interval (in days, hours, etc). The only overloaded operator that makes sense then is:

```
int operator-(DATE d1, DATE d2) { ... }
```

Operator overloading is best done for numbers, and similar objects. This facility allows you to implement any kind of numbers you would like to design, it is fully general. Just look at qfloat.h and see what would happen with qfloat being any other number type you want.

Note too, that nothing has changed about C. There are no classes or any other underlying context, and you are free to build the context you wish, as always C has been.

1.32.2 References

References are a special kind of pointers that are always dereferenced when used. When you declare a reference, you must declare immediately the object they point to. There are no invalid references since they can't be assigned. Once a reference is declared and initialized, you can't reassign them to another object.

They are safer pointers than normal pointers, since they are guaranteed correct, unless the object they point to is destroyed, of course. References are initialized with the construct:

```
int a;
int &pa = a;
```

The "pa" variable is a reference to an integer (an "int &"), and it is immediately initialized to point to the integer "a". Note that you do not have to take the address of "a", but just put its name. The compiler takes the address.

Again, here is a short pointer only. A complete description is found in the user manual.

1.32.3 Generic functions

You can declare a function that receives several types of arguments, i.e. a generic function by using the "overloaded" keyword. Suppose a function that receives as arguments a qfloat or a double.

```
int overloaded docalcs(qfloat *pq) { …}

int overloaded docalcs(double *pd) { … }
```

This function can receive either a "qfloat" number or a double number as input. The compiler notices the type of argument passed in the call and arranges for calling the right function.

Notice that you define two internally different functions, and that the decision of which one will be called will be done according to the type of arguments in the call.

It is not possible to declare a function overloaded *afte*r a call to this function is already generated. The following code will NOT work:

```
docals(2.3);
int overloaded docals(double *pd);
```

Here the compiler will signal an error.

1.32.4 Default arguments

Default arguments allow you to simplify the interface of a function by providing arguments that if not present will assume a default value.

For instance:

```
int fn(int a,int b = 78);
```

This declares a function called «fn», that takes one argument and optionally a second. When the second is not given, the compiler will arrange for it being 78. For instance

```
fn(2);
```

is equivalent to

```
fn(2,78);
```

1.32.5 Structured exception handling

1.32.5.1 Why exception handling?

As you know very well, the following program will provoke a trap.

```
#include <stdio.h>
int main(void)
{
     char *p=NULL;

     *p = 0;
     printf(“This will never be reached\n”);
     return 0;
}
```

There is no way you can catch this trap, and try to recover, or, at least, exit the program with an error message.

This means that there isn’t any way for you to prevent your program from failing catastrophically at the smallest error. It suffices to have a bad pointer somewhere and you are doomed. If you use a third party library you have to trust it 100%, meaning that the slightest problem in the library will provoke the end of your application.

The whole application is fragile and very difficult to trust. A small error in ten thousands of lines is almost inevitable, as you may know...

Of course there is already an exception handling mechanism. When a machine trap occurs, windows displays the well known “This application...” message and shuts down the program. Couldn’t the system make something better?

Well, it can do something better, and it does. Structured exception handling within lcc-win32 is built using the general exception handling mechanism of windows. It has been around since the beginning of the win32 API, i.e. 1995.

1.32.5.2 How do I use SEH?

Structured exception handling allows you to build a solution for the above problems. You enclose the possibly buggy code within special markers, like this:

```
#include <stdio.h>
#include <seh.h>
int main(void)
{
     char *p = NULL;
     __try {
          *p = 0;
     }
     __except(EXCEPTION_EXECUTE_HANDLER) {
          printf("Ooops, there was a problem with this program.\n");
          printf("Please call the maintenance team at 0-800-XXX\n");
     }
```

```
        return 0;
    }
```

This will print

```
D:\lcc\mc38\test>test
Ooops, there was a problem with this program.
Please call the maintenance team at 0-800-XXX
```

If we change it to:

```
#include <stdio.h>
#include <seh.h>
int main(void)
{
     char p[10];
     __try {
          *p = 0;
          printf("No exceptions\n");
     }
     __except(EXCEPTION_EXECUTE_HANDLER) {
          printf("Ooops, there was a problem with this program.\n");
          printf("Please call the maintenance team at 0-800-XXX\n");
     }
     return 0;
}
```

This will now print:

```
No exceptions
```

We have three parts here.

The first one is the protected code block, enclosed by `__try { }`. This block can contain code that provokes exceptions. To leave it prematurely you use the `__leave` expression.

The second part is an integer expression that appears between parentheses after the `__except` keyword. This expression should eventually evaluate to one of the three constants defined in the `<seh.h>` header file.

`EXCEPTION_EXECUTE_HANDLER` (1) means that the code in the following expression should be executed. `EXCEPTION_CONTINUE_SEARCH` (0) means that the system should go on looking for another handler. `EXCEPTION_CONTINUE_EXECUTION` (-1) means that the system should attempt to go on at the same point in the program where the exception happened.

In this case we decided that any exception would provoke the execution of the block following the `__except()`.

1.32.5.3 Auxiliary functions

Of course, you are surely keenly interested in the actual exception code. The pseudo-function GetExceptionCode() will return the current exception code. If we change our program to:

```
#include <stdio.h>
#include <seh.h>
int main(void)
{
     char *p=NULL;
     unsigned code;
     __try {
          *p = 0;
          printf("No exceptions\n");
     }
     __except(code=GetExceptionCode(),EXCEPTION_EXECUTE_HANDLER) {
          printf("Ooops, there was a problem with this program.”);
```

```
            printf("Error code: %#x\n",code);
        }
        return 0;
    }
```

This will print:

```
Ooops, there was a problem with this program.
Error code: 0xc0000005
```

We use a comma expression within the except() to catch the value of the exception code into a local variable. The comma expression returns EXCEPTION_EXECUTE_HANDLER as its result, with the side effect of storing the exception code in a local variable.

In other compilers, the GetExceptionCode() function can be called only within an except expression. Lcc-win32 let's you call it anywhere you want, it will always return the exception code of the last exception that was executed, or zero, if there weren't any.

Another useful function is

```
void RaiseException(unsigned code,unsigned flags,
                unsigned nbArgs,unsigned *args);
```

We can use it to call the exception handling mechanism in the same way as a real exception would. Example:

```
#include <stdio.h>
#include <seh.h>
int main(void)
{
    char p[10];
    unsigned code;
    __try {
        *p = 0;
        printf("No exceptions\n");
        RaiseException(0xdeadbeef,0,0,NULL);
    }
    __except(code=GetExceptionCode(),EXCEPTION_EXECUTE_HANDLER) {
        printf("Ooops, there was a problem with this program.\n");
        printf("Error code: %#x\n",code);
    }
    return 0;
}
```

This will print:

```
No exceptions
Ooops, there was a problem with this program.
Error code: 0xceadbeef
```

Note that the system will change the exception code from `0xdeadbeef` to `0xceadbeef.`

Why?

The explanation is in the MSDN site: [74]

I quote:

To make sure that you do not define a code that conflicts with an existing exception code, set the third most significant bit to 1. The four most-significant bits should be set as shown in the following table.

74. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/html/_core_raising_software_exceptions.asp.

Bits	Recommended setting	Description
31-30	11	These two bits describe the basic status of the code: 11 = error, 00 = success, 01 = informational, 10 = warning.
29	1	Client bit. Set to 1 for user-defined codes.
28	0	Reserved bit. (Leave set to 0.)

You can set the first two bits to a setting other than 11 binary if you want, although the error setting is appropriate for most exceptions. The important thing to remember is to set bits 29 and 28 as shown in the previous table.

End quote.

Note that nowhere will be said that the actual code will be changed by the operating system. Just recommendations... But anyway. Follow this schema and you (may) be happy with it.

1.32.5.4 Giving more information

A 32 bit integer can be too little data for describing what happened. Additional data can be channeled through the exception arguments using the RaiseException API.

You can pass up to 15 arguments when you raise an exception. For instance, here we will pass the name of the function that provoked the error in the first parameter:

```
#include <stdio.h>
#include <seh.h>
int fn(void)
{
     unsigned argstable[1];

     argstable[0] = (unsigned)__func__;
     RaiseException(0xe0000001,0,1,argstable);
     return 0; // Not reached
}

int main(void)
{
     unsigned code;
     __try {
          fn();
     }
     __except(code=GetExceptionCode(),EXCEPTION_EXECUTE_HANDLER) {
          EXCEPTION_POINTERS *ex = GetExceptionInformation();
          printf("Error code: %#x\n",code);
          printf("In function %s\n",
               (char *)ex->ExceptionRecord->ExceptionInformation[0]);
     }
     return 0;
}
```

This will print:

```
Error code: 0xe0000001
In function fn
```

The keyword __func__ will be substituted by the name of the current function by the compiler. We put that name in the first parameter of the table. We call then RaiseException() with 1 argument and our table. The exception handling code will get the exception information, and print the first argument.

The function GetExceptionInformation() returns a pointer to a copy of the exception information received by the system by the lcc-win32 runtime. The structure `EXCEPTION_POINTERS` contains two pointers, the first points to an `EXCEPTION_RECORD`, and the second to a `CONTEXT` structure containing the machine registers at the time of the exception. In the example we use the exception record to get to the parameters array, and we print the first one.

Nowhere in the shown code there is a test to verify that the exception code is the one we are expecting. If another exception had happened that has no parameters, or a different set of parameters we would have crashed.

1.32.5.5 Catching stack overflow

The stack space allocated by windows to a program at start is 4K. This stack can grow up to a size of 1MB. If you go beyond this limit you get a stack overflow exception (0xC00000FD). This exception will be catched by the structured exception handling framework however, and at first it looks like a normal exception. Consider this program:

```
#include <windows.h>
#include <seh.h>
#include <stdio.h>
#include <intrinsics.h>
void StackOverflow(int depth)
{
    char blockdata[90000];
    printf("Overflow: %d\n", depth);
    StackOverflow(depth+1);
}

int main(int argc, char* argv[])
{
     for( ; ; ) {
          __try  {
               StackOverflow(0);
          }
          __except(EXCEPTION_EXECUTE_HANDLER) {
               printf("Exception handler %lX\n", GetExceptionCode());
          }
     }
     return 0;
}
```

When run, this program will print:

```
Overflow: 0
Overflow: 1
Overflow: 2
Overflow: 3
Overflow: 4
Overflow: 5
Overflow: 6
Overflow: 7
Overflow: 8
Overflow: 9
Overflow: 10
Exception handler C00000FD
Overflow: 0
```

```
Overflow: 1
Overflow: 2
Overflow: 3
Overflow: 4
Overflow: 5
Overflow: 6
Overflow: 7
Overflow: 8
Overflow: 9
Overflow: 10
```

We see that the first time the handler is invoked, but the second time there is no handler invoked, the program exits abruptly without any further messages.

The reason for this behavior is that if a thread in your application causes an `EXCEPTION_STACK_OVERFLOW`, then your thread has left its stack in a damaged state. This is in contrast to other exceptions such as `EXCEPTION_ACCESS_VIOLATION` or `EXCEPTION_INT_DIVIDE_BY_ZERO`, where the stack is not damaged. This is because the stack is set to an arbitrarily small value when the program is first loaded, just 4096 bytes, 1 "page". The stack then grows on demand to meet the needs of the thread. This is implemented by placing a page with `PAGE_GUARD` access at the end of the current stack. When your code causes the stack pointer to point to an address on this page, an exception occurs. The system then does the three following things:

1 Remove the PAGE_GUARD protection on the guard page, so that the thread can read and write data to the memory.

2 Allocate a new guard page that is located one page below the last one.

3 Rerun the instruction that raised the exception.

If a thread in your program grows the stack beyond its limit (1MB with normal programs), the step 1 above will succeed, but the step 2 will fail. The system generates an exception and the first try block is executed. This is OK, but the stack has been left *without* a guard page. Since the second time that we get a stack overflow there is no more stack left, the program will provoke an access violation when it attempts to use the stack within the code of the exception handling procedure __except_handler3, that receives the exception from windows. This will provoke the dreaded double fault exception that terminates immediately the program no questions asked. Note that not even the "this program has attempted an illegal operation" dialog box appears.

The correct way of handling stack overflow is then, to do the following:

1 Get the page size from the system.

2 Set the page to guard page again

An example is given below:

```
#include <intrinsics.h> // For _GetStackPointer()
int main(int argc, char* argv[])
{
     for (;;)
     {
          __try {
               StackOverflow(0);
          }
          __except(EXCEPTION_EXECUTE_HANDLER)
          {
               LPBYTE lpPage;
               static SYSTEM_INFO si;
               static MEMORY_BASIC_INFORMATION mi;
```

```
                static DWORD dwOldProtect;

                // Get page size of system
                GetSystemInfo(&si);
                // Find SP address
                lpPage = _GetStackPointer();
                // Get allocation base of stack
                VirtualQuery(lpPage, &mi, sizeof(mi));
                // Go to page beyond current page
                lpPage = (LPBYTE)(mi.BaseAddress)-si.dwPageSize;
                // Free portion of stack just abandoned
                if (!VirtualFree(mi.AllocationBase,
                   (LPBYTE)lpPage - (LPBYTE)mi.AllocationBase,
                   MEM_DECOMMIT)) {
                   // If we get here, exit, there is nothing that can be done
                   exit(1);
                }
                // Reintroduce the guard page
                if (!VirtualProtect(lpPage, si.dwPageSize,
                   PAGE_GUARD | PAGE_READWRITE,
                   &dwOldProtect)) {
                   exit(1);
                }
                printf("Exception handler %lX\n", GetExceptionCode());
                Sleep(2000);
            }
        }
        return 0;
}
```

1.32.5.6 The __retry construct

You can correct the error condition that provoked the exception and then restart again using the __retry construct. A simple example of this is the following:

```
#include <stdio.h>
#include <seh.h>
int main(void)
{
        char *p = NULL;

        __try  {
                *p ='A';
                printf("%s\n",p);
        }
        __except(EXCEPTION_EXECUTE_HANDLER) {
                p = "abc";
                __retry;
        }
        return 0;
}
```

The first execution of the __try yields an exception since the pointer p is NULL. Within the except block we correct this condition by assigning to that pointer a valid value, then we restart with the __retry. The output of this program is:

Abc

1.32.6 The signal function

Besides the structured exception handling mechanism outlined above, lcc-win32 offers the standard `signal()` utility of the standard C language.

1.32.6.1 Software signals

Software signals are the same thing as the exceptions we described in the proceeding paragraphs. They can be raised by the CPU (traps) or simply raised by the function `raise()`. To give you an idea of how this mechanism works let's repeat the example of above (structured exception handling) using the signal mechanism:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h> // For exit()

// This function will be called by the signal mechanism when a SIGSEGV is raised.
void traphandler(int s)
{
     printf("Illegal address exception\n");
     exit(1);
}
int main(void)
{
     char *p = NULL;

     // Set the function to handle SIGSEGV
     signal(SIGSEGV,traphandler);
     *p = 0;
     return 0;
}
```

This will produce the following output:

```
D:\lcc\test>tsignal
Illegal address exception
```

To make it easy to understand, the code above doesn't test the result of setting the signal handler. A more robust version looks like this:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h> // For exit()
[snip for brevity]
int main(void)
{
     char *p = NULL;
     void (*oldhandler)(int);

     oldhandler = signal(SIGSEGV,traphandler);
     if (oldhandler == SIG_ERR) {
          printf("Impossible to establish a signal handler\n");
          exit(1);
     }
     *p = 0;
     signal(SIGSEGV,oldhandler);
     return 0;
}
```

This code tests if the return value of the signal() call is OK. Besides, before the function exists, the old handler will be restored.

The software signals that lcc-win32 supports are described in the standard header file <signal.h>. They include SIGSEGV for illegal addresses, SIGFPE (floating point exceptions), and others. In general the signal() mechanism is barely supported by the operating system under Windows.

1.32.6.2 Using the signal mechanism

What can we do within our rescue function?

Not much.

We can't call any library functions, since they could call signal too, and they are not reentrant. Besides, when we return from the signal handling function we will end up crashing anyway since the signal mechanism will continue at the point where the program failed. Since signal() disables the specific signal that was raised, the second time we trap the program will crash.

A way out of this dilemma is to use signal() together with the setjmp/longjmp facility to recover somewhere else in the program. Here is an example:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h> // For exit()
#include <setjmp.h>
jmp_buf jumpbuffer;

// This function will call longjmp to reestablish a previous context assumed in the jumpbuffer
// global variable.
void traphandler(int s)
{
     psignal(s,"Error");
     longjmp(jumpbuffer,1);
}
int main(void)
{
     char *p = NULL;
     void (*oldhandler)(int);

     oldhandler = signal(SIGSEGV,traphandler);
     if (oldhandler == SIG_ERR) {
          printf("Impossible to establish a signal handler\n");
          exit(1);
     }
     if (setjmp(jumpbuffer)) {
          ;
     }
     else {
          *p = 0;
     }
     signal(SIGSEGV,oldhandler);
     printf("Normal exit\n");
     return 0;
}
```

This will print:

```
Error: Segmentation fault
Normal exit
```

What happens?

The first time we arrive at the statement:

```
if (setjmp(jumpbuffer))
```

the setjmp() function will return zero, and we will enter the else clause. There we produce a trap (SIGSEGV), what will raise the signal mechanism, that will call the traphandler() function. In there, we print the name of the signal using the standard library function psignal()[75] and then we make a longjmp that will return to the same

```
if (setjmp(jumpbuffer))
```

but this time with a value of 1. Instead of getting into the else arm of the if statement, we enter the branch with the empty statement, and we go out normally after the trap.

Note how this code starts to look very similar to the structured exception handling code. The difference is that here we have to trap all possible signals, then run our code, then unset all the signals to their former values.

The only reason to prefer using signal() to the structured exception handling is portability. The signal mechanism will work in many other environments. For instance the program above will run unmodified in a UNIX system producing the same output.

75. The psignal() function is not part of the standard but it exists in many implementation, specially in UNIX systems.

1.33 Numerical programming

Computers are done for making calculations, well, at least originally that was their objective. Playing games, internet browsing, word processing, etc., came later.

The problem of storing numbers in a computer however, is that the continuum of numbers is infinite and computers are limited. Yes, we have now many times the RAM amount that we had just a few years ago, but that amount is finite and the numbers we can represent in it are just a finite approximation to the infinite mathematical continuum.

The moment we convert a problem from the domain of mathematical analysis to the range of numbers that can be handled in a machine, even in a paper and pencil "machine", we are going to necessarily introduce approximations that can lead to errors, truncation errors, rounding errors, whatever.

Suppose we want to evaluate exp(x) by using a series expansion:

$$\exp(x) = 1 + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + O(x^9)$$

We have to stop somewhere. No way out. Here we get tired at the 9th term. And no matter how much effort we put into this, there will be always a truncation error. In this case the truncation error can be accurately calculated. Analyzing and estimating the error bounds is the art of numerical analysis.

Computers use bit patterns to represent any object that can be *represented* in a computer. In the section about structures we represented a person by a bit pattern like this:

```
structure person {
     char *Name;
     int age;
     ...
};
```

A person is surely not a bit pattern. We use the bit pattern to abstract some characteristics of the person we are interested in. Numbers aren't different. They can be represented by a bit pattern too. We can use 32 bits, what allows us to represented almost 4294967296 numbers. But obviously there are more numbers (infinitely more) than that, so *any* representation will always fail somewhere.

Any computation involving computer-representable numbers (that map onto the real-line) as its arguments can potentially produce a result that lies in-between two representable numbers. In that case, that number is rounded off to one of the grid-points. And this incurs round off error. Any practical numerical analyst (one who uses a computer to carry out numerical computations) must have a method of bounding, estimating and controlling both the round off error at each numerical step as well as the total round off error.

What do we want from the representation of numbers then?

1) The grid points should be as dense as possible

2) The range (the span between the smallest and the largest number) should be as wide as possible

3) The number of bits used should be as small as possible.

4) The rules of arithmetic should be mimicked as closely as possible.

5) The rules should be such that they can be implemented in hard-wired computer logic.

Note that all those requirements are completely contradictory. If we want a dense representation we have to use more bits. If we increase the range we have to thin the spacing between numbers, etc.

1.33.1 Floating point formats

In lcc-win32 we have a plethora of numeric types, ranging from the smallest single precision format, to the bignum indefinite precision numbers.

All the numeric types (including the complex numbers type but excluding the bignums) are based in different floating point formats. In this formats, a number is represented by its sign, an exponent, and a fraction.

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in the calculations. The floating point formats are an approximation of the continuum, leaving wide gaps between each representable number. The range and precision of this subset is determined by the IEEE Standard 754 for the types float, double and long double. The qfloat and the bignum data types have their own formats.

1.33.1.1 Float (32 bit) format

This format uses one bit for the sign, 8 bits for the exponent, and 23 bits for the fraction.

```
struct floatFormat {
     unsigned Fraction:23;
     unsigned Exponent:8;
     unsigned sign:1;
};
```

Bits 0:22 contain the 23-bit fraction, f, with bit 0 being the least significant bit of the fraction and bit 22 being the most significant; bits 23:30 contain the 8-bit biased exponent, e, with bit 23 being the least significant bit of the biased exponent and bit 30 being the most significant; and the highest-order bit 31 contains the sign bit, s. The normalized numbers are represented using this formula:

$$(-1)s \times 2^{e-127} \times 1.f$$

Here we have an exponent that uses -127 as bias. This means that a constant is added to the actual exponent so that the number is always a positive number. The value of the constant depends on the number of bits available for the exponent. In this format the bias is 127, but in other formats this number will be different.

The range of this format is from `7f7fffff` to `00800000`, in decimal `3.40282347e+38` to `1.17549435e-38`. This numbers are defined in the standard header file <float.h> as FLT_MAX and FLT_MIN. The number of significant digits is 6, defined in float.h as FLT_DIG.

1.33.1.2 Double (64 bit) format

This format uses one bit for the sign, 11 bits for the exponent, and 52 bits for the fraction.

```
struct doubleFormat {
     unsigned FractionLow;
     unsigned FractionHigh:22;
     unsigned sign:1;
}
```

Bits 0..51 contain the 52 bit fraction f, with bit 0 the least significant and bit 51 the most significant; bits 52..62 contain the 11 bit biased exponent e, and bit 63 contains the sign bit s. The normalized numbers are represented with:

$$(-1)s \times 2^{e-1023} \times 1.f$$

The bias for the exponent in this case is -1023. The range of this format is from `7feffffff ffffffff` to `00100000 00000000` in decimal from `1.7976931348623157e+308` to `2.2250738585072014e-308`. This numbers are defined in float.h as DBL_MAX and DBL_MIN respectively. The number of significant digits is 15, defined as DBL_DIG.

1.33.1.3 Long double (80 bit) format

This format uses one bit for the sign, 15 bits for the biased exponent, and 64 bits for the fraction. Bits 0..63 contain the fraction, the bits 64..78 store the exponent, and bit 79 contains the sign.

```
struct longdoubleFormat {
     unsigned FractionLow;
     unsigned FractionHigh;
     unsigned Exponent:15;
     unsigned sign:1;
}
```

The bias for the exponent is 16383 and the formula is:

$$(-1)s \times 2^{e-16383} \times 1.f$$

The range of this format is from `7ffe ffffffff ffffffff` to `0001 80000000 00000000`, or, in decimal notation, from `1.189731495357231765 05e+4932` to `3.3621031431120935062 6e-4932`. Quite enough to represent the number of atoms in the whole known universe. Those numbers are defined as LDBL_MAX and LDBL_MIN in float.h. The number of significant digits is 18, defined as LDBL_DIG.. Note that even if the number of bits of the long double representation is 80, or ten bytes, `sizeof(long double)` is 12, and not 10. The reason for this is the alignment of this numbers in memory. To avoid having numbers not aligned at addresses that are not multiple of four, two bytes of padding are left empty.

1.33.1.4 The qfloat format

This format is specific to lcc-win32 and was designed by Stephen Moshier, the author of the "Cephes" mathematical library that lcc-win32 uses internally. Its description is as follows:

```
#define _NQ_    12
struct qfloatFormat {
     unsigned int sign;
     int exponent;
     unsigned int mantissa[_NQ_];
} ;
```

This is defined in the "qfloat.h" header file.It provides 104 significant digits, a fraction of 352 bits, (one word is left empty for technical reasons) and a biased exponent of 32 bits. The bias uses 0x8001, or 32769.

1.33.1.5 Special numbers

All the floating point representations include two "numbers" that represent an error or NAN, and signed infinity (positive and negative infinity). The representation of NANs in the IEEE formats is as follows:

type	NAN	Infinity (+ and -)
float	7fc00000	7f800000 ff800000
double	7ff80000 00000000	7ff00000 00000000 fff00000 00000000
long double	7fff ffffffff ffffffff	7fff 80000000 00000000 ffff 80000000 00000000

We have actually two types of NANs: quiet NANs and signalling NANs.

A Quiet NaN, when used as an operand in any floating point operation, quietly (that is without causing any trap or exception) produces another quiet NaN as the result, which, in turn, propagates. A Quiet NaN has a 1 set in the most significant bit-position in the mantissa field.

A Signaling NaN has no business inside an FPU. Its very presence means a serious error. Signaling NaNs are intended to set off an alarm the moment they are fetched as an operand of a floating point instruction. FPUs are designed to raise a trap signal when they touch a bit pattern like that.

Quiet NaNs are produced, when you do things like try to divide by zero, or you pass incorrect arguments to a standard FPU function, for instance taking the square root of -1. Modern FPUs have the ability to either produce a quiet NaN, or raise a signal of some sort, when they encounter such operands on such instructions. They can be initialized to do either of the two options, in case the code runs into these situations. We will see this in more details in the “exceptions” section further down.

1.33.2 What can we do with those numbers then?

Let’s do some experiments with this formats

1.33.2.1 Range

OK. We have seen how floating point numbers are stored in memory. To give us an idea of the range and precision of the different formats let’s try this simple program. It calculates the factorial of its argument, and it is not very efficient, since it will repeat all calculations at each time.

```
#include <stdio.h>
#include <math.h>
float factf(float f)
{
        float result=1.0;

        while (f > 0) {
                result *= f;
                if (!isfinitef(f))
                        break;
                f--;
        }
        return result;
}
int main(void)
{
     float ff=1.0f,fctf = 1.0;
```

```
            while (1) {
                    ff = factf(fctf);
                    if (!isfinitef(ff))
                            break;
                    printf("%10.0f! = %40.21g\n",fctf,ff);
                    fctf++;
            }
            printf("Max factorial is %g\n",fctf-1);
        return 0;
}
```

We start with the smallest format, the `float` format. We test for overflow with the standard function is_finitef, that returns 1 if its float argument is a valid floating point number, zero otherwise. We know that our `fact()` function will overflow, and produce a NAN (**N**ot **A** **N**umber) after some iterations, and we rely on this to stop the infinite loop. We obtain the following output:

```
         1! =                                     1
         2! =                                     2
         3! =                                     6
         4! =                                    24
         5! =                                   120
         6! =                                   720
         7! =                                  5040
         8! =                                 40320
         9! =                                362880
        10! =                               3628800
        11! =                              39916800
        12! =                             479001600
        13! =                            6227020800
        14! =                           87178289152
        15! =                         1307674279936
        16! =                        20922788478976
        17! =                       355687414628352
        18! =                      6402374067290112
        19! =                    121645096004222980
        20! =                   2432902298041581600
        21! =                  51090945235216237000
        22! =                1.124000724806013e+021
        23! =               2.5852017444594486e+022
        24! =                6.204483105838766e+023
        25! =               1.5511209926324736e+025
        26! =                4.032915733765936e+026
        27! =                1.088886923454107e+028
        28! =               3.0488839051318128e+029
        29! =               8.8417606614675607e+030
        30! =               2.6525290930453747e+032
        31! =               8.2228384475874814e+033
        32! =                2.631308303227994e+035
        33! =               8.6833178760213554e+036
        34! =                2.952328838437621e+038
Max factorial is 34
```

This measures the range of numbers stored in the float format, i.e. the capacity of this format to store big numbers. We modify slightly our program by replacing the `float` numbers by `double` numbers, and we obtain this:

```
         1! =                                     1
         2! =                                     2
         3! =                                     6
         4! =                                    24
```

```
          5! =                                    120
      ...
        168! =                  2.5260757449731988e+302
        169! =                  4.2690680090047056e+304
        170! =                   7.257415615308004e+306
Max factorial is 170
```

The range of double precision floats is much greater. We can go up to 170!, quite a big number. Even greater (as expected) is the range of long doubles, where we obtain:

```
       1751 =              3.674156538617319512e+4920
       1752 =              6.437122255657543772e+4923
       1753 =              1.128427531416767423e+4927
       1754 =              1.979261890105010059e+4930
Max factorial is 1754
```

Changing the declarations of the numbers to qfloats (and increasing the printing precision) increases even more the range:

```
3207! = 2.6860353624721360247253929832838122123693779548460 7e+9853
3208! = 8.6168014428106123673190606903744695772809644791461 8e+9856
3209! = 2.7651315829979255086726865755411672873494615013580 1e+9860
Max factorial is 3209
```

The range increases by more than 3,000 orders of magnitude.

1.33.2.2 Precision

What is the precision of those numbers?

We modify the first program as follows:

```
int main(void)
{
        float f=1.0f,fctf = 1.0;

        fctf = factf(34.0f);
        f = fctf+1.0f; // Add one to fctf
        if (fctf != f) { // 1+fctf is equal to fctf ???
                printf("OK\n");
        }
        else
                printf("Not ok\n");
        return 0;
}
```

We obtain the factorial of 34. We add to it 1. Then we compare if it is equal or not to the same number. Against all mathematical expectations, our program prints “Not ok”. **In floating point maths, 1+N = N !!!**

Why this?

The density of our format makes the gaps between one number and the next one bigger and bigger as the magnitude of the numbers increases. At the extreme of the scale, almost at overflow, the density of our numbers is extremely thin. We can get an idea of the size of the gaps by writing this:

```
int main(void)
{
        float f=1.0f,fctf = 1.0;

        fctf = factf(34.0f);
        f = 1.0f;
        while (fctf == (f+fctf)) {
                f *= 10.0f;
```

```
        }
        printf("Needs: %e\n",f);
        return 0;
}
```

We get the output:

```
Needs: 1.000000e+019
```

We see that the gap between the numbers is huge: 1e19!

What are the results for double precision?

We modify our program and we get:

```
Needs: 1.000000e+019
```

What????

Why are the results of double precision identical to the floating point precision? We should find that the smallest format would yield gaps much wider than the other, more precise format!

Looking at the assembler code generated by our floating point program, we notice:

```
; while (fctf == (f+fctf)) {
        flds    -16(%ebp) ; loads fctf in the floating point unit
        fadds   -4(%ebp)  ; adds f to the number stored in the FPU
        fcomps  -16(%ebp) ; compares the sum with fctf
```

Looking at the manuals for the pentium processor we see that the addition is done using the full FPU precision (80 bits) and not in floating point precision. Each number is loaded into the FPU and automatically converted to a 80 bits precision number. We modify our program to avoid this:

```
int main(void)
{
        float f,fctf,sum;

        fctf = factf(34.0f);
        f = 1.0f;
        sum = f+fctf;
        while (fctf == sum) {
                f *= 2.0f;
                sum = fctf+f;
        }
        printf("Needs: %e\n",f);
        return 0;
}
```

Note that this modified program is mathematically equivalent to the previous one. When we run it, we obtain:

```
Needs: 1.014120e+031
```

OK, now we see that the gap between numbers using the float format is much bigger than the one with double precision.

Note that **both versions of the program are mathematically equivalent but numerically completely different!**

Note too that the results differ by 12 orders of magnitude just by modifying slightly the calculations.

We modify our program for double precision, and now we obtain:

```
Needs: 3.777893e+022
```

The gap using double precision is much smaller (9 orders of magnitude) than with single precision.[76]

Using qfloats now, we write:

```
#include <qfloat.h>
#include <stdio.h>
int main(void)
{
        qfloat f=34,fctf;

        fctf = factq(f);
        f = fctf+1;
        if (fctf != f) {
                printf("OK\n");
        }
        else
                printf("Not ok\n");
        return 0;
}
```

This prints OK at the first try. Using the extremely precise qfloat representation we obtain gaps smaller than 1 even when the magnitude of the numbers is 10^{34}.

This extension of lcc-win32 allows you to use extremely precise representation only in the places where it is needed, and revert to other formats when that precision is no longer needed.

1.33.2.3 Going deeper

Let's take a concrete example: 178.125.

Suppose this program:

```
#include <stdio.h>
// No compiler alignment
#pragma pack(1)
// In this structure we describe a simple precision floating point
// number.
typedef union {
     float fl;
     struct {
          unsigned f:23; // fraction part
          unsigned e:8;// exponent part
          unsigned sign:1;// sign
        };[77]
} number;

// This function prints the parts of a floating point number
// in binary and decimal notation.
void pfloat(number t)
{
     xprintf("Sign %d, exponent %d (-127= %d), fraction: %023b\n",
               t.sign,t.e,t.e-127,t.f);
}
```

76. Note that there are as many IEEE754 numbers between 1.0 and 2.0 as there are between 2^56 and 2^57 in double format. 2^57 - 2^56 is quite a big number: 72,057,594,037,927,936.

77. This is an unnamed structure within another structure or union. lcc-win32 and other compilers (like visual C) allow to access the members of an unnamed union/structure without having to write: t.u.sign,t.u.e,etc.

```
int main(void)
{
     number t;

     t.fl = 178.125;
     pfloat(t);
     return 0;
}
```

This will produce the output:

```
Sign 0, exponent 134 (-127= 7), fraction: 01100100010000000000000
```

To calculate the fraction we do:

```
fraction = 01100100001 =
0 * 1/2 +
1 * 1/4 +
1 * 1/8 +
0 * 1/16+
0 * 1/32+
1 * 1/64 +
 ... +
1 * 1/1024
```

This is:

```
0.25+0.125+0.015625+0.0009765625 = 0.3916015625
```

Then, we add `1` to `0.3916015625` obtaining `1.3916015625.`

This number, we multiply it by 2^7 = 128:

```
1,3916015625 * 128 = 178.125.
```

1.33.2.4 Rounding modes

When the result of a computation does not hit directly a number in our representation grid, we have to decide which number in the grid we should use as the result. This is called rounding. We have the following rounding modes:

1) Round to the nearest grid point. This is the default setting when a program compiled with lcc-win32 starts executing.

2) Round upwards. Choose always the next higher grid point in the direction of positive infinity.

3) Round downwards. Choose always the next lower grid point in the direction of negative infinity.

4) Round to zero. We choose always the next grid point in the direction of zero. If the number is positive we round down, if it is negative we round up.

This rounding modes are defined in the standard header file fenv.h as:

```
/* Rounding direction macros */
#define FE_TONEAREST    0
#define FE_DOWNWARD     1
#define FE_UPWARD       2
#define FE_TOWARDZERO   3
```

You can change the default rounding precision by using the function `fesetround(int)`, also declared in the same header file.

The rationale for this "rounding modes" is the following: To know if an algorithm is stable, change your rounding mode using the same data and run your program in all rounding modes. Are the results the same? If yes, your program is numerically stable. If not, you got a big problem and you will have to debug your algorithm.

For a total of N floating point operations you will have a rounding error of:[78]

1) For round to nearest is sqrt(N) * machine epsilon

2) For round up is N * machine epsilon.

3) For round down is -N * machine epsilon.

4) For round to zero is -N * machine epsilon if the number is positive, N * Machine Epsilon if the number is negative.

The number you actually obtain will depend of the sequence of the operations.

The machine epsilon is the smallest number that changes the result of an addition operation at the point where the representation of the numbers is the densest. In IEEE754 representation this number has an exponent value of the bias, and a fraction of 1. If you add a number smaller than this to 1.0, the result will be 1.0. For the different representations we have in float.h:

```
#define FLT_EPSILON 1.19209290e-07F // float
#define DBL_EPSILON 2.2204460492503131e-16 // double
#define LDBL_EPSILON 1.084202172485504434007452e-19L //long double
// qfloat epsilon truncated so that it fits in this page...
#define QFLT_EPSILON 1.09003771904865842969737513593110651 ... E-106
```

This defines are part of the C99 ANSI standard and should be defined in all compilers that implement that standard.

When in C you convert a floating point number into an integer, the result is calculated using rounding towards zero. To see this in action look at this simple program:

```
#include <stdio.h>
void fn(double a)
{
        printf("(int)(%g)=%d (int)(%g)=%d\n",a,(int)a,-a,(int)-a);
}
int main(void) {
        for (double d = -1.2; d < 2.0; d += 0.3)
                fn(d);
        return 0;
}
```

This leads to the following output (note the lack of precision: 0.3 can't be exactly represented in binary):

```
(int)(-1.5)=-1   (int)(1.5)=1
(int)(-1.2)=-1   (int)(1.2)=1
(int)(-0.9)=0    (int)(0.9)=0
(int)(-0.6)=0    (int)(0.6)=0
(int)(-0.3)=0    (int)(0.3)=0
(int)(1.11022e-016)=0    (int)(-1.11022e-016)=0
(int)(0.3)=0     (int)(-0.3)=0
(int)(0.6)=0     (int)(-0.6)=0
(int)(0.9)=0     (int)(-0.9)=0
```

78. See http://serc.iisc.ernet.in/~ghoshal/fpv.html#nodeliver, or the home page of the Siddhartha Kumar Ghoshal, Senior Scientific Officer at the Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore.

```
(int)(1.2)=1     (int)(-1.2)=-1
(int)(1.5)=1     (int)(-1.5)=-1
(int)(1.8)=1     (int)(-1.8)=-1
```

To round away from zero you can use:

```
#define Round(x) ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
```

This will result in nonsense results if there is an overflow. A better version would be:

```
#define Round(x) \
     ((x) < LONG_MIN-0.5 || (x) > LONG_MAX+0.5 ?\
     error() :
          ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
```

The standard library function round() does this too.[79]

To round towards positive infinity you use:

```
#define RoundUp(x) ((int)(x+0.5))
```

Negative infinity is similar.

1.33.3 Numerical stability[80]

Suppose we have a starting point and a recurrence relation for calculating the numerical value of an integral. The starting value is given by:

$$I_0=\left[\ln(x+5)\right]^1_0 = \ln 6 - \ln 5 = 0.182322$$

The recurrence relation is given by:

$$I_1 = 1/1 - 5I_0$$
$$I_2 = 1/2 - 5I_1$$
$$I_3 = 1/3 - 5I_2$$
$$\text{etc}$$

We calculate this, starting with 0.182322. We use the following program:

```
#include <stdio.h>
#include <math.h>
int main(void)
{
        float i = 0.182322;

        for (int z = 1; z<9;z++) {
                i = 1.0f/(float)z - 5.0*i;
                printf("I%-3d: %9.6g\n",z,i);
        }
        return 0;
}
```

We use single precision. Note the notation 1.0f meaning 1 in float precision.

```
I1  :   0.08839
I2  : 0.0580499
```

---

79. The round functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction. Ansi C standard page 232.
80. I have this example from the very good book “Numerical Mathematics and Scientific Computation” by Germund Dahlquist and Åke Björck. Available on line at: http://www.mai.liu.se/~akbjo/NMbook.html

```
I3  : 0.0430839
I4  : 0.0345805
I5  : 0.0270974
I6  : 0.0311798
I7  : -0.0130418
I8  :  0.190209
```

The first few numbers look correct, but I6 is bigger than I5, what after the recurrence relation should never happen. Moreover I7 is negative and later numbers are complete nonsense.

Why?

Well, because the roundoff error e in I0 is *multiplied* by 5 in the first iteration, then multiplied again by 5 in the next iteration so that after a few iterations the error becomes bigger than the result.

Writing this is double precision, and replacing the precalculated constant with a computation of log(6.0) - log(5.0) we get better results.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
        double i = log(6.0) - log(5.0);

        for (int z = 1; z<29;z++) {
                i = 1.0/(double)z - 5.0*i;
                printf("I%-3d: %9.6g\n",z,i);
        }
        return 0;
}
```

We get:

```
I1  : 0.0883922
I2  : 0.0580389
I3  : 0.0431387
I4  : 0.0343063
I5  : 0.0284684
I6  : 0.0243249
I7  : 0.0212326
I8  : 0.0188369
I9  : 0.0169265
I10 : 0.0153676
```

```
I11 : 0.0140713
I12 : 0.0129767
I13 : 0.0120398
I14 : 0.0112295
I15 : 0.0105192
I16 : 0.00990385
I17 : 0.00930427
I18 : 0.00903421
I19 : 0.00746051
I20 : 0.0126975
```

```
I21 : -0.0158682
I22 :  0.124796
I23 :   -0.5805
I24 :   2.94417
I25 :  -14.6808
I26 :   73.4427
I27 :  -367.176
I28 :   1835.92
```

We see that now we can go up to the 19th iteration with apparently good results. We see too that the increased precision has only masked a fundamental flaw of the algorithm itself. The manner the calculation is done produces a five fold increase in the error term at each iteration. This is an example of a numerically unstable algorithm.

1.33.4 Complex numbers

Complex numbers have a real and an imaginary part. According to which concrete number representation is used to store those parts, they can be

1: float _Complex.
2: double _Complex
3: long double _Complex

4: qfloat _Complex

In lcc-win32, however, all complex types will be condensed in the long double complex type. This implementation can be changed later, but in any case it is better that you are aware of this feature.

There are several reasons for this.

Using this representation allows memory stored numbers to be identical to numbers as used by the floating point unit. Programs will give the same results even if they store intermediate results in memory.

It simplifies the compiler and the complexity of the code generation machinery.

All this types can be mixed in an expression, and most operators like + or * are defined for them, with the exception of less, less equal, greater, and greater than.

You should always include the “complex.h” standard header file when you use this type of numbers. In that header file, the basic keyword _Complex is replaced by just “complex”, and it will be used here as an equivalent keyword.

1.33.4.1 Complex constants:

They can be written in two ways:

```
#include <complex.h>
double complex s = 123.0+12.8i;
```

or

```
#include <complex.h>
double complex s = 123.0+12.8*I;
```

Note that the first notation is not standard (but used in the gcc compiler system), and it applies only to explicit constants. Please do not assume that if you have a variable “x”, just writing “xi” will transform it into a complex number!

When in your code you build a complex number using variables, you use the standard notation:

```
double complex w;
w = (65.0*angle/2)+45.8*angle*I;
```

In this example the real part of the number is (65.0*angle/2) and the imaginary part is 45.8*angle. The constant “I” represents 0,1 or csqrt(-1) and is defined in complex.h.

1.34 Programming with security in mind

The C language doesn't give you many security nets. Actually, there is no security net. You are supposed to make no mistakes. This is very difficult to achieve, of course. Here, I will try to give you some simple advice that you can use in your code to make it safer and more robust.

1.34.1 Always include a 'default' in every switch statement

It is very common that you write a switch statement just taking into account the "normal" cases, i.e. the ones you expect. This is not very clever if the data doesn't fit into the expected cases, since the program will go on after the switch statement, even with a completely unexpected result. This will lead to strange errors further down, what makes the error discovering more difficult. Always include a "default" case, even if you just write:

```
switch (input) {
     case xx:
     ...
     default:
          assert(0);
}
```

The assert macro (defined in <assert.h>) will stop the program with a clear error message indicating the source line and file of the faulty code. This means that you will be informed when an unexpected data item arrives, and you can correct this during the debug phase.

1.34.2 Pay attention to strlen and strcpy

As we have seen before, (See page 63) the `strlen` and `strcpy` functions expect a well formed C string. The `strlen` function will not stop until a zero byte is found. This is not very dangerous, since strlen just reads memory, without modifying it. Our `strcpy` function however, expects a large enough destination buffer, and a well formed source string. If one of those requirements is not met, a catastrophe happens: strcpy will start destroying memory without any restriction. Always check for those two conditions *before* using it.

This is easier said than done though... Consider this code:

```
void func(char *p) {
   char buf[10+1];
   memset(buf,0,sizeof(buf));

   // limit string to 10 chars
   sprintf(buf,"%10s",p);
   printf("Hello, %s\n",buf);
}
```

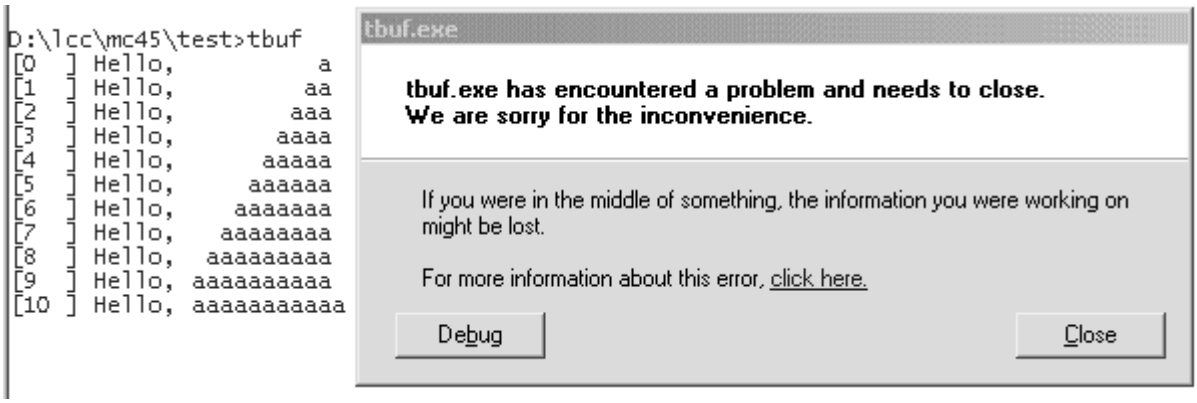
Where is the error here?

We see that the destination buffer is cleaned with memset, setting it to all zeroes. We see that the size of the destination buffer is 10+1, i.e. the programmer takes into account the terminating zero of the destination string. We see too that the sprintf format string indicates a field of 10 positions. Since the eleventh position is already set to zero with memset, the destination buffer should be always a correct string of at most 10 characters finished with zero.

Well, let's test our assumptions. We make a test program like this:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
        char buf[30];
```

```
        int i;

        memset(buf,0,sizeof(buf));
        for (i=0; i<14;i++) {
                buf[i] = 'a';
                printf("[%-3d] ", i);
                func(buf);
        }
        return 0;
}
```



Yes, we are sorry for the inconvenience. And no, it doesn't help to call Microsoft. It is not their fault this time.

What happened?

Obvious. The printf specification `"%10s"` will NOT limit the copy to 10 characters. It just says that the string will be padded at the left with blanks if it is *shorter* than 10 chars. If not, sprintf will continue to copy the characters without any limit.

Our function can be made safe very easily, by just using snprintf, instead of its dangerous cousin sprintf. The `snprintf` function will take an optional integer argument after the format string, telling it how many characters it should write at most into its destination buffer.

```
void func(char *p) {
   char buf[10+1];
   memset(buf,0,sizeof(buf));

   // limit string to 10 chars by using snprintf
   snprintf(buf,10,"%10s",p);
   printf("Hello, %s\n",buf);
}
```

This time, we see the expected output:

```
[0  ] Hello,          a
[1  ] Hello,         aa
[2  ] Hello,        aaa
[3  ] Hello,       aaaa
[4  ] Hello,      aaaaa
[5  ] Hello,     aaaaaa
[6  ] Hello,    aaaaaaa
[7  ] Hello,   aaaaaaaa
[8  ] Hello,  aaaaaaaaa
[9  ] Hello, aaaaaaaaaa
[10 ] Hello, aaaaaaaaaa
[11 ] Hello, aaaaaaaaaa
[12 ] Hello, aaaaaaaaaa
```

1.34.3 Do not assume correct input

Do you remember the Blaster worm?

Here is the code that allowed the Blaster worm to exist:[81]

```
HRESULT GetMachineName(WCHAR *pwszPath) {
    WCHAR  wszMachineName[N + 1])
    LPWSTR pwszServerName = wszMachineName;
    while (*pwszPath != L'\\' )
        *pwszServerName++ = *pwszPath++;
    ...
}
```

The error here is the assumption that the path is a reasonable path that will not overrun the destination buffer. Machine names are normally at most 10-15 chars long. If you are not expecting unreasonable input, you can just use the above code. And that code was running for quite a long time before this time bomb exploded and a clever hacker found out that he/she could gain control of the machine with this sloppy programming snippet.

1.34.4 Watch out for trojans

The windows API CreateProcess, can be given the name of the program to execute in two ways: as its first argument, or as a whole command line in its second parameter, leaving the first empty as NULL. Suppose this:

```
CreateProcess(NULL, "C:\\Program Files\\myprogram.exe", ...)
```

Suppose now, that somebody has left a “c:\program.exe” executable in the right place. What will happen? CreateProcess will start executing “c:\program” instead of the indicated program. A simple way to avoid this is to enclose the name of the program in quotes.

```
CreateProcess(NULL, "\"C:\\Program Files\\myprogram.exe\" -K -F",
...)
```

Notice that the related API WinExec will scan its input for the first space, making it worst than CreateProcess. In the above example, the code:

```
WinExec(“C:\\program files\\myprogram.exe”);
```

will *always* try to execute c:\program if it exists.

81. Microsoft Security Bulletin MS03-026: Buffer Overrun In RPC Interface Could Allow Code Execution (823980). Originally posted: July 16, 2003. Revised: September 10, 2003. Cited by Michael Horward in his excellent series about security in msdn. The URL for one of its articles is: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure09112003.asp

1.35 Pitfalls of the C language

Look. I am not a religious person. C is not a religion for me, and this means that I see some of the pitfalls of the language. I write this so that you see them at the start, and they do not bite you.

1.35.1 Defining a variable in a header file

If you write:

```
static int foo = 7;
```

in a header file, each C source file that includes that header will have a different copy of “foo”, each initialized to 7, but each in a different place. These variables will be totally unrelated, even if the intention of the programmer is to have a single variable “foo”.

If you omit the static, at least you will get an error at link time, and you will see the bug.

Golden rule: Never define something in a header file. Header files are for declarations only!

1.35.2 Confusing = and ==

If you write

```
if (a = 6) {
}
```

you are assigning to “a” the number 6, instead of testing for equality. The “if” branch will be always taken because the result of that assignment is different from zero. The compiler will emit a warning whenever an assignment in a conditional expression is detected.

1.35.3 Forgetting to close a comment

If you write:

```
a=b; /* this is a bug
c=d; /* c=d will never happen */
```

The comment in the first line is not terminated. It goes one through the second line and is finished with the end of the second line. Hence, the assignment of the second line will never be executed. Wedit helps you avoid this by coloring commentaries in another color as normal program text.

1.35.4 Easily changed block scope.

Suppose you write the following code:

```
if  (someCondition)
     fn1();
else
     OtherFn();
```

This is OK, but if you add some code to debug, for instance, you end up with:

```
if  (someCondition)
     fn1();
else
     printf(“Calling OtherFn\n”);
     OtherFn();
```

The else is not enclosed in curly braces, so only one statement will be taken. The end result is that the call to OtherFn is always executed, no matter what.

Golden rule: ALWAYS watch out for scopes of “if” or “else” not between curly braces when adding code.

1.35.5 Using the ++ or -- more than once in an expression.

The ANSI C standard[82] specifies that an expression can change the value of a variable only once within an expression. This means that a statement like:

```
i++ = ++i;
```

is invalid, as are invalid this, for instance:

```
i = i+++++i;
```

(in clear `i++ + ++i`)

1.35.6 Unexpected Operator Precedence

The code fragment,

```
if( chr = getc() != EOF ) {
printf( "The value of chr is %d\n", chr );
}
```

will always print 1, as long as end-of-file is not detected in getc. The intention was to assign the value from getc to chr, then to test the value against EOF. The problem occurs in the first line, which says to call the library function getc. The return value from getc (an integer value representing a character, or EOF if end-of-file is detected), is compared against EOF, and if they are not equal (it’s not end-of-file), then 1 is assigned to the object chr. Otherwise, they are equal and 0 is assigned to chr. The value of chr is, therefore, always 0 or 1.

The correct way to write this code fragment is,

```
if( (chr = getc()) != EOF ) {
     printf( "The value of chr is %d\n", chr );
}
```

The extra parentheses force the assignment to occur first, and then the comparison for equality is done.[83]

---

82. Doing assignment inside the controlling expression of loop or selection statements is not a good programming practice. These expressions tend to be difficult to read, and problems such as using = instead of == are more difficult to detect when, in some cases, = is desired.

83. How does it work?

He tests first if the lower 16 bits contain a 1 bit. The number 65535 consists of eight 1s, in binary notation, since $65535 = 2^{16} - 1$.

If the test fails, this means that the least significant bit can’t be in the lower 16 bits. He increases the counter by 16, and shifts right the number to skip the 8 bits already known to contain zero. If the test succeeds, this means that there is at least a bit set in the lower 16 bits. Nothing is done, and the program continues to test the lower 16 bits.

He uses the same reasoning with the 16 bits in x that now contain either the high or the lower word of x. 255 is $2^8 - 1$. This is applied then recursively. At each step we test 16, then 8, then 4, then 2 and at the end the last bit.

1.35.7 Extra Semi-colon in Macros

The next code fragment illustrates a common error when using the preprocessor to define constants:

```
#define MAXVAL 10; // note the semicolon at the end
/* ... */
if( value >= MAXVAL ) break;
```

The compiler will report an error. The problem is easily spotted when the macro substitution is performed on the above line. Using the definition for MAXVAL, the substituted version reads,

```
if( value >= 10; ) break;
```

The semi-colon (;) in the definition was not treated as an end-of-statement indicator as expected, but was included in the definition of the macro MAXVAL. The substitution then results in a semi-colon being placed in the middle of the controlling expression, which yields the syntax error. Remember: the pre-processor does only a textual substitution of macros.

1.35.8 Watch those semicolons!

Yes, speaking about semicolons, look at this:

```
if (x[j] > 25);
     x[j] = 25;
```

The semicolon after the condition of the if statement is considered an empty statement. It changes the whole meaning of the code to this:

```
if (x[j] > 25) { }
x[j] = 25;
```

The `x[j] = 25` statement will be always executed.

1.35.9 Assuming pointer size is equal to integer size

Today under lcc-win32 the sizeof(void *) is equal to the sizeof(int). This is a situation that will change in the future when we start using the 64 bit machines where int can be 32 bits but pointers would be 64 bits. This assumption is deeply rooted in many places even under the windows API, and it will cause problems in the future. Never assume that a pointer is going to fit in an integer, if possible.

1.35.10 Careful with unsigned numbers

Consider this loop:

```
int i;
for (i = 5; i >= 0; i --) {
     printf("i = %d\n", i);
}
```

This will terminate after 6 iterations. This loop however, will never terminate:

```
unsigned int i;
for (i = 5; i >= 0; i --) {
     printf("i = %d\n", i);
}
```

The loop variable i will decrease to zero, but then the decrement operation will produce the unsigned number 4294967296 that is bigger than zero. The loop goes on forever.Note too that the common windows type DWORD is **unsigned**!

1.35.11 Changing constant strings

Constant strings are the literal strings that you write in your program. For instance, you write:

```
outScreen(“Please enter your name”);
```

This constant string “Please enter your name” is stored (under lcc-win32) in the data section of your program and can be theoretically modified. For instance suppose that the routine “outScreen” adds a “\r\n” to its input argument. This will be in almost all cases a serious problem since:

1) The compiler stores identical strings into the same place. For instance if you write

```
a = “This is a string”;
b = “This is a string”;
```

there will be only one string in the program. The compiler will store them under the same address, and if you modify one, the others will be automatically modified too since they are all the same string.

2) If you add characters to the string (with strcat for instance) you will destroy the other strings or other data that lies beyond the end of the string you are modifying.

Some other compilers like gcc will store those strings in read memory marked as read only, what will lead to an exception if you try to modify this. Lcc-win32 doesn’t do this for different reasons, but even if you do not get a trap it is a bad practice that should be avoided.

A common beginner error is:

```
char *a = “hello”;
char *b = “world”;
strcat(a,b);
```

In this case you are adding at the end of the space reserved for the character array “hello” another character string, destroying whatever was stored after the “a” character array.

1.35.12 Indefinite order of evaluation

Consider this code:

```
fn(pointer->member, pointer = &buffer[0]);
```

This will work for some compilers (gcc, lcc-win32) and not for others. The order of evaluation of arguments in a function call is *undefined*. Keep this in mind. If you use a compiler that will evaluate the arguments from left to right you will get a trap or a nonsense result.

1.35.13 A local variable shadows a global one

Suppose you have a global variable, say “buf”. If you declare a local variable of the same name at an inner scope, the local variable will take precedence over the global one, i.e. when you write:

```
unsigned char buf[BUFSIZ];
int fn(int a)
{
     char buf[3];
     ...
     buf[BUFSIZ-1] = 0; // Error! the local variable is accessed,
                        // not the global one
}
```

Giving the command line option “-shadows” to the compiler will generate a warning when this happens.

1.35.14 Careful with integer wraparound

Consider this code:[84]

```
bool func(size_t cbSize) {
   if (cbSize < 1024) {
      // we never deal with a string trailing null
      char *buf = malloc(cbSize-1);
      memset(buf,0,cbSize-1);

      // do stuff

      free(buf);

      return true;
   } else {
      return false;
   }
}
```

Everything looks normal and perfect in the best of all worlds here. We test if the size if smaller than a specified limit, and we then allocate the new string. But... what happens if `cbSize` is zero???

Our call to malloc will ask for 0-1 bytes, and using 32 bit arithmetic we will get an integer wrap-around to `0xffffffff`, or `-1`. We are asking then for a string of 4GB. The program has died in the spot.

1.35.15 Problems with integer casting

In general, the compiler tries to preserve the value, but casting from signed to unsigned types can produce unexpected results. Look, for instance, at this sequence:

```
char c = 0x80; //-128
//now we cast it to short
short s = (short)c;    //now s = 0xff80 still -128
unsigned short us = (unsigned short)s; //us = 0xff80, which is 65408!
```

In general you should not try to mix signed/unsigned types in casts.

Casts can occur automatically as a result of the operations performed. The operators + (addition), ~ (bitwise negation) - (minus) will cast any type shorter than an int into a signed int. If the type is larger than an integer, it will be left unchanged.

1.35.16 Octal numbers

Remember that numbers beginning with zero are treated as number written in base 8. The number 012 is decimal 10, not 12. This error is difficult to find because everything will compile without any error, after all, this is a legal construct.

84. I got this example from the excellent column of Michael Horward in the msdn security column.

Chapter

2

Windows Programming

2.1 Introduction

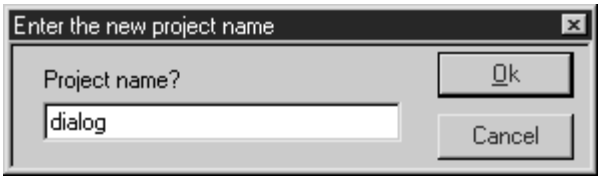
OK, up to now we have built a small program that receives all its input from a file. This is more or less easy, but a normal program will need some input from the user, input that can't be passed through command line arguments, or files. At this point, many introductory texts start explaining scanf, and other standard functions to get input from a command line interface. This can be OK, but I think a normal program under windows uses the features of windows.[85]

We will start with the simplest application that uses windows, a dialog box with a single edit field, that will input a character string, and show it in a message box at exit.

The easiest way to do this is to ask wedit to do it for you. You choose 'new project' in the project menu, give a name and a sources directory, and when the software asks you if it should generate the application skeleton for you, you answer yes.

You choose a dialog box application, when the main dialog box of the "wizard" appears, since that is the simplest application that the wizard generates, and will fit our purposes quite well.

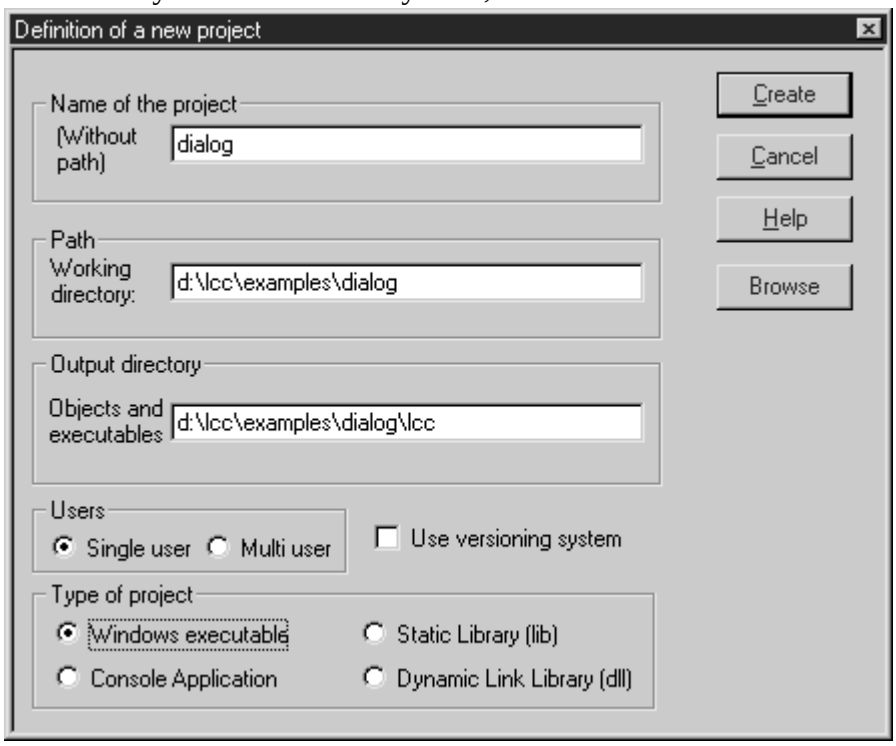
But let's go step by step. First we create a project. The first thing you see is a dialog box, not very different from the one we are going to build, that asks for a name for the new project. You enter a name like this:



You press OK, and then we get a more complicated one, that asks quite a lot of questions.

85. note 82

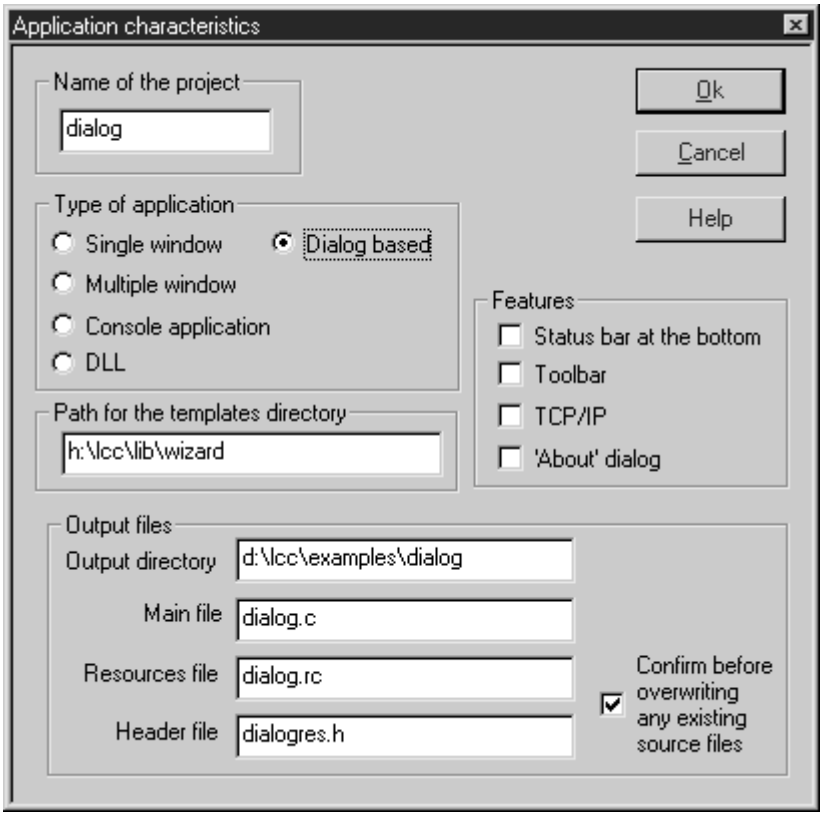
You enter some directory in the second entry field, make sure the “windows executable” at the



bottom is selected, and press ok. Then we get:

You press the “yes” button. This way, we get into the wizard.

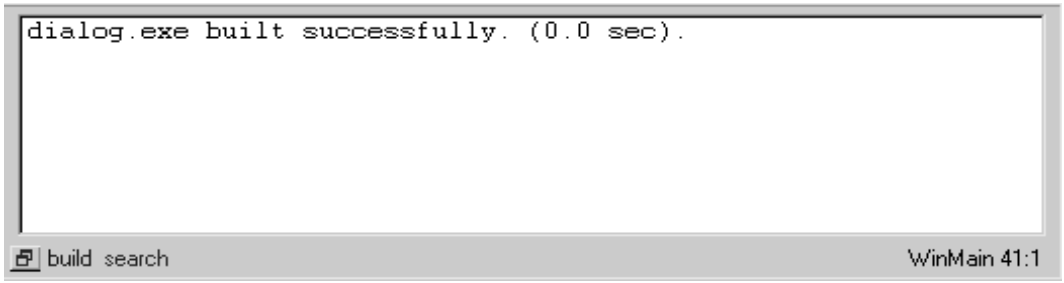
The first panel of the wizard is quite impressing, with many buttons, etc. Ignore all but the type of application panel. There, select a “dialog based” application, like this:



You see, the “Dialog based’ check button at the upper left is checked. Then press the OK button.

Then we get to different dialogs to configure the compiler. You leave everything with the default values, by pressing Next at each step. At the end, we obtain our desired program. For windows standards, this is a very small program: 86 lines only, including the commentaries. We will study this program in an in-depth manner. But note how short this program actually is. Many people say that windows programs are impossible huge programs, full of fat. This is just not true!

But first, we press F9 to compile it. Almost immediately, we will obtain:



Dialog.exe built successfully. Well, this is good news![86] Let's try it. You execute the program you just built using Ctrl+F5. When we do this, we see our generated program in action:

Just a dialog box, with the famous OK/Cancel buttons, and nothing more. But this is a start. We close the dialog, either by pressing the "x" button at the top right corner, or just by using OK or Cancel, they both do the same thing now, since the dialog box is empty.

We come back to the IDE, and we start reading the generated program in more detail. It has three functions:

- WinMain
- InitializeApp
- DialogFunc

If we ignore the empty function "InitializeApp", that is just a hook to allow you to setup things before the dialog box is shown, only two functions need to be understood. Not a very difficult undertaking, I hope.

2.1.1 WinMain

Command line programs, those that run in the ill named "msdos window", use the "main" function as the entry point. Windows programs, use the WinMain entry point.[87]

The arguments WinMain receives are a sample of what is waiting for you. They are a mess of historical accidents that make little sense now. Let's look at the gory details:

```
int APIENTRY WinMain(HINSTANCE hinst,
HINSTANCE hinstPrev,
LPSTR lpCmdLine,
int nCmdShow);
```

This is a function that returns an int, uses the stdcall calling convention[88] denoted by APIENTRY, and that receives (from the system) 4 parameters.

hinst, a "HANDLE" to an instance of the program. This will always be 0x400000 in hexadecimal, and is never used. But many window functions need it, so better store it away.

---

86. This has only historical reasons, from the good old days of windows 2.0 or even earlier. You can use "main" as the entry point, and your program will run as you expect, but traditionally, the entry point is called WinMain, and we will stick to that for now.

87. A calling convention refers to the way the caller and the called function agrees as to who is going to adjust the stack after the call. Parameters are passed to functions by pushing them into the system stack. Normally it is the caller that adjusts the stack after the call returns. With the stdcall calling convention, it is the called function that does this. It is slightly more efficient, and contributes to keeping the code size small.

88. API means Application Programmer Interface, i.e. entry points into the windows system for use by the programmers, like you and me.

hinstPrev. Again, this is a mysterious “HANDLE” to a previous instance of the program. Again, an unused parameter, that will always contain zero, maintained there for compatibility reasons with older software.

lpCmdLine. This one is important. It is actually a pointer to a character string that contains the command line arguments passed to the program. Note that to the contrary of “main”, there isn’t an array of character pointers, but just a single character string containing all the command line.

nCmdShow. This one contains an integer that tells you if the program was called with the instruction that should remain hidden, or should appear normally, or other instructions that you should use when creating your main window. We will ignore it for now.

OK OK, now that we know what those strange parameters are used (or not used) for, we can see what this function *does*.

```
int APIENTRY WinMain(HINSTANCE hinst,
                     HINSTANCE hinstPrev,
                     LPSTR lpCmdLine,
                     int nCmdShow)
{
     WNDCLASS wc; // A structure of type WNDCLASS

     memset(&wc,0,sizeof(wc)); // We set it to zero
     wc.lpfnWndProc = DefDlgProc; // Procedure to call for handling messages
     wc.cbWndExtra = DLGWINDOWEXTRA;
     wc.hInstance = hinst;
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
     wc.lpszClassName = "dialog";
     RegisterClass(&wc);

     return DialogBox(hinst,
                      MAKEINTRESOURCE(IDD_MAINDIALOG),
                      NULL,
                     (DLGPROC) DialogFunc);
}
```

We see that the main job of this function is filling the structure wc, a WNDCLASS structure with data, and then calling the API[89] DialogBox. What is it doing?

We need to register a class in the window system. The windows system is object oriented, since it is derived from the original model of the window and desktop system developed at Xerox, a system based in SmallTalk, an object oriented language. Note that all windows systems now in use, maybe with the exception of the X-Window system, are derived from that original model. The Macintosh copied it from Xerox, and some people say that Microsoft copied it from the Macintosh. In any case, the concept of a *class* is central to windows.

A class of windows is a set of window objects that share a common procedure. When some messages or events that concern this window are detected by the system, a message is sent to the window procedure of the concerned window. For instance, when you move the mouse over the surface of a window, the system sends a message called WM_MOUSEMOVE to the windows procedure, informing it of the event.

There are quite a lot of messages, and it would be horrible to be forced to reply to all of them in all the windows you create. Fortunately, you do not have to. You just treat the messages that interest you, and pass all the others to the default windows procedure.

89. PLEASE never do this if you use the garbage collector!

There are several types of default procedures, for MDI windows we have MDIDefWindowProc, for normal windows we have DefWindowProc, and for dialog boxes, our case here, we have the DefDlgProc procedure.

When creating a class of windows, it is our job to tell windows which procedure should call when something for this window comes up, so we use the class registration structure to inform it that we want that all messages be passed to the default dialog procedure and we do not want to bother to treat any of them. We do this with:

```
wc.lpfnWndProc = DefDlgProc;
```

As we saw with the qsort example, functions are first class objects in C, and can be passed around easily. We pass the address of the function to call to windows just by setting this field of our structure.

This is the most important thing, conceptually, that we do here. Of course there is some other stuff. Some people like to store data in their windows[90]. We tell windows that it should reserve some space, in this case the `DLGWINDOWEXTRA` constant, that in win.h is #defined as 30. We put in this structure too, for obscure historical reasons, the hinst handle that we received in `WinMain`. We tell the system that the cursor that this window uses is the system cursor, i.e. an arrow. We do this with the API LoadCursor that returns a handle for the cursor we want. The brush that will be used to paint this window will be white, and the class name is the character string “dialog”.

And finally, we just call the RegisterClass API with a pointer to our structure. Windows does its thing and returns.

The last statement of WinMain, is worth some explanation. Now we have a registered class, and we call the DialogBox API, with the following parameters:

```
DialogBox(hinst,
          MAKEINTRESOURCE(IDD_MAINDIALOG),
          NULL, (DLGPROC) DialogFunc);
```

The hinst parameter, that many APIs still want, is the one we received from the system as a parameter to WinMain. Then, we use the MAKEINTRESOURCE macro, to trick the compiler into making a special pointer from a small integer, IDD_MAINDIALOG that in the header file generated by the wizard is defined as 100. That header file is called dialogres.h, and is quite small. We will come to it later.

What is this `MAKEINTRESOURCE` macro?

Again, history, history. In the prototype of the DialogBox API, the second parameter is actually a char pointer. In the days of Windows 2.0 however, in the cramped space of MSDOS with its 640K memory limit, passing a real character string was out of the question, and it was decided (to save space) that instead of passing the name of the dialog box resource as a real name, it should be passed as a small integer, in a pointer. The pointer should be a 32 bit pointer with its upper 16 bits set to zero, and its lower 16 bits indicating a small constant that would be searched in the resource data area as the “name” of the dialog box template to load.

Because we need to load a template, i.e. a series of instructions to a built-in interpreter that will create all the necessary small windows that make our dialog box. As you have seen, dialog boxes can be quite complicated, full of edit windows to enter data, buttons, trees, what have you. It would be incredible tedious to write all the dozens of calls to the `CreateWindow` API, passing it all the coords of the windows to create, the styles, etc.

---

90. When the IDE asks you if you want to open it as a resource say NO. We want to look at the text of that file this time.

To spare you this Herculean task, the designers of the windows system decided that a small language should be developed, together with a compiler that takes statements in that language and produce a binary file called *resource file*.

This resource files are bound to the executable, and loaded by the system from there automatically when using the DialogBox primitive. Among other things then, that procedure needs to know which dialog template should load to interpret it, and it is this parameter that we pass with the `MAKEINTRESOURCE` macro.

Ok, that handles (at least I hope) the second parameter of the DialogBox API. Let's go on, because there are still two parameters to go!

The third one is NULL. Actually, it should be the parent window of this dialog box. Normally, dialog boxes are written within an application, and they have here the window handle of their parent window. But we are building a stand-alone dialog box, so we left this parameter empty, i.e. we pass NULL.

The last parameter, is the DialogFunc function that is defined several lines below. The DefDlgProc needs a procedure to call when something important happens in the dialog box: a button has been pushed, an edit field receives input, etc.

Ok, this closes the call of the DialogBox API, and we are done with WinMain. It will return the result of the DialogBox function. We will see later how to set that result within our Dialog box procedure.

2.1.2 Resources

We mentioned before, that there is a compiler for a small resource language that describes our dialog boxes. Let's look at that with a little bit more detail before we go to our dialog procedure.

Open that file that should be called dialog.rc if you gave the project the "dialog" name[91], and look at this lines:

```
IDD_MAINDIALOG DIALOG 7, 20, 195, 86                              (1)
STYLE DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SYSMENU
(2)
CAPTION "dialog"                                                  (3)
FONT 8, "Helv"                                                    (4)
BEGIN
    DEFPUSHBUTTON   "OK", IDOK, 149, 6, 40, 14                    (5)
    PUSHBUTTON      "Cancel", IDCANCEL, 149, 23, 40, 14           (6)
END
```

We see that all those statements concern the dialog box, its appearance, the position of its child windows, etc. Let's go statement by statement:

1) We find here the same identifier `IDD_MAINDIALOG`, and then the `DIALOG` statement, together with some coordinates. Those coordinates are expressed in Dialog Units, not in pixels. The motivation behind this, is to make dialog boxes that will look similar at all resolutions and with different screen sizes. The units are based somehow in the size of the system font, and there are APIs to get from those units into pixels, and from pixels into those units.

91. You will be prompted for a header file, where are stored the definitions for things like IDD_MAINDIALOG. Choose the one generated by the wizard. Its name is <project name>res.h, i.e. for a project named "test" we would have "testres.h".

2) The `STYLE` statement tells the interpreter which things should be done when creating the window. We will see later when we create a real window and not a dialog box window, that there can be quite a lot of them. In this case the style indicates the appearance (`DS_MODALFRAME`), that this window is visible, has a caption, and a system menu.

3) The `CAPTION` statement indicates just what character string will be shown in the caption.

4) In a similar way, the `FONT` statement tells the system to use Helv

5) The following statements enumerate the controls of the dialog box, and their descriptions are enclosed in a BEGIN/END block. We have two of them, a push button that is the default push button, and a normal pushbutton

6) the Cancel button. Both of them have a certain text associated with them, a set of coords as all controls, and an ID, that in the case of the OK button is the predefined symbol `IDOK`, with the numerical value of 1, and in the case of the Cancel button `IDCANCEL` (numerical value 2).

To convert this set of instruction in this language into a binary resource file that windows can interpret, we use a compiler called a *resource compiler*. Microsoft's one is called rc, Borland's one is called "brc", and lcc-win32's one is called lrc. All of them take this resource language with some minor extensions depending on the compiler, and produce a binary resource file for the run time interpreter of windows.

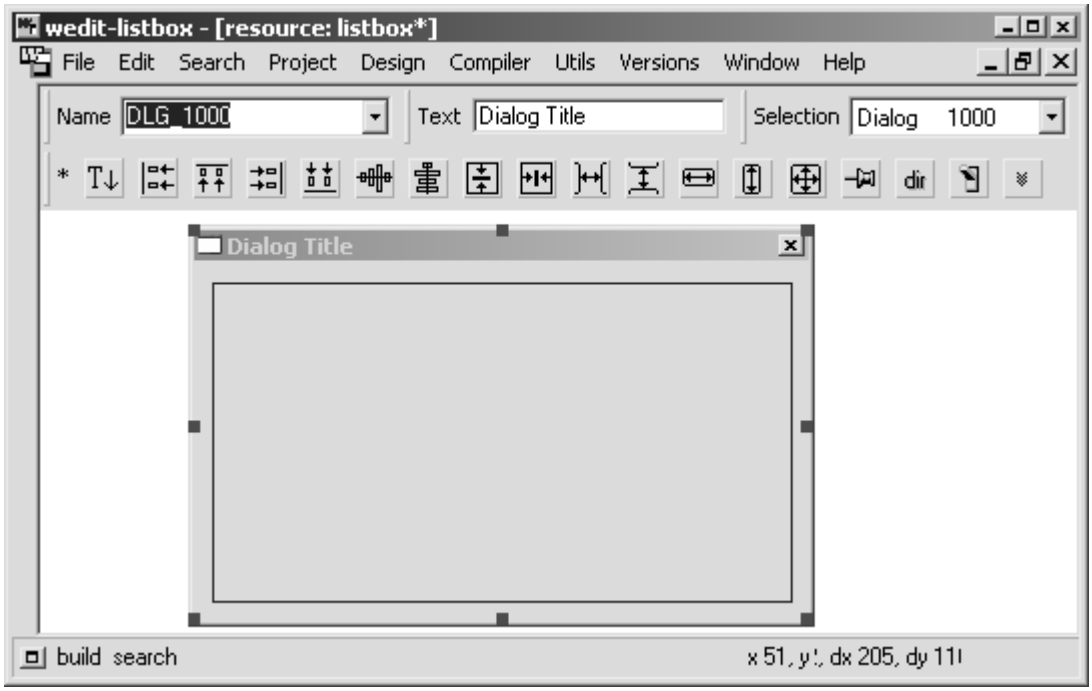
The resource compiler of lcc-win32 is explained in detail in the technical documentation, and we will not repeat that stuff again here. For our purposes it is enough to know that it is compatible with the other ones.

The binary resource files generated by the resource compiler are passed to the linker that converts them into resource specifications to be included in the executable.

Note that actually you do not need to know this language, because the IDE has a resource editor that can be used to build graphically using drag and drop the dialog box. But the emphasis here is to introduce you to the system so that you know not only what button should you push, but *why* you should push that button too.

But we wanted originally to make a dialog box containing an edit field. We are far away from our objective yet.

We come back to Wedit, after closing our text file “dialog.rc”, and we go to the “Design” menu bar and press “Open/new”. [92]The resource editor opens up, and we see the following display:



Near the top left border you see a button like this:

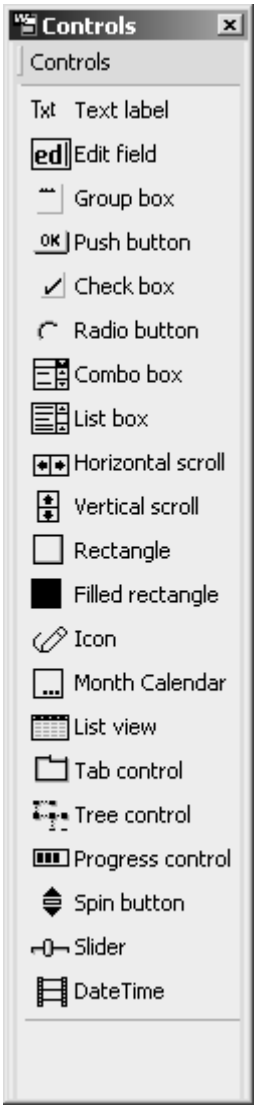
If you click in that button, the controls palette opens:

92. Why should introductory texts be clear?

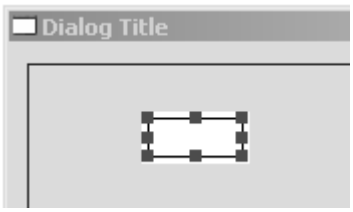
Why not make obfuscated sentences?

There is even an obfuscated C contest. Who writes the most incomprehensible program? Quite a challenge! With the documentation you can do even better: you write docs that are so incomprehensible that nobody reads them!

Controls palette



The whole operation of the editor is quite simple: The smaller window represents all the controls that you can put in a dialog box: entry fields, buttons, checkboxes, and several others. You select one button with the mouse, and drag it to your dialog box. There you drop it at the right position. To add an entry field then, we just push the edit field icon, the third one from the left in the upper row of icons, and drag it to our dialog box in the main wedit window. After doing this, it will look like this:



Our entry field becomes the selected item, hence the red handles around it. After resizing if necessary, we need to enter its identifier, i.e. the symbolic name that we will use in our program to refer to it.

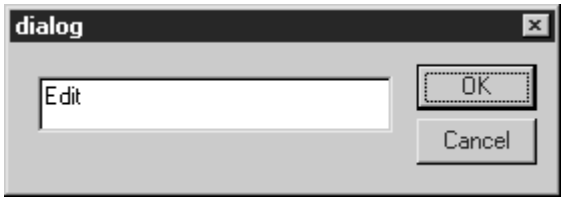
We will refer then in our program to this entry field with the name IDENTRYFIELD, maybe not a very cute name, but at least better than some bare number. The editor will write a

```
#define IDENTRYFIELD 101
```

in the generated header file.

The number 101 is an arbitrary constant, chosen by the editor. We resize the dialog box a bit (I like dialogs that aren't bigger than what they should be), and we press the "test" button.

We see a display like this:



We can enter text in the entry field, and pushing Cancel or OK will finish the test mode and return us to the dialog box editor.

OK, seems to be working. We save, and close the dialog box editor. We come back to our dialog procedure, where we will use this new entry field to get some text from the user.

2.1.3 The dialog box procedure

```
static BOOL CALLBACK DialogFunc(HWND hwndDlg, UINT msg, WPARAM wParam,
LPARAM lParam)
{
     switch (msg) {
     case WM_INITDIALOG:
          InitializeApp(hwndDlg,wParam,lParam);
          return TRUE;
     case WM_COMMAND:
          switch (LOWORD(wParam)) {
               case IDOK:
                    EndDialog(hwndDlg,1);
                    return 1;
               case IDCANCEL:
                    EndDialog(hwndDlg,0);
                    return 1;
          }
          break;
     case WM_CLOSE:
          EndDialog(hwndDlg,0);
          return TRUE;
     }
     return FALSE;
}
```

A dialog box procedure is called by the system. It has then, a fixed argument interface, and should return a predefined value. It receives from the system the handle of the dialog box window, the message, and two extra parameters.

Normally these procedures are a big switch statement that handles the messages the program is interested in. The return value should be TRUE if the dialog box procedure has handled the message passed to it, FALSE otherwise.

The general form of a switch statement is very simple: you have the switch expression that should evaluate to an integer and then you have several “cases” that should evaluate to compile time constants.

All those names in the switch statement above are just integers that have been given a symbolic name in windows.h using the preprocessor #define directive. A “break” keyword separates one “case” from the next one.

Note that in C a case statement can finish without a break keyword.

In that case (!) execution continues with the code of the next case. In any case, of course, a return statement finishes execution of *that* case, since control is immediately passed to the calling function.[93]

In this procedure we are interested in only three messages, hence we have only three “cases” in our switch:

1) `WM_INITDIALOG`. This message is sent after the window of the dialog box has been created, but before the dialog is visible in the screen. Here is done the initialization of the dialog box data structures, or other things. The wizard inserts here a call to a procedure for handling this message.

93. You put the cursor under the WM_INITDIALOG identifier and press F1.

2) WM_COMMAND. This message is sent when one of the controls (or child windows if you want to be exact) has something to notify to the dialog: a button has been pressed, a check box has been pressed, data has been entered in an entry field, etc. Since we can have several controls, we use again a switch statement to differentiate between them. Switch statements can be nested of course.

3) WM_CLOSE. This message arrives when the user has pressed the "close" button in the system menu, or has typed the Alt+F4 keyboard shortcut to close the dialog.

Now, the whole purpose of this exercise is to input a character string. The text is entered by the user in our entry field. It is important, from a user's perspective, that when the dialog box is displayed, the cursor is at the beginning of the entry field. It could be annoying to click each time in the entry field to start editing the text. We take care of this by forcing the *focus* to the entry field.

Under windows, there is always a single window that has the *focus*, i.e. receives all the input from the keyboard and the mouse. We can force a window to have the focus using the SetFocus API.

```
static int InitializeApp(HWND hDlg,WPARAM wParam, LPARAM lParam)
{
     SetFocus(GetDlgItem(hDlg,IDENTRYFIELD));
     return 1;
}
```

We add this call in the procedure InitializeApp. We test, and… it doesn't work. We still have to click in the edit field to start using it. Why?

Because, when we read the documentation of the WM_INITDIALOG message[94] it says:

```
WM_INITDIALOG
hwndFocus = (HWND) wParam; // handle of control to receive focus
lInitParam = lParam;       // initialization parameter
Parameters
hwndFocus
```

Value of wParam. Identifies the control to receive the default keyboard focus. *Windows assigns the default keyboard focus only if the dialog box procedure returns TRUE.*

Well, that is it! We have to return FALSE, and our SetFocus API will set the focus to the control we want. We change that, and… it works! Another bug is gone.[95]

Note that the SetFocus API wants a window handle. To get to the window handle of a control in the dialog box we use its ID that we took care of defining in the dialog editor. Basically we give to the SetFocus function the result of calling the API GetDlgItem. This is nice, since we actually need only one window handle, the window handle of the dialog box that windows gives to us, to get all other window handles of interest.

Now comes a more difficult problem. When the user presses the OK button, we want to get the text that he/she entered. How do we do that?

We have two problems in one: the first is to decide when we want to get the text, and the other is how to get that text.

---

94. This example shows you how to get rid of those problems, and the kind of problems you will encounter when programming under Windows. The only solution in most cases is a detailed reading of the documentation. Fortunately, Windows comes with a clear documentation that solves most problems.

95. All the notifications messages from edit fields begin with the EN_ prefix, meaning **E**dit field **N**otification.

For the first one the answer is clear. We want to read the text only when the user presses the OK button. If the Cancel button is pressed, or the window is closed, we surely aren't interested in the text, if any. We will read the text then when we handle the message that the OK button window sends to us when is pressed. We change our dialog procedure like this:

```
case WM_COMMAND:
     switch (LOWORD(wParam)) {
          case IDOK:
               ReadText(hwndDlg);
               EndDialog(hwndDlg,1);
               return 1;
          case IDCANCEL:
               EndDialog(hwndDlg,0);
               return 1;
     }
     break;
```

We add a call to a function that will get the text into a buffer. That function looks like this:

```
static char buffer[1024];
int ReadText(HWND hwnd)
{
     memset(buffer,0,sizeof(buffer));
     if (GetDlgItemText(hwnd,
                        IDENTRYFIELD,
                        buffer,
                        sizeof(buffer))) {
          return 1;
     }
     return 0;
}
```

We define a buffer that will not be visible from other modules, hence static. We set a fixed buffer with a reasonable amount of storage.

Our function cleans the buffer before using it, and then calls one of the workhorses of the dialog procedures: the API GetDlgItemText. This versatile procedure will put in the designated buffer, the text in a control window, in this case the text in the entry field. We again indicate to the API which control we are interested in by using its numerical ID. Note that GetDlgItemText returns the number of characters read from the control. If there isn't anything (the user pressed OK without entering any text), GetDlgItemText returns zero.

The first time that we do that; we will surely will want to verify that the text we are getting is the one we entered. To do this, we use the API MessageBox that puts up a message in the screen without forcing us to register a window class, define yet another window procedure, etc.

We add then to our window procedure, the following lines:

```
case IDOK:
     if (ReadText(hwndDlg)) {
          MessageBox(hwndDlg,buffer,
                          "text entered",MB_OK);
          EndDialog(hwndDlg,1);
     }
     return 1;
```

MessageBox takes a parent window handle, in this case the handle of the dialog box procedure, a buffer of text, a title of the message box window, and several predefined integer constants, that indicate which buttons it should show. We want just one button called OK, so we pass that constant.

Note too, that if the user entered no text, we do NOT call the EndDialog API, so the dialog box will refuse to close, even if we press the OK button. We *force* the user to enter some text before closing the dialog. Since we haven't changed anything in the logic for the Cancel button, the dialog box will still close when the user presses those buttons. Only the behavior of the OK button will change.

The EndDialog API takes two parameters: the dialog box window handle that it should destroy, and a second integer parameter. The dialog box will return these values as the result of the DialogBox call from WinMain remember?

Since WinMain returns itself this value as its result, the value returned by the DialogBox will be the return value of the program.

2.1.4 A more advanced dialog box procedure

Doing nothing and not closing the dialog box when the user presses OK is not a good interface. You expect a dialog box to go away when you press OK don't you?

A user interface like this makes for an unexpected behavior. Besides, if we put ourselves in the user's shoes, how can he/she find out what is wrong? The software doesn't explain anything, doesn't tell the user what to do to correct the situation, it just silently ignores the input. This is the worst behavior we could imagine.

Well, there are two solutions for this. We can disable the OK button so that this problem doesn't appear at all, or we could put up a message using our MessageBox API informing the user that a text must be entered.

Let's see how we would implement the first solution.

To be really clear, the OK button should start disabled, but become active immediately after the user has typed some text. If, during editing, the user erases all text that has been entered, the OK button should revert to its inactive state.

We can do this by processing the messages that our edit field sends to us. Edit fields are very sophisticated controls, actually a full-blown mini-editor in a small window. Each time the user types anything in it, the edit field procedure sends us a WM_COMMAND message, informing us of each event.

We change our dialog procedure as follows:

```
case WM_COMMAND:
     switch (LOWORD(wParam)) {
          case IDOK:
               // suppressed, stays the same
          case IDCANCEL:
               EndDialog(hwndDlg,0);
               return 1;
          case IDENTRYFIELD:
               switch (HIWORD(wParam)) {
                    case EN_CHANGE:
                         if (GetDlgItemText( hwndDlg, IDENTRYFIELD,
                             buffer,sizeof(buffer))) {
                              EnableWindow(
                              GetDlgItem(hwndDlg,IDOK),1);
                         }
                         else
                             EnableWindow(
                              GetDlgItem(hwndDlg,IDOK),0);
                         break;
                    }
                    break;
```

```
            }
            break;
```

We add a new case for this message. But we see immediately that this nested switch statements are getting out of hand. We have to split this into a function that will handle this message. We change again our dialog box procedure as follows:

```
        case IDCANCEL:
             EndDialog(hwndDlg,0);
             return 1;
        case IDENTRYFIELD:
             return EntryFieldMessages(hwndDlg,wParam);
```

This is much clearer. We put the code for handling the entry field messages in its own procedure, “EntryFieldMessages”. Its code is:

```
int EntryFieldMessages(HWND hDlg,WPARAM wParam)
{
     HWND hIdOk = GetDlgItem(hDlg,IDOK);

     switch (HIWORD(wParam)) {
     case EN_CHANGE:
          if (GetDlgItemText(hDlg,IDENTRYFIELD,
               buffer, sizeof(buffer))) {
               // There is some text in the entry field. Enable the IDOK button.
               EnableWindow(hIdOk,1);
          }
          else // no text, disable the IDOK button
               EnableWindow(hIdOk,0);
          break;
     }
     return 1;
}
```

Let’s look at this more in detail. Our switch statement uses the HIWORD of the first message parameter. This message carries different information in the upper 16 bits (the HIWORD) than in the lower 16 bits (LOWORD). In the lower part of wParam we find the ID of the control that sent the message, in this case IDENTRYFIELD, and in the higher 16 bits we find which sub-message of WM_COMMAND the control is sending to us, in this case EN_CHANGE[96], i.e. a change in the text of the edit field.

There are many other notifications this small window is sending to us. When the user leaves the edit field and sets the focus with the mouse somewhere else we are notified, etc. But all of those notifications follow the same pattern: they are sub-messages of WM_COMMAND, and their code is sent in the upper 16 bits of the wParam message parameter.

Continuing the analysis of EntryFieldMessages, we just use our friend GetDlgItemText to get the length of the text in the edit field. If there is some text, we enable the IDOK button with the API EnableWindow. If there is NO text we disable the IDOK button with the same API. Since we are getting those notifications each time the user types a character, the reaction of our IDOK button will be immediate.

But we have still one more thing to do, before we get this working. We have to modify our InitializeApp procedure to start the dialog box with IDOK disabled, since at the start there is no text in the entry field.

```
static int InitializeApp(HWND hDlg,WPARAM wParam, LPARAM lParam)
{
```

96. I remember calling the software support when installing some hardware: many of the options of the installation software were disabled but there was no way of knowing why.

```
        SetFocus(GetDlgItem(hDlg,IDENTRYFIELD));
        // Disable the IDOK button at the start.
        EnableWindow(GetDlgItem(hDlg,IDOK),0);
        return 1;
    }
```

We recompile, and it works. The OK button starts disabled (grayed), and when we type the first character it becomes active, just as we wanted. When we select all the text in the entry field, and then erase it, we observe that the button reverts to the inactive state.

2.2 User interface considerations

There was another way of informing the user that text must be entered: a MessageBox call, telling him/her precisely what is wrong. This alternative, making something explicit with a message, or implicit, like the solution we implemented above appears very often in windows programming, and it is very difficult to give a general solution to it. It depends a lot of course, upon the application and its general style. But personally, *I prefer explicit error messages rather than implicit ones*. When you receive an error message, you know exactly what is wrong and you can take easily steps to correct it. When you see a menu item disabled, it is surely NOT evident what the hell is happening and why the software is disabling those options.[97]

But there are other user-interface considerations in our dialog box to take into account too.

One of them is more or less evident when you see how small the letters in the edit field are. Dialog boxes use a default font that shows very thin and small characters. It would be much better if we would change that font to a bigger one.

In the initialization procedure, we set the font of the edit field to a predefined font. Windows comes with several predefined items, ready for you to use without much work. One of them is the system font that comes in two flavors: monospaced, and proportional. We use the monospaced one. Our initialization procedure then, looks now like this:

```
static int InitializeApp(HWND hDlg,WPARAM wParam, LPARAM lParam)
{
     HFONT font;

     font = GetStockObject(ANSI_FIXED_FONT);
     SendDlgItemMessage(hDlg,IDENTRYFIELD,
               WM_SETFONT,(WPARAM)font,0);
     EnableWindow(GetDlgItem(hDlg,IDOK),0);
     SetFocus(GetDlgItem(hDlg,IDENTRYFIELD));
     return 1;
}
```

A HFONT is a font “handle”, i.e. an integer that represents a font for windows. We get that integer using the GetStockObject API. This function receives an integer code indicating which object we are interested in and returns it. There are several types of object we can get from it: fonts, brushes, pens, etc.[98]

Yet another point missing in our dialog box is a correct title, or prompt. The title of our dialog is now just “dialog”. This tells the user nothing at all. A friendlier interface would tell the user

97. Now is a good time to read the documentation for that API. It will not be repeated here.

98. We introduced that to see if we were really getting a string. Since now we are returning that data to the calling program, that message should disappear.

what data the software is expecting from him/her. We could change the title of the dialog to a more meaningful string.

The program calling our dialog procedure could give this string as a parameter to the dialog box. Dialog boxes can receive parameters, as any other procedure. They receive them in the parameters passed to the WM_INITDIALOG message.

A closer look to the documentation of the WM_INITDIALOG message tell us that the lParam message parameter contains for the WM_INITDIALOG 32 bits of data passed in the last parameter of an API called DialogBoxParam.

We have to modify our calling sequence to the dialog, and instead of using DialogBox we use the DialogBoxParam API. Looking into our program, we see that the DialogBox API was called in our WinMain function (see above). We should modify this call then, but a new problem appears: where does WinMain know which string to pass to the DialogBoxParam API?

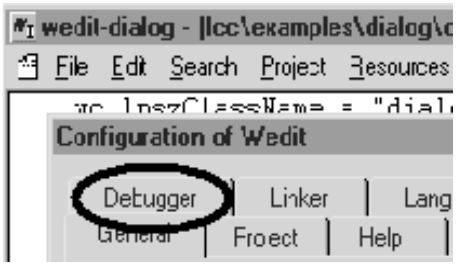
Well, we could decide that this string would be the parameters passed to WinMain in the lpCmdLine argument. This is the most flexible way. We modify then the call to DialogBox like follows:

```
return DialogBoxParam (hinst,
     MAKEINTRESOURCE(IDD_MAINDIALOG),
     NULL, (DLGPROC) DialogFunc,
     (int)lpCmdLine);
```

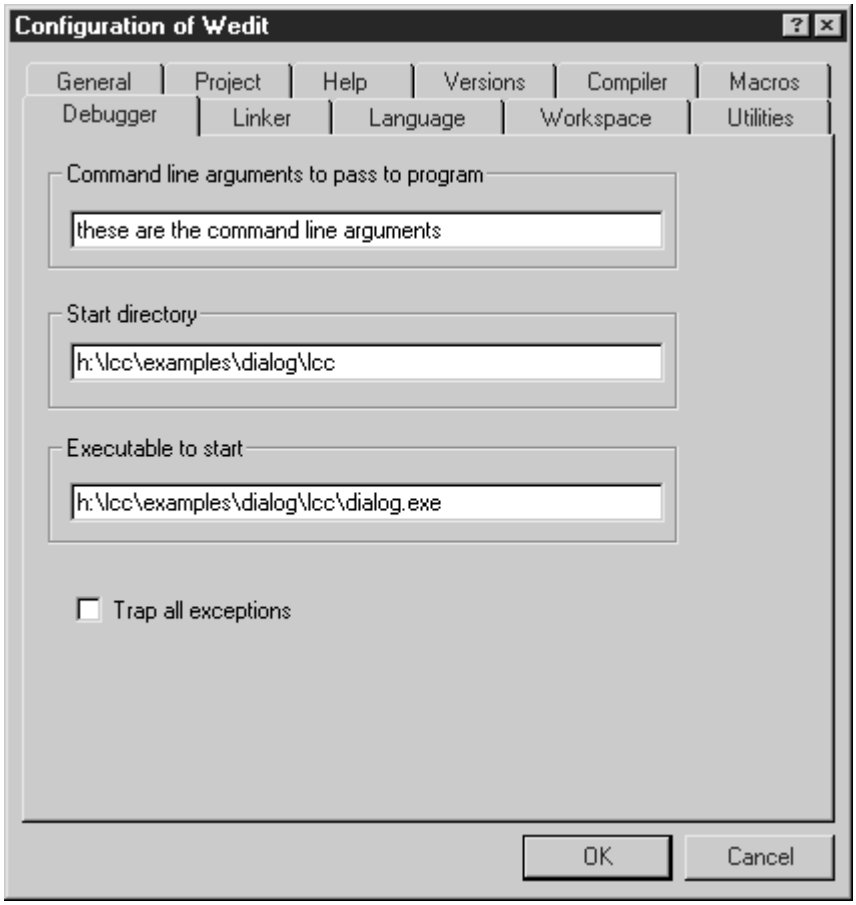
Since our dialog box is now constructed with DialogBoxParam, we receive in the lParam message parameter the same pointer that we gave to the DialogBoxParam API. Now, we have just to set that text in the caption of our dialog box and it's done. We do that (again) in our initialization procedure by adding:

```
SetWindowText(hDlg, (char *)lParam);
```

The SetWindowText API sets the caption text of a window, if that window has a caption bar of course. To test this, we have to tell Wedit to pass a command line argument to the program when it calls the debugger or executes the program with Ctrl+F5. We do this by selecting the "debugger" tab in the configuration of wedit:

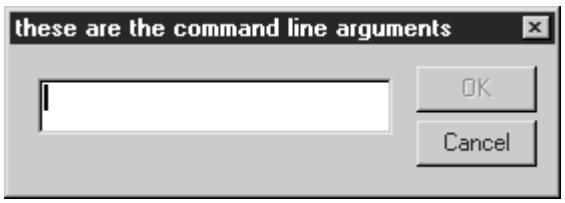


The debugger tab is in the upper left corner. When we select it, we arrive at the following tab:



Note the first line “Command line arguments to pass to program”. There, we write the string that we want shown in the dialog box.

When now we press Ctrl+F5, we see our dialog box like this:



Nice, we can pass our dialog box a “prompt” string. This makes our dialog box more useful as a general input routine. Remember that the objective of this series of sections was to introduce you a general routine to input a character string from the user. We are getting nearer.

Still, there is one more problem that we haven’t solved yet. We have a buffer of a limited length, i.e. 1024 characters. We would like to limit the text that the user can enter in the dialog box so that we avoid overflowing our buffer. We can do this with the message `EM_SETLIMITTEXT`. We have to send this message to the control when we start the dialog box, so that the limit will be effective before the user has an occasion of overflowing it. We add then

```
SendDlgItemMessage(hDlg,IDENTRYFIELD,
                   EM_SETLIMITTEXT,512,0);
```

2.3 Libraries

What we would like is a way of using this dialog box in our applications of course. How could we do that?

One way would be to call it as an independent program. We could use the facilities for calling a program within the windows system, and pass our prompt in the command line parameters. This would work, but the problem of getting the string from the user would be quite complicated to solve. Programs can only return an error code, and in some situations this error code can only be from zero to 255… We can't pass pointers just like that from one program to another; we can't just pass a pointer to a character string as the result of the program.

Why?

Because windows, as other systems like linux, Solaris, and UNIX in general, uses a protected virtual address schema. The machine addresses that the program uses are virtual, as if the program was the only one running in the machine. It is the operating system and the CPU that does the translation of those virtual addresses into real RAM locations in the machine you are using. This means the addresses of each program aren't meaningful for another program. We can pass special pointers (shared memory) within windows, but that's too advanced stuff for an introductory text, sorry.

But there are many other ways of solving this problem without costly interface development. What do we want? Let's get that clear first, worrying about implementation details later.

```
int GetString(char *prompt, char *buffer,int bufferlen);
```

This routine would return either true or false, depending if the user has pressed OK or cancelled the operation. If the user pressed OK, we would find the string that was entered in the buffer that we pass to GetString. To avoid any overflow problems, we would pass the length of the character string buffer, so that the GetString routine stops input when we reach that limit.

The C language supports code reuse. You can compile code that is useful in many situations and build libraries of routines that can be reused over and over again. The advantages are many:

- The code has to be written once and debugged once.
- The size of our program stays small, since we call a routine instead of repeating the code all over the place.

Function calls are the mechanism of code reuse since a function can be used in many situations. Libraries are just a collection of routines that are linked either directly or indirectly with the main program.

From the standpoint of the user of the library, not the one who is building it, the usage is quite simple:

1) You have to include the header file of the library to make the definition available to the compiler.

2) You use the functions or data

3) You add the library to the set of files that the linker uses to build your program.

This simplicity of usage makes libraries a good way to reuse and share the code between different applications.

Under windows we have two types of libraries:

Static libraries. These libraries are built in files that normally have the .lib extension and are linked with the program directly, i.e. they are passed to the linker as arguments. The linker takes the code of the needed functions from the library and copies the code into the program.

Dynamic libraries. These aren't copied into the main program, but are resolved at load time by the program loader. When you double-click a program's icon you activate a system program of windows called *program loader* that goes to the disk, finds the code for the executable you have associated with the icon, and loads it from disk into RAM. When doing this, the loader finds if the program needs any dynamic libraries, that normally have the .DLL extension, and reads their code too, linking it dynamically to the program being loaded.

Which one should we use in this application?

Static or not static? That is the question!

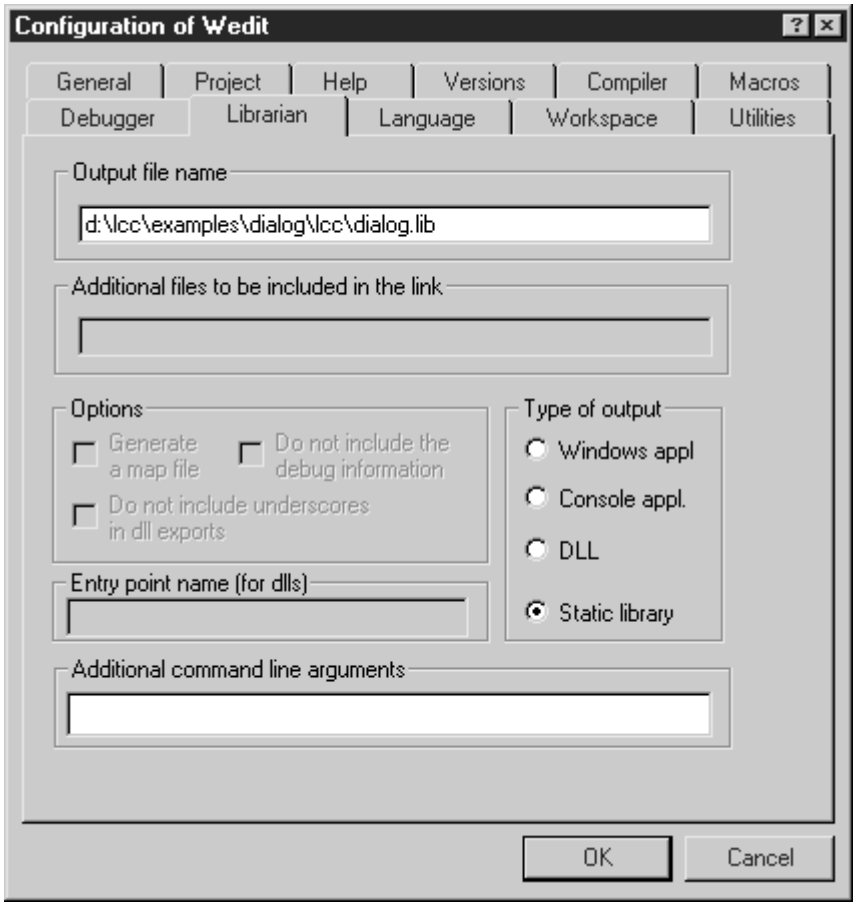
Our program needs a class registration before it can call the DialogBoxParam API. If we use the static library approach, we would have to require that the user of the library calls some initialization routine before, to allow us to register our class with windows.

But this would complicate the interface. We introduce with this requirement yet another thing that can go wrong with the program, yet another thing to remember.

A way out of this dilemma would be to take care of doing the registration automatically. We could setup an integer variable that would start as zero. Before calling our DialogBoxParam procedure we would test the value of this variable.

If it is zero it means that our class wasn't registered. We would register our class and set this variable to one, so that the next call finds a value different than zero and skips the class registration code.

We have to tell the IDE to produce a static library now, instead of a normal executable file. We do this by going into the linker configuration tab, and checking the library radio-button, like this:



You see the "Static library" radio-button checked. The name of the library is the name of the project with the .lib extension.

Now we are ready to change our WinMain. We change our WinMain function to be the GetString procedure, like this:

```
static char buffer[1024];

static int classRegistered;

int APIENTRY GetString(char *prompt, char *destbuffer,int bufferlen)
{
     WNDCLASS wc;
     int result;
     HANDLE hinst;

     hinst = GetModuleHandle(NULL);
     if (classRegistered == 0) {
          memset(&wc,0,sizeof(wc));
          wc.lpfnWndProc = DefDlgProc;
          wc.cbWndExtra = DLGWINDOWEXTRA;
          wc.hInstance = hinst;
          wc.hCursor = LoadCursor(NULL, IDC_ARROW);
          wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
          wc.lpszClassName = "dialog";
          RegisterClass(&wc);
          classRegistered = 1;
     }
```

```
        result = DialogBoxParam(hinst,
                        MAKEINTRESOURCE(IDD_MAINDIALOG),
                        NULL,
                        (DLGPROC) DialogFunc,
                        (int)prompt);

        if (result == 1) {
             strncpy(destbuffer,buffer,bufferlen-1);
             destbuffer[bufferlen-1] = 0;
        }
        return result;
    }
```

We have several things to explain here.

1) We move the declaration of our static buffer that was before further down, to the beginning of the file, so that we can use this buffer in the GetString procedure to copy its contents into the destination buffer.

2) We declare our flag for testing if the class has been registered as a static int, i.e. an integer visible only in this module. We do not need to initialize it to zero, since the C language guarantees that all non-explicitly initialized static variables will be set to zero when the program starts.

3) We modify the declarations of local variables in the GetString procedure, adding a result integer variable, and a HANDLE that will hold the instance of the current module. Before, we received this as a parameter in the arguments of WinMain, but now we have to get it by some other means. The solution is to call the GetModuleHandle API, to get this. We indicate it that we want the handle of the currently running executable by passing it a NULL parameter.

4) We test then our global flag classRegistered. If it is zero, we haven't registered the class, and we do it now. Afterwards, we set the variable to one, so that this piece of code will not be executed again.

5) We call our DialogBox procedure just like we did before, but now we assign its result to an integer variable and we test if the result is one (i.e. the user pressed OK). If that is the case, we copy the string from the temporary buffer to the destination buffer. Note that we use the strncpy function. This standard library function takes an extra parameter, a maximum length to copy. We do not want to overflow the destination buffer under any circumstances, so we only copy a maximum of bufferlen characters minus one, to account for the terminating zero of the string. We ensure afterwards that the string is zero terminated, and we return the result.

The rest of the program remains the same, so it is not shown. It is important to remember to get rid of that MessageBox call however![99]

We compile the library, and we want to test it, but… we need a test program. A library is not an executable by itself.

Besides this, we need a header file that the user of the library will use to get the prototype of our function. Its contents are very simple:

```
int GetString(char *prompt, char *destBuffer,int bufferlen);
```

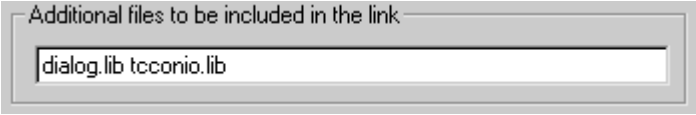
and that's all.

---

99. Well, this is not great deal; we have just to answer YES when it proposes to create the skeleton.

Now, we have to define a new project that will contain the test code. We do that in the same way as we have created the other projects: we choose the 'Create project' option in the 'Project' menu bar, and we name it appropriately "testdialog".

We do NOT specify a windows application. Since our library should be independent whether it is called from a windows program or a console application, we should test that now.



Now, when creating the project, we ask the wizard to create a console application.[100] We leave everything by default, but when we arrive at the linker settings dialog, we add our dialog.lib to the libraries entry field, like this:

Another issue to remember, is the following:

We need a resource file. Since in our library there are no resources, we have to add those resources to our test program. This is easy to do in this case: we just add the dialog.rc resource file to the project. The interface for the users of the library however is terrible. All programs that use our library will be forced to include somehow the resource for the dialog box! Including a resource in another resource file is difficult, to say the least.

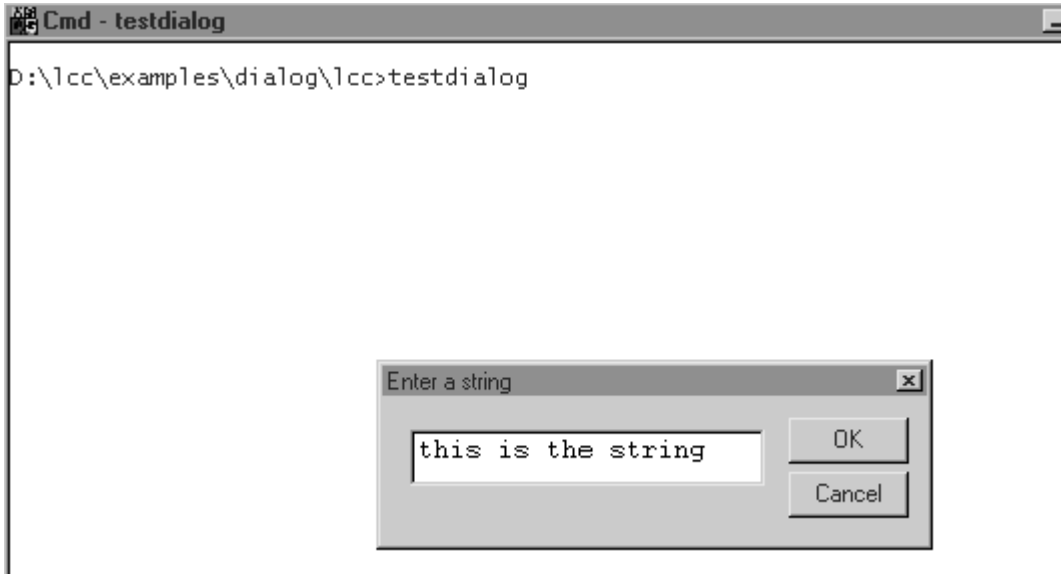
Wow, this looks like a showstopper actually.

OK. This presents an unexpected and serious trouble for our library project, but we will not leave things at midway. We finish our test program by changing our "main" function, like this:

```
extern int APIENTRY GetString(char *prompt,char *buf,int len);

int main(void)
{
        char buffer[1024];
        if (GetString("Enter a string",buffer,sizeof(buffer))) {
               printf("String is %s\n",buffer);
        }
        else printf("User cancelled!\n");
        return 0;
}
```

100. It could be argued that this could be done with DLLs too: the linker should export all externally visible symbols. In practice is better only to export symbols that should be visible. This avoids name clashes.

When we start this program from the command line, we see:



This isn't that bad, for a start. We have a sophisticated line editor, complete with arrow interface to move around in the text, clipboard support built in, delete and backspace keys already taken care of, etc. If we would have to write ourselves an equivalent program, it would cost us probably days of development. To develop the clipboard interface already is quite a challenge.

But we are confronted to the problem of resources. Our static library idea was a dead end. We have to find something else.

Summary: The C language supports the concept of code reuse in the form of libraries. The static libraries are combined with the main application at link time (statically). They can't contain resources.

2.4 Dynamically linked libraries (DLLs)

A dynamically linked library is just like a static library: it contains a set of useful functions that can be called from other software. As with normal .lib libraries, there is no main function.

Unlike static libraries however, they have several features that make them a very interesting alternative to static libraries:

- When they are loaded, the loader calls a function of the library to allow load time initializations. This allows us to register our class, for instance, or do other things.
- When the program that loads them starts or ends a new thread the system arranges for calling the same function. This allows us to take special actions when this event occurs. We do not need this feature for our application here, but other software do.
- When the library is unloaded, either because the program explicitly does it or because simply the program is finished, we get notified. Here we can reverse the actions we performed when the library was loaded: we can, for instance, unregister our window class.
- DLLs can contain resources. This solves the problem of forcing the user of the library to link a bunch of resources to his/her program.

DLLs need to specify which functions are going to be exported, i.e. made visible to the outside world. With static libraries this is not really necessary since the librarian will write all the symbols with the external type to the library symbol table automatically.[101]

We can declare that a symbol will be exported using two methods:

1) We can put in the declaration of the symbol the __declspec(dllexport) mark.

2) We can write the name of the symbol in a special file called definitions file (with the .def extension) and pass this file to the linker.

Which method you use is a matter of taste. Writing __declspec(dllexport) in the source code is quite ugly, and may be non-portable to other systems where the conventions for dynamically linked code may be completely different. A definitions file spares us to hard wire that syntax in the source code.

The definitions file has its drawbacks too however. We need yet another file to maintain, another small thing that can go wrong.

For our example we will use the __declspec(dllexport) syntax since we have only one function to export.

We return to our library project, and reopen it. We go again to the linker configuration tab, that now is called “librarian” since we are building a static library, and we check the radio-button corresponding to a DLL project. We answer yes when Wedit says whether it should rebuild the makefile and there we are. Now we have to make the modifications to our small library.

We have to define a function that will be called when the library is loaded. Traditionally, the name of this function has been LibMain since the days of Windows 3.0 or even earlier. We stick to it and define the following function:

```
int WINAPI LibMain(HINSTANCE hDLLInst, DWORD Reason, LPVOID Reserved)
{
    switch (Reason)
    {
```

101. You can change this by pressing the corresponding button in the linker configuration tab, or by giving the argument –nounderscores to the linker, when building the DLL.

```
            case DLL_PROCESS_ATTACH:
                  hinst = hDLLInst;
                  DoRegisterClass();
                  break;
            case DLL_PROCESS_DETACH:
                  UnregisterClass("dialog",hDLLInst);
                  break;
            case DLL_THREAD_ATTACH:
                  break;
            case DLL_THREAD_DETACH:
                  break;
        }
        return TRUE;
    }
```

This function, like our dialog function or many other functions under windows, is a *callback* function, i.e. a function that is called by the operating system, not directly from our code. Because of this fact, its interface, the arguments it receives, and the result it returns is fixed. The operating system will always pass the predefined arguments to it, and expect a well-defined result.

The arguments that we receive are the following:

1) We receive a `HANDLE` to the instance of the DLL. Note that we have to pass to several functions this handle later on, so we will store it away in a global variable.

2) We receive a `DWORD` (an unsigned long) that contains a numerical code telling us the reason why we are being called. Each code means a different situation in the life cycle of the DLL. We have a code telling us that the we were just loaded (`DLL_PROCESS_ATTACH`), another to inform us that we are going to be unloaded (`DLL_PROCESS_DETACH`), another to inform us that a new thread of execution has been started (`DLL_THREAD_ATTACH`) and another to tell us that a thread has finished (`DLL_THREAD_DETACH`).

3) The third argument is reserved by the system for future use. It is always zero.

The result of LibMain should be either TRUE, the DLL has been correctly initialized and loading of the program can continue, or zero meaning a fatal error happened, and the DLL is unable to load itself.

Note that we return always TRUE, even if our registration failed.

Why?

If our registration failed, this module will not work. The rest of the software could go on running however, and it would be too drastic to stop the functioning of the whole software because of a small failure in a routine that could be maybe optional.

Why the registration of our class could fail?

One of the obvious reasons is that the class is already registered, i.e. that our calling program has already loaded the DLL, and it is loading it again. Since we do not unregister our class, this would provoke that we try to register the class a second time.

For the time being, we need to handle only the event when the DLL is loaded or unloaded. We do two things when we receive the `DLL_PROCESS_ATACH` message: we store away in our global variable the instance of the DLL, and then we register our string dialog box class. We could have just done it in the LibMain function, but is clearer to put that code in its own routine. We write then:

```
static void DoRegisterClass(void)
```

```
{
     WNDCLASS wc;

     memset(&wc,0,sizeof(wc));
     wc.lpfnWndProc = DefDlgProc;
     wc.cbWndExtra = DLGWINDOWEXTRA;
     wc.hInstance = hinst;
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
     wc.lpszClassName = "dialog";
     RegisterClass(&wc);
}
```

You see that the code is the same as the code we had originally in WinMain, then in our GetString procedure, etc.

To finish LibMain, we handle the message `DLL_PROCESS_DETACH` unregistering the class we registered before.

With this, our GetString procedure is simplified: We do not need to test our flag to see if the class has been registered any more. We can be certain that it was.

```
int APIENTRY __declspec(dllexport)
GetString(char *prompt, char *destbuffer,int bufferlen)
{
     int result;

     result = DialogBoxParam(hinst,
                  MAKEINTRESOURCE(IDD_MAINDIALOG),
                  NULL,
                  (DLGPROC) DialogFunc,
                  (int)prompt);

     if (result == 1) {
          strncpy(destbuffer,buffer,bufferlen-1);
          destbuffer[bufferlen-1] = 0;
     }
     return result;
}
```

We compile and link our DLL by pressing F9. Note that when we are building a DLL, lcc-win32 will generate three files and not only one as with a normal executable.

1) We obtain of course a dialog.dll file that contains the DLL.

2) We obtain an import library that allows other programs to be linked with this DLL. The name will be dialog.lib, but this is not a normal library. It is just a library with almost no code containing stubs that indicate the program loader that a specific DLL is used by the program.

3) We obtain a text file called dialog.exp that contains in text form the names that are exported from the DLL. If, for any reason we wanted to regenerate the import library for the DLL, we could use this file together with the buildlib utility of lcc-win32 to recreate the import library. This can be important if you want to modify the names of the exported functions, establish synonyms for some functions or other advanced stuff.

2.5 Using a DLL

To use our newly developed DLL we just plug-in the older test program we had done for our static library. The interface is the same; nothing has changed from the user code of our library. The only thing that we must do differently is the link step, since now we do not need to add the resource file to our program.

Wedit leaves our DLL in the lcc directory under the project main directory. We just recompile our testdialog.c program that we had before. Here is the code for testdialog.c again:

```
extern int APIENTRY GetString(char *prompt,char *buffer,int len);

int main(int argc,char *argv[])
{
     char buffer[1024];

     if (GetString("Enter a string",
                   buffer,sizeof(buffer))) {
          printf("String is %s\n",buffer);
     }
     else printf("User cancelled\n");
     return 0;
}
```

We compile this in the directory of the project, without bothering to create a project. Suppose that our project is in

```
h:\lcc\projects\dialog
```

and the dll is in

```
h:\lcc\projects\dialog\lcc
```

We compile with the command:

```
lcc testdialog.c
```

then we link with the command:

```
lcclnk testdialog.obj lcc\dialog.lib
```

Perfect! We now type the name of the program to test our dll.

testdialog

but instead of the execution of the program we see a dialog box like this:

The system tells us that it can't find the dll.

Well, if you reflect about this, this is quite normal. A DLL must be linked when the execution of the program starts. The system will search in the start directory of the program, and in all directories contained in the PATH environment variable. If it doesn't find a "dialog.dll" anywhere it will tell the user that it can't execute the program because a missing DLL, that's all.

The solution is to copy the DLL into the current directory, or copy the DLL in one of the directories in your PATH variable. Another solution of course is to go to the directory where the DLL is, and start execution of the program there.

This dependency on the DLL is quite disturbing. All programs that use the DLL in this fashion would need to have the DLL in their startup directory to be able to work at all.

A way to get rid of this is to avoid linking with the DLL import library. Yes you will say, but how will we use the DLL?

DLLs can be loaded into the program's address space with the API LoadLibrary. This API will do what the program loader does when loading a program that contains a reference to a DLL. If the load succeeds, the API will return us a handle to the library, if not, it will return us an INVALID_HANDLE as defined in windows.h.

After loading the DLL, we can get the address of any exported function within that DLL just by calling another windows API: GetProcAddress. This API receives a valid DLL handle, and a character string containing the name of the function to find, and will return an address that we can store in a function pointer.

Let's do this.

```
#include <windows.h>                                    (1)
#include <stdio.h>                                      (2)
int (APIENTRY *pfnGetString)(char *,char *,int);        (3)

int main(int argc,char *argv[])
{
     char buffer[1024];
     HANDLE dllHandle = LoadLibrary(                   (4)
          "h:\\lcc\\examples\\dialog\\lcc\\dialog.dll");

     if (dllHandle == INVALID_HANDLE_VALUE) {           (5)
          fprintf(stderr,"Impossible to load the dll\n");
          exit(0);
     }
     pfnGetString = (int (APIENTRY *)(char *,char *,int))
          GetProcAddress(dllHandle,"_GetString@12");   (6)
     if (pfnGetString == NULL) {
          fprintf(stderr,
         "Impossible to find the procedure GetString\n");
          exit(1);
     }
     if (pfnGetString(
          "Enter a string",buffer,sizeof(buffer))) {
          printf("String is %s\n",buffer);
     }
     else printf("User cancelled\n");
     return 0;
}
```

We go step by step:

We need to include windows.h for getting the prototypes of LoadLibrary, and GetProcAddress, besides some constants like `INVALID_HANDLE_VALUE`.

stdio.h is necessary too, since we use fprintf

This is a function pointer called pfnGetString, that points to a function that returns an int and takes a char *, another char * and an int as arguments. If you do not like this syntax please bear with me. Even Dennis Ritchie says this syntax isn't really the best one.

We store in our dllHandle, the result of calling LoadLibrary with an absolute path so it will always work, at least in this machine. Note that the backslashes must be repeated within a character string.

We test if the handle received is a correct one. If not we put up some message and we exit the program.

We are now ready to assign our function pointer. We must cast the return value of GetProcAddress to a function like the one we want. The first part of the statement is just a cast, using the same construction that the declaration of the function pointer before, with the exception that we do not include the identifier of course, since this is a cast. But the arguments to `GetProcAddress` are weird. We do not pass really the name of the function `GetString`, but a name `_GetString@12`. Where does this name come from?

The rest of the program stays the same.

To understand where this weird name comes from, we have to keep in mind the following facts:

lcc-win32 like many other C compilers, adds always an underscore to the names it generates for the linker.[102]

Since our function is declared as _stdcall, windows conventions require that we add to the name of the function the character '@' followed by the size of the function arguments. Since our function takes a char pointer (size 4) another char pointer, and an integer (size 4 too), we have 12 bytes of procedure arguments, hence the 12. Note that all types smaller than an integer will be automatically be promoted to integers when passing them to a function to keep the stack always aligned, so that we shouldn't just take the size of the arguments to make the addition. All of this can become really complicated if we have structures that are pushed by value into the stack, or other goodies.

The best thing would be that our DLL would export _GetString@12 as GetString. PERIOD.

Well, this is exactly where our dialog.def file comes handy. Here is a dialog.def that will solve our problem.

```
LIBRARY dialog
EXPORTS
_GetString@12=GetString
```

We have in the first line just the name of the DLL in a LIBRARY statement, and in the second line two names. The first one is the name as exported by the compiler, and the second one is the name as it should be visible from outside. By default, both are the same, but now we can separate them. With these instructions, the linker will put in the export table of the DLL the character string "GetString", instead of the compiler-generated name.[103]

Once this technical problems solved, we see that our interface is much more flexible now. We could just return FALSE from our interface function if the DLL wasn't there, and thus disable some parts of the software, but we wouldn't be tied to any DLL. If the DLL isn't found, the functionality it should fulfill can't be present but nothing more, no catastrophes.

---

102. In the documentation of windows, you will find out that in the .def file you can write other kinds of statements. None of them are supported by lcc-win32 and are silently ignored since they are redundant with the command line arguments passed to the linker. Do not write anything else in the .def file besides the names of the exports.

103. The Macintosh works in the same manner, albeit with a much more primitive system.

2.6 A more formal approach.

2.6.1 New syntax

Now that we know how to make a DLL, we should be able to understand more of C, so let's come back again to the syntax and take a closer look.

A problem area is function pointer casting, what leads straight into gibberish-looking code.

```
int (APIENTRY *pfn)(char *,char *,int);
```

Better is to typedef such constructs with:

```
typedef int (APIENTRY *pfn)(char *,char *,int);
```

Now we declare our function pointer just with

```
pfn pfnGetString;
```

and we can cast the result of GetProcAddress easily with:

```
pfnGetString = (pfn)GetProcAddress( ... );
```

Hiding the gibberish in a typedef is quite useful in this situations, and much more readable later.

2.6.2 Event oriented programming

In another level, what those programs show us is how to do event-oriented programs, i.e. programs designed to react to a series of events furnished by the event stream.

Under windows, each program should make this event-stream pump turn, by writing somewhere:

```
while (GetMessage()) {
     ProcessMessage();
}
```

we will describe the exact syntax later.[104] This message-pump is hidden now from view under the DefDlgProc procedure, but it is the source of all the messages passed to the dialog procedure that we defined.

A windows program is designed to react to those messages, or events. It will process them in sequence, until it decides to stop the message pump ending the program.

The general structure is then:

---

104. Application frameworks like MFC introduce an additional simplifying layer between you and the operating system. Much has been said and written about them, and here I will not discuss this in much more detail. Suffice to note that the purpose of lcc-win32 is to let you be always in control of what is going on. You can do here anything, contrary to a framework, where you can only do what the framework provides for, and nothing else.

True, an application framework can simplify the coding, and many people use them. It would be feasible to build such a framework with lcc-win32, but … I will leave this problem "as an exercise to the reader"…

- Initialize the application, register the window classes, etc.
- Start the message pump
- Process events until a predefined event (generally the closing of the main window) provokes the stop of the message pump.
- Cleanup

This was the standard way of windows programming until the C++ "wizards" decided that this message pump was too ugly to be shown to programmers and hid it behind an "application object". Later, several years later, things changed again and the ATL environment made the famous "message pump" visible again. [105]

Texts started appearing explaining the surprised programmers what was WinMain and that "message pump" that was again the non-plus-ultra of modernity.[106] Luckily for people programming in C, all those fads were invisible. Since C programming emphasizes low-level knowledge of what is going on, the "message pump" has always been there and we knew about it.

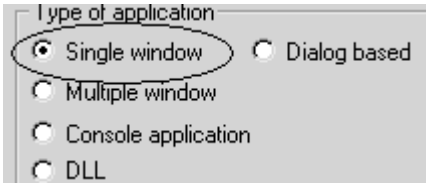
Messages are sent to window objects that are members of a class of similar objects or *window class*. The procedure SendMessage can also send messages to a window object, as if they would have been sent by the system itself. We send, for instance, the message EM_SETLIMITTEXT to an entry field to set the maximum number of characters that the entry field will process.

The system relies on the window procedure passing all messages that it doesn't handle to the default message procedure for that class of objects, to maintain a coherent view of the desktop and to avoid unnecessary code duplication at each window.

We are now ready to start our next project: building a real window, not just a simple dialog.

2.7 A more advanced window

We will create the next example with the wizard too. It is a handy way of avoiding writing a lot of boilerplate code. But it is obvious that we will need to understand every bit of what the wizard generates. We create a new project, as usual with the "project" then "Create" menu option, and we give it the name of "winexample". The wizard shows us a lot of dialogs with many options that will be explained later, so just stick with the defaults. Choose a single window application:



After pressing the "next" button till the dialog boxes disappear, we press F9, compile the whole, and we run it. We see a single white window, with a status bar at the bottom, a sum-

105. In the data processing field, we have been always recycling very old ideas as "new, just improved". Object oriented programming was an idea that came from the old days of Simula in the beginning of the seventies but was "rediscovered" or rather "reinvented" in the late 80s. Garbage collection was standard in lisp systems in the seventies, and now has been discovered again by Mr. Bill Gates, in the next century, with his proposal for the C# language.

106. Remember the basics of Boolean logic: a bit ANDed with another will be one only if both bits are 1. A bit Ored with another with return 1 only if one or both of them is 1.

mary menu, and nothing else. Well, this is the skeleton of a windows application. Let's see it in more detail.

We start as always in the same place. We go to the WinMain function, and we try to figure out what is it doing. Here it is:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, INT nCmdShow)
{
     MSG msg;
     HANDLE hAccelTable;

     // Saves in this global variable the instance handle, that must be passed as an
     // argument to many window functions.
     hInst = hInstance;
     // If the initialization of the application fails, WinMain exits
     if (!InitApplication())
          return 0;
     // Loads the keyboard accelerators for common menu options
     hAccelTable = LoadAccelerators(hInst,MAKEINTRESOURCE(IDACCEL));
     // Creates the main window, and exits if it can't be created
     if ((hwndMain = CreateApplWndClassWnd()) == (HWND)0)
          return 0;
     // Creates the status bar at the bottom of the main window
     CreateSBar(hwndMain,"Ready",1);
     // Shows the main window
     ShowWindow(hwndMain,SW_SHOW);
     // Starts the message loop. When the main window post the quit message, GetMessage
     // will return NULL
     while (GetMessage(&msg,NULL,0,0)) {
          if (!TranslateAccelerator(msg.hwnd,hAccelTable,&msg)) {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
          }
     }
     // Returns the wParam of the WM_QUIT message
     return msg.wParam;
}
```

We have the same schema that we saw before, but this time with some variations. We start the application (registering the window class, etc.), we load the keyboard accelerators, we create the window, the status bar, we show our window, and then we enter the message loop until we receive a `WM_QUIT`, that breaks it. We return the value of the "wParam" parameter of the last message received (`WM_QUIT` of course).

Simple isn't it?

Now let's look at it in more detail.

The "InitApplication" procedure initializes the `WNDCLASS` structure with a little more care now, since we are not using our `DefDialogProc` any more, there are a lot of things we have to do ourselves. Mostly, that procedure uses the standard settings:

```
static BOOL InitApplication(void)
{
     WNDCLASS wc;

     memset(&wc,0,sizeof(WNDCLASS));
     // The window style
     wc.style = CS_HREDRAW|CS_VREDRAW |CS_DBLCLKS ;
     wc.lpfnWndProc = (WNDPROC)MainWndProc;
```

```
        wc.hInstance = hInst;
        // The color of the background
        wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
        wc.lpszClassName = "winexampleWndClass";
        // The menu for this class
        wc.lpszMenuName = MAKEINTRESOURCE(IDMAINMENU);
        // default cursor shape: an arrow.
        wc.hCursor = LoadCursor(NULL,IDC_ARROW);
        // default icon
        wc.hIcon = LoadIcon(NULL,IDI_APPLICATION);
        if (!RegisterClass(&wc))
             return 0;
        // ---TODO--- Call module specific initialization routines here

        return 1;
    }
```

The style of the window used is a combination of integer constants like `CS_HREDRAW`, and others, combined using the OR operator, the vertical bar. What does this mean?

This is a standard way of using bit flags in C. If you go to the definition of `CS_HREDRAW` (right-click in that name and choose the "Goto definition" option), you will see that the value is 2. Other constants like `CS_DBLCLKS` have a value of 8. All those numbers are a power of two. Well, a power of two by definition will always have a *single* bit set. All other bits will be zero. If you OR those numbers with each other, you will obtain a number that has the bits set that correspond to the different values used. In our case this statement is equivalent to:

```
wc.style = 2 | 1 | 8;
```

8 ored with 1 is 1 0 0 1, ored with 2 is 1 0 1 1, what is equal to 11 in decimal notation.

This is a very common way of using flags in C. Now, if you want to know if this window is a window that answers to double-clicks, you just have to query the corresponding bit in the style integer to get your answer. You do this with the following construct:

```
if (wc.style & CS_DBLCLKS) {
}
```

We test in this if expression, if the value of the "style" integer ANDed with 8 is different than zero. Since CS_DBLCLKS is a power of two, this AND operation will return the value of that single bit.[107] Note too that 1 is a power of two since 2 to the power of zero is one.

We will return to this at the end of this section.

Coming back to our initialization procedure, there are some new things, besides this style setting. But this is just a matter of reading the windows documentation. No big deal. There are many introductory books that augment their volume just with the easy way of including a lot of windows documentation in their text. Here we will make an exception.

But what is important however is that you know *how* to look in the documentation! Suppose you want to know what the hell is that `CS_DBLCLKS` constant, and what does it exactly mean. You press F1 in that identifier and nothing. It is not in the index.

Well, this constant appears in the context of RegisterClass API. When we look at the documentation of RegisterClass, we find a pointer to the doc of the `WNDCLASS` structure. Going there, we find in the description of the *style* field, all the CS_* constants, neatly explained.

Note that not all is in the index. You have to have a feeling of where to look. Lcc-win32 comes with a help file of reasonable size to be downloaded with a standard modem. It is 13MB compressed, and it has the essentials. A more detailed documentation complete with the latest stuff is in the Software Development Kit (SDK) furnished by Microsoft. It is available at their Web site, and it has a much more sophisticated help engine.

The WinMain procedure is the right place for initializing other things like CoInitialize() if you are going to use COM, or WSAStartup() if you are going to use the sockets layer, etc. The command line parameters are available in the lpCmdLine parameter.

The messages for the main window are handled in the MainWndProc function, that is passed as the message handling function when registering the window class. The standard version looks like this: each event is mapped to a case in a big switch of possible events.

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM
lParam)
{
     switch (msg) {
     case WM_SIZE:
          // Windows has been resized. Resize any child windows here
          SendMessage(hWndStatusbar,msg,wParam,lParam);
          InitializeStatusBar(hWndStatusbar,1);
          break;
     case WM_MENUSELECT:
          // The user is browsing the menu. Here you can add code
          // to display some text about the menu item being browsed for instance
          return MsgMenuSelect(hwnd,msg,wParam,lParam);
          // The WM_COMMAND message reports an item of the menu has been selected.
     case WM_COMMAND:
          HANDLE_WM_COMMAND(hwnd,wParam,lParam,MainWndProc_OnCommand);
          break;
     // This message reports that the window will be destroyed. Close the application now.
     case WM_DESTROY:
          PostQuitMessage(0);
          break;
     // All other messages go to the default procedure provided by Windows
     default:
          return DefWindowProc(hwnd,msg,wParam,lParam);
```

---

107. You may wonder what that variable "msg" stands for. It is a structure of type MSG, that is defined in windows.h as follows:

```
typedef struct tagMSG
{  HWND    hwnd;
   UINT    message;
   WPARAM  wParam;
   LPARAM  lParam;
   DWORD   time;
   POINT   pt;   } MSG, *PMSG, *NPMSG, *LPMSG;
```

Note that in C you can append as many names to a pointer to a structure as you like, and in windows this is used a lot. The reason is an historical one. In windows 16 bits there were several types of pointers: near (16 bit pointers), far (long pointers of 32 bits) and generally the near pointers were prefixed with NP, the 32 bit ones with LP. This was retained for compatibility reasons until today, even if there are only 32 bit pointers now.

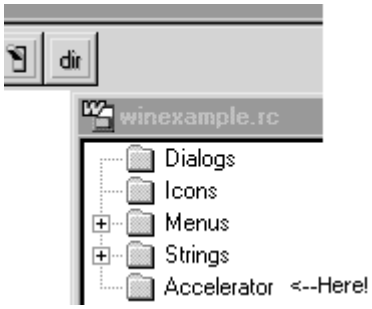
This structure contains then, the following fields:

- **hwnd**: The handle of the specific window to which the message is directed.
- **message:** A 16-bit value identifying the message.
- **wparam:** A 32-bit value identifying the first message parameter. Its meaning depends on the message being sent.
- **lParam:** A 32-bit value identifying the second message parameter.
- **time:** A 32-bit value identifying the time when the event that provoked this message happened.
- **pt:** This is a POINT structure containing the coordinates of the mouse in the instant the event happened that provoked this message.

```
        }
        return 0;
    }
```

2.7.1 Working with keyboard accelerators

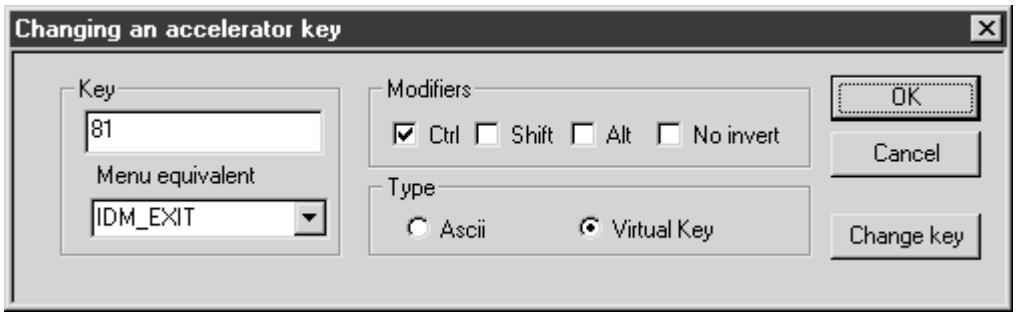
After initializing the window class, the WinMain function loads the accelerators for the application. This table is just a series of keyboard shortcuts that make easy to access the different menu items without using the mouse and leaving the keyboard. In the resource editor you can edit them, add/delete/change, etc. To do this you start the resource editor and you press the "dir" button in the upper right. You will see the following display.



You click in the "Accelerator" tree tab, and you will see the following:

We have here the accelerator called `IDM_EXIT` that has the value of 300. This is just Ctrl+Q for quit. The key value is 81, the ASCII value of the letter 'q', with a flag indicating that the control key must be pressed, to avoid quitting just when the user happens to press the letter q in the keyboard!

Double-clicking in the selected line leads us to yet another dialog:



Here you can change the accelerator as you want. The flags are explained in the documentation for the resource editor.

But this was just another digression, we were speaking about WinMain and that statement: LoadAccelerators… Well, let's go back to that piece of code again.

After loading the accelerators, the status bar is created at the bottom of the window, and then, at last, we show the window. Note that the window is initially hidden, and it will be shown only when all things have been created and are ready to be shown. This avoids screen flickering, and saves execution time. It would be wasteful to redraw the window before we created the status bar, since we would have to do that again after it is created.

We will speak about status bar later, since that is not crucial now. What really is important is the message loop that begins right after we call ShowWindow.

```
while (GetMessage(&msg,NULL,0,0)) {
     if (!TranslateAccelerator(msg.hwnd,hAccelTable,&msg)) {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
}
```

This loop calls the API GetMessage. If that returns TRUE, we call the API to translate the accelerators. This API will convert a sequence of keystrokes into a WM_COMMAND message, as it was sent by a menu, if it finds a correspondence between the keystrokes and the accelerator table that we loaded just a few lines above. If TranslateAccelerator doesn't find any correspondence, we go on by calling TranslateMessage, that looks for sequences of key pressing and key up messages, and does the dirty work of debouncing the keyboard, handling repeat sequences, etc. At last, we dispatch our message to the procedure indicated in the window class.

And that is it. We loop and loop and loop, until we eventually get a `WM_QUIT`, that provokes that GetMessage returns FALSE, and the while loop is finished.[108]

Wow. Finished?

We have hardly started. What is interesting now, is that we have a skeleton to play with. We will show in the next sections how we add things like a dialog box, etc.

Summary: Windows programming looks intimidating at first. But it is just the looks. Before we go on, however, as promised, let's look in more details to how flags are set/unset in C.

Flags are integers where each bit is assigned a predefined meaning. Usually with a preprocessor *define* statement, powers of two are assigned a symbolic meaning like in the case of `CS_DBLCLKS` above. In a 32-bit integer we can stuff 32 of those. We *test* those flags with:

108. Never forget this: local variables do NOT retain their value from one call of the function to the next one!

```
if (flag & CONSTANT) {
}
```

we *set* them with:

```
flag |= CONSTANT;
```

we *unset* them with:

```
flag &= ~CONSTANT;
```

This last statement needs further explanations. We use the AND operator with the complement of the constant. Since those constants have only one bit set, the complement of the constant is an integer with all the bits turned into ones except the bit that we want to unset. We AND that with our flag integer: since all bits but the one we want to set to zero are one, we effectively turn off that bit only, leaving all others untouched.

2.8 Customizing the wizard generated sample code

The generator can be configured for different types of applications. Use first the simple window application. This setup produces a standard WinMain procedure, like this:

You can add code for the initialization of your application in the InitApplication function. The standard version just registers the class of the main window.

In the next paragraphs we will try to see how we modify this skeleton to add different items you may want to add to your application.

2.8.1 Making a new menu or modifying the given menu.

Add your item with the resource editor, and give it an identifier, normally written in uppercase like: `IDMENU_ASK_PARAMETERS`, or similar. This is surely not the text the user will see, but a symbolic name for an integer constant, that windows will send to the program when the user presses this option. We can then, continue our beloved programming traditions.

Once that is done, and your menu displays correctly, go to the wizard code in theMainWndProc function.[109] There, you will see a big switch with different code to handle the events we are interested in. The menu sends a command event, called `WM_COMMAND`. In there you see that this event will be handled by the `HANDLE_COMMAND` macro. It is just a shorthand to break down the 64 bits that windows sends us in smaller pieces, disassembling the message information into its constituent parts. This macro ends calling the `MainWndProc_OnCommand` function that is a bit higher in the text. There, you will find a switch with the comment:

```
//---TODO--- Insert new commands here.
```

Well, do exactly that, and add (as a new case in the switch) your new identifier `IDMENU_ASK_PARAMETERS.` There you can do whatever you want to do when that menu item is clicked.

2.8.2 Adding a dialog box.

Draw the dialog box with controls and all that in the resource editor, and then open it as a result of a menu option, for instance. You would use `DialogBox`, that handy primitive explained in detail in the docs to fire up your dialog.[110] You have to write the procedure to handle the dialog box's messages first. You can start the dialog box in a non-modal mode with the `CreateDialog` API.

To make this a bit more explicit, let's imagine you have defined your dialog under the name of `IDD_ASK_PARAMS` in the resource editor.[111] You add a menu item corresponding to the dialog in the menu editor, one that will return `IDM_PARAMETERS`, say. You add then in the function `MainWndProc_OnCommand` code like this:

```
case IDM_PARAMETERS:
     r = DialogBox(hInst,
```

109. Go to "help", then click in Win32 API, get to the index and write the name of that function.

110. Again, this is the #defined identifier of the dialog, not the dialog's title!

111. The registry has been criticized because it represents a single point of failure for the whole system. That is obviously true, but it provides as a redeeming value, a standard way of storing and retrieving configuration data and options. It allows your application to use the same interface for storing this data, instead of having to devise a schema of files for each application. The software is greatly simplified by this, even if it is risky, as a general principle.

```
                MAKEINTRESOURCE( IDD_ASK_PARAMS),
                ghwndMain,ParamsDlgProc);
            break;
```

You give to that API the instance handle of the application, the numerical ID of the dialog enclosed in the `MAKEINTRESOURCE` macro, the handle of the parent window, and the name of the procedure that handles the messages for the dialog. You will need the prototype of the function in some header file to be able to use the name of it, so it is a good idea to write a "dlgboxprotos.h" header file that is included in every source file of the application.

If you want to pass a parameter to the dialog box procedure you can use DialogBoxParam.

2.8.3 Drawing the window

You have to answer to the `WM_PAINT` message. See the documentation for a longer description. This will provoke drawing when the window needs repainting only. You can force a redraw if you use the `InvalidateRect` API.

You add code like this:

```
case WM_PAINT:
     PAINTSTRUCT ps;
     HDC hDC = BeginPaint(hwnd,&ps);
     // Code for painting using the HDC goes here
     EndPaint(hwnd,&ps);
     break;
```

You use the API BeginPaint to inform windows that you are going to update the window. Windows gives you information about the invalid rectangles of the window in the `PAINTSTRUCT` area. You pass to windows the address of such an area, and the handle to your window. The result of `BeginPaint` is an `HDC`, a Handle to a Device Context, that is required by most drawing primitives like `TextOut`, `LineTo`, etc. When you are finished, you call EndPaint, to inform windows that this window is updated.

To draw text you use `TextOut`, or `DrawText`. Note that under windows there is no automatic scrolling. You have to program that yourself or use a multi-line edit control.

2.8.4 Initializing or cleaning up

You can write your initialization code when handling the `WM_CREATE` message. This message is sent only once, when the window is created. To cleanup, you can rely on the `WM_CLOSE` message, or better, the `WM_DESTROY` message. Those will be sent when the window is closed/destroyed. Note that you are not forced to close the window when you receive the `WM_CLOSE` message. Even if this is not recommended, you can handle this message and avoid passing it to the `DefWndProc` procedure. In this case the window is not destroyed. Another thing is when you receive the `WM_DESTROY` message. There, you are just being informed that your window is going to be destroyed anyway.

2.8.5 Getting mouse input.

You can handle the `WM_LBUTTONDOWN`, or `WM_RBUTTONDOWN` messages. To follow the mouse you handle the `WM_MOUSEMOVE` messages. In the information passed with those message parameters you have the exact position of the mouse in pixel coordinates.

Message	*Meaning*

WM_LBUTTONBDBLCLK	Double click in left mouse button
WM_LBUTTONDOWN WM_LBUTTONUP	Left mouse button click. For the middle and right button the messages used use the letters ‘M’ and ‘R’ instead of ‘L’ here.
WM_MOUSEWHEEL	The mouse wheel has been used.
WM_MOUSEMOVE	This is posted to a window when the cursor moves. If the mouse is not captured, the message is posted to the window that contains the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_MOUSEHOVER	This is posted to a window when the cursor hovers over the client area of the window for the period of time specified in a prior call to TrackMouseEvent.
WM_MOUSELEAVE	This is posted to a window when the cursor leaves the client area of the window specified in a prior call to TrackMouseEvent.

2.8.6 Getting keyboard input

Handle the WM_KEYDOWN message or the WM_CHAR message. Windows allows for a fine-grained control of the keyboard, with specific messages when a key is pressed, released, repeat counts, and all the keyboard has to offer. Here is a small description of each:

Message	*Meaning*
WM_ACTIVATE	Window is being either activated or desactivated.
WM_CHAR	The user has typed a character. Here you receive characters like ‘m’ for instance.
WM_KEYDOWN	A key has been pressed. Here you can use keys like the arrows of the keyboard, Del Insr, etc.
WM_KEYUP	A key has been released

2.8.7 Handling moving/resizing

You get WM_MOVE when your window has been moved, WM_SIZE when your window has been resized. In the parameters of those messages you find the new positions or new size of your window. You may want to handle those events when you have child windows that you should move with your main window, or other measures to take, depending on your application.

Message	*Meaning*
WM_MOVE	Sent after a window has been moved.
WM_MOVING	Sent to a window when the user is moving it. By processing this message, you can monitor the position of the drag rectangle and, if needed, change its position.

Message	*Meaning*
WM_SIZE	Sent after a window has been resized.
WM_SIZING	Sent to a window that the user is resizing. By processing this message, an application can monitor the size and position of the drag rectangle and, if needed, change its size or position.
WM_WINDOWPOSCHANGED	Sent to a window whose size, position, or place in the Z order has changed as a result of a call to the SetWindowPos function or another window-management function
WM_WINDOWPOSCHANGING	Same as above message but sent before the changes.
WM_GETMINMAXINFO	Sent to a window when the size or position of the window is about to change. You can use this message to override the window's default maximized size and position, or its default minimum or maximum tracking size.
WM_ENTERSIZEMOVE	Sent one time to a window after it enters the moving or sizing modal loop. The window enters the moving or sizing modal loop when the user clicks the window's title bar or sizing border, or when the window passes the WM_SYSCOMMAND message to the DefWindowProc function and the wParam parameter of the message specifies the SC_MOVE or SC_SIZE value. The operation is complete when DefWindowProc returns.
WM_EXITSIZEMOVE	Sent one time to a window, after it has exited the moving or sizing modal loop.

2.9 Window controls

A “control” is a technical name for a child window that has a characteristic appearence and fills a functionality for user input. A list box for instance is a child window that has a rectangular appearence with lines and scrollbars that displays a list of items.

When the system creates a dialog box, it creates all its child windows according to the specifications it finds in the executable file. The dialog box procedure is the parent window of all the controls that appear in it.

When an event happens that possible needs that the control notifies the application about it, the control sends a notification message. For instance, when the user types a character into an edit field, the edit control sends a notification to the parent window specifying the character received.

Notifications can be sent in the form of a WM_COMMAND message, or in the form of a WM_NOTIFY message.

The following controls send a WM_COMMAND message to notify the parent window of an event:

Control	*Notifications*
Button	Notifies the parent window when it has been selected. See the documentation for the BN_XXX messages
Combobox	These controls are a combination of a list box and an edit field, so the notifications it sends are a combination of both. See the documentation for CBN_XXX messages.
Edit	These controls are used for text input. They notify the parent when a character has been received or in general when the text has been modified. See the EN_XXX messages
Listbox	These controls display a variable length list. They notify the parent window when an item has been selected, or clicked.
Scrollbar	These controls let the user choose the amount of scrolling that should be done for the information being displayed. They notify the parent window when they are used.
Static	These controls do not send any notifications (hence they are called "static"). They are text strings, or lines, rectangles.
Rich edit controls	They are used to display text with more than a single font or color. They have a long list of notifications, you can even specify that you want to be notified when the user clicks in an URL in the text. This control uses both the WM_COMMAND interface and the WM_NOTIFY interface.

The interface using the WM_COMMAND message was described above (See page 225). Another interface used by many controls is the WM_NOTIFY interface. This message has the following format:

```
lResult = SendMessage(     // returns LRESULT
     (HWND) hWndControl,   // handle to parent window
     (UINT) WM_NOTIFY,     // message ID
     (WPARAM) wParam,      // = (WPARAM) (int) idCtrl;
     (LPARAM) lParam       // = (LPARAM) (LPNMHDR) pnmh;
);
```

This code is executed by the control when it sends a message to the parent window. The important parameters of this message are:

1 ***idCtrl*** This is a numerical constant (an integer) that is used as the ID of this control.

2 ***pnmh*** This is a pointer to a structure that contains further information about the message. It points to a NMHDR structure that contains the following fields:

```
typedef struct tagNMHDR {
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
} NMHDR;
```

3 This struture can be followed by more data in some of the notifications that a control can send. In that case this fixed fields will be membbers of a larger structure and they will be always at the same position.

The controls that use this interface were introduced into windows later (in 1995) than the first ones, that were present in all the earlier versions of windows. They are the following:

Control	*Notifications*
Animation control	An animation control is a window that displays an Audio-Video Interleaved (AVI) clip. An AVI clip is a series of bitmap frames like a movie. Animation controls can only display AVI clips that do not contain audio. This control is currently not supported by the resource editor.
Date and time picker controls	They allow the user to input a date or a time. The notifications use the prefix DTN_XXX.
Header controls	A header control is a window that is usually positioned above columns of text or numbers. It contains a title for each column, and it can be divided into parts. The user can drag the dividers that separate the parts to set the width of each column. The notifications use the prefix HDN_XXX. Not currently supported in the resource editor.
Hot key controls	A hot key control is a window that enables the user to enter a combination of keystrokes to be used as a hot key. A hot key is a key combination that the user can press to perform an action quickly. For example, a user can create a hot key that activates a given window and brings it to the top of the z-order. The hot key control displays the user's choices and ensures that the user selects a valid key combination. Not supported in the resource editor.
IP address control	Allows to input an IP address. Not currently supported in the resource editor.
Month calendar	Allows the user to choose a month or a calendar date. The notifications use the prefix MCN_XXX.
Pager control	A pager control is a window container that is used with a window that does not have enough display area to show all of its content. The pager control allows the user to scroll to the area of the window that is not currently in view. Not supported in the editor.
Progress bar.	A progress bar is a window that an application can use to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled with the system highlight color as an operation progresses. This control sends no notifications but receives messages that use the PBM_XXX prefix.

Control	*Notifications*
Tab controls	A tab control is analogous to the dividers in a notebook or the labels in a file cabinet. By using a tab control, an application can define multiple pages for the same area of a window or dialog box. Each page consists of a certain type of information or a group of controls that the application displays when the user selects the corresponding tab. The notifications specific to this control use the TCN_XXX prefix.
Trackbar control	A trackbar is a window that contains a slider and optional tick marks. When the user moves the slider, using either the mouse or the direction keys, the trackbar sends notification messages to indicate the change. The notifications used are only the NM_CUSTOMDRAW and the NM_REALEASEDCAPTURE messages.
Tree views	A tree-view control is a window that displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk. Each item consists of a label and an optional bitmapped image, and each item can have a list of subitems associated with it. By clicking an item, the user can expand or collapse the associated list of subitems. The notifications sent use the TVN_XXX prefix
Up-down control	An up-down control is a pair of arrow buttons that the user can click to increment or decrement a value, such as a scroll position or a number displayed in a companion control. The value associated with an up-down control is called its current position. An up-down control is most often used with a companion control, which is called a buddy window.

The documentation for all this controls is very detailed and it would be impossible to reproduce here.[112] Note that many of the controls not directly supported in the resource editor can be created "by hand" by just calling the CreateWindow procedure. Most of them are not specially useful in the context of a dialog box.

To handle the messages with the WM_NOTIFY interface you should add code to handle the message of interest in the window procedure. Suppose, for instance, that you have a tree control and you want to be notified when the user has clicked on an an item with the right mouse button.

```
case WM_NOTIFY:
     LPNMHDR nmhdr = (LPNMHDR)lParam;
     switch (nmhdr->code) {
          case NM_RCLICK:
          TV_HITTESTINFO testInfo;
          // The structure testInfo will be filled with the coordinates
          // of the item at this particular point.
          memset(&testInfo,0,sizeof(TV_HITTESTINFO));
          // Get the cursor coordintes
          GetCursorPos(&testInfo.pt);
          pt1 = testInfo.pt;
          // Translate the coordinates into coordinates of the tree window
          MapWindowPoints(HWND_DESKTOP,hwndTree,&testInfo.pt,1);
```

112.See "The windows user interface" in the online documentation of lcc-win32.

```
            // Now ask the tree view control if there is an item at this position
            hti = TreeView_HitTest(hwndTree,&testInfo);
            // If nothing is found stop.
            if (hti == (HTREEITEM)0) break;
            // There is something. Show the context menu using the information returned by
            // the control in the item handle hti.
            hnewMenu = CreateContextMenu(hti);
            // If the creation of the menu did not work stop.
            if (hnewMenu == (HMENU)0) break;
            menuItem = TrackPopupMenu(hnewMenu,
                      TPM_RIGHTBUTTON|TPM_TOPALIGN|TPM_RETURNCMD,
                      pt1.x,pt1.y,
                      0,hwnd,NULL);
            DestroyMenu(hnewMenu);
            // Here we would do some action indicated by the menuItem code
            break;
        }
```

This code begins with a cast of the lParam message parameter into a pointer to a NMHDR structure. Then it examines the passed notification code. If it is a right click it fills a TV_HITTESTINFO structure with the coordinates of the mouse, and translates those into the coordinates of the tree control that this code supposes in a global variable "hwndTree". The control should return either NULL, meaning there is no item at that position, or the handle of an item that was right-clicked with the mouse. If there was an item, a context menu is created, shown, and the result is an integer indicating which menu item was chosen, if any.

This is one of the many ways you can interact with the window system. Each control will have its own interface since they are all different and perform quite different actions.

You can see the above code in action when you click with the right button in an item of the project workspace window in Wedit.

2.9.1 Using controls without a dialog box

You use the `CreateWindow` API with a predefined window class. You pass it as the parent-window parameter the handle of the window where you want to put the control on. For instance if you want to create somewhere in your window an edit field, you would write:

```
hEditWnd = CreateWindow("EDIT", // window class is "edit"
     NULL,
     WS_CHILD | WS_VISIBLE |
          ES_MULTILINE |
          WS_VSCROLL | WS_HSCROLL |
          ES_AUTOHSCROLL | ES_AUTOVSCROLL,
     0,
     0,
     (Rect.right - Rect.left), //Assume a rectangle structure
     (Rect.bottom - Rect.top), // that contains the coordinates
     hParentWnd,
     (HMENU) 14376, // Window identifier
     hInst, // Application instance
     NULL);
```

The controls "edit", "listbox" and others are already predefined when windows starts. You can use them at any time. Some others however are NOT like that, and you need to call the API `InitCommonControlsEx()` before you can use them.

2.10 A more complex example: a "clone" of spy.exe

What can we do with the empty window that Wedit generates?

Let's do a more difficult problem: We want to find out all the windows that are opened at a given time in the system. We will display those windows in a tree control, since the child windows give naturally a tree structure. When the user clicks in a window label, the program should display some information about the window in the status bar.

We generate a skeleton with Wedit, as described above. We create a new project, and generate a simple, single window application.

2.10.1 Creating the child windows

OK. Now we come back to the task at hand. The first thing to do is to create the tree control window. A good place to do these kinds of window creations is to use the opportunity the system gives to us, when it sends the `WM_CREATE` message to the main window. We go to the procedure for the main window, called `MainWndProc`, and we add the `WM_CREATE` case to the switch of messages:

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM
lParam)
{
     static HWND hwndTree;

     switch (msg) {
     case WM_CREATE:
          hwndTree = CreateTree(hwnd,IDTREEWINDOW);
          break;
```

This is "top down" design. We hide the details of the tree window creation in a function that returns the window handle of the tree control. We save that handle in a static variable. We declare it as static, so that we can retrieve its value at any call of `MainWndProc`.[113]

Our `CreateTree` function, looks like this:

```
static HWND _stdcall CreateTree(HWND hWnd,int ID)
{
     return
CreateWindowEx(WS_EX_CLIENTEDGE,
WC_TREEVIEW,"",
     WS_VISIBLE|WS_CHILD|WS_BORDER|TVS_HASLINES|
TVS_HASBUTTONS|TVS_DISABLEDRAGDROP,
          0,0,0,0,
          hWnd,(HMENU)ID,hInst,NULL);
}
```

This function receives a handle to the parent window, and the numeric ID that the tree control should have. We call the window creation procedure with a series of parameters that are well described in the documentation. We use the value of the `hInst` global as the instance, since the code generated by Wedit conveniently leaves that variable as a program global for us to use.

Note that we give the initial dimensions of the control as zero width and zero height. This is not so catastrophic as it seems, since we are relying in the fact that after the creation message, the main window procedure will receive a `WM_SIZE` message, and we will handle the sizing

113. In page 218. Published by St Martin’s Press. 1990. ISBN 0-312-06179-X (pbk)

of the tree control there. This has the advantage that it will work when the user resizes the main window too.

2.10.2 Moving and resizing the child windows

We add code to the `WM_SIZE` message that Wedit already had there to handle the resizing of the status bar at the bottom of the window.

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM
lParam)
{
     static HWND hwndTree;
     RECT rc,rcStatus;

     switch (msg) {
     case WM_CREATE:
          hwndTree = CreateTree(hwnd,IDTREEWINDOW);
          break;
     case WM_SIZE:
          SendMessage(hWndStatusbar,msg,wParam,lParam);
          InitializeStatusBar(hWndStatusbar,1);
          GetClientRect(hwnd,&rc);
          GetWindowRect(hWndStatusbar,&rcStatus);
          rc.bottom -= rcStatus.bottom-rcStatus.top;
          MoveWindow(hwndTree,0,0,rc.right,rc.bottom,1);
          break;
```

We ask windows the current size of the main window with `GetClientRect`. This procedure will fill the rectangle passed to it with the width and height of the client area, i.e. not considering the borders, title, menu, or other parts of the window. It will give us just the size of the drawing surface.

We have a status bar at the bottom, and the area of the status bar must be subtracted from the total area. We query this time using the `GetWindowRect` function, since we are interested in the whole surface of the status bar window, not only in the size of its drawing surface. We subtract the height of the window from the height that should have the tree control, and then we move it to the correct position, i.e. filling the whole drawing surface of the window. And we are done with drawing.

2.10.3 Starting the scanning.

Now we pass to the actual task of our program. We want to fill the tree control with a description of all the windows in the system. A convenient way to do this is to change the "New" menu item into "Scan", and start scanning for windows when the user chooses this item.

To do this, we add an entry into the `MainWndProc_OnCommand` function:

```
void MainWndProc_OnCommand(HWND hwnd,
int id, HWND hwndCtl, UINT codeNotify)
{
     switch(id) {
     case IDM_NEW:
          BuildTree(hwnd);
          break;
     case IDM_EXIT:
          PostMessage(hwnd,WM_CLOSE,0,0);
          break;
     }
}
```

Simple isn't it? We just call "BuildTree" and we are done.

2.10.4 Building the window tree.

We start with the desktop window, we add it to the tree, and then we call a procedure that will enumerate all child windows of a given window. We have two directions to follow: the child windows of a given window, and the sibling windows of a given window. This is true for the desktop window too.

Let's look at the code of "`BuildTree`":

```
int BuildTree(HWND parent)
{
     HWND Start = GetDesktopWindow();
     HWND hTree = GetDlgItem(parent,IDTREEWINDOW);
     TV_INSERTSTRUCT TreeCtrlItem;
     HTREEITEM hNewNode;

     memset(&TreeCtrlItem,0,sizeof(TreeCtrlItem));
     TreeCtrlItem.hParent = TVI_ROOT;
     TreeCtrlItem.hInsertAfter = TVI_LAST;
     TreeCtrlItem.item.mask = TVIF_TEXT | TVIF_PARAM;
     TreeCtrlItem.item.pszText = "Desktop";
     hNewNode = TreeView_InsertItem(hTree,&TreeCtrlItem);
     Start = GetWindow(Start,GW_CHILD);
     Scan(hTree,hNewNode,Start);
     return 1;
}
```

We start at the start, and we ask windows to give us the window handle of the desktop window. We will need the tree window handle too, so we use "`GetDlgItem`" with the parent window of the tree control, and it's ID. This works, even if the parent window is a normal window, and not a dialog window.

We go on by filling our `TV_INSERTSTRUCT` with the right values. This is a common interface for many window functions. Instead of passing n parameters, we just fill a structure and pass a pointer to it to the system. Of course, it is always a good idea to clean the memory space with zeroes before using it, so we zero it with the "`memset`" function. Then we fill the fields we need. We say that this item is the root item, that the insertion should happen after the last item, that the item will contain the text "Desktop", and that we want to reserve place for a pointer in the item itself (`TVIF_PARAM`). Having done that, we use the macro for inserting an item into the tree.

The root item created, we should then scan the siblings and child windows of the desktop. Since the desktop is the root of all windows it has no siblings, so we start at its first child. The `GetWindow` function, gives us a handle to it.

2.10.5 Scanning the window tree

We call our "Scan" function with the handle of the tree control, the handle to the just inserted item, and the window handle of the first child that we just obtained.

The "Scan" function looks like this:

```
void Scan(HWND hTree,HTREEITEM hTreeParent,HWND Start)
{
     HWND hwnd = Start,hwnd1;
     TV_INSERTSTRUCT TreeCtrlItem;
     HTREEITEM htiNewNode;
     char bufTxt[256],bufClassName[256],Output[1024];
```

```
        while (hwnd != NULL) {
             SendMessage(hwnd,WM_GETTEXT,250,(LPARAM) bufTxt);
             GetClassName(hwnd,bufClassName,250);
             wsprintf(Output,"\"%s\" %s",bufTxt,bufClassName);
             memset(&TreeCtrlItem,0,sizeof(TreeCtrlItem));
             TreeCtrlItem.hParent = hTreeParent;
             TreeCtrlItem.hInsertAfter = TVI_LAST;
             TreeCtrlItem.item.mask = TVIF_TEXT | TVIF_PARAM;
             TreeCtrlItem.item.pszText = (LPSTR) Output;
             TreeCtrlItem.item.lParam = (LPARAM) hwnd;
             htiNewNode =
                  TreeView_InsertItem ( hTree,&TreeCtrlItem);
             if((hwnd1=GetWindow(hwnd,GW_CHILD))!=NULL)
                  Scan(hTree,htiNewNode,hwnd1);
             hwnd=GetWindow(hwnd,GW_HWNDNEXT);
        }
   }
```

We loop through all sibling windows, calling ourselves recursively with the child windows.

In our loop we do:

We get the text of the window, to show it in our tree. We do this by sending the `WM_GETTEXT` message to the window.

We get the class name of the window.

We format the text (enclosed in quotes) and the class name in a buffer.

We start filling the `TV_INSERTSTRUCT`. These steps are very similar to what we did for the desktop window.

After inserting our node in the tree, we ask if this window has child windows. If it has, we call Scan recursively with the new node and the new child window.

Then we ask if this window has sibling windows. If it has, the main loop will go on since GetWindow will give us a non-null window handle. If it hasn't we are done and we exit.

2.10.6 Review

Let's look at our "BuildTree" function again and ask us:

How could this fail?

We notice immediately several things.

We always add items to the tree at the end, but we never cleanup the tree control. This means that after a few times the user has clicked in the menu, we will have several times all the windows of the system in our tree. All nodes should be deleted when we start.

The tree control will redraw itself several times when we add items. This is unnecessary and produces a disturbing blinking in the display. We should hold the window without any redrawing until all changes are done and then redraw once at the end.

We modify the "BuildTree" procedure as follows:

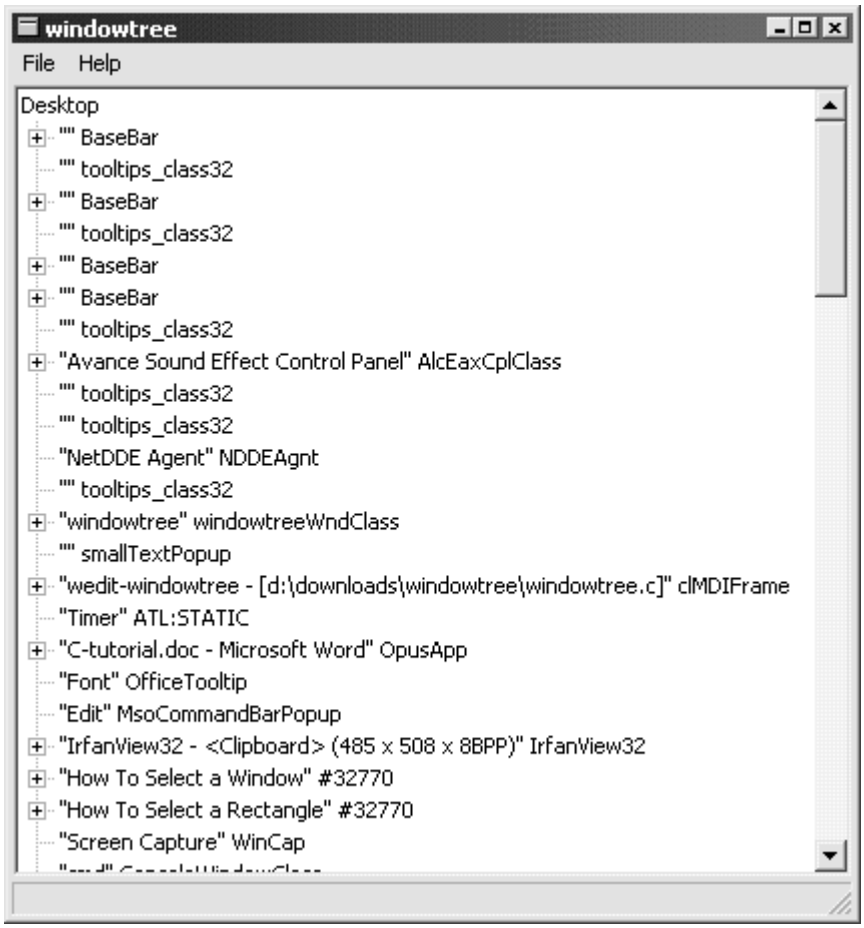
```
int BuildTree(HWND parent)
{
     HWND Start = GetDesktopWindow();
     HWND hTree = GetDlgItem(parent,IDTREEWINDOW);
     TV_INSERTSTRUCT TreeCtrlItem;
     HTREEITEM hNewNode;

     SendMessage(hTree,WM_SETREDRAW,0,0);
```

```
        TreeView_DeleteAllItems(hTree);
        memset(&TreeCtrlItem,0,sizeof(TreeCtrlItem));
        TreeCtrlItem.hParent = TVI_ROOT;
        TreeCtrlItem.hInsertAfter = TVI_LAST;
        TreeCtrlItem.item.mask = TVIF_TEXT | TVIF_PARAM;
        TreeCtrlItem.item.pszText = "Desktop";
        hNewNode = TreeView_InsertItem(hTree,&TreeCtrlItem);
        Start = GetWindow(Start,GW_CHILD);
        Scan(hTree,hNewNode,Start);
        TreeView_Expand(hTree,hNewNode,TVE_EXPAND);
        SendMessage(hTree,WM_SETREDRAW,1,0);
        return 1;

    }
```

We enclose all our drawing to the control within two calls to the SendMessage function, that tell essentially the tree control not to redraw anything. The third parameter (i.e. the wParam of the message) is a Boolean flag that indicates whether redrawing should be on or off. This solves the second problem.

After setting the redraw flag to off, we send a command to the control to erase all items it may have. This solves our first problem.

Here is the output of the program after we press the "Scan" menu item.



A lot of code is necessary to make this work, but thankfully it is not our code but window's. The window resizes, redraws, etc., without any code from us.

2.10.7 Filling the status bar

Our task consisted in drawing the tree, but also of displaying some useful information about a window in the status bar when the user clicks on a tree item.

First, we have to figure out how we can get notified when the user clicks in an item.

The tree control (as many other controls) sends notifications through its WM_NOTIFY message. We add a snippet of code to our MainWndProc procedure:

```
case WM_CREATE:
     hwndTree = CreateTree(hwnd,IDTREEWINDOW);
     break;
case WM_NOTIFY:
     return HandleWmNotify(hwnd,wParam,lParam);
```

The function HandleWmNotify looks as follows:

```
LRESULT HandleWmNotify(HWND hwnd, WPARAM wParam, LPARAM lParam)
{
     NMHDR *nmhdr;
     TV_HITTESTINFO testInfo;
     HWND hTree = GetDlgItem(hwnd,IDTREEWINDOW);
     HTREEITEM hti;
     HWND hwndStart;

     nmhdr = (NMHDR *)lParam;
     switch (nmhdr->code) {
     case NM_CLICK:
          memset(&testInfo,0,sizeof(TV_HITTESTINFO));
          GetCursorPos(&testInfo.pt);
          MapWindowPoints(HWND_DESKTOP,hTree,&testInfo.pt,1);
          hti = TreeView_HitTest(hTree,&testInfo);
          if (hti == (HTREEITEM)0) break;
          hwndStart = GetTreeItemInfo(hTree,hti);
          SetTextInStatusBar(hwnd,hwndStart);
          break;
     }
     return DefWindowProc(hwnd,WM_NOTIFY,wParam,lParam);
}
```

We just handle the NM_CLICK special case of all the possible notifications that this very complex control can send. We use the NMHDR part of the message information that is passed to us with this message in the lParam message parameter.

Our purpose here is to first know if the user has clicked in an item, or somewhere in the background of the tree control. We should only answer when there is actually an item under the coordinates where the user has clicked. The algorithm then, is like this:

Get the mouse position. Since windows has just sent a click message, the speed of current machines is largely enough to be sure that the mouse hasn't moved at all between the time that windows sent the message and the time we process it. Besides, when the user is clicking it is surely not moving the mouse at super-sonic speeds.

Map the coordinates we received into the coordinates of the tree window.

Ask the tree control if there is an item under this coordinates.

If there is none we stop

Now, we have a tree item. We need to know which window is associated with this item, so that we can query the window for more information. Since we have left in each item the window handle it is displaying, we retrieve this information. We hide the details of how we do this in a subroutine "GetTreeItemInfo", that returns us the window handle.

Using that window handle we call another function that will display the info in the status bar.

We pass all messages to the default window procedure. this is a non-intrusive approach. The tree control could use our notifications for something. We just need to do an action when this event happens, but we want to disturb as little as possible the whole environment.

2.10.8 Auxiliary procedures

To retrieve our window handle from a tree item, we do the following:

```
static HWND GetTreeItemInfo(HWND hwndTree,HTREEITEM hti)
{
     TV_ITEM tvi;

     memset(&tvi,0,sizeof(TV_ITEM));
     tvi.mask = TVIF_PARAM;
     tvi.hItem = hti;
     TreeView_GetItem(hwndTree,&tvi);
     return (HWND) tvi.lParam;
}
```

As you can see, it is just a matter of filling a structure and querying the control for the item. we are interested only in the PARAM part of the item.

More complicated is the procedure for querying the window for information. Here is a simple approach:

```
void SetTextInStatusBar(HWND hParent,HWND hwnd)
{
     RECT rc;
     HANDLE pid;
     char info[4096],*pProcessName;

     GetWindowRect(hwnd,&rc);
     GetWindowThreadProcessId(hwnd,&pid);
     pProcessName = PrintProcessNameAndID((ULONG)pid);
     wsprintf(info,
          "Handle: 0x%x %s, left %d, top %d, right %d, bottom %d,
                    height %d, width %d, Process: %s",
          hwnd,
          IsWindowVisible(hwnd)? "Visible" : "Hidden",
          rc.left,rc.top,rc.right,rc.bottom,
          rc.bottom-rc.top,
          rc.right-rc.left,
          pProcessName);
     UpdateStatusBar(info, 0, 0);
}
```

The algorithm here is as follows:

Query the window rectangle (in screen coordinates).

We get the process ID associated with this window

We call a subroutine for putting the name of the process executable file given its process ID.

We format everything into a buffer

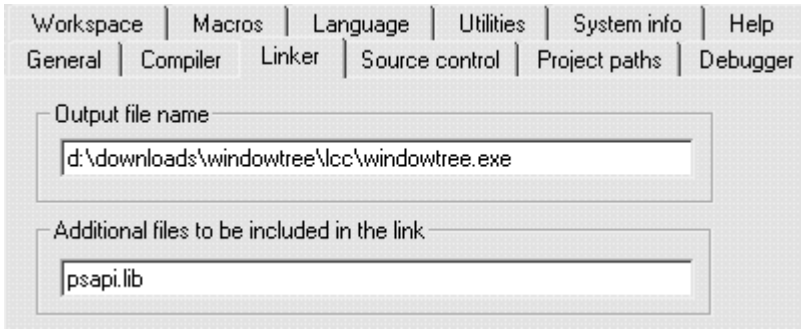
We call UpdateStatusBar, generated by wedit, with this character string we have built.

The procedure for finding the executable name beginning with a process ID is quite advanced, and here we just give it like that.

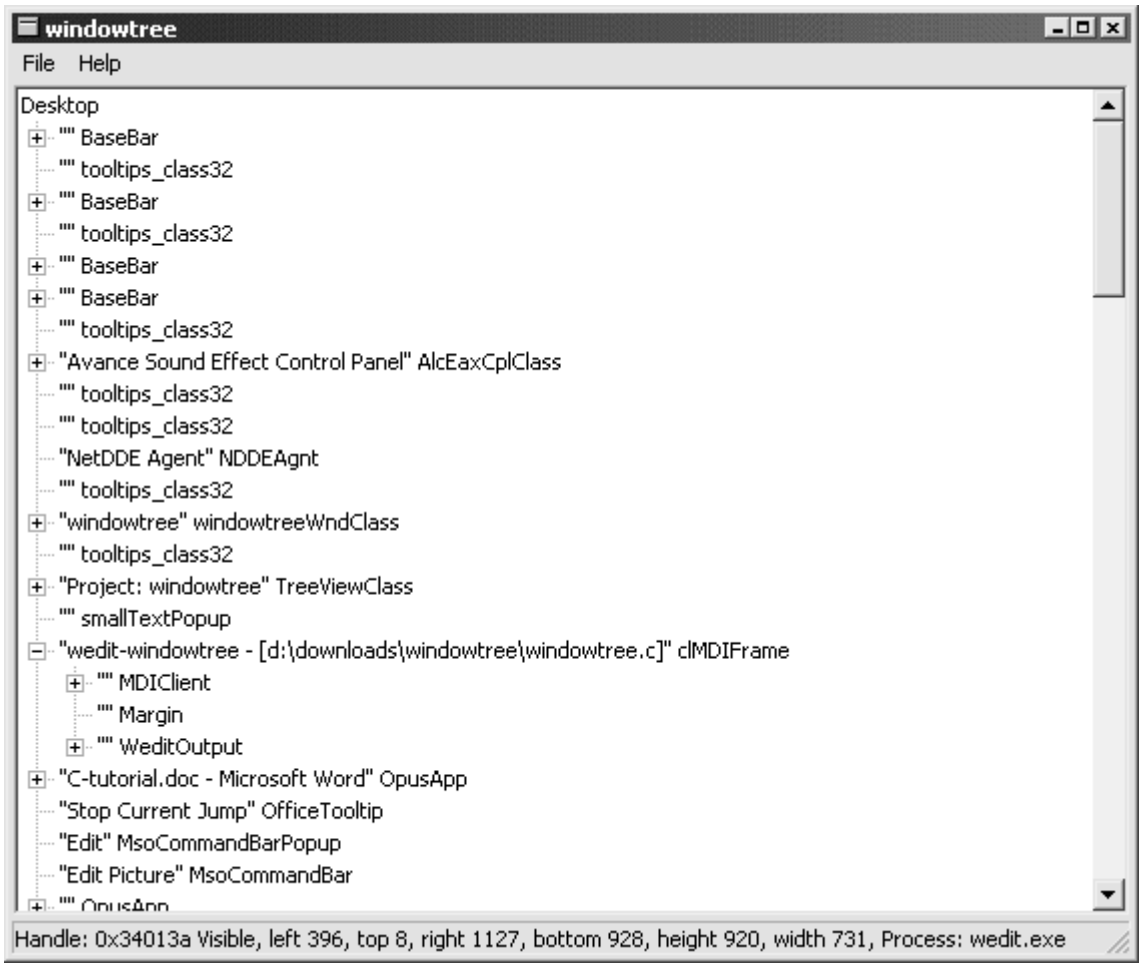
```
static char * PrintProcessNameAndID( DWORD processID )
{
```

```
        static char szProcessName[MAX_PATH];
        HMODULE hMod;
        DWORD cbNeeded;
        HANDLE hProcess = OpenProcess( PROCESS_QUERY_INFORMATION |
                  PROCESS_VM_READ,
             FALSE, processID );
        szProcessName[0] = 0;
        if ( hProcess ) {
             if ( EnumProcessModules( hProcess, &hMod, sizeof(hMod),
                  &cbNeeded) ) {
                  GetModuleBaseName( hProcess, hMod, szProcessName,
                       sizeof(szProcessName) );
             }
             CloseHandle( hProcess );
        }
        return szProcessName;
    }
```

Note that you should add the library `PSAPI.LIB` to the linker command line. You should do this in the linker tab in the configuration of wedit:



And now we are done. Each time you click in an item window, the program will display the associated information in the status bar:



Summary: There are many things that could be improved in this small program. For instance, it could be useful to have a right mouse menu, or a dialog box with much more information etc. This is just a blueprint to get you started however.

The whole code for this program is in the appendix 4.

2.11 Numerical calculations in C.

Well, we have a beautiful window with nothing in it. Blank. It would look better if we would draw something in it isn't it? By the way, this is an introduction to C, not to windows…

What can we draw?

Let's draw a galaxy. In his wonderful book "Computers Pattern Chaos and Beauty", Clifford A. Pickover[114] writes:

> *We will approximate a galaxy viewed from above with logarithmic spirals. They are easily programmable in a computer, representing their stars with just dots. One arm is 180 degrees out of phase with the other. To obtain a picture of a galactic distribution of dots, simply plot dots at* (r,θ) *according to:*
>
> $$r_1 = e^{[\theta \tan \phi]}$$
>
> $$r_2 = e^{[(\pi+\theta) \tan \phi]}$$
>
> *where r1 and r2 correspond to the intertwined spiral arms. The curvature of the galactic arms is controlled by* ϕ *which should be about 0.2 radians for realistic results. In addition,* $0 < \theta < 1000$ *radians. For greater realism, a small amount of random jitter may be added to the final points.*

He is kind enough to provide us with a formal description of the program in some computer language similar to BASIC. Here it is:

Algorithm: How to produce a galaxy.

Notes: The program produces a double logarithmic spiral. The purpose of the random number generator is to add jitter to the distribution of stars.

Variables:

in = curvature of galactic arm (try in = 2)

maxit = maximum iteration number

scale = radial multiplicative scale factor

cut = radial cutoff

f = final cutoff

Code:

```
loop1: Do i = 0 to maxit;
theta = float(i)/50;
r = scale*exp(theta*tan(in));
if r > cut then leave loop1;
x = r * cos(theta)+50;
y = r * sin(theta)+50;
call rand(randx);
call rand(randy);
PlotDotAt(x+f*randx,y+f*randy);
end
loop2: Do i = 0 to maxit;
```

114. The resource editor has several editors specialized for each kind of resource. You get a dialog box editor, a menu editor, a string table editor, an accelerators editor, and an image editor. Each one is called automatically when clicking in a resource from the menu, obtained with the *dir* button.

```
theta = float(i)/50;
theta2 = (float(i)/50)-3.14;
r = scale*exp(theta2*tan(in));
if r > cut then leave loop2;
x = r * cos(theta)+50;
y = r*sin(theta)+50;
call rand(randx);
call rand(randy);
PlotDotAt(x+f*randx,y+f*randy);
end
```

This are quite clear specs. Much clearer than other "specs" you will find in your future career as programmer… So let's translate this into C. We can start with the following function:

```
void DrawGalaxy(HDC hDC,double in,
                int maxit,double scale,
                double cut, double f)
{
     double theta, theta2, r, x, y, randx, randy;
     for (int i = 0; i <= maxit; i++) {                        (1)
          theta = ((double)i)/CENTER;                         (2)
          r = scale*exp(theta*tan(in));                       (3)
          if (r > cut) break;                                 (4)
          x = r * cos(theta)+CENTER;                          (5)
          y = r * sin(theta)+CENTER;                          (6)
          randx = (double)rand() / (double)RAND_MAX;          (7)
          randy = (double)rand() / (double)RAND_MAX;          (8)
          PlotDotAt(hDC,x+f*randx,y+f*randy,RGB(0,0,0));
     }
     for (int i = 0; i <= maxit; i++) {
          theta = ((double)i)/CENTER;
          theta2 = ( ((double)i)/CENTER) –3.14;
          r = scale * exp(theta2*tan(in));                    (9)
          if (r > cut) break;
          x = r*cos(theta)+CENTER;
          y = r*sin(theta)+CENTER;
          randx = (double)rand() / (double) RAND_MAX;
          randy = (double)rand() / (double) RAND_MAX;
          PlotDotAt(hDC,x+f*randx,y+f*randy,RGB(255,0,0));  (10)
     }
}
```

We translate both loops into two for statements. The exit from those loops before they are finished is done with the help of a break statement. This avoids the necessity of naming loops when we want to break out from them, what could be quite fastidious in the long term…

I suppose that in the language the author is using, loops go until the variable is equal to the number of iterations. Maybe this should be replaced by a strictly smaller than… but I do not think a point more will do any difference.

Note the cast of `i (double)i`. Note too that I always write 50.0 instead of 50 to avoid unnecessary conversions from the integer 50 to the floating-point number 50.0. This cast is not necessary at all, and is there just for "documentation" purposes. All integers when used in a double precision expression will be automatically converted to double precision by the compiler, even if there is no cast.

The functions exp and tan are declared in math.h. Note that it is imperative to include math.h when you compile this. If you don't, those functions will be assumed to be external functions that return an int, the default. this will make the compiler generate code to read an integer instead of reading a double, what will result in completely nonsensical results.

A break statement "breaks" the loop.

This statement means

```
r = (r*cos(theta)) + 5 and NOT
r = r * (cos(theta)+CENTER;
```

In line 8 we use the rand() function. This function will return a random number between zero and RAND_MAX. The value of RAND_MAX is defined in stdlib.h. If we want to obtain a random number between zero and 1, we just divide the result of rand() by RAND_MAX. Note that the random number generator must be initialized by the program before calling rand() for the first time. We do this in WinMain by calling

```
srand((unsigned)time(NULL));
```

This seeds the random number generator with the current time.

We are missing several pieces. First of all, note that CENTER is

```
#define CENTER 400
```

because with my screen proportions in my machine this is convenient. Note that this shouldn't be a #define but actually a calculated variable. Windows allows us to query the horizontal and vertical screen dimensions, but… for a simple drawing of a spiral a #define will do.

The function PlotPixelAt looks like this:

```
void PlotDotAt(HDC hdc,double x,double y,COLORREF rgb)
{
     SetPixel(hdc,(int)x,(int)y,rgb);
}
```

The first argument is an "HDC", an opaque pointer that points to a "device context", not further described in the windows documentation. We will speak about opaque data structures later. A COLORREF is a triple of red, green, and blue values between zero (black) and 255 (white) that describe the colors of the point. We use a simple schema for debugging purposes: we paint the first arm black (0,0,0) and the second red (255,0,0).

In event oriented programming, the question is "which event will provoke the execution of this code"?

Windows sends the message WM_PAINT to a window when it needs repainting, either because its has been created and it is blank, or it has been resized, or when another window moved and uncovered a portion of the window. We go to out MainWndProc function and add code to handle the message. We add:

```
     case WM_PAINT:
          dopaint(hwnd);
          break;
```

We handle the paint message in its own function. This avoids an excessive growth of the MainWndProc function. Here it is:

```
void dopaint(HWND hwnd)
{
     PAINTSTRUCT ps;
     HDC hDC;
     hDC = BeginPaint(hwnd,&ps);
     DrawGalaxy(hDC,3.0,20000,2500.0,4000.0,18.1);
     EndPaint(hwnd,&ps);
}
```

We call the windows API BeginPaint, passing it the address of a PAINTSTRUCT, a structure filled by that function that contains more information about the region that is to be painted, etc. We do not use it the information in it, because for simplicity we will repaint the

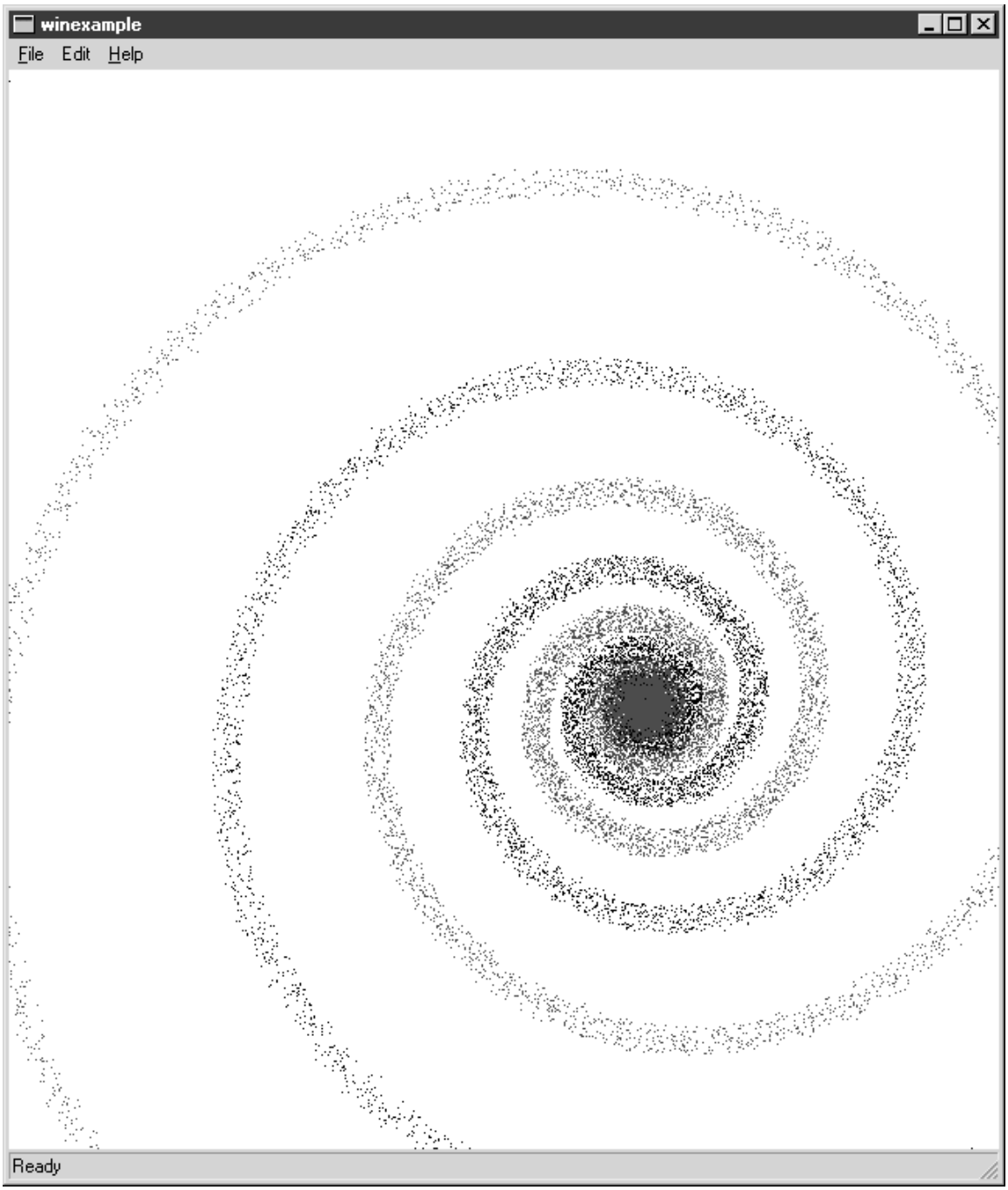
whole window each time we receive the message, even if we could do better and just repaint the rectangle that windows passes to us in that parameter. Then, we call the code to draw our galaxy, and inform windows that we are done with painting.

Well, this finishes the coding. We need to add the

```
#include <math.h>
#include <time.h>
```

at the beginning of the file, since we use functions of the math library and the time() function to seed the srand() function.

We compile and we obtain:



It would look better, if we make a better background, and draw more realistic arms, but for a start this is enough.

There are many functions for drawing under windows of course. Here is a table that provides short descriptions of the most useful ones:

Function	*Purpose*
AngleArc	Draws a line segment and an arc.
Arc	Draws an elliptical arc using the currently selected pen. You specify the bounding rectangle for the arc.
ArcTo	ArcTo is similar to the Arc function, except that the current position is updated.
GetArcDirection	Returns the current arc direction for the specified device context. Arc and rectangle functions use the arc direction.
LineTo	Draws a line from the current position up to, but not including, the specified point.
MoveToEx	Updates the current position to the specified point and optionally returns the previous position.
PolyBezier	Draws one or more Bézier curves.
PolyBezierTo	Same as PolyBézier but updates the current position.
PolyDraw	Draws a set of line segments and Bézier curves.
PolyLine	Draws a series of line segments by connecting the points in the specified array.
PolyLineTo	Updates current position after doing the same as PolyLine.
PolyPolyLine	Draws multiple series of connected line segments.
SetArcDirection	Sets the drawing direction to be used for arc and rectangle functions.

There are many other functions for setting color, working with rectangles, drawing text (TextOut), etc. Explaining all that is not the point here, and you are invited to read the documentation.

Summary: C converts integer and other numbers to double precision when used in a double precision expression. This will be done too when an argument is passed to a function. When the function expects a double and you pass it an int or even a char, it will be converted to double precision by the compiler.

All functions that return a double result must declare their prototype to the compiler so that the right code will be generated for them. An unprototyped function returning a double will surely result in incorrect results!

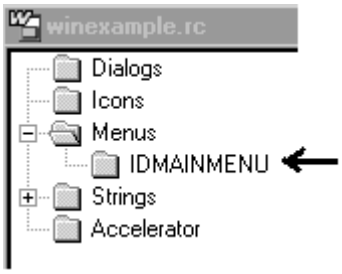
Opaque data structures are hidden from client code (code that uses them) by providing just a void pointer to them. This way, the client code is not bound to the internal form of the

structure and the designers of the system can modify it without affecting any code that uses them. Most of the windows data structures are used this way: an opaque "HANDLE" is given that discloses none of the internals of the object it is pointing to.

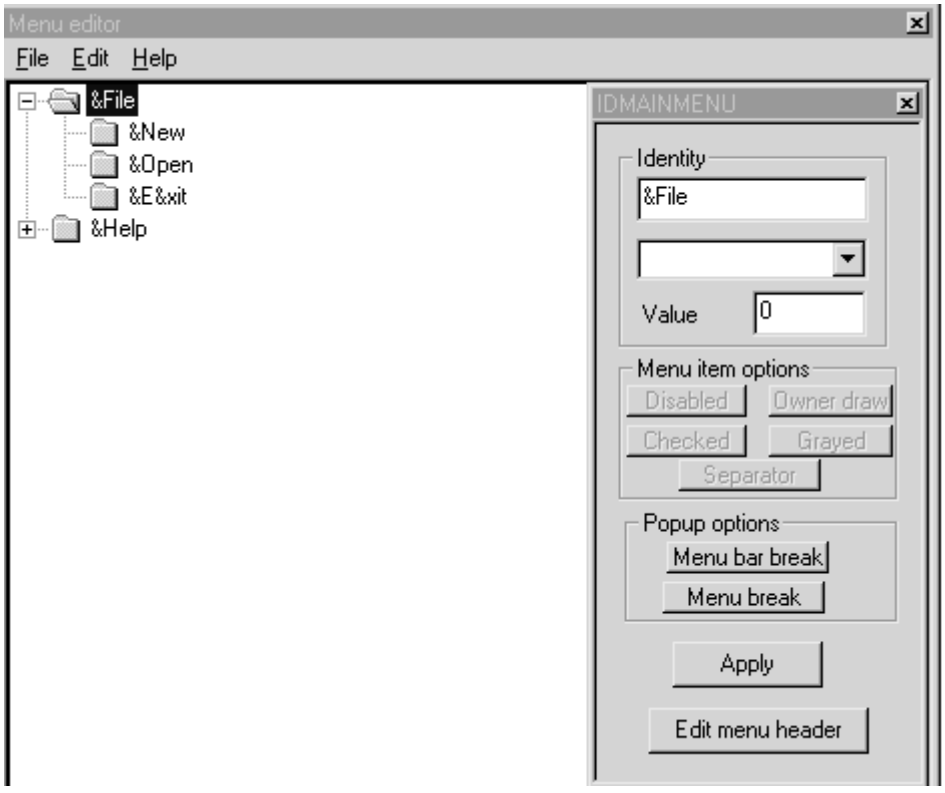
2.12 Filling the blanks

Input goes through dialog boxes under windows. They are ubiquitous; so let's start to fill our skeleton with some flesh. Let's suppose, for the sake of the example that we want to develop a simple text editor. It should read some text, draw it in the screen, and provide some utilities like search/replace, etc.

First, we edit our menu, and add an "edit" item. We open the directory window, and select the menu:

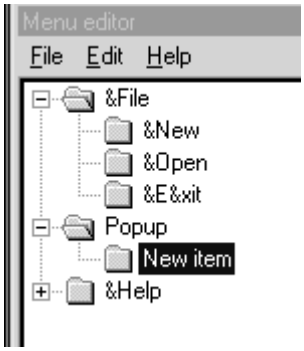


We arrive at the menu editor[115]. If we open each branch of the tree in its left side, it looks like this:



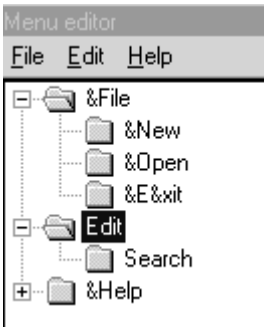
115. This type of interface requires an *action from the part of the user* to indicate when it is finished modifying the name and desires to "apply" the changes. Another possibility would be that the resource editor applies the changes letter by letter as the user types them in, as some other editors do. This has the advantage of being simpler to use, but the disadvantage of being harder to program and debug. As always, an the appearance of the user interface is not only dictated by the user comfort, but also by the programming effort necessary to implement it. You will see this shortly when you are faced with similar decisions.

We have at the left side the tree representing our menu. Each submenu is a branch, and the items in the branch; the leaves are the items of the submenu. We select the "File" submenu and press the "insert" key. We obtain a display like this:



A new item is inserted after the currently selected one. The name is "Popup", and the first item is "New item". We can edit those texts in the window at the right: We can change the symbolic name, and set/unset several options. When we are finished, we press "Apply" to write our changes to the resource. [116]

OK, we change the default names to the traditional "Edit" and "Search", to obtain this display:



We will name the new item `IDM_SEARCH`. I am used to name all those constants starting with IDM_ from ID Menu, to separate them in my mind from IDD_ (ID Dialog).

We can now start drawing the "Search" dialog. Just a simple one: a text to search, and some buttons to indicating case sensitivity, etc. We close the menu editor, and we start a new dialog.

---

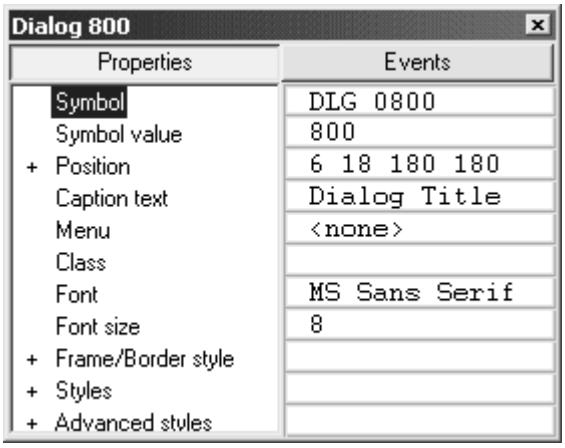
116. A debugger is a program that starts another program, the "program to be debugged" or "debuggee", and can execute it under the control of the user, that directs the controlled execution. All C development systems offer some debugger, and lcc-win32 is no exception. The debugger is described in more detail in the user's manual, and it will not be described here. Suffice to note that you start it with F5 (or Debugger in the compiler menu), you can single step at the same level with F4 and trace with F8. The debugger shows you in yellow the line the program will execute next, and marks breakpoints with a special symbol at the left. Other debuggers may differ from this of course, but the basic operations of all of them are quite similar. Note that lcc-win32 is binary compatible with the debugger of Microsoft: you can debug your programs using that debugger too.

To be able to use the debugger you need to compile with the g2 flag on. That flag is normally set by default. It directs the compiler to generate information for the debugger, to enable it to show source lines and variable values. The compiler generates a whole description of each module and the structures it uses called "debug information". This information is processed by the linker and written to the executable file. If you turn the debugging flag *off* the debugger will not work. The best approach is to leave this flag *on* at all times. Obviously the executable size will be bigger, since the information uses up space on disk. If you do not want it, you can instruct the linker to ignore it at link time. In this way, just switching that linker flag *on* again will allow you to debug the program.

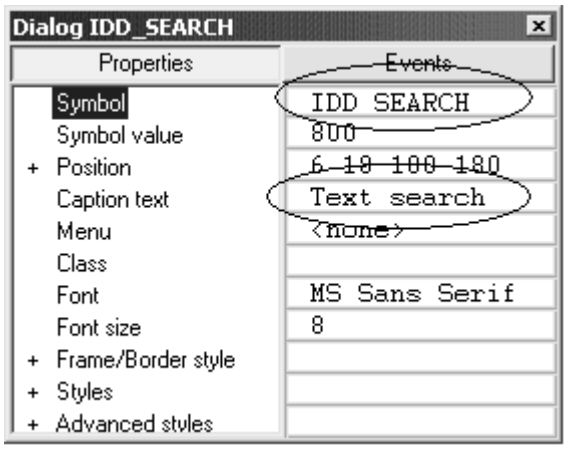
The debug information generated by lcc-win32 uses the NB09 standard as published by Microsoft and Intel. This means that the programs compiled with lcc-win32 can be debugged using another debugger that understands how to use this standard.

In the “Resources” submenu, we find a “New” item, with several options in it. We choose the “dialog” option.

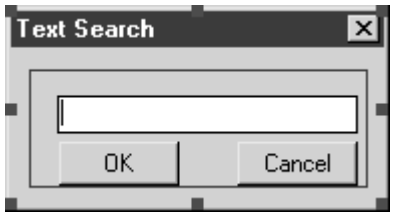
The dialog editor builds an empty dialog and we are shown the following parameters dialog:



Even if this quite overwhelming, we are only interested in two things: the title of the dialog and the symbolic identifier. We leave all other things in their default state. We name the dialog IDD_SEARCH, and we give it the title “Text search”. After editing it looks like this:



We press the OK button, and we do what we did with the dialog in the DLL, our first example. The finished dialog should look roughly like this:



An edit field, and two push button for OK and Cancel. The edit field should receive the ID `IDTEXT`.

Now comes the interesting part. How to connect all this?

We have to first handle the WM_COMMAND message, so that our main window handles the menu message when this menu item is pressed. We go to our window procedure MainWnd-Proc. Here it is:

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM
lParam)
{
     switch (msg) {
     case WM_SIZE:
          SendMessage(hWndStatusbar,msg,wParam,lParam);
          InitializeStatusBar(hWndStatusbar,1);
          break;
     case WM_MENUSELECT:
          return MsgMenuSelect(hwnd,msg,wParam,lParam);
     case WM_COMMAND:
     HANDLE_WM_COMMAND(hwnd,wParam,lParam,MainWndProc_OnCommand);
          break;
     case WM_DESTROY:
          PostQuitMessage(0);
          break;
     default:
          return DefWindowProc(hwnd,msg,wParam,lParam);
     }
     return 0;
}
```

We can see that it handles already quite a few messages. In order,

We see that when the main window is resized, it resizes its status bar automatically.

When the user is going through the items of our menu, this window receives the `WM_MENUSELECT` message from the system. We show the appropriate text with the explanations of the actions the menu item in the status bar.

When a command (from the menu or from a child window) is received, the parameters are passed to a macro defined in windowsx.h that breaks up the wParam and lParam parameters into their respective parts, and passes those to the `MainWndProc_OnCommand` function.

When this window is destroyed, we post the quit message.

The function for handling the commands looks like this:

```
void MainWndProc_OnCommand(HWND hwnd,
     int id, HWND hwndCtl, UINT codeNotify)
{
     switch(id) {
     // ---TODO--- Add new menu commands here
     case IDM_EXIT:
     PostMessage(hwnd,WM_CLOSE,0,0);
     break;
     }
}
```

We find a comment as to where we should add our new command. We gave our menu item "Search" the symbolic ID of `IDM_SEARCH`. We modify this procedure like this:

```
void MainWndProc_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT
codeNotify)
{
     switch(id) {
          // ---TODO--- Add new menu commands here
          case IDM_SEARCH:
               {
                    char text[1024];
                    if (CallDialog(IDD_SEARCH,SearchDlgProc,
                                   (LPARAM)text))
                         DoSearchText(text);
               }
```

```
                break;
            case IDM_EXIT:
            PostMessage(hwnd, WM_CLOSE,0,0);
            break;
        }
    }
```

When we receive the menu message then, we call our dialog. Since probably we will make several dialogs in our text editor, it is better to encapsulate the difficulties of calling it within an own procedure: CallDialog. This procedure receives the numeric identifier of the dialog resource, the function that will handle the messages for the dialog, and an extra parameter for the dialog, where it should put the results. We assume that the dialog will return TRUE if the user pressed OK, FALSE if the user pressed the Cancel button.

If the user pressed OK, we search the text within the text that the editor has loaded in the function DoSearch.

How will our function CallDialog look like?

Here it is:

```
int CallDialog(int id,DLGPROC proc,LPARAM parameter)
{
     int r = DialogBoxParam(hInst,MAKEINTRESOURCE(id),
hwndMain, proc, parameter);
     return r;
}
```

We could have returned the value of the DialogBoxParam API immediately but I like storing function return values in local variables. You never know what can happen, and those values are easily read in the debugger.

We have to write a dialog function, much like the one we wrote for our string DLL above. We write a rough skeleton, and leave the details for later:

```
BOOL CALLBACK SearchDlgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
     switch (msg) {
          case WM_INITDIALOG:
               return TRUE;
          case WM_CLOSE:
               EndDialog(hwnd,0);
               break;
     }
     return FALSE;
}
```

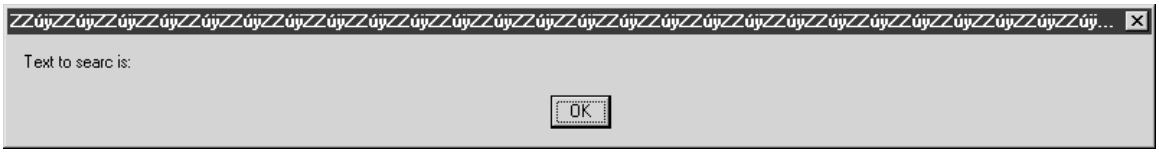
This does nothing, it doesn't even put the text in the received parameter, but what we are interested in here, is to first ensure the dialog box shows itself. Later we will refine it. I develop software like this, as you may have noticed: I try to get a working model first, a first approximation. Then I add more things to the basic design. Here we aren't so concerned about design anyway, since all this procedures are very similar to each other.

The other procedure that we need, DoSearchText, is handled similarly:

```
int DoSearchText(char *txt)
{
     MessageBox(NULL,"Text to search:",txt, MB_OK );
     return 1;
}
```

We just show the text to search. Not very interesting but…

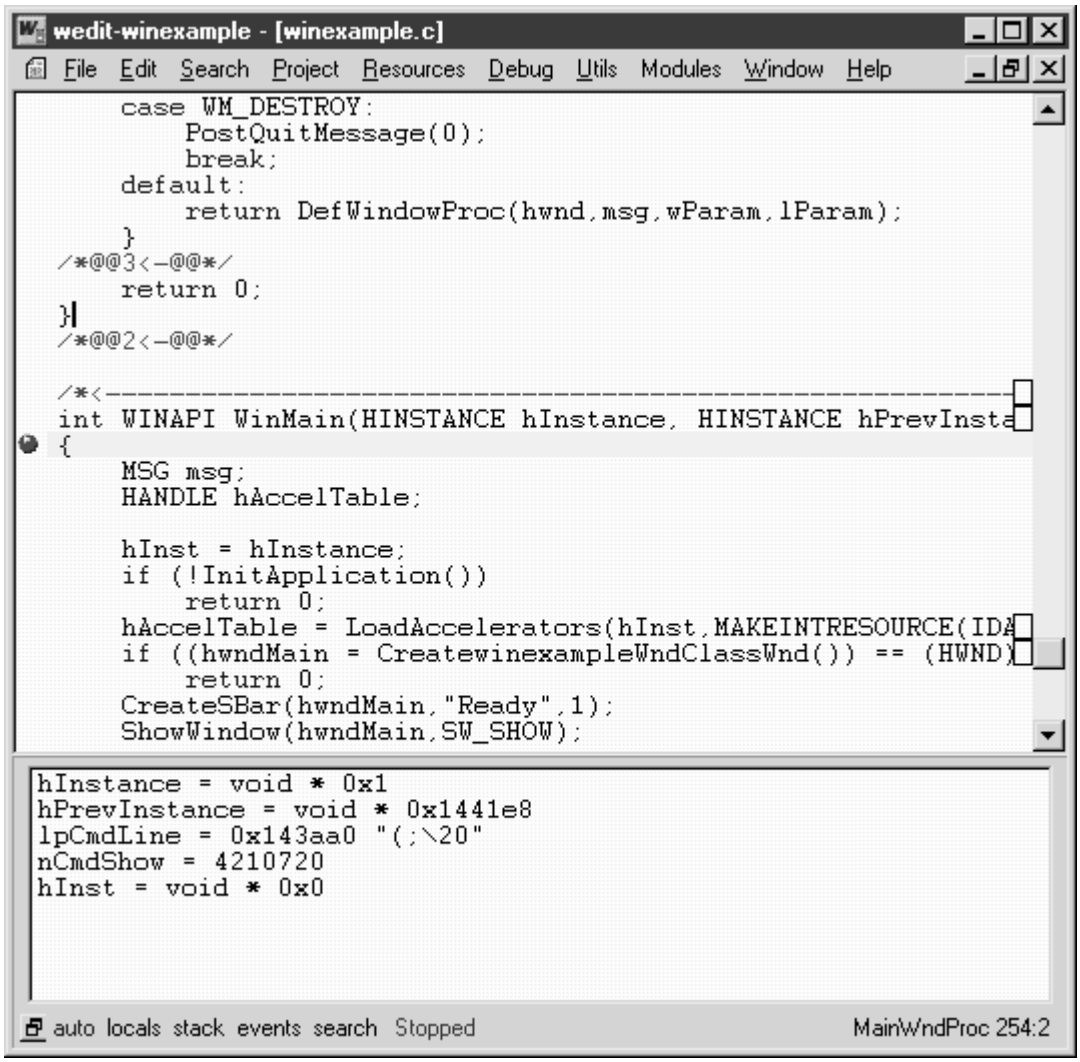
We compile, link, the main application window appears, we select the "Search" menu, and… we see:



What's wrong?????

Well, we have inverted the parameters to the `MessageBox` procedure, but that's surely not the point. Why is the dammed dialog box not showing?

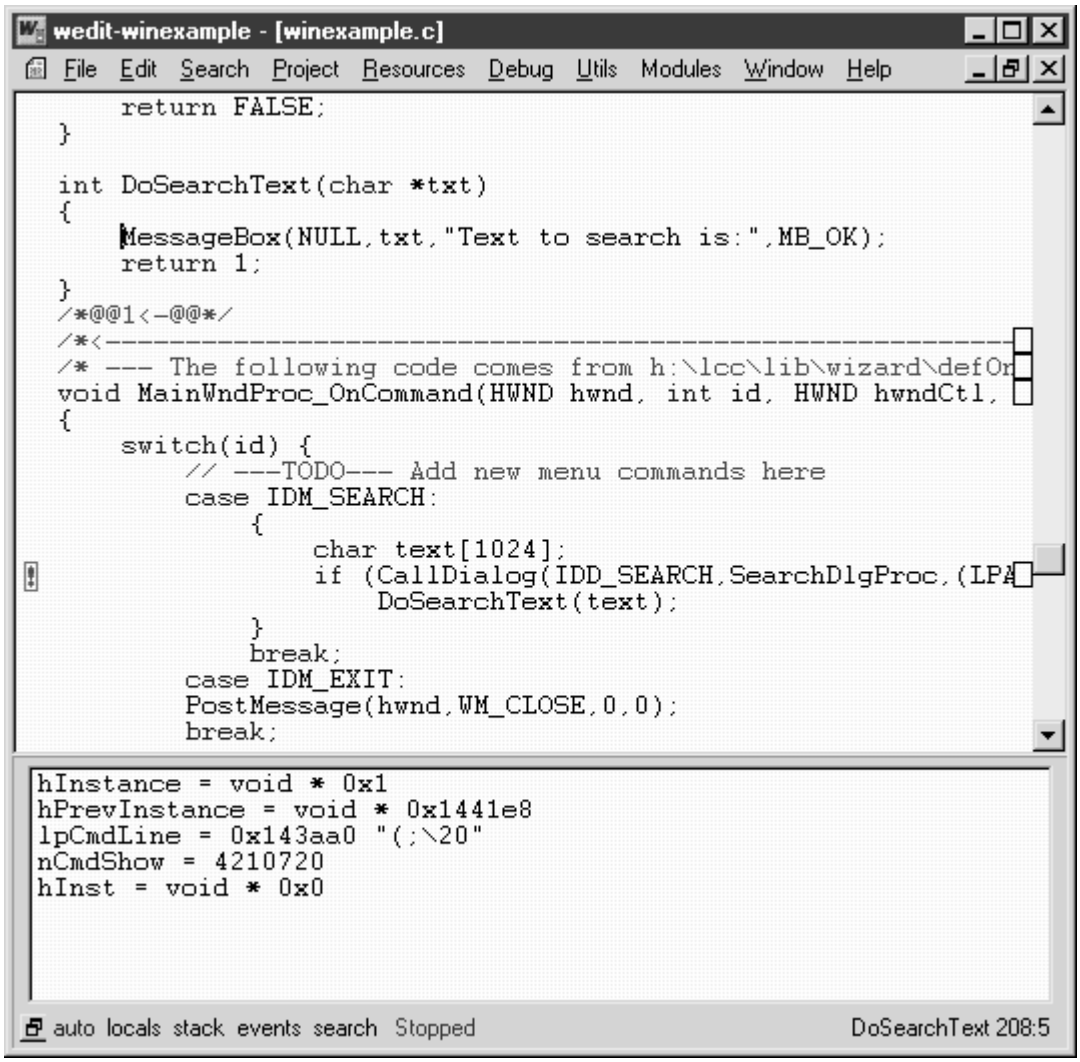
Well, here we need a debugger.[117] We need to know what is happening when we call the dialog box. We press F5, and we start a debugging session. The debugger stops at WinMain.



Now, wait a minute, our window procedure that receives the message from the system is called indirectly from Windows, we can't just follow the program blindly. If we did that, we would end up in the main loop, wasting our time.

117. Why didn't we use the DLL to ask for the string to search? Mostly because I wanted to give you an overview of the whole process. A good exercise would be to change the program to use the DLL. Which changes would be necessary? How would you link?

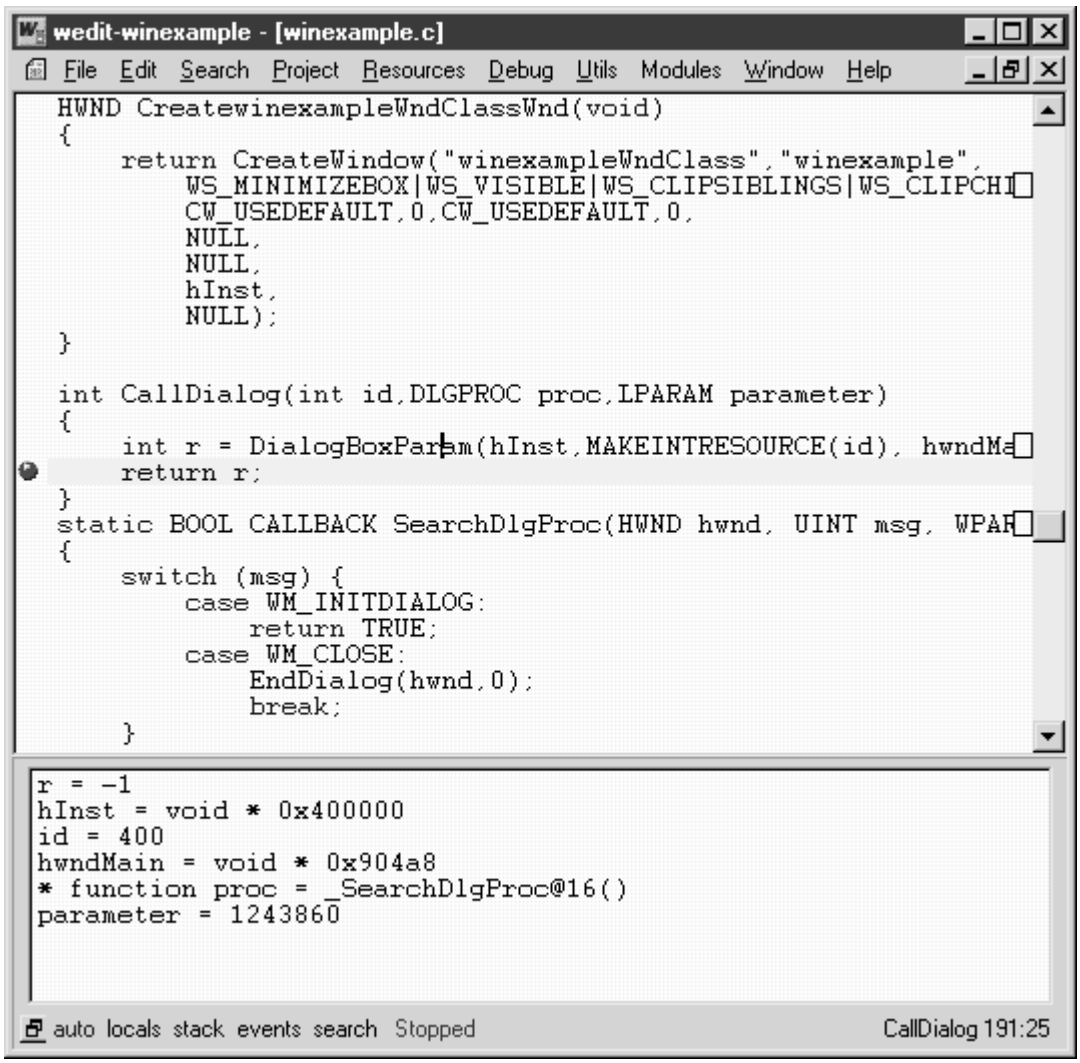
No, we have to set a breakpoint there. We set a breakpoint when we call the dialog using the F2 accelerator key. We see that Wedit sets a sign at the left to indicate us that there is a breakpoint there. Then we press F5 again to start running the program.



Our program starts running, we go to the menu, select the search item, and Wedit springs into view. We hit the breakpoint. Well that means at least that we have correctly done things until here: the message is being received. We enter into the CallDialog procedure using the F8 accelerator. We step, and after going through the DialogBoxParam procedure we see no dialog and the return result is –1. The debugger display looks like this:

We see the current line highlighted in yellow, and in the lower part we see the values of some variables. Some are relevant some are not. Luckily the debugger picks up r as the first one. Its value is –1.

Why –1?



A quick look at the doc of `DialogBoxParam` tells us "If the function fails, the return value is -1."

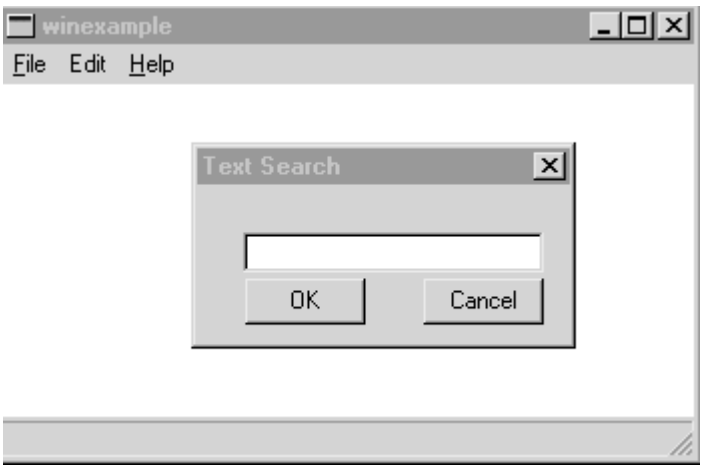
Ahh, how clear. Yes of course, it failed. But why?

Mystery. There are no error codes other than just general failure. What could be wrong?

Normally, this –1 means that the resource indicated by the integer code could not be found. I have learned that the hard way, and I am writing this tutorial for you so that you learn it the easy way. The most common cause of this is that you forgot to correctly give a symbolic name to your dialog.

We close the debugger, and return to the resource editor. There we open the dialog box properties dialog box (by double clicking in the dialog box title bar) and we see… that we forgot to change the name of the dialog to `IDM_SEARCH`!!! We correct that and we try again.

OK, this looks better. The dialog is showing.



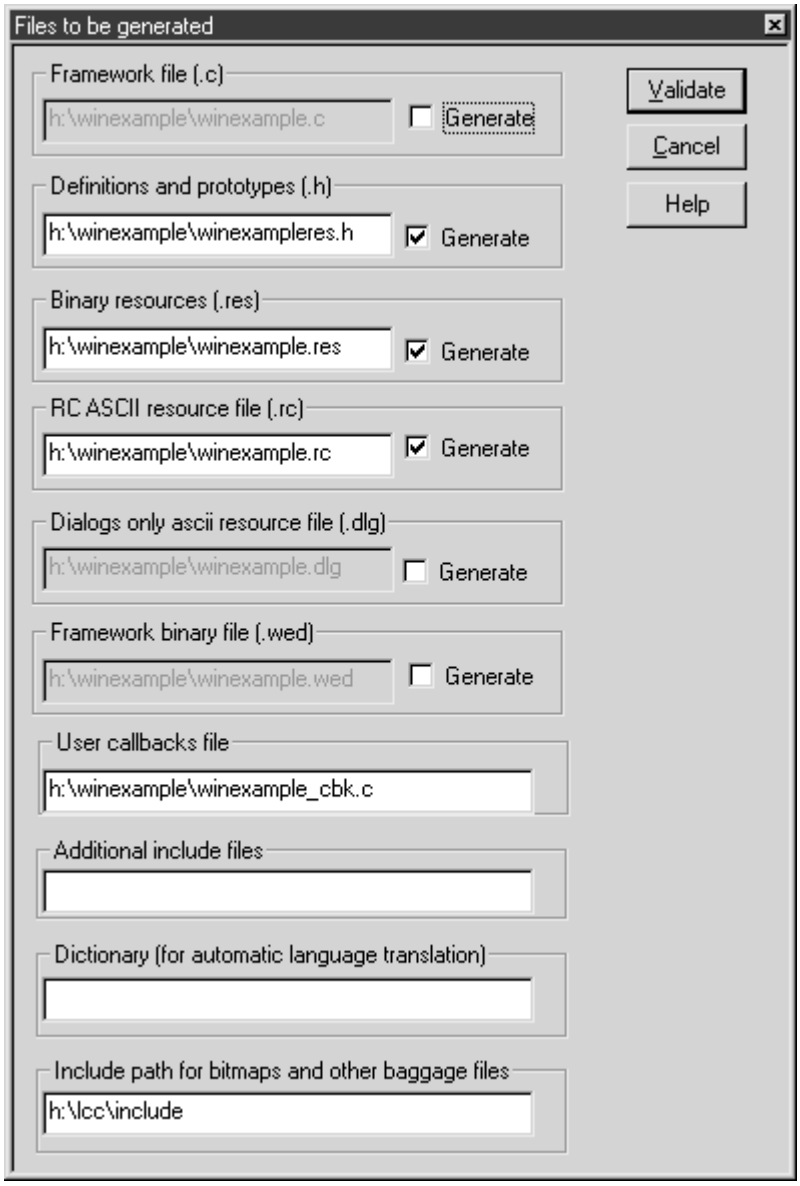
The rest is quite trivial most of the wok was done when building the DLL. Actually, the dialog box is exactly the same.[118]

118. Newer versions of Wedit check for this. Older ones aren't so sophisticated so please take care.

2.13 Using the graphical code generator

As we saw before, writing all that code for a simple dialog is quite a lot of work. It is important that you know what is going on, however. But now we can see how we can make Wedit generate the code for us.

The code generator is not enabled by default. You have to do it by choosing the "Output" item in the resources menu. This leads to an impressing looking box like this:



This dialog shows you the files that will be generated or used by Wedit. Those that will be generated have a box at the right, to enable or disable their generation. The others are used if available, but only for reading. The last item is an additional path to look for bitmaps, icons and other stuff that goes into resources.

You notice that the first item is disabled. You enable it, and type the full path of a file where you want Wedit to write the dialog procedures. Notice that by default, the name of this file is <name of the project>.c. This could provoke that your winexample.c that we worked so hard to write, could be overwritten.[119] Choose another name like "dialogs.c" for instance.

119. Look in the weditreslib folder. The file commmsg.c contains the default procedure and all the machinery necessary for the run time of the framework.

Now, when you save your work, all the dialog procedures will be written for you. But before, we have to tell the framework several things.

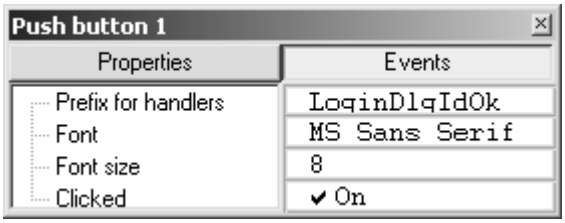
The first will be the prefix to use for all the procedures that will be used in this dialog box. We define this in the main properties dialog box obtained when you double click in the dialog title.

"Dlg400" is an automatic generated name, not very convincing. We can put in there a more meaningful name like "DlgSearch" for instance. We will see shortly where this name is used.

What we want to do now is to specify to the editor where are the events that interest us. For each of those events we will specify a *callback procedure* that the code generated by the editor will call when the event arrives. Basically all frameworks, no matter how sophisticated, boil down to that: a quick way of specifying where are the events that you want to attach some code to.

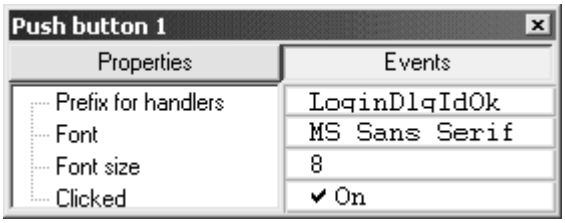
The events we can choose from are in the properties dialog boxes, associated with each element of the dialog box. You have the general settings in the properties window associated with the dialog box (that appears when you double click in the title), and you have the buttons properties that appear when you right click in a button.

Those dialog boxes have generally a standard or «Properties» part that allows you to change things like the element's text, or other simple properties, and a part that is visible only when you have selected the C code generation. That part is normally labeled "events" and appears at the right. That label tells us which kind of events the framework supports: window messages. There are many events that can possibly happen of course, but the framework handles only those.



We see at the right tab a typical "messages" part: We have some buttons to choose the messages we want to handle, the function name prefix we want for all callback procedures for this element, and other things like the font we want to use when the elements are displayed.

We see again that "Dlg400"… but this allows us to explain how those names are generated actually. The names of the generated functions are built from the prefix of the dialog box, then the prefix for the element, and then a fixed part that corresponds to the event name. We edit the prefix again, following this convention.



The "Selected" message is on, so the framework will generate a call to a function called `DlgSearchOnOkSeelected()`. Happily for us, we do not have to type those names ourselves.

Without changing anything else we close the button properties and save our work. We open the c source file generated by the editor.

We obtain the following text:

```
/* Wedit Res Info */
#ifndef __windows_h
#include <windows.h>
#endif
#include "winexampleres.h"

BOOL APIENTRY DlgSearch(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
     static WEDITDLGPARAMS WeditDlgParams;

     switch(msg)
     {
          case WM_INITDIALOG:
               SetWindowLong(hwnd,DWL_USER,
                    (DWORD)&WeditDlgParams);
               DlgSearchInit(hwnd,wParam,lParam);
               /* store the input arguments if any */
               SetProp(hwnd,"InputArgument",(HANDLE)lParam);
               break;
          case WM_COMMAND:
               switch (LOWORD(wParam))
               {
                    case IDOK:
                         DlgSearchOnOKSelected(hwnd);
                    break;
               }
               break;
     }
     return(HandleDefaultMessages(hwnd,msg,wParam,lParam));
}
```

We have here a normal callback procedure for our dialog. It handles two messages: `WM_INITDIALOG` and `WM_COMMAND`. The callback procedures are in bold type. There are two of them: The initialization callback called **"DlgSearchInit"**, and the one we attached to the OK button above, **"DlgSearchOnOkSelected"**.

There are more things in there, but for the time being we are interested in just those ones, because they have to be written by you!

What you want to do when the dialog box has been created but it is not yet visible?

This is the purpose of the first callback. In our text search dialog we could write in the edit field the last searched word, for instance, to avoid retyping. Or we could fill a combo box with all the words the user has searched since the application started, or whatever. Important is that you remember that in that function all initializations for this dialog box should be done, including memory allocation, populating list boxes, checking check buttons or other chores.

The second callback will be called when the user presses the OK button. What do you want to do when this event happens? In our example we would of course start the search, or setup a set of results so that the procedure that called us will know the text to search for, and possibly other data.

Different controls will react to different events. You may want to handle some of them. For instance you may want to handle the event when the user presses a key in an edit field, to check input. You can use the framework to generate the code that will trap that event for you, and concentrate in a procedure that handles that event.

How would you do this?

You open the properties dialog of the edit control and check the “Update” message. This will prompt the editor to generate a call to a function of yours that will handle the event. The details are described in the documentation of the resource editor and will not be repeated here. What is important to retain are the general principles at work here. The rest is a matter of reading the docs, to find out which events you can handle for each object in the dialog, and writing the called functions with the help of the windows documentation.

But what happens if there is an event that is not foreseen by the framework? With most frameworks this is quite a problem, happily not here. You have just to check the button “All” in the dialog properties and the framework will generate a call to a default procedure (named <prefix>Default) at each message. There you can handle all events that the operating system sends. I have tried to keep the framework open so that unforeseen events can be still treated correctly.

Another way to customize the framework is to modify the default procedure provided in the library weditres.lib. The source of that procedure is distributed in the source distributions of lcc-win32[120] and is relatively easy to modify.

120. To find that easily just press F12 and click in its name in the function list.

2.14 Customizing controls

Under windows you have a set of building blocks at your disposal that is, even it is quite rich, fixed. A plain listbox will look like a listbox and there is no obvious way to modify the appearence of it, not even for doing such simple things like writing the text in red, or painting the background in brown.

The reason for this is simple: there are infinitely many ways of customizing a control, and if Windows would provide an API for each one, the windows API would not fit in a computer no matter what.

The solution provided by the system is a general interface that gives you enough elements to be able to do whatever you want to do. You can:

1) customize the appearance of a control by making some of its parts “owner draw”, i.e. drawed by yourself.

2) Process some special windows messages that allow you to change the font, the background color, and other characteristics by modifying the settings of a passed device context (HDC).

In both cases the interface is quite simple: Windows sends a message to the procedure of the parent window of the control, to ask it to draw a specific part of the control. Some (or all) of the control behavior is retained, but the appearance can be changed as you like.

2.14.1 Processing the WM_CTLCOLORXXX message

We can start with a simple example that draws a listbox in a special way. Let’s use the second method first, since it is easier.

Suppose we want to display the text in a listbox in red color. In the dialog procedure which owns the list box (or in the windows procedure if it is a list box created independently) you receive the WM_CTLCOLORLISTBOX message. In the wparam parameter you receive an HDC that you can customize using SetTextColor to draw the text with the color you wish. To inform windows that you have processed this message, you should return a handle to a brush that will be used to paint the background of the listbox. Notice that windows does not destroy this object. You should destroy it when the dialog box finishes.

Creating and destroying the brush that you should return is cumbersome. If you are not interested in modifying the background, you can use the API GetStockObject() returning the handle of the white brush like this:

```
return GetStockObject(WHITE_BRUSH);
```

Objects created with GetStockObejct do not need to be destroyed, so you spare yourself the job of creating and destroying the brush. The code would look like this:

```
BOOL WINAPI MyDlgProc(HWND hwnd, UINT msg,WPARAM wParam,LPARAM lParam)
{
     HDC hDC;
     COLORREF oldcolor;
     switch (msg) {
          ...
          case WM_CTLCOLORLISTBOX:
               hDC = (HDC)wParam;
               color = RGB(192,0,0); // red color
               SetTextColor(hDC,color);
               return GetStockObject(WHITE_BRUSH);
          ...
     }
}
```

If you want to return another brush than white, you should handle the WM_INITDIALOG message to create the brush, and the WM_NCDESTROY message to destroy the brush you created to avoid memory leaks. In this case, the above example should be augmented with:

```
BOOL WINAPI MyDlgProc(HWND hwnd, UINT msg,WPARAM wParam,LPARAM lParam)
{
     HDC hDC;
     COLORREF oldcolor;
     static HBRUSH hBrush;
     switch (msg) {
          case WM_INITDIALOG:
               hBrush = CreateSolidBrush(BK_COLOR);
               // other initializations go here
          ...
          case WM_CTLCOLORLISTBOX:
               hDC = (HDC)wParam;
               color = RGB(192,0,0); // red color
               SetTextColor(hDC,color);
               return hBrush;
          ...
          case WM_NCDESTROY:
               DeleteObject(hBrush);
               break;
     }
}
```

We create the brush at the initialization of the list box, we use it when we receive the WM_CTLCOLORLISTBOX message, and we destroy it when we receive the WM_NCDESTROY message.

This technique can be used not only with list boxes but with many other controls. We have:

Message	*Control*
WM_CTLCOLORBTN	Button
WM_CTLCOLOREDIT	Edit field
WM_CTLCOLORDLG	Dialog box
WM_CTLCOLORLISTBOX	List box
WM_CTLCOLORSCROLLBAR	Scroll bar
WM_CTLCOLORSTATIC	Static controls (text, etc.)
WM_CTLCOLORLISTBOX + WM_CTLCOLOREDIT	A combo box sends both messages to the parent window

All this messages have the same parameters as in the list box example and the procedure is exactly the same.

The WM_CTLCOLORSTATIC is interesting because it allows you to add text in any font and color combination to a window or dialog by just calling SelectObject with a special font or font size using the passed HDC of this message.

2.14.2 Using the WM_DRAWITEM message

Suppose we want to draw a list box that draws an image at the left side of the text in each line. In this case, just setting some properties of the HDC we receive with WM_CTLCOLORLISTBOX will not cut it. We need to do the entire drawing of each line of the list box.

We can do this by setting the style of the list box to owner draw (the resource editor will do this for you) and we need to process the WM_DRAWITEM message. Windows sends us a pointer to a DRAWITEMSTRUCT that contains all the necessary information to do the drawing in the LPARAM parameter. This structure is defined as follows:

Member	*Description*
`unsigned int CtlType`	This member can be one of the following values. ODT_BUTTON button ODT_COMBOBOX combo box ODT_LISTBOX list box ODT_LISTVIEW List-view control ODT_MENU Owner-drawn menu item ODT_STATIC static control ODT_TAB Tab control
`unsigned int CtlID`	Specifies the identifier of the combo box, list box, button, or static control. This member is not used for a menu item.
`unsigned int itemID`	Specifies the menu item identifier for a menu item or the index of the item in a list box or combo box. For an empty list box or combo box, this member can be–1. This allows the application to draw only the focus rectangle at the coordinates specified by the rcItem member even though there are no items in the control
`unsigned int itemAction`	Specifies the drawing action that windows expect us to perform. This action is indicated with the following flags: ODA_DRAWENTIRE The entire control needs to be drawn. ODA_FOCUS The control has lost or gained the keyboard focus. The itemState member should be checked to determine whether the control has the focus. ODA_SELECT The selection status has changed. The itemState member should be checked to determine the new selection state.
`HWND hwndItem`	Window handle of the control
`RECT rcItem`	Rectangle where we should do our drawing
`unsigned long *itemData`	Specifies the application-defined value associated with the menu item.

Member	*Description*
`unsigned int itemState`	State of the item after the current drawing action takes place. This can be a combination of the following values: ODS_CHECKED This bit is used only in a menu. ODS_COMBOBOXEDIT The drawing takes place in the edit control of an owner-drawn combo box. ODS_DEFAULT The item is the default item. ODS_DISABLED The item is to be drawn as disabled. ODS_FOCUS The item has the keyboard focus. ODS_GRAYED This bit is used only in a menu. ODS_HOTLIGHT The item is being hot-tracked, that is, the item will be highlighted when the mouse is on the item. ODS_INACTIVE The item is inactive and the window associated with the menu is inactive. ODS_NOACCEL The control is drawn without the keyboard accelerator cues. ODS_NOFOCUSRECT The control is drawn without focus indicator cues. ODS_SELECTED The menu item's status is selected.

With all this information, we can do whatever we want inside the rcItem rectangle. Here is the example for our problem of drawing a bitmap in each line of the combo box.

First, in our dialog procedure we handle the message WM_DRAWITEM, of course:

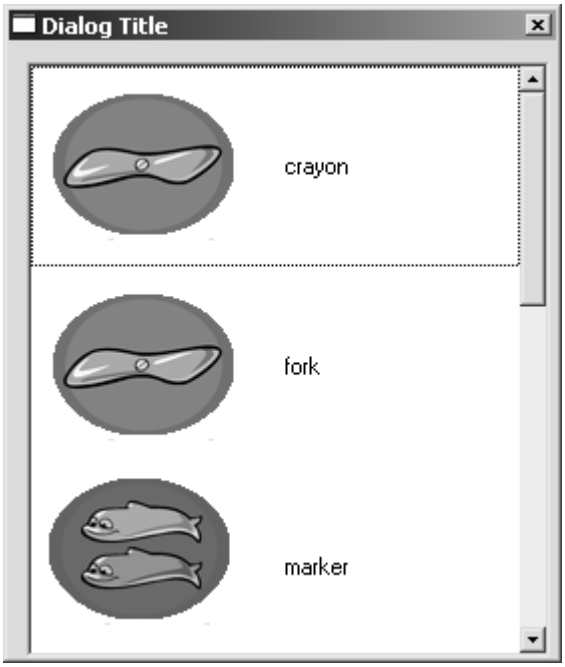
```
BOOL WINAPI MyDlgProc(HWND hwnd, UINT msg,WPARAM wParam,LPARAM lParam)
{
     HDC hDC;
     COLORREF oldcolor;
     switch (msg) {
          case WM_INITDIALOG:
          LoadListboxBitmaps(hwnd);
          // other initializations can follow
          ...
          case WM_DRAWITEM:
          DrawListboxLine(hwnd,(LPDRAWITEMSTRUCT)lParam);
          return 1; // We processed this message
          ...
          case WM_NCDESTROY:
          DestroyListboxBitmaps(hwnd);
          // other cleanup tasks can go here
     }
}
```

Our procedure to draw each line of the dialog box can look like this:

```
long DrawListboxLine(HWND hwndDlg,LPDRAWITEMSTRUCT lpdis)
{
     HBITMAP hbmpPicture,hbmpOld;
     int y,state;
     HDC hdcMem;
     TEXTMETRIC tm;
     RECT rcBitmap;
     COLORREF oldcolor;
     char tchBuffer[200];

     // If there are no list box items, skip this message.
     if (lpdis->itemID == -1) return 0;
```

```
        switch (lpdis->itemAction) {
        case ODA_SELECT:
        case ODA_DRAWENTIRE:
             // Windows DOES NOT erase the background of owner draw list boxes.
             // Erase the background to start with a coherent state.
             FillRect(lpdis->hDC,&lpdis->rcItem,
                  GetStockObject (WHITE_BRUSH) );

             // We get the handle of the bitmap to draw from the item-data slot in each
             // list box line. The initialization procedure sets this
             hbmpPicture =(HBITMAP)SendMessage(lpdis->hwndItem,
                  LB_GETITEMDATA, lpdis->itemID, (LPARAM) 0);

             // We select the image in a memory DC, then we select it. This is a
             // standard technique for copying a bitmap into the screen
             hdcMem = CreateCompatibleDC(lpdis->hDC);
             hbmpOld = SelectObject(hdcMem, hbmpPicture);
             BitBlt(lpdis->hDC,
                  lpdis->rcItem.left, lpdis->rcItem.top,
                  lpdis->rcItem.right - lpdis->rcItem.left,
                  lpdis->rcItem.bottom - lpdis->rcItem.top,
                  hdcMem, 0, 0, SRCCOPY);

             // We are done with our memory DC. Clean up.
             SelectObject(hdcMem, hbmpOld);
             DeleteDC(hdcMem);

             // Now, we retrieve the text for this line.
             SendMessage(lpdis->hwndItem, LB_GETTEXT,
                  lpdis->itemID, (LPARAM) tchBuffer);

             // How big is the text of this item?
             GetTextMetrics(lpdis->hDC, &tm);

             // Vertically center the text
             y = (lpdis->rcItem.bottom+lpdis->rcItem.top-tm.tmHeight)/2;

             // Send the text to the screen 6 pixels after the bitmap
             TextOut(lpdis->hDC,XBITMAP+6,y,tchBuffer,strlen(tchBuffer));

             // Is the item selected?
             if (lpdis->itemState & ODS_SELECTED) {
                  // Yes. Invert the whole rectangle (image+text) to signal the selection
                  BitBlt(lpdis->hDC,
                       lpdis->rcItem.left, lpdis->rcItem.top,
                       lpdis->rcItem.right - lpdis->rcItem.left,
                       lpdis->rcItem.bottom - lpdis->rcItem.top,
                       lpdis->hDC, 0, 0, PATINVERT);
             }
             break;
        // Do nothing if we gain or lose the focus. This is a laxist view but... it works!
        case ODA_FOCUS:
             break;
        }
        return TRUE;
   }
```

And after all this work, here it is:



2.15 Building custom controls

In many cases it is a good idea to develop a class of windows that can be used in several applications with the same interface. The idea is to encapsulate into a dll module or just in a static library, a specialized window that will do one kind of thing: a grid for a database application, or, for using a simpler example, an lcd simulation display.

2.15.1 An lcd display

The objective is to build a simple display window that simulates a small "lcd" display that shows a few lines of text. To build this application we follow the usual procedure: we generate an application with the wizard, and we change the display of the window: instead of drawing a blank window we add a handler for the WM_PAINT message like this:

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM
lParam)
{
     switch (msg) {
          ...
     case WM_PAINT:
          DoPaint(hwnd);
          break;
     case WM_DESTROY:
          ...
     }
     return 0;
}
```

In our "DoPaint" procedure we will do just that: paint the lcd window. We need first to draw a simple character set, from which we will copy the characters into the display. Here is a simple one. In this bitmap we use three colors: a light blue for drawing the body of the characters, a

darker blue for drawing the background of the characters, and black, to draw the space between each character.[121] Our drawing procedure "DoPaint" will make a mapping between the bitmap colors and the color we want to give to our simulated display.

To draw the characters, we will make a memory display context, where we will store the bitmap of characters. When we want to draw one char, we will copy from the memory bitmap to the display window the portion of the bitmap that contains our character, translating the colors as we go. We have to take care of scrolling if necessary, and many other things. The purpose here is not to show you how you implement the best ever lcd display in a PC, but to show the interfaces of a custom control so we will keep the logic to the bare minimum.

Let's see our DoPaint procedure:

```
static void DoPaint(HWND hwnd)
{
1    PAINTSTRUCT ps;
2    HDC hDC = BeginPaint(hwnd,&ps);
3    int len = strlen(Text);
4    HBITMAP bmp,oldBitmap;
5    HDC dcMemory;
6    COLORMAP ColorMap[4];
7    int charcount = 0;
8    int linecount = 1;
9    RECT m_rect,rc;
10   int x,y;

12   ColorMap[0].from = SEGM_COLORS[0];
13   ColorMap[0].to  = PixelOnColor;
14   ColorMap[1].from = SEGM_COLORS[1];
15   ColorMap[1].to  = PixelOffColor;
16   ColorMap[2].from = SEGM_COLORS[2];
17   ColorMap[2].to  = PixelBackColor;
18   bmp = CreateMappedBitmap(hInst, ResourceId, 0, ColorMap, 3);
19   dcMemory = CreateCompatibleDC(hDC);
20   oldBitmap = SelectObject(dcMemory,bmp);
21   HBRUSH hbBkBrush = CreateSolidBrush(PixelBackColor);
22   GetClientRect(hwnd,&m_rect);
23   FillRect(hDC, &m_rect, hbBkBrush);
24   x = y = 0;
25   for (int ix = 0; ix < len; ix++){
26        GetCharBmpOffset(&rc, (char)Text[ix],CharacterWidth,
27             CharacterXSpacing,CharacterHeight,CharacterYSpacing);
28        BitBlt(hDC,x, y, (CharacterWidth + CharacterXSpacing),
29             (CharacterHeight+CharacterYSpacing), dcMemory, rc.left,
30             rc.top, SRCCOPY);
31        x += CharacterWidth + CharacterXSpacing;
32        charcount++;
33        if ((charcount == MaxXCharacters) && MaxYCharacters == 1)
34             break;
35        else if ((charcount == MaxXCharacters) && MaxYCharacters > 1)
36        {
37             if (linecount == MaxYCharacters)
38                  break;
39             x = charcount = 0;
40             y += CharacterHeight + CharacterYSpacing;
```

121.The exact values are: RGB(63,181,255), RGB(23,64,103), and RGB(0,0,0).

```
41              linecount++;
42          }
43      }
44      EndPaint(hwnd,&ps);
45      SelectObject(dcMemory,oldBitmap);
46      DeleteDC(dcMemory);
47      DeleteObject(bmp);
48      DeleteObject(hbBkBrush);
}
```

The first thing our procedure does is to obtain a device context from Windows (2). This device context is already clipped to the area we should draw. We suppose that the text of the lcd is stored in the variable “Text”, and we get its length (3).

To make the translation from the colors in the bitmap to the colors of our lcd window we use the primitive CreateMappedBitmap, that translates a bitmap with a set of source colors into a bitmap with the colors translated according to a translation table. This table is prepared in lines 12 to 17, then passed as argument to the API.

Now we have to create our memory device context. We make a compatible context in line 19, and then select into it the bitmap we have just obtained, after color translation. We will use this memory bitmap to copy bits into our display.

Before that, however, we make a solid brush and fill the whole window with the background color (lines 21-23).

For each character in our text, we do:

1 Calculate a rectangle with the position in our memory bitmap of the bits of the given character. Since we assume that all characters have the same width, it is a simple multiplication of a known constant by the width of the characters. We leave that calculation to the GetCharBmpOffset function (lines 26-27).

2 Using that rectangle, we copy the bits from our memory bitmap to the display using the BitBlt API (lines 28-30).

3 The rest of the lines in the loop updates the position of the destination, and take care of advancing to the next line.

4 Lines 44-48 concern the cleanup. We tell windows that we are finished with drawing using the EndPaint API (line 44). We deselect our bitmap from the memory device context (line 45), then we delete the memory DC. Note that we must do things in that order, since a bitmap can’t be deleted if it is selected in a DC. Then, we delete the background brush we created.

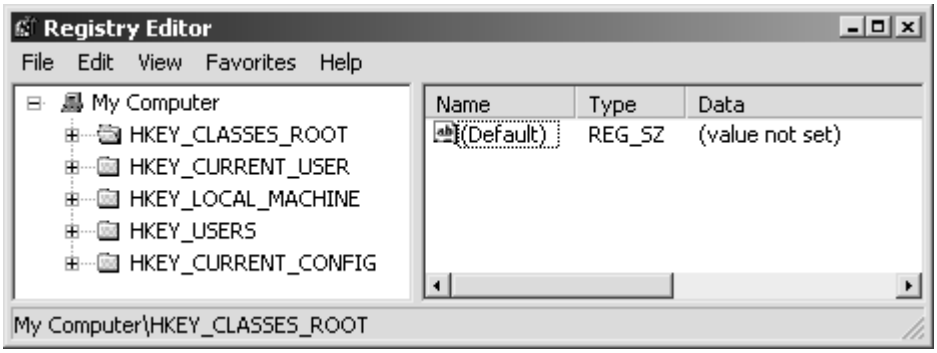
2.16 The Registry

The registry stores data in a hierarchically structured tree. Each node in the tree is called a key. Each key can contain both sub keys and values. Sometimes, the presence of a key is all the data that an application requires; other times, an application opens a key and uses the values associated with the key. A key can have any number of values, and the values can be in any form. Registry values can be any of the following types:

- Binary data
- 32 bit numbers
- Null terminated strings
- Arrays of null terminated strings. The array ends with two null bytes.
- Expandable null terminated strings. These strings contain variables like `%PATH%` or `%windir%` that are expanded when accessed.

2.16.1 The structure of the registry

To view and edit the registry you can use the system tool "regedit". When it starts, it shows the basic structure of the root trees:



1 **`HKEY_CLASSES_ROOT.`** Registry entries under this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key. File viewers and user interface extensions store their OLE class identifiers in `HKEY_CLASSES_ROOT`, and in-process servers are registered in this key. To change the settings for the interactive user, store the changes under `HKEY_CURRENT_USER\Software\Classes` rather than `HKEY_CLASSES_ROOT.`

2 **`HKEY_CURRENT_USER.`** Registry entries under this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences. This key makes it easier to establish the current user's settings; the key maps to the current user's branch in `HKEY_USERS.` In `HKEY_CURRENT_USER`, software vendors store the current user-specific preferences to be used within their applications. Lcc-win32, for example, creates the `HKEY_CURRENT_USER\Software\lcc` key for its applications to use, with each application creating its own subkey under the lcc key.

3 **`HKEY_LOCAL_MACHINE`**. Registry entries under this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software. It contains subkeys that hold current configuration data, including Plug and Play information (the Enum branch, which includes a complete list of all hardware that has ever been on the system), network logon preferences, network security information,

software-related information (such as server names and the location of the server), and other system information. Note that you must be the administrator user to be able to modify this tree.

4 **HKEY_USERS**. Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

5 **HKEY_CURRENT_CONFIG**. Contains information about the current hardware profile of the local computer system. The information under `HKEY_CURRENT_CONFIG` describes only the differences between the current hardware configuration and the standard configuration. Information about the standard hardware configuration is stored under the Software and System keys of `HKEY_LOCAL_MACHINE.` Actually, this key is an alias for `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Hardware Profiles\Current`.

This keys provide you with an entry point into the registry. To open a key, you must supply a handle to another key in the registry that is already open. The system defines predefined keys that are always open. Those keys help you navigate in the registry and make it possible to develop tools that allow a system administrator to manipulate categories of data. Applications that add data to the registry should always work within the framework of predefined keys, so administrative tools can find and use the new data.

2.16.2 Enumerating registry subkeys

The following example demonstrates the use of the RegQueryInfoKey, RegEnumKeyEx, and RegEnumValue functions. The hKey parameter passed to each function is a handle to an open key. This key must be opened before the function call and closed afterward.

```
// QueryKey - Enumerates the subkeys of key, and the associated
//    values, then copies the information about the keys and values
//    into a pair of edit controls and list boxes.
// hDlg - Dialog box that contains the edit controls and list boxes.
// hKey - Key whose subkeys and values are to be enumerated.
void QueryKey(HWND hDlg, HANDLE hKey)
{
    CHAR     achKey[MAX_PATH];
    CHAR     achClass[MAX_PATH] = "";  // buffer for class name
    DWORD    cchClassName = MAX_PATH;  // size of class string
    DWORD    cSubKeys;                 // number of subkeys
    DWORD    cbMaxSubKey;              // longest subkey size
    DWORD    cchMaxClass;              // longest class string
    DWORD    cValues;              // number of values for key
    DWORD    cchMaxValue;          // longest value name
    DWORD    cbMaxValueData;       // longest value data
    DWORD    cbSecurityDescriptor; // size of security descriptor
    FILETIME ftLastWriteTime;      // last write time

    DWORD i, j;
    DWORD retCode, retValue;

    CHAR  achValue[MAX_VALUE_NAME];
    DWORD cchValue = MAX_VALUE_NAME;
    CHAR  achBuff[80];

    // Get the class name and the value count.
    RegQueryInfoKey(hKey,        // key handle
        achClass,                // buffer for class name
        &cchClassName,           // size of class string
        NULL,                    // reserved
```

```
        &cSubKeys,               // number of subkeys
        &cbMaxSubKey,            // longest subkey size
        &cchMaxClass,            // longest class string
        &cValues,                // number of values for this key
        &cchMaxValue,            // longest value name
        &cbMaxValueData,         // longest value data
        &cbSecurityDescriptor,   // security descriptor
        &ftLastWriteTime);       // last write time

    SetDlgItemText(hDlg, IDE_CLASS, achClass);
    SetDlgItemInt(hDlg, IDE_CVALUES, cValues, FALSE);

    SendMessage(GetDlgItem(hDlg, IDL_LISTBOX),
        LB_ADDSTRING, 0, (LONG) "..");

    // Enumerate the child keys, until RegEnumKeyEx fails. Then
    // get the name of each child key and copy it into the list box.

    SetCursor(LoadCursor(NULL, IDC_WAIT));
    for (i = 0, retCode = ERROR_SUCCESS;
            retCode == ERROR_SUCCESS; i++)
    {
        retCode = RegEnumKeyEx(hKey,
                    i,
                    achKey,
                    MAX_PATH,
                    NULL,
                    NULL,
                    NULL,
                    &ftLastWriteTime);
        if (retCode == (DWORD)ERROR_SUCCESS)
        {
            SendMessage(GetDlgItem(hDlg, IDL_LISTBOX),
                LB_ADDSTRING, 0, (LONG) achKey);
        }
    }
    SetCursor(LoadCursor (NULL, IDC_ARROW));

     // Enumerate the key values.
    SetCursor(LoadCursor(NULL, IDC_WAIT));

    if (cValues)
    {
        for (j = 0, retValue = ERROR_SUCCESS;
                j < cValues; j++)
        {
            cchValue = MAX_VALUE_NAME;
            achValue[0] = '\0';
            retValue = RegEnumValue(hKey, j, achValue,
                &cchValue,
                NULL,
                NULL,    // &dwType,
                NULL,    // &bData,
                NULL);   // &bcData

            if (retValue == (DWORD) ERROR_SUCCESS )
            {
                achBuff[0] = '\0';

                // Add each value to a list box.
```

```
                    if (!lstrlen(achValue))
                        lstrcpy(achValue, "<NO NAME>");
                    wsprintf(achBuff, "%d) %s ", j, achValue);
                    SendMessage(GetDlgItem(hDlg,IDL_LISTBOX2),
                        LB_ADDSTRING, 0, (LONG) achBuff);
                }
            }

        SetCursor(LoadCursor(NULL, IDC_ARROW));
    }
```

2.16.3 Rules for using the registry

Although there are few technical limits to the type and size of data an application can store in the registry, certain practical guidelines exist to promote system efficiency. An application should store configuration and initialization data in the registry, and store other kinds of data elsewhere.

Generally, data consisting of more than one or two kilobytes (K) should be stored as a file and referred to by using a key in the registry rather than being stored as a value. Instead of duplicating large pieces of data in the registry, an application should save the data as a file and refer to the file. Executable binary code should never be stored in the registry. For instance, lccwin32 stores just the name of the path to the lcc installation directory in the registry, and all other data is stored in files in the \lcc\lib directory. This saves registry space.

A value entry uses much less registry space than a key. To save space, an application should group similar data together as a structure and store the structure as a value rather than storing each of the structure members as a separate key. (Storing the data in binary form allows an application to store data in one value that would otherwise be made up of several incompatible types.)

2.16.4 Interesting keys

The registry is an endless source of “useful tips”. There is for instance, a nice site that provides

Functionality	*Key*
Change the Location of System and Special Folders	HKEY_CURRENT_USER\Software\ Microsoft \Windows\CurrentVersion \Explorer \User Shell Folders
Configure CoolSwitch Application Swapping CoolSwitch is a feature available within Windows to quickly switch between tasks and programs by using the Alt + TAB keys.	HKEY_CURRENT_USER\Control Panel\Desktop Value Name: CoolSwitch, CoolSwitchColumns, CoolSwitchRows Data Type: REG_SZ (String Value) Value Data: (0 = disabled, 1 = enabled) Modify the value named "CoolSwitch" and set it to "0" to disable task switching or "1" to enable it. To change the size of the CoolSwitch popup modify the values named "CoolSwitchColumns" and "CoolSwitchRows". Logoff or restart Windows for the changes to take effect.
Change the Registered Owner and Organization	Windows 95/98 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion Windows NT/2000/XP HKEY_LOCAL_MACHINE\SOFTWARE\ Microsoft\Windows NT\CurrentVersion
Change the version of the lcc-win32 compiler to a new location (or to an older/newer version)	HKEY_CURRENT_USER\software\lcc
Disabling tooltips	Although ToolTips may be helpful, you might find it annoying. To disable it, go to HKEY_USERS\.DEFAULT\Control Panel\Desktop. On the right pane, double-click UserPreferencemask and change the value to 3E 00 00 00. To enable it again, change the value back to BE 00 00 00.

a “Windows registry guide”: http://www.winguides.com/registry, where I took some of the examples above. They distribute (for free) a very interesting guide in windows help format full of tips like that. Another similar site is http://www.activewin.com/tips/reg/index.shtml.

2.17 Etc.

At last count, there was around 17000 APIs in the file \lcc\lib\apilist.txt. This file is used by the linker to figure out in which library a given API is found. You can look at it to get an impression of the richness of the windows API. There are functions for many kinds of stuff, and here I will just try to give a very superficial overview of all that. You are invited to download the SDK documentation (for free) from the Microsoft MSDN site.

Lcc-win32 gives you access to all this:

Clipboard	Just that. A common repository for shared data. Quite a few formats are available, for images, sound, text, etc.
Communications	Read and write from COM ports.
Consoles and text mode support	The “msdos” window improved.
Debug Help	Why not? Write a debugger. Any one can do it. It is quite interesting as a program.
Device I/O	Manage all kind of cards with DeviceIOControl.
Dynamically linked libraries (DLLs)	Yes, I know. It is hot in DLL Hell. But at least you get separate modules, using binary interfaces that can be replaced one by one. This can lead to confusion, but it is inherent in the method.
Files	The disk is spinning anyway. Use it!
File Systems	Journaling file systems, NTFS, FAT32. As you like it.
Graphics	Windows are graphical objects. The GDI machinery allows you to draw simple objects with lines or regions, but you can go to higher dimensions with DirectX or OpenGl.
Handles and Objects	Objects that the system manages (windows, files, threads, and many others) are described by a numerical identifier. A handle to the object.
Hooks	Install yourself in the middle of the message queue, and hear what is being passed around: you receive the messages before any other window receives them.
Inter-Process Communications	Client/Server, and many other forms of organizing applications are available. You have all the primitives to do any kind of architecture. Synchronization, pipes, mailslots, you name it.
Mail	Send/receive mail messages using the Messaging API.
Multimedia	Sound, video, input devices.

Network	Yes, TCP/IP. Send data through the wire; develop your own protocol on top of the basic stuff. You have all the tools in here.
Virtual memory	Use those megabytes. They are there anyway. Build huge tables of data. Use virtual memory, reserve contiguous address space, etc.
Registry.	A global database for configuration options.[1]
Services	Run processes in the background, without being bothered by a console, window, or any other visible interface.
Shell programming	Manage files, folders, shortcuts, and the desktop.
Terminal services	Terminal Services provides functionality similar to a terminal-based, centralized host, or mainframe, environment in which multiple terminals connect to a host computer. A user can log on at a terminal, and then run applications on the host computer, accessing files, databases, network resources, and so on. Each terminal session is independent, with the host operating system managing conflicts between multiple users contending for shared resources
Windows	Yes, Windows is about windows. You get a huge variety of graphical objects, from the simple check box to sophisticated, tree-displaying stuff. An enormous variety of things that wait for you, ready to be used.

1. **M**edia **C**ontrol **I**nterface

2.17.1 Clipboard

The data in the clipboard is tagged with a specific format code. To initiate the data transfer to or from the clipboard you use `OpenClipboard`, `GetClipboardData` allows you to read it, `SetClipboardData` to write it, etc. You implement this way the famous Cut, Copy and Paste commands that are ubiquitous in most windows applications. Predefined data formats exist for images (`CF_BITMAP`,`CF_METAFILEPICT`, `CF_TIFF`), sound (`CF_WAVE`, `CF_RIFF`), text (`CF_TEXT`), pen data (`CF_PENDATA`), a list of files (`CF_HDROP`) and several others.

When you pass a block of data to the clipboard, that memory should be allocated, since windows expects that it can free it when it is no longer used. Since it is the clipboard that should free the allocated memory, you can't use the standard allocation functions, you should use the GlobalAlloc API to get memory to be used by the clipboard.

To transfer a character string to the clipboard then, we could use the following function:

```
int CopyToClipboard(char *str,HWND hwnd)
{
     int len = strlen(str)+1;
     HANDLE h;

     h = GlobalAlloc(GHND|GMEM_DDESHARE,len);
     if (h == (HANDLE)0)
```

```
            return 0;
        txt = GlobalLock(h);
        strcpy(txt,str);
        if (OpenClipboard(hwnd)) {
            EmptyClipboard();
            SetClipboardData(CF_TEXT,h);
            CloseClipboard();
            return TRUE;
        }
        return FALSE;
    }
```

We allocate space for our character string, then obtain a pointer from the given handle. We can then copy the data into this memory block.

We obtain access to the clipboard by opening it giving it the handle of a window that will own the clipboard. If the operation succeeds, we clear the previous contents, set our data and we are done.

2.17.2 Serial communications.

You use the same primitives that you use for files to open a communications port. Here is the code to open COM1 for instance:

```
HANDLE hComm;
char *gszPort = “COM1”;
hComm = CreateFile( gszPort,
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    0,
                    OPEN_EXISTING,
                    FILE_FLAG_OVERLAPPED,
                    0);
```

You use that handle to call `ReadFile` and `WriteFile` APIs. Communications events are handled by SetCommMask, that defines the events that you want to be informed about (break, clear-to-send, ring, rxchar, and others). You can change the baud rate managing the device control block (`SetCommState`), etc. As with network interfaces, serial line programming is a black art.

2.17.3 Files

Besides the classical functions we have discussed in the examples, Windows offers you more detailed file control for accessing file properties, using asynchronous file input or output, for managing directories, controlling file access, locking, etc. In a nutshell, you open a file with CreateFile, read from it with ReadFile, write to it with WriteFile, close the connection to it with CloseHandle, and access its properties with GetFileAttributes. Compared with the simple functions of the standard library those functions are more difficult to use, since they require more parameters, but they allow you a much finer control. Here is a very small list of some of the file functions provided. For more information read the associated documentation for the functions given here.

One thing you will wonder is how do you get a HANDLE starting with a FILE pointer. We

Function name	*Description*
CreateFile	Opens or creates a file. It can open a COM port (serial communications) or a pipe.
CloseHandle	Use it to close a file.
ReadFile or ReadFileEx	Reads data from a file, starting at the current file pointer. Both synchronous and asynchronous versions exist.
WriteFile or WriteFileEx	Writes data to a file, starting at the current file pointer. Both synchronous and asynchronous versions exist.
CopyFile or CopyFileEx	Copy one file to another.
MoveFile or MoveFileEx	Moves a file.
DeleteFile	Deletes a file.
GetFileAttributes or GetFileAttributesEx	Reads the file attributes like if its read only, hidden, system, etc.
SetFileAttributes or SetFileATtributesEx	Sets the file attributes
GetFileSize	Returns the size of a file.
FindFirstFile and FindNextFile	Searches for a file matching a certain wildcard.

discussed above files and used always a FILE structure as specified in <stdio.h>. To obtain a HANDLE that can be used with the windows file functions you write:

```
#include <windows.h>
#include <stdio.h>
FILE *fptr;
HANDLE h = (HANDLE)_get_osfhandle(_fileno(fptr));
```

2.17.4 File systems

These days files are taken for granted. File systems not. Modern windows file systems allow you to track file operations and access their journal data. You can encrypt data, and at last under windows 2000 Unix's mount operation is recognized. You can establish symbolic links for files, i.e, consider a file as a pointer to another one. This pointer is dereferenced by the file system when accessing the link.

Windows supports several file systems:

1) NTFS.NTFS is the preferred file system on Windows. It was designed to address the requirements of high-performance file servers and server networks as well as desktop computers, and in doing so, address many of the limitations of the earlier FAT16 and FAT32 file systems.

2) FAT32 The File Allocation Table (FAT) file system organizes data on fixed disks and floppy disks. The main advantage of FAT volumes is that they are accessible by MS-DOS, Windows, and OS/2 systems. FAT is the only file system currently supported for floppy disks and other removable media. FAT32 is the most recently defined FAT-based file system format, and it's included with Windows 95 OSR2, Windows 98, and Windows Millennium Edition. FAT32 uses 32-bit cluster identifiers but reserves the high 4 bits, so in effect it has 28-bit cluster identifiers.

3) UDF file system. The implementation is compliant with ISO 13346 and supports UDF versions 1.02 and 1.5. OSTA (Optical Storage Technology Association) defined UDF in 1995 as a format to replace CDFS for magneto-optical storage media, mainly DVD-ROM. UDF is included in the DVD specification and is more flexible than CDFS.

2.17.5 Graphics

GDI is the lowest level, the basic machinery for drawing. It provides you:

- Bitmap support
- Brush support for painting polygons.
- Clipping that allows you to draw within the context of your window without worrying that you could overwrite something in your neighbor’s window. Filled shapes, polygons ellipses, pie rectangle, lines and curves.
- Color management, palettes etc.
- Coordinate spaces, and transforms.
- Text primitives for text layout, fonts, captions and others.
- Printing

But higher levels in such a vast field like graphics are surely possible. Lcc-win32 offers the standard jpeg library of Intel Corp to read/write and display jpeg files. Under windows you can do OpenGl, an imaging system by Silicon Graphics, or use DirectX, developed by Microsoft.

2.17.6 Handles and Objects

An object is implemented by the system with a standard header and object-specific attributes. Since all objects have the same structure, there is a single object manager that maintains all objects. Object attributes include the name (so that objects can be referenced by name), security descriptors to rule access to the information stored in those objects, and others, for instance properties that allow the system to enforce quotas. The system object manager allows mapping of handles from one process to another (the `DuplicateHandle` function) and is responsible for cleanup when the object handle is closed.

2.17.7 Inter-Process Communications

You can use the following primitives:

- Atoms. An atom table is a system-defined table that stores strings and corresponding identifiers. An application places a string in an atom table and receives a 16-bit integer, called an atom that can be used to access the string. The system maintains a global atom

table that can be used to send information to/from one process to another: instead of sending a string, the processes send the atom id.

- Clipboard. This is the natural way to do inter-process communications under windows: Copy and Paste.
- Mailslots. A mailslot is a pseudofile; it resides in memory, and you use standard Win32 file functions to access it. Unlike disk files, however, mailslots are temporary. When all handles to a mailslot are closed, the mailslot and all the data it contains are deleted. A mailslot server is a process that creates and owns a mailslot. A mailslot client is a process that writes a message to a mailslot. Any process that has the name of a mailslot can put a message there. Mailslots can broadcast messages within a domain. If several processes in a domain each create a mailslot using the same name, the participating processes receive every message that is addressed to that mailslot and sent to the domain. Because one process can control both a server mailslot handle and the client handle retrieved when the mailslot is opened for a write operation, applications can easily implement a simple message-passing facility within a domain.
- Pipes. Conceptually, a pipe has two ends. A one-way pipe allows the process at one end to write to the pipe, and allows the process at the other end to read from the pipe. A two-way (or duplex) pipe allows a process to read and write from its end of the pipe.
- Memory mapped files can be used as a global shared memory buffer.

2.17.8 Mail

The Messaging API (MAPI) allows you to program your messaging application or to include this functionality into your application in a vendor-independent way so that you can change the underlying message system without changing your program.

2.17.9 Multimedia

Audio. You can use Mixers, MIDI, and waveform audio using MCI.DirectSound offers a more advanced sound interface.

Input devices. You can use the joystick, precise timers, and multimedia file input/output.

Video. Use AVI files to store video sequences, or to capture video information using a simple, message-based interface.

2.17.10 Network

Windows Sockets provides you will all necessary functions to establish connections over a TCP/IP network. The TCPIP subsystem even supports other protocols than TCPIP itself. But whole books have been written about this, so here I will only point you to the one I used when writing network programs: Ralph Davis “Windows NT Network programming”, from Addison Wesley.

For an example of network functions see “Retrieving a file from the internet” page 279.

2.17.11 Hooks

A hook is a mechanism by which a function can intercept events (messages, mouse actions, keystrokes) before they reach an application. The function can act on events and, in some cases, modify or discard them. This filter functions receive events, for example, a filter function might want to receive all keyboard or mouse events. For Windows to call a filter function,

the filter function must be installed—that is, attached—to an entry point into the operating system, a hook (for example, to a keyboard hook). If a hook has more than one filter function attached, Windows maintains a chain of those, so several applications can maintain several hooks simultaneously, each passing (or not) its result to the others in the chain.

2.17.12 Shell Programming

Windows provides users with access to a wide variety of objects necessary for running applications and managing the operating system. The most numerous and familiar of these objects are the folders and files, but there are also a number of virtual objects that allow the user to do tasks such as sending files to remote printers or accessing the Recycle Bin. The shell organizes these objects into a hierarchical name space, and provides users and applications with a consistent and efficient way to access and manage objects.

2.17.13 Services

A service application conforms to the interface rules of the Service Control Manager (SCM). A user through the Services control panel applet can start it automatically at system boot, or by an application that uses the service functions. Services can execute even when no user is logged on to the system

2.17.14 Terminal Services

When a user logs on to a Terminal Services-enabled computer, a session is started for the user. Each session is identified by a unique session ID. Because each logon to a Terminal Services client receives a separate session ID, the user-experience is similar to being logged on to multiple computers at the same time; for example, an office computer and a home computer. The console session on the Terminal Server is assigned the session ID 0.

The Remote Desktop Protocol (RDP) provides remote display and input capabilities over network connections for Windows-based applications running on a server. RDP is designed to support different types of network topologies and multiple LAN protocols.

On the server, RDP uses its own video driver to render display output by constructing the rendering information into network packets using RDP protocol and sending them over the network to the client. On the client, RDP receives rendering data and interprets the packets into corresponding graphics device interface API calls. For the input path, client mouse and keyboard events are redirected from the client to the server. On the server, RDP uses its own virtual keyboard and mouse driver to receive these keyboard and mouse events.

To optimize performance, it is good practice for applications to detect whether they are running in a Terminal Services client session. For example, when an application is running on a remote session, it should eliminate unnecessary graphic effects like:

1) Splash screens. Transmitting a splash screen to a Terminal Services client consumes extra network bandwidth and forces the user to wait before accessing the application.

2) Animations which consume CPU and network bandwidth

3) Direct input or output to the video display.

You can detect if you are running in a remote session by using the following code:

```
BOOL IsRemoteSession(void)
{
   return GetSystemMetrics( SM_REMOTESESSION );
}
```

2.17.15 Windows

Here is a short overview of the types of controls available to you.

Control	*Description*
Edit	Single or multi line text editor.
Checkbox	For a set of multiple choices
Listbox	For displaying lists
Combobox	A list + an edit control
Static	Group boxes, static text, filled rectangles. Used for labels, grouping and separation.
Push buttons	Used to start an action
Radio buttons	Used for choosing one among several possible choices.
Scroll bars	Used for scrolling a view.
Animation controls	Display AVI files
Date and Time	Used to input dates
Headers	Allow the user to resize a column in a table
List view	Used to display images and text.
Pager	Used to make a scrollable region that contains other controls. You scroll the controls into view with the pager.
Progress bar	Used in lengthy operations to give a graphic idea of how much time is still needed to finish the operation.
Property Sheets	Used to pack several dialog boxes into the same place, avoiding user confusion by displaying fewer items at the same time.
Richedit	Allows editing text with different typefaces (bold, italic) with different fonts, in different colors… The basic building block to build a text processor.
Status bars	Used to display status information at the bottom of a window
Tab controls	The building block to make property sheets.
Toolbar controls	A series of buttons to accelerate application tasks.
Tooltips	Show explanations about an item currently under the mouse in a pop-up window.
Trackbars	Analogical volume controls, for instance.

Tree view	Displays hierarchical trees.

2.18 Advanced windows techniques

Windows is not only drawing of course. It has come a long way since windows 3.0, and is now a sophisticated operating system. You can do things like memory-mapped files for instance, that formerly were possible only under UNIX. Yes, “mmap” exists now under windows, and it is very useful.

2.18.1 Memory mapped files

Memory mapped files allow you to see a disk file as a section of RAM. The difference between the disk and the RAM disappears. You can just seek to a desired position by incrementing a pointer, as you would do if you had read the whole file into RAM, but more efficiently. It is the operating system that takes care of accessing the disk when needed. When you close the connection, the operating system handles the clean up.

Here is a small utility that copies a source disk file to a destination using memory-mapped files.

```
int main (int argc, char **argv)
{
   int   fResult = FAILURE;
   ULARGE_INTEGER liSrcFileSize, // See[122]
                  liBytesRemaining,
                  liMapSize,
                  liOffset;
   HANDLE hSrcFile    = INVALID_HANDLE_VALUE,
          hDstFile    = INVALID_HANDLE_VALUE,
          hSrcMap     = 0,
          hDstMap     = 0;

   BYTE * pSrc = 0,
        * pDst = 0;

   char * pszSrcFileName = 0,
        * pszDstFileName = 0;

   if (argc != 3) // test if two arguments are given in the command line
   {
      printf("usage: copyfile <srcfile> <dstfile>\n");
      return (FAILURE);
   }

   pszSrcFileName = argv[argc-2];  // Src is second to last argument
```

122. ULARGE_INTEGER is defined in the windows headers like this:

```
typedef union _ULARGE_INTEGER {
        struct {DWORD LowPart; DWORD HighPart;};
        long long QuadPart;
} ULARGE_INTEGER,*PULARGE_INTEGER;
```

The union has two members: a anonymous one with two 32 bit integers, and another with a long long integer, i.e. 64 bits. We can access the 64-bit integer’s low and high part as 32 bit numbers. This is useful for functions returning two results in a 64 bit number.

```
   pszDstFileName = argv[argc-1];  // Dst is the last argument
```
/* We open the source file for reading only, and make it available for other processes even if we are copying it. We demand to the operating system to ensure that the file exists already */
```
   hSrcFile = CreateFile (pszSrcFileName,
GENERIC_READ, //Source file is opened for reading only
FILE_SHARE_READ,// Shareable
         0, OPEN_EXISTING, 0, 0);
   if (INVALID_HANDLE_VALUE == hSrcFile)
   {
      printf("couldn't open %s\n", pszSrcFileName);
      goto DONE;
   }
```
/* We open the destination file for reading and writing with no other access allowed. We demand the operating system to create the file if it doesn't exist. */
```
   hDstFile = CreateFile (pszDstFileName,
GENERIC_READ|GENERIC_WRITE, 0,
         0, CREATE_ALWAYS, 0, 0);
   if (INVALID_HANDLE_VALUE == hDstFile)
   {
      printf("couldn't create %s\n", pszSrcFileName);
      goto DONE;
   }
```
/*
We need to query the OS for the size of this file. We will need this information later when we create the file-mapping object. Note that we receive a 64 bit number splitted in two. We receive a 32-bit integer containing the result's lower 32 bits, and we pass to the function the address where it should put the remaining bits! Well, if you find this interface strange (why not return a 64 bit integer?) please do not complain to me. Note too the strange form of the error checking afterwards: we check for a return value of all bits set to one, and check the result of the GetLastError() API.
```
   */
   SetLastError(0);
   liSrcFileSize.LowPart = GetFileSize(hSrcFile,
&liSrcFileSize.HighPart)
   if (0xFFFFFFFF == liSrcFileSize.LowPart &&
GetLastError() != NO_ERROR){
      printf("couldn't get size of source file\n");
      goto DONE;
   }
```
/*
Special case: If the source file is zero bytes, we don't map it because there's no need to and anyway CreateFileMapping cannot map a zero-length file. But since we've created the destination, we've successfully "copied" the source.
```
   */
   if (0 == liSrcFileSize.QuadPart)
   {
      fResult = SUCCESS;
      goto DONE;
   }
```
/*
Map the source file into memory. We receive from the OS a HANDLE that corresponds to the opened mapping object.
```
   */
      hSrcMap = CreateFileMapping (hSrcFile,
0, PAGE_READONLY, 0, 0, 0);
   if (!hSrcMap){
      printf("couldn't map source file\n");
      goto DONE;
```

```
   }
  /*
```
Now we create a file mapping for the destination file using the size parameters we got above.
```
  */
   hDstMap = CreateFileMapping (hDstFile, 0,
              PAGE_READWRITE,
               liSrcFileSize.HighPart,
               liSrcFileSize.LowPart, 0);
   if (!hDstMap)
   {
      DEBUG_PRINT("couldn't map destination file\n");
      goto DONE;
   }
  /*
```
Now that we have the source and destination mapping objects, we build two map views of the source and destination files, and do the file copy.
To minimize the amount of memory consumed for large files and make it possible to copy files that couldn't be mapped into our virtual address space entirely (those over 2GB), we limit the source and destination views to the smaller of the file size or a specified maximum view size (MAX_VIEW_SIZE--which is 96K).
If the size of file is smaller than the max view size, we'll just map and copy it. Otherwise, we'll map a portion of the file, copy it, then map the next portion, copy it, etc. until the entire file is copied.
MAP_SIZE is 32 bits because MapViewOfFile requires a 32-bit value for the size of the view. This makes sense because a Win32 process's address space is 4GB, of which only 2GB (2^31) bytes may be used by the process. However, for the sake of making 64-bit arithmetic work below for file offsets, we need to make sure that all 64 bits of limpest are initialized correctly.
```
  */
    liBytesRemaining.QuadPart = liSrcFileSize.QuadPart;
    /* This assignment sets all 64 bits to this value */
    liMapSize.QuadPart = MAX_VIEW_SIZE;

    do {
   /* Now we start our copying loop. The “min” macro returns the smaller of two numbers. */
liMapSize.QuadPart = min(liBytesRemaining.QuadPart,
liMapSize.QuadPart)

     liOffset.QuadPart = liSrcFileSize.QuadPart –
liBytesRemaining.QuadPart;

     pSrc = (BYTE *)MapViewOfFile(hSrcMap, FILE_MAP_READ,
liOffset.HighPart,
          liOffset.LowPart, liMapSize.LowPart);
     pDst = (BYTE *)MapViewOfFile(hDstMap,FILE_MAP_WRITE,
liOffset.HighPart,
          liOffset.LowPart, liMapSize.LowPart);
     /* We use memcpy to do the actual copying */
memcpy(pDst, pSrc, liMapSize.LowPart);

     UnmapViewOfFile (pSrc);
     UnmapViewOfFile (pDst);

     liBytesRemaining.QuadPart -= liMapSize.QuadPart;
   }
   while (liBytesRemaining.QuadPart > 0);

   fResult = SUCCESS;
DONE:
   /* We are done, Note the error treatment of this function. We use gotos to reach the end of the function, and here we cleanup everything.   */
```

```
   if (hDstMap) CloseHandle (hDstMap);

   if(hDstFile!=INVALID_HANDLE_VALUE) CloseHandle(hDstFile);

   if (hSrcMap) CloseHandle(hSrcMap);

   if (hSrcFile != INVALID_HANDLE_VALUE)
      CloseHandle (hSrcFile);

   if (fResult != SUCCESS)
   {
      printf("copying %s to %s failed.\n",
pszSrcFileName, pszDstFileName);
      DeleteFile (pszDstFileName);
   }
   return (fResult);
}
```

Summary: To get a pointer to a memory mapped file do the following:

- Open the file with CreateFile
- Create a mapping object with CreateFileMapping using the handle you receive from CreateFile.
- Map a portion (or all) of the file contents, i.e. create a view of it, with MapViewOfFile.

2.18.2 Letting the user browse for a folder: using the shell

A common task in many programming situations is to let the user find a folder (directory) in the file system hierarchy. When you want to search for certain item, for instance, or when you need to allow the user to change the current directory of your application. The windows shell offers a ready-made set of functions for you, and the resulting function is quite short. Let's first see its specification, i.e. what do we want as an interface to this function.

Required is a character string where the result will be written to, and a title for the user interface element. The result should be 1 if the path contains a valid directory, 0 if there was any error, the user cancelled, whatever.

To be clear about the specifications let's look at this example:

```
int main(void)
{
        char path[MAX_PATH];
        if (BrowseDir("Choose a directory",path)) {
                printf("You have choosen %s\n",path);
        }
        else printf("action cancelled\n");
        return 0;
}
```

How do we write "BrowseDir" in windows?

Here it is:

```
#include <shlobj.h>
#include <stdio.h>

int BrowseDir(unsigned char *Title,char *result)
{
    LPMALLOC pMalloc;       (1)
    BROWSEINFO browseInfo;(2)
    LPITEMIDLIST ItemIDList;(3)
```

```
        int r = 0;              (4)

        if (S_OK != SHGetMalloc(&pMalloc))(5)
                    return 0;
        memset(&browseInfo,0,sizeof(BROWSEINFO));(6)
        browseInfo.hwndOwner = GetActiveWindow();(7)
        browseInfo.pszDisplayName = result;(8)
        browseInfo.lpszTitle = Title;(9)
        browseInfo.ulFlags = BIF_NEWDIALOGSTYLE;(10)
        ItemIDList = SHBrowseForFolder(&browseInfo);(11)
        if (ItemIDList != NULL) {
           *result = 0;
           if (SHGetPathFromIDList(ItemIDList,result))(12)
           {
              if (result[0]) r = 1;(13)
              pMalloc->lpVtbl->Free(pMalloc,ItemIDList);(14)
           }
        }
        pMalloc->lpVtbl->Release(pMalloc);(15)
        return r;
    }
```

Small isn't it?

Let's see the gory details.

We need a local variable that will hold a pointer to a shell defined function that will allocate and release memory. The shell returns us a result that needs memory to exist. We need to free that memory, and we have to take care of using the same function that the shell uses to allocate memory. This pointer to an interface (the malloc interface) will be in our local variable pMalloc.

The shell needs information about the environment, and some pointers to put the results of the interaction with the user. We will see more of this when we fill this structure below.

The shell uses a lot of stuff, and we have to take care to avoid filling our brain with unending details. What is an ITEMLIST? Actually I haven't even bothered to read the docs about it, since the only use I found is to pass it around to other shell functions.

The result of the function is initialized to zero, i.e. we will set this result to 1 only and only if there is a valid path in the buffer.

OK. Here we start. The first thing to do then, is to get the pointer to the shell allocator. If anything goes wrong there, there is absolutely nothing we can do and the best thing is to return immediately with a FALSE result.

We have to clean up the structure (note that this is a local variable, so its contents are as yet undefined). We use the primitive memset and set all members of the structure to zero. This is a common idiom in C: clear all memory before using and assign to it a known value. Since the default value of many structure members is zero, this easies the initialization of the structure since we do not have to explicitly set them to NULL or zero.

We start the filling of the relevant members. We need to put the owner window handle in the `hwndOwner` member. We get the handle of the current active window using a call to a windows API.

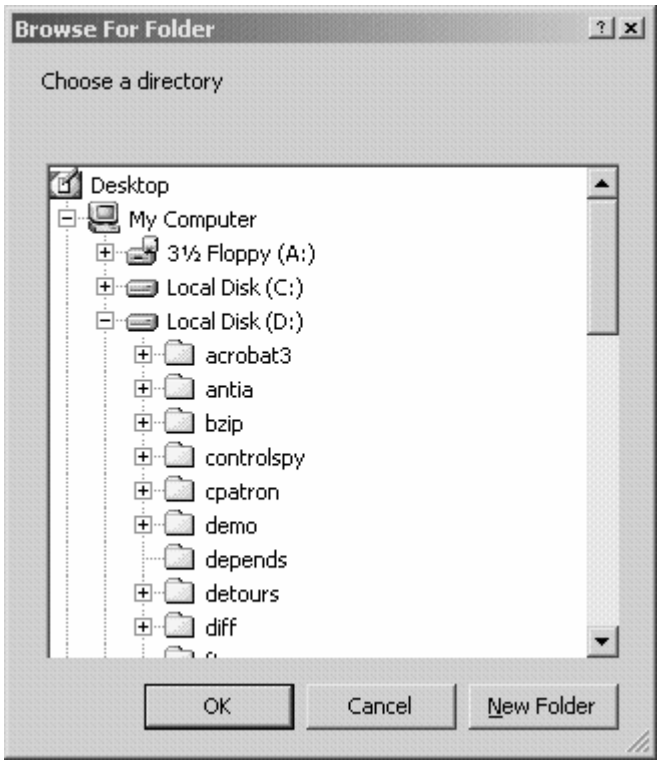
The shell needs place to store the display name of the chosen path. Note that this is not what we want (the full path) but just the last item in the path, i.e. the last directory. Why this is so? Because the user could choose a path like "My documents", and in this case we could detect that the user has chosen a "standard well known name" for the directory. But we do not use

this feature, and just give the shell some place in our… result variable. Since we overwrite this data later, this is harmless and avoids a new variable.[123]

We set the lpszTitle member to the value we passed in. This is a text that will be displayed at the top of the user interface window that appears, and should remain the user what kind of folder he/she is looking for.

As flags we just pass `BFID_USENEWUI`, meaning that we want the shell to use the improved user interface, with drag and drop, new folder, the possibility of deleting folders, whatever.

And we are done with the filling of the structure! We just have to call our famous `SHBrowseForFolder`, and assign the result to ItemIdList. Here is an image of the user interface display that appears in a windows 2000 machine; in other versions of windows it will look different. The user interface is quite sophisticated, and it is all at our disposal without writing any code (well almost!). What is better; even if we had spent some months developing a similar thing, we would have to maintain it, test it, etc. Note that you can transparently browse the network, give a symbolic path like "My computer" or other goodies.



If the call worked, i.e. if the user interface returns a valid pointer and not NULL, we should translate the meaningless stuff we receive into a real path, so we call `SHGetPathFromIDList`, to do exactly that. We pass it a pointer to the result character string that we receive as second argument.

If that function returns OK, we verify that the returned path is not empty, and if it is, we set the result of this function to TRUE.

Now we have to clean-up. We use the COM interface pointer we received from `SHGetMalloc`, and use its only member (lpVtbl) to get into the Free function pointer member. There are other function pointers in that structure for sure, but we do not use them in this application. We pass to that Free function a pointer to the interface it gave to us, and then the object to free.

123.Passing NULL works too.

When we are done with a COM interface we have to remember to call a function to release it, passing it again a pointer to the returned object. We are done now, we return the result and exit.

How can this program fail?

There are only three API calls here, and all of them are tested for failure. In principle there is no way this can fail, although it could fail for other reasons: it could provoke a memory leak (for instance if we had forgotten the call to the `Release` method at the end), or it could use resources that are never released (the list of identifiers that we obtain from the system, etc.

2.18.3 Retrieving a file from the internet

The basic way of receiving/sending data in the internet is TCP/IP, a protocol originally developed for Unix systems, and adapted later to windows by Microsoft under the name of "winsock" for window sockets.

The first versions of TCP/IP for windows were very difficult to use, and required a good deal of network programming knowledge: the code for establishing an FTP connection under windows 3.1 was several thousand lines, and required watching each transaction and following each possible error.

This has changed a lot since the "internet wave" required that each coffee machine in the world should have an internet connection. Higher level interfaces were developed and added to the standard windows API. Here is an example of a working program that will download the "README" file from the lcc-win32 home "q-software-solutions" and write it to the disk of your machine in the current directory.

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <direct.h>
#include <wininet.h>

static void ErrorOut(char *where)
{
     int err = GetLastError();
     printf("Error %d (0x%x) when calling %s\n",err,err,where);
}

int main(void)
{
     DWORD dwType = FTP_TRANSFER_TYPE_BINARY;
     HANDLE hConnect,hOpen;
     char *szUser,*szPass,*szHost,*szFile1,*szFile2;

     hOpen = InternetOpen ( "FTPGET",  LOCAL_INTERNET_ACCESS ,
               NULL, 0, 0);
     if (!hOpen) {
          ErrorOut("InternetOpen");
          return 1;
    }
     szHost = "www.q-software-solutions.com";
     szUser = "anonymous";
     szPass = "foo@foo.com";
     hConnect = InternetConnect(hOpen, szHost ,
          INTERNET_INVALID_PORT_NUMBER,szUser,
          szPass,INTERNET_SERVICE_FTP,INTERNET_FLAG_PASSIVE , 0);
     if (hConnect) {
          szFile1 = "/pub/lcc/README";
```

```
            szFile2 = "README";
            if (!FtpGetFile (hConnect,szFile1, szFile2,
                 FALSE,INTERNET_FLAG_RELOAD, dwType, 0)) {
                 ErrorOut ("FtpGetFile");
            }
            else printf("Remote file %s --> local file %s",
                 szFile1,szFile2);
            if (!InternetCloseHandle (hConnect)) {
                 ErrorOut ("InternetCloseHandle");
            }
       }
       else {
            ErrorOut("InternetConnect");
       }
       InternetCloseHandle(hOpen);
       return 0;
  }
```

This program is about 400 bytes of code and 200 bytes of data for character strings. When linked it is only 4K long.

The whole hard part of networking is hidden behind the windows interface, provided by the "wininet.lib" library, and the "wininet.h" header file.

Basically, to get a file from a remote machine using the FTP (File Transfer Protocol) you need to open a connection to the remote server, send it the path of the file to retrieve, and that is it. The rest is some logic for opening and initializing the wininet library, and taking care of possible errors.

Please be aware that all this supposes that you have a working TCP/IP network connection and that the lower layers of the network are working. Besides, note that the paths given here may change, etc etc. Network programming is a huge subject, impossible to cover here.

2.19 Error handling under windows

The windows API is an ancient software construct. Pieces of the original design started with windows 2.x under MSDOS with the 64K limitations of real mode are still buried within it. Pieces of the Windows 3.1 (16 bit protected mode) software are still very much in evidence, and the Win32 API is already 8 years old and counting.

Because those historical reasons, there exists a bewildering set of failure results. We have actually all possibilities:[124]

1 **The boolean convention**. Returning TRUE for success, FALSE for failure. This is a simple minded approach that is used in many APIs. You code then:

```
if (!DoSomething()) {
     // failure handling
}
```

The drawback of this convention is that with only two different values there is no room for explaining the failure reasons.

2 **The status convention.** Returning a status like `HRESULT`. A `HRESULT` >= 0, such as `S_OK`, is a success, and a `HRESULT` < 0, such as `E_FAIL`, is a failure.This would be OK but there is the problem that other API functions return an error code defined in windows.h. There is the BIG problem that `ERROR_SUCCESS`[125] is defined there as zero! In this schema the value zero is reserved to indicate that everything is OK, and all other values

124.This part has been adapted from an excellent article by Jon Pincus published in MSDN June 2000.

mean an error code. Note that the value of zero that in the boolean convention above indicates failure, means the contrary here.

3 **Returning a NULL pointer**. This failure indication is used in the standard C library. malloc, realloc, and many others return NULL for failure. As with the boolean convention, some other means are necessary to distinguish the different kinds of failures.

4 **Returning an "impossible" value**. Sometimes this value is zero, for instance in the API GetWindowsDirectory, sometimes is -1 for instance the fgets function in the C library, or many other values. This convention becomes very difficult to follow (and memorize) when applied to a large API like the windows API.

5 **Throwing a C++ exception**. This is very nice in C++, but it will never work with C or with COM. It is illegal to throw a C++ exception in COM. Besides, the runtimes of different compilers handle C++ exceptions in a different way, so they are incompatible with each other. Anyway, if you work with lcc-win32 you will be spared this.

6 **Using the RaiseException API.** This is the normal way of using structured exception handling in lcc-win32. It has its drawbacks too, since it doesn't fit very well with COM, for instance, and it is (maybe) not compatible with some other compilers. It is compatible with Microsoft compilers, and maybe Borland, since all of them use the same binary approach to this problem, imposed by the windows API.

7 **Using some other mechanism** of structured exception handling. Many other libraries exists for structured exception handling, sometimes using setjmp/longjmp, sometimes using the signal() mechanism, or some other conventions. If you find yourself working with some code that does this, either you use it without any modifications or you just drop it. Trying to fix/modify this kind of code is best left to very experienced programmers.

8 **Using GetLastError()** to get information about the error. This would seem evident but in their infinite wisdom, the windows API designers decided to provide different error codes for the same failures in the different versions of the operating system. Under windows 95/98 the error codes would be different than the error codes for the same failure under windows NT/2000.

When you mix all those error handling strategies in a single program, as you *have* to do to program with the API, the resulting mix can be very entertaining. Consider this:

```
extern BOOL DoSomething(void); // Uses the boolean strategy
BOOL ok = DoSomething();
if (FAILED(ok))     // WRONG!!!
   return;
```

The macro `FAILED` tests if the `HRESULT` is less than zero. Here is the definition:

```
#define FAILED(S) ((HRESULT)((S)<0))
```

In the above case the code for failure (0) will *not* be less than zero, and it will not be detected as a failure code. Rule:

Do not use the `FAILED` or `SUCCEEDED` macros for anything other than `HRESULTS`!

Since you know that `FALSE` is defined as zero, `FAILED(FALSE)` is zero, not really an expected result.

The opposite problem leads to headaches too. Always use the `FAILED/SUCCEEDED` macros with `HRESULTS` and never write things like this:

```
HRESULT hr;
```

125.Simply looking at this name: "ERROR_SUCCESS" implies that there is a big problem with the people that designed that stuff. It is an error code or a success indication?

```
hr = ISomething->DoSomething();
if (! hr)     // WRONG!!!
   return;
```

Note too that in their infinite wisdom, the designers of this API decided that S_FALSE is a success code, NOT a failure code, and it is defined as 1 (one...!!!). This means that

```
SUCCEEDED(S_FALSE) == TRUE
```

Nice isn't it?

2.19.1 Some tips for debugging

2.19.1.1 Check the return status of any API call.

Consider this code:

```
TCHAR cDriveLetter;
TCHAR szWindowsDir[MAX_PATH];
GetWindowsDirectory(szWindowsDir, MAX_PATH);
cDriveLetter = szWindowsDir[0];   // WRONG!!!
```

If the call to GetWindowsDirectory fails, the program will access uninitialized memory. This means that the drive value will be a random number between 0 and 255. Of course it is quite difficult to imagine that GetWindowsDirectory fails, or is it?

In windows using Terminal Server, it becomes much more complicated to figure out where is the windows directory of a particular user. The GetWindowsDirectory API has to do a lot more stuff, including allocating memory for buffers. And that allocation can fail and the developer at Microsoft did the right thing and tested if that memory allocation call worked or not, and if not it returns an error failure.

Besides, that API will work now, but who knows what the future versions of windows will do? Why leave a possible bug in there?

But this is difficult to do. Checking the return value of ALL API calls is feasible but is very complex and time-consuming. And as everyone knows it is not a matter of the developer only. How big is the time budget allocated to you for developing this application? How much effort should be spent in making this application 100% bullet proof?

Those questions should be answered but they are not only the responsibility of the developer.

2.19.1.2 Always check allocations

Even in this days of 512MB RAM being common configurations, and with swap files extending into the gigabyte memory failures *can* happen. Specially in the case of infinite loops where all bets are off and any memory allocation *will* fail. And here the memory manager of lcc-win32 will not help since it will return NULL also, it can't do anything else.

Allocations can fail because of corrupted input data to the memory allocation function. Suppose this code:

```
int allocsiz;
...
buffer = GC_malloc(allocsiz);
```

If for any reason allocsiz goes negative, GC_malloc will receive a negative number that will interpret as a size_t, ie. as an unsigned integer. This means that a negative request will be interpreted as an allocation request of more than 2GB. Testing the result of an allocation allows a clean exit, or an exception handling construct, or other solutions that allow you to pinpoint with greater easy where the failure happened.

You should be aware of this problem when using flexible arrays. When you write

```
int fn(int a)
{
     int array[a];
}
```

Here you are allocating “a” integers in the stack when control reaches the function prologue. There is no way for you to check if this worked or not, the only failure notification will be a stack overflow if “a” has the wrong value.

2.20 Some Coding Tips

Here are some useful receipts for solving specific windows problems.

2.20.1 Determining which version of Windows is running

```
BOOL  InWinNT() //test for NT
{
     OSVERSIONINFO osv;
     GetVersionEx(&osv);
     osv.dwOSVersionInfoSize=sizeof(osv);
     return osv.dwPlatformId==VER_PLATFORM_WIN32_NT;
}
```

2.20.2 Translating the value returned by GetLastError() into a readable string

This utility function uses the system "FormatMessage" to do the bulk of the work of caring about locale settings like language setup, etc.

```
BOOL GetFormattedError(LPTSTR dest,int size)
{
 DWORD dwLastError=GetLastError();
 if(!dwLastError)
    return 0;
 BYTE width=0;
 DWORD flags;
 flags  = FORMAT_MESSAGE_MAX_WIDTH_MASK &width;
 flags |= FORMAT_MESSAGE_FROM_SYSTEM;
  flags |= FORMAT_MESSAGE_IGNORE_INSERTS;
 return 0 != FormatMessage(flags,
         NULL,
         dwLastError,
         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
         dest,
         size,
         NULL);
}
```

2.20.3 Clearing the screen in text mode

The following code will clear the screen in text mode.

```
#include <windows.h>

/* Standard error macro for reporting API errors */
#define PERR(bSuccess, api){if(!(bSuccess)) \
     printf("%s:Error %d from %s \
   on line %d\n", __FILE__, GetLastError(), api, __LINE__);}

void cls( HANDLE hConsole )
{
   COORD coordScreen = { 0, 0 };   /* Home the cursor here */
   BOOL bSuccess;
   DWORD cCharsWritten;
   CONSOLE_SCREEN_BUFFER_INFO csbi; /* to get buffer info */
   DWORD dwConSize;       /* number of character cells in the current buffer */
  /* get the number of character cells in the current buffer */

   bSuccess = GetConsoleScreenBufferInfo( hConsole, &csbi );
```

```
    PERR( bSuccess, "GetConsoleScreenBufferInfo" );
    dwConSize = csbi.dwSize.X * csbi.dwSize.Y;

    /* fill the entire screen with blanks */

    bSuccess = FillConsoleOutputCharacter( hConsole, (TCHAR) ' ',
       dwConSize, coordScreen, &cCharsWritten );
    PERR( bSuccess, "FillConsoleOutputCharacter" );

      /* get the current text attribute */

    bSuccess = GetConsoleScreenBufferInfo( hConsole, &csbi );
    PERR( bSuccess, "ConsoleScreenBufferInfo" );

       /* now set the buffer's attributes accordingly */

    bSuccess = FillConsoleOutputAttribute(hConsole, csbi.wAttributes,
       dwConSize, coordScreen, &cCharsWritten );
    PERR( bSuccess, "FillConsoleOutputAttribute" );
      /* put the cursor at (0, 0) */

    bSuccess = SetConsoleCursorPosition( hConsole, coordScreen );
    PERR( bSuccess, "SetConsoleCursorPosition" );
    return;
}
```

This function can be called like this:

```
cls(GetStdHandle(STD_OUTPUT_HANDLE));
```

The library TCCONIO.LIB contains many other functions for text manipulation using the console interface. The corresponding header file is TCCONIO.H, which is automatically included when you include conio.h

This library was contributed by Daniel Guerrero (daguer@geocities.com)

2.20.4 Getting a pointer to the stack

To get a pointer to the stack, use the following code:

```
far MyFunction()
{
int x;
int far *y = &x;
}
```

NOTE: This pointer will not be valid when the function is exited, since the stack contents will change.

2.20.5 Disabling the screen saver from a program

Under Windows NT, you can disable the screen saver from your application code. To detect if the screen saver is enabled, use this:

```
SystemParametersInfo( SPI_GETSCREENSAVEACTIVE,
                      0,
                      pvParam,
                      0
                    );
```

On return, the parameter pvParam will point to TRUE if the screen saver setting is enabled in the system control panel applet and FALSE if the screen saver setting is not enabled.

To disable the screen saver setting, call SystemParametersInfo() with this:

```
SystemParametersInfo( SPI_SETSCREENSAVEACTIVE,
                      FALSE,
                      0,
                      SPIF_SENDWININICHANGE
                    );
```

2.20.6 Drawing a gradient background

You can draw a smooth gradient background using the following code:

```
void DrawBackgroundPattern(HWND hWnd)
{
  HDC hDC = GetDC(hWnd);  // Get the DC for the window.
  RECT rectFill;          // Rectangle for filling band.
  RECT rectClient;        // Rectangle for entire client area.
  float fStep;            // How large is each band?
  HBRUSH hBrush;
  int iOnBand;  // Loop index

 // How large is the area you need to fill?
  GetClientRect(hWnd, &rectClient);

 // Determine how large each band should be in order to cover the
 // client with 256 bands (one for every color intensity level).
  fStep = (float)rectClient.bottom / 256.0f;

 // Start filling bands
  for (iOnBand = 0; iOnBand < 256; iOnBand++) {

    // Set the location of the current band.
    SetRect(&rectFill,
            0,                            // Upper left X
            (int)(iOnBand * fStep),       // Upper left Y
            rectClient.right+1,           // Lower right X
            (int)((iOnBand+1) * fStep)); // Lower right Y

  // Create a brush with the appropriate color for this band.
    hBrush = CreateSolidBrush(RGB(0, 0, (255 – iOnBand)));


  // Fill the rectangle.
    FillRect(hDC, &rectFill, hBrush);

  // Get rid of the brush you created.
    DeleteObject(hBrush);
  };

  // Give back the DC.
  ReleaseDC(hWnd, hDC);
}
```

2.20.7 Capturing and printing the contents of an entire window

```
//
// Return a HDC for the default printer.
//
HDC             GetPrinterDC(void)
{
    PRINTDLG        pdlg;
```

```
    memset(&pdlg, 0, sizeof(PRINTDLG));
    pdlg.lStructSize = sizeof(PRINTDLG);
    pdlg.Flags = PD_RETURNDEFAULT | PD_RETURNDC;
    PrintDlg(&pdlg);
    return pdlg.hDC;
}
//
// Create a copy of the current system palette.
//
HPALETTE        GetSystemPalette()
{
    HDC             hDC;
    HPALETTE        hPal;
    HANDLE          hLogPal;
    LPLOGPALETTE    lpLogPal;
    // Get a DC for the desktop.
     hDC = GetDC(NULL);
  // Check to see if you are a running in a palette-based
  // video mode.
    if (!(GetDeviceCaps(hDC, RASTERCAPS) & RC_PALETTE)) {
        ReleaseDC(NULL, hDC);
        return NULL;
    }
  // Allocate memory for the palette.
    lpLogPal = GlobalAlloc(GPTR, sizeof(LOGPALETTE) + 256 *
                           sizeof(PALETTEENTRY));
    if (!hLogPal)
        return NULL;
     // Initialize.
     lpLogPal->palVersion = 0x300;
     lpLogPal->palNumEntries = 256;
     // Copy the current system palette into the logical palette.
    GetSystemPaletteEntries(hDC, 0, 256,
                            (LPPALETTEENTRY) (lpLogPal->palPalEntry));
    // Create the palette.
     hPal = CreatePalette(lpLogPal);
    // Clean up.
     GlobalFree(lpLogPal);
     ReleaseDC(NULL, hDC);
    return hPal;
}
//
// Create a 24-bit-per-pixel surface.
//
HBITMAP         Create24BPPDIBSection(HDC hDC, int iWidth, int iHeight)
{
    BITMAPINFO      bmi;
    HBITMAP         hbm;
    LPBYTE          pBits;
    // Initialize to 0s.
     ZeroMemory(&bmi, sizeof(bmi));
    // Initialize the header.
    bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    bmi.bmiHeader.biWidth = iWidth;
    bmi.bmiHeader.biHeight = iHeight;
    bmi.bmiHeader.biPlanes = 1;
    bmi.bmiHeader.biBitCount = 24;
    bmi.bmiHeader.biCompression = BI_RGB;   // Create the surface.
    hbm = CreateDIBSection(hDC,&bmi, DIB_RGB_COLORS,&pBits,NULL, 0);
    return (hbm);
```

```
}
//
// Print the entire contents (including the non-client area) of
// the specified window to the default printer.
BOOL            PrintWindowToDC(HWND hWnd)
{
    HBITMAP         hbm;
    HDC             hdcPrinter;
    HDC             hdcMemory;
    HDC             hdcWindow;
    int             iWidth;
    int             iHeight;
    DOCINFO         di;
    RECT            rc;
    DIBSECTION      ds;
    HPALETTE        hPal;
     // Do we have a valid window?
     if (!IsWindow(hWnd))
     return FALSE;
     // Get a HDC for the default printer.
     hdcPrinter = GetPrinterDC();
     if (!hdcPrinter)
        return FALSE;
     // Get the HDC for the entire window.
     hdcWindow = GetWindowDC(hWnd);
      // Get the rectangle bounding the window.
     GetWindowRect(hWnd, &rc);
     // Adjust coordinates to client area.
     OffsetRect(&rc, -rc.left, -rc.top);
     // Get the resolution of the printer device.
     iWidth = GetDeviceCaps(hdcPrinter, HORZRES);
     iHeight = GetDeviceCaps(hdcPrinter, VERTRES);
     // Create the intermediate drawing surface at window resolution.
     hbm = Create24BPPDIBSection(hdcWindow, rc.right, rc.bottom);
     if (!hbm) {
        DeleteDC(hdcPrinter);
        ReleaseDC(hWnd, hdcWindow);
        return FALSE;
     }
     // Prepare the surface for drawing.
     hdcMemory = CreateCompatibleDC(hdcWindow);
     SelectObject(hdcMemory, hbm);
     // Get the current system palette.
    hPal = GetSystemPalette();  // If a palette was returned.
    if (hPal) {                 // Apply the palette to the source DC.
        SelectPalette(hdcWindow, hPal, FALSE);
        RealizePalette(hdcWindow);
        // Apply the palette to the destination DC.
        SelectPalette(hdcMemory, hPal, FALSE);
        RealizePalette(hdcMemory);
    }
     // Copy the window contents to the memory surface.
    BitBlt(hdcMemory, 0, 0, rc.right, rc.bottom,
           hdcWindow, 0, 0, SRCCOPY);
     // Prepare the DOCINFO.
    ZeroMemory(&di, sizeof(di));
    di.cbSize = sizeof(di);
    di.lpszDocName = "Window Contents"; // Initialize the print job.
    if (StartDoc(hdcPrinter, &di) > 0) {// Prepare to send a page.
        if (StartPage(hdcPrinter) > 0) {
```

```
            // Retrieve the information describing the surface.
            GetObject(hbm, sizeof(DIBSECTION), &ds);
            // Print the contents of the surface.
            StretchDIBits(hdcPrinter,
                          0, 0, iWidth, iHeight,
                          0, 0, rc.right, rc.bottom, ds.dsBm.bmBits,
                          (LPBITMAPINFO) & ds.dsBmih, DIB_RGB_COLORS,
                          SRCCOPY);
            // Let the driver know the page is done.
            EndPage(hdcPrinter);
        }
        // Let the driver know the document is done.
        EndDoc(hdcPrinter);
    }
    // Clean up the objects you created.
    DeleteDC(hdcPrinter);
    DeleteDC(hdcMemory);
    ReleaseDC(hWnd, hdcWindow);
    DeleteObject(hbm);
    if (hPal)
        DeleteObject(hPal);
}
```

2.20.8 Centering a dialog box in the screen

Use the following code:

```
{
RECT rc;
GetWindowRect(hDlg, &rc);
SetWindowPos(hDlg, NULL,
     ((GetSystemMetrics(SM_CXSCREEN) - (rc.right - rc.left)) / 2),
     ((GetSystemMetrics(SM_CYSCREEN) - (rc.bottom - rc.top)) / 2),
           0, 0, SWP_NOSIZE | SWP_NOACTIVATE);
}
```

2.20.9 Determining the number of visible items in a list box

In a list box, if the number of lines is greater than the number of lines in the list box, some of them will be hidden. In addition, it could be that the list box is an owner draw list box, making the height of each line a variable quantity. Here is a code snippet that will handle all cases, even when all of the items of the list box are visible, and some white space is left at the bottom.

The basic idea is to subtract each line height from the total height of the client area of the list box.

```
int ntop, nCount, nRectheight, nVisibleItems;
RECT rc, itemrect;
// First, get the index of the first visible item.
ntop = SendMessage(hwndList, LB_GETTOPINDEX, 0, 0);
// Then get the number of items in the list box.
nCount = SendMessage(hwndList, LB_GETCOUNT, 0, 0);
// Get the list box rectangle.
GetClientRect(hwndList, &rc);
// Get the height of the list box's client area.
nRectheight = rc.bottom – rc.top;
// This counter will hold the number of visible items.
nVisibleItems = 0;
// Loop until the bottom of the list box. or the last item has been reached.
```

```
while ((nRectheight > 0) && (ntop < nCount))
{
     // Get current line's rectangle.
     SendMessage(hwndList, LB_GETITEMRECT,
                    ntop,(DWORD)(&itemrect));
     // Subtract the current line height.
      nRectheight = nRectheight – (itemrect.bottom – itemrect.top);
      nVisibleItems++;            // Increase item count.
      Ntop++;                     // Move to the next line.
}
```

2.20.10 Starting a non-modal dialog box

Non-modal dialog boxes behave as independent top level windows. They can be started using the `CreateDialog` function.

```
HWND CreateDialog(
     HINSTANCE hInstance,// handle to application instance
     LPCTSTR lpTemplate,// Identifies dialog box template name.
     HWND hWndParent,// Handle to owner window.
     DLGPROC lpDialogFunc // Pointer to dialog box procedure.
   );
```

Non-modal dialog boxes should be used with care, since they are equivalent to starting another thread of execution. In addition, you should never forget that the user can restart the same sequence of events that led to that dialog box, causing a second dialog box to appear.

2.20.11 Propagating environment variables to the parent environment

Under Windows, there is no 'export' directive as in Unix systems. To propagate the value of an environment variable to the rest of the system, use the following registry key:

```
HKEY_CURRENT_USER \
         Environment
```

You can modify system environment variables by editing the following registry key:

```
   HKEY_LOCAL_MACHINE \
               SYSTEM \
    CurrentControlSet \
              Control \
      Session Manager \
          Environment
```

Note that any environment variable that needs to be expanded (for example, when you use %SYSTEM%) must be stored in the registry as a REG_EXPAND_SZ registry value. Any values of type REG_SZ will not be expanded when read from the registry.

The problem with this method, however, is that changes will be effective only after the next logoff, probably not exactly what you want.

To effect these changes without having to log off, broadcast a WM_SETTINGCHANGE message to all windows in the system, so that any applicable applications (such as Program Manager, Task Manager, Control Panel, etc.) can perform an update.

```
SendMessageTimeout(HWND_BROADCAST, WM_SETTINGCHANGE, 0,
    (LPARAM) "Environment", SMTO_ABORTIFHUNG,
    5000, &dwReturnValue);
```

In theory, this will work, but there is a problem with Windows 95. Under that system, there is no other way to set those variables than rebooting the system!

2.20.12 Restarting the shell under program control

In many cases it can be necessary to restart the shell. To do this, find the window of the Explorer, send it a quit message, and restart it. The following code snippet will work:

```
HWND hwndShell = FindWindow("Progman", NULL);
PostMessage(hwndShell, WM_QUIT, 0, 0L);
WinExec("Explorer.exe",SW_SHOW);
```

2.20.13 Translating client coordinates to screen coordinates

To determine the screen coordinates for the client area of a window, call the ClientToScreen function to translate the client coordinates returned by GetClientRect into screen coordinates. The following code demonstrates how to use the two functions together:

```
RECT rMyRect;
GetClientRect(hwnd, (LPRECT)&rMyRect);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.left);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.right);
```

2.20.14 Passing an argument to a dialog box procedure

You can pass a void * to a dialog box procedure by calling:

```
result = DialogBoxParam(hInst,// Instance of the application.
MAKEINTRESOURCE(id),     // The resource ID or dialog box name.
GetActiveWindow(),       // The parent window.
Dlgfn,                   // The dialog box procedure.
(DWORD) "Hello");        // The arguments.
```

In your dialog box procedure (here DlgFn), you will find those arguments in the lParam parameter of the WM_INITDIALOG message.

2.20.15 Calling printf from a windows application

Windows applications do not have a console, i.e., the ‘DOS’ window. To access the console from a Windows application, create one, initialize stdout, and use it as you would normally use it from a native console application.

```
#include <windows.h>
#include <stdio.h>
#include <fcntl.h>
int main(void)
{
     int hCrt;
     FILE *hf;

     AllocConsole();
     hCrt = _open_osfhandle((long)  GetStdHandle (
               STD_OUTPUT_HANDLE),_O_TEXT );
     hf = fdopen( hCrt, "w" );
     *stdout = *hf;
     setvbuf( stdout, NULL, _IONBF, 0 );
     printf("Hello world\n");
     return 0;
}
```

2.20.16 Enabling or disabling a button or control in a dialog box.

You should first get the window handle of the control using

```
hwndControl = GetDlgItem(hwndDlg,IDBUTTON);
```

Using that window handle, call EnableWindow.

2.20.17 Making a window class available for all applications in the system.

Under 32-bit Windows and Windows NT, a style of CS_GLOBALCLASS indicates that the class is available to every DLL in the process, not every application and DLL in the system, as it does in Windows 3.1.

To have a class registered for every process in the system under Windows NT:

- Register the class in a DLL.
- Use a style of CS_GLOBALCLASS.
- List the DLL in the following registry key.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
   Windows\AppInit_DLLs
```

This will force the DLL to be loaded into every process in the system, thereby registering the class in each and every process.

NOTE: This technique does not work under Windows 95.

2.20.18 Accessing the disk drive directly without using a file system

To open a physical hard drive for direct disk access (raw I/O) in a Win32-based application, use a device name of the form

```
\\.\PhysicalDriveN
```

where N is 0, 1, 2, and so forth, representing each of the physical drives in the system.

To open a logical drive, direct access is of the form

```
\\.\X:
```

where X: is a hard-drive partition letter, floppy disk drive, or CD-ROM drive.

You can open a physical or logical drive using the CreateFile() application programming interface (API) with these device names provided that you have the appropriate access rights to the drive (that is, you must be an administrator). You must use both the CreateFile() FILE_SHARE_READ and FILE_SHARE_WRITE flags to gain access to the drive.

Once the logical or physical drive has been opened, you can then perform direct I/O to the data on the entire drive. When performing direct disk I/O, you must seek, read, and write in multiples of sector sizes of the device and on sector boundaries. Call DeviceIoControl() using IOCTL_DISK_GET_DRIVE_GEOMETRY to get the bytes per sector, number of sectors, sectors per track, and so forth, so that you can compute the size of the buffer that you will need.

Note that a Win32-based application cannot open a file by using internal Windows NT object names; for example, attempting to open a CD-ROM drive by opening

```
\Device\CdRom0
```

does not work because this is not a valid Win32 device name. You can use the `QueryDosDevice()` API to get a list of all valid Win32 device names and see the mapping between a particular Win32 device name and an internal Windows NT object name. An application running

at a sufficient privilege level can define, redefine, or delete Win32 device mappings by calling the DefineDosDevice() API.

2.20.19 Retrieving the Last-Write Time

The following example uses the GetFileTime function to retrieve the last-write time for a file. It converts the time to local time based on the current time-zone settings, and creates a date and time string that can be shown to the user.

```
BOOL GetLastWriteTime(HANDLE hFile, LPSTR String)
{
    FILETIME ftCreate, ftAccess, ftWrite;
    SYSTEMTIME stUTC, stLocal;

     // Retrieve the file times for the file.
    if (!GetFileTime(hFile, &ftCreate, &ftAccess, &ftWrite))
        return FALSE;

     // Convert the last-write time to local time.
    FileTimeToSystemTime(&ftWrite, &stUTC);
    SystemTimeToTzSpecificLocalTime(NULL, &stUTC, &stLocal);

     // Build a string showing the date and time.
    sprintf(String, "%02d/%02d/%d  %02d:%02d",
        stLocal.wDay, stLocal.wMonth, stLocal.wYear,
        stLocal.wHour, stLocal.wMinute);

    return TRUE;
}
```

2.20.20 Setting the System Time

The following example sets the system time using the SetSystemTime function.

```
BOOL SetNewTime(WORD hour, WORD minutes)
{
    SYSTEMTIME st;
    char *pc;

    GetSystemTime(&st);       // gets current time
    st.wHour = hour;          // adjusts hours
    st.wMinute = minutes;     // and minutes
    if (!SetSystemTime(&st))  // sets system time
        return FALSE;
    return TRUE;
}
```

2.20.21 Getting the list of running processes

You can use pdh.lib to get a list of all the running processes. here is a simple code that will print this list in the standard output.

```
/*
 * Using pdh.lib to get a list of the running processes. This sample is
 * adapted from the similar sample code in the windows SDK. Compile with:
 * lc listproc.c pdh.lib
 */
#include <windows.h>
#include <winperf.h>
#include <malloc.h>
```

```
#include <stdio.h>
#include <pdh.h>
#include <pdhmsg.h>

int main(void)
{
     PDH_STATUS  pdhStatus              = ERROR_SUCCESS;
     LPTSTR      szCounterListBuffer    = NULL;
     DWORD       dwCounterListSize      = 0;
     LPTSTR      szInstanceListBuffer   = NULL;
     DWORD       dwInstanceListSize     = 0;
     LPTSTR      szThisInstance         = NULL;


     // call the function to determine the required buffer size for the data
     pdhStatus = PdhEnumObjectItems(
          NULL,                  // reserved
          NULL,                  // local machine
          "Process",             // object to enumerate
          szCounterListBuffer,   // pass in NULL buffers
          &dwCounterListSize,    // an 0 length to get
          szInstanceListBuffer,  // required size
          &dwInstanceListSize,   // of the buffers in chars
          PERF_DETAIL_WIZARD,    // counter detail level
          0);
     if (pdhStatus != ERROR_SUCCESS && pdhStatus != PDH_MORE_DATA) {
          printf("\nUnable to determine the buffer size required");
          return 1;
     }
     // allocate the buffers and try the call again
     // PdhEnum functions will return ERROR_SUCCESS in WIN2K, but
     // PDH_MORE_DATA in XP and later.
     // In either case, dwCounterListSize and dwInstanceListSize should contain
     // the correct buffer size needed.
     //
     szCounterListBuffer = (LPTSTR)malloc (dwCounterListSize );
     szInstanceListBuffer = (LPTSTR)malloc(dwInstanceListSize);
     if ((szCounterListBuffer == NULL) ||
          (szInstanceListBuffer == NULL)) {
          printf ("\nUnable to allocate buffers");
          return 1;
     }
     pdhStatus = PdhEnumObjectItems (NULL,   // reserved
          NULL,   // local machine
          "Process", // object to enumerate
          szCounterListBuffer,    // pass in NULL buffers
          &dwCounterListSize,     // an 0 length to get
          szInstanceListBuffer,   // required size
          &dwInstanceListSize,    // of the buffers in chars
          PERF_DETAIL_WIZARD,     // counter detail level
          0);
     if (pdhStatus == ERROR_SUCCESS){
          printf ("\nRunning Processes:");
          // walk the return instance list
          for (szThisInstance = szInstanceListBuffer;
        *szThisInstance != 0;
        szThisInstance += strlen(szThisInstance) + 1) {
               printf ("\n  %s", szThisInstance);
          }
     }
```

```
    if (szCounterListBuffer != NULL) free (szCounterListBuffer);
    if (szInstanceListBuffer != NULL) free (szInstanceListBuffer);
    return 0;
}
```

2.20.22 Changing a File Time to the Current Time

The following example sets the last-write time for a file to the current system time using the SetFileTime function.

```
BOOL SetFileToCurrentTime(HANDLE hFile)
{
    FILETIME ft;
    SYSTEMTIME st;
    BOOL f;

    GetSystemTime(&st);              // gets current time
    SystemTimeToFileTime(&st, &ft);  // converts to file time format
    f = SetFileTime(hFile,           // sets last-write time for file
        (LPFILETIME) NULL, (LPFILETIME) NULL, &ft);
    return f;
}
```

2.20.23 Displaying the amount of disk space for each drive

```
#include <windows.h>
#include <stdio.h>
// This program will loop through all drives in the system and will print the
// capacity of each drive, the number of bytes free/used, and the percentage free.
// At the end of the loop it will print the totals.
int main(void)
{
     long long BytesAvailable,TotalBytesAvailable=0;
     long long capacity,TotalCapacity=0;
     long long userFree,TotalUserFree=0;
     long long used,TotalUsed=0;
     long double percent;
     int counter = 'C'; // Start with C drive ignoring floppies
     char diskname[512];

     strcpy(diskname,"C:\\");
     printf("%-6s %15s %15s %15s %6s\n","Drive", "Capacity",
     "Available","Used","Free");
     while (counter != (1+'Z')) {
     diskname[0] = counter;
     if (GetDiskFreeSpaceEx(diskname, &BytesAvailable,
          &capacity,&userFree)) {
          percent = 100.0L*(((long double)BytesAvailable)/(long
          double)capacity);
          used = capacity-BytesAvailable;
               /*
               printf formats:
                    %-6s  format string in 6 position, left justified  (negative width)
                    %15'lld format 64 bit in 15 positions separating digits in groups (')
                    %6.2Lf  format long double (Lf) in 6 positions with 2 decimals
               */
          printf("%-6s %15'lld %15'lld %15'lld %6.2Lf%%\n",
          diskname,capacity,BytesAvailable,used,percent);
          TotalBytesAvailable+=BytesAvailable;
```

```
            TotalCapacity+=capacity;
            TotalUsed+=used;
        }
        counter++;
        }
        // Now print the totals
        percent = 100.0L*(((long double)TotalBytesAvailable)/(long
             double)TotalCapacity);
        printf("\n%-6s %15'lld %15'lld %15'lld %6.2Lf%%\n", "Total:",
        TotalCapacity,TotalBytesAvailable,TotalUsed,percent);
        return 0;
    }
```

The output of this program can look like this:

```
Drive          Capacity       Available            Used   Free
C:\       6,292,303,872   2,365,452,288   3,926,851,584  37.59%
D:\      10,487,197,696   3,563,794,432   6,923,403,264  33.98%
E:\      31,453,437,952  17,499,627,520  13,953,810,432  55.64%
F:\      15,726,731,264  10,327,638,016   5,399,093,248  65.67%
H:\         569,366,528               0     569,366,528   0.00%
I:\      31,790,673,920  27,672,530,944   4,118,142,976  87.05%

Total:   96,319,711,232  61,429,043,200  34,890,668,032  63.78%
```

Since it doesn't check if a drive is a CD drive, drive H: is listed, and it has obviously 0 bytes available since it is read only.

2.20.24 Mounting and unmounting volumes in NTFS 5.0

The windows file system allows you to "mount" disk drive in another directory, and see the contents of the drive as if they were subdirectories of the directory where the drive is mounted. Suppose, for instance, that you make a directory C:\mnt, and in this directory you make a subdirectory called cdrom. Using the utility below, you can mount the CDROM drive under the c:\mnt\cdrom directory, and you will see the contents of the cdrom drive in c:\mnt\cdrom.

2.20.24.1 Mount

```
#include <windows.h>
#include <stdio.h>
#define BUFSIZE MAX_PATH
int main( int argc, char *argv[] )
{
   BOOL bFlag;
   char Buf[BUFSIZE];     // temporary buffer for volume name
   char buf1[BUFSIZE];     // temporary buffer for volume name
   DWORD nl,flags;
   char fsname[512];

   if( argc != 3 ) {
      printf("Usage: mount <directory> <drive>\n");
      printf("For example\nmount c:\\mnt\\cdrom g:\\\n");
      return( -1 );
   }

  // We should do some error checking on the inputs. Make sure
  // there are colons and backslashes in the right places, etc.

   bFlag = GetVolumeNameForVolumeMountPoint(
              argv[2], // input volume mount point or directory
                  Buf, // output volume name buffer
```

```
                BUFSIZE // size of volume name buffer
            );

    if (bFlag != TRUE) {
       printf( "Retrieving volume name for %s failed.\n", argv[2] );
       return (-2);
    }
    // Check that the file system supports mounting
    bFlag = GetVolumeInformation(argv[2],buf1,BUFSIZE,
         NULL,&nl,&flags,fsname,256);
    if (0 == (flags & FILE_SUPPORTS_REPARSE_POINTS)) {
         printf("File system doesn't support mount points\n");
         return (-3);
    }

    printf( "Volume name of %s is %s\n", argv[2], Buf );
    if (strlen(argv[1]) < 2) {
       printf("Incorrect path name %s\n",argv[1]);
       return -4;
    }
    strncpy(fsname,argv[1],sizeof(fsname)-1);
    fsname[sizeof(fsname)-1] = 0;
    // Add a trailing backslash, if not this call will not work
    if (fsname[strlen(fsname)-1] != '\\') {
       strcat(fsname,"\\");
    }
    bFlag = SetVolumeMountPoint(
               fsname, // mount point
                    Buf // volume to be mounted
            );

    if (!bFlag) {
      printf ("Attempt to mount %s at %s failed. Error code %d\n",
               argv[2], argv[1],GetLastError());
    }
    else printf("%s mounted in %s\n",argv[2],argv[1]);

    return (!bFlag);
}
```

2.20.24.2 Umount

Once we have mounted a drive, we can unmount it with the following utility:

```
#include <windows.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
   BOOL bFlag;
   char mpath[512];

   if (argc != 2)
   {
      printf("%s unmounts a volume from the volume mount point.\n",
         argv[0]);
      printf("For example:\numount c:\\mnt\\cdrom\n");
      return (-1);
   }

   strncpy(mpath,argv[1],sizeof(mpath)-3);
   mpath[sizeof(mpath)-2] = 0;
```

```
    if (mpath[strlen(mpath)-1] != '\\') {
       strcat(mpath,"\\");
    }
    bFlag = DeleteVolumeMountPoint( mpath );

    printf ("%s %s in unmounting the volume at %s.\n",
            argv[0], bFlag ? "succeeded" : "failed", mpath);

    return (!bFlag);
}
```

2.21 FAQ

Here are some answer to questions users of lcc-win32 have asked, or questions that I imagine you will find useful.

2.21.1 How do I create a progress report with a Cancel button?

Here is the answer from the MSDN Knowledge base, article Q76415

The following article describes the mechanisms necessary to implement a progress or activity indicator with a Cancel Operation option, to be used for lengthy and CPU-intensive subtasks of an application. Examples of operations using this include: copying multiple files, directory searches, or printing large files.

The progress dialog box is implemented in two steps:

- Initialize the dialog box before starting lengthy or CPU intensive subtask.
- After each unit of the subtask is complete, call ProgressYield() to determine if the user has canceled the operation and to update the progress or activity indicator.

This is the description of the progress dialog procedure. The procedure uses a global variable (Cancel) to inform the CPU-intensive subtask that the user has indicated a desire to terminate the subtask.

```
WORD Cancel = FALSE;    /* This must be global to all modules */
                        /* which call ProgressYield()         */
BOOL FAR PASCAL ProgressDlgProc(hDlg, message, wParam, lParam)
HWND hDlg;
unsigned message;
WORD wParam;
DWORD lParam;
{
switch (message)
   {
/* Use other messages to update the progress or activity */
/* indicator.                                            */
   .
   case WM_COMMAND:
      switch (wParam)
         {
         case ID_CANCEL:    /* ID_CANCEL = 2 */
            Cancel = TRUE;

         default:
            return FALSE;
         }
   .
```

```
      .
      default:
         return FALSE;
      }

}
```

The following describes the ProgressYield procedure, which should be called after each unit of the CPU-intensive subtask is completed. The ProgressYield procedure uses the IsDialogMessage function (described in the "Microsoft Windows Software Development Kit Reference Volume 1"). IsDialogMessage will convert keyboard messages into selection commands for the corresponding dialog box.

```
void ProgressYield(HWND hwnd)
{
MSG   msg;

/* Remove all available messages for any window that belong */
/* to the current application.                               */
while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
   {
   /* Translate and Dispatch the given message if the window */
   /* handle is null or the given message is not for the    */
   /* modeless dialog box hwnd.                             */
   if (!hwnd || !IsDialogMessage(hwnd, &msg))
      {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
      }
   }
}
```

The following describes how to incorporate the progress dialog as part of an application's subtask that is CPU-intensive. The PROGRESS_DLG resource should contain a button with an ID of 2, because this is the wParam of the WM_COMMAND that will be sent when the user presses the ESC key. The button should also have the BS_DEFPUSHBUTTON style so that the ENTER key will also result in the termination of the CPU-intensive subtask.

```
FARPROC lpProgressProc;
HWND    hwndProgress;           /* This needs to be global if */
                                /* accessed by other modules.  */
.
.
/* Initialize before starting CPU-intensive work.              */
lpProgressProc = MakeProcInstance(ProgressDlgProc,
                                  hInst);  /* Current instance. */
hwndProgress = CreateDialog(hInst,         /* Current instance. */
                            "PROGRESS_DLG", /* Resource.        */
                            hwndParent,     /* Parent handle.   */
                            lpProgressProc);/* Instance address. */
ShowWindow(hwndProgress);
.
.
/* Start CPU intensive work here. */
   .
   .
   /* Before or after each unit of work, the application */
   /* should do the following:                           */
   ProgressYield(hwndProgress);
   if (Cancel == TRUE)
      break;   /* Terminate CPU-intensive work immediately. */
```

```
    .
    .
/* End CPU-intensive work here. */
.
.
DestroyWindow(hwndProgress);
FreeProcInstance(lpProgressProc);
.
```

2.21.2 How do I show in the screen a print preview?

If a screen font is available that exactly matches (or at least very closely corresponds to) the chosen printer font, then the process is very straightforward and consists of seven steps:

- Retrieve a Device Context (DC) or an Information Context (IC) for the printer.
- Call EnumFontFamilies() to obtain a LOGFONT structure for the selected printer font. The nFontType parameter to the EnumFontFamilies() callback function specifies if a given font is a device font.
- Get a DC for the screen.
- Convert the lfHeight and lfWidth members of the LOGFONT structure from printer resolution units to screen resolution units. If a mapping mode other than MM_TEXT is used, round-off error may occur.
- Call CreateFontIndirect() with the LOGFONT structure.
- Call SelectObject(). GDI will select the appropriate screen font to match the printer font.
- Release the printer device context or information context and the screen device context.

If a screen font that corresponds to the selected printer font is not available, the process is more difficult. It is possible to modify the character placement on the screen to match the printer font to show justification, line breaks, and page layout. However, visual similarity between the printer fonts and screen fonts depends on a number of factors, including the number and variety of screen fonts available, the selected printer font, and how the printer driver describes the font. For example, if the printer has a serifed Roman- style font, one of the GDI serifed Roman-style fonts will appear to be very similar to the printer font. However, if the printer has a decorative Old English-style font, no corresponding screen font will typically be available. The closest available match would not be very similar.

To have a screen font that matches the character placement of a printer font, do the following:

- Perform the preceding seven steps to retrieve an appropriate screen font.
- Get the character width from the TEXTMETRIC structure returned by the EnumFonts function in step 2 above. Use this information to calculate the page position of each character to be printed in the printer font.
- Allocate a block of memory and specify the spacing between characters. Make sure that this information is in screen resolution units.
- Specify the address of the memory block as the lpDx parameter to ExtTextOut(). GDI will space the characters as listed in the array.

2.21.3 How do I change the color of an edit field?

See page 252.

2.21.4 How do I draw a transparent bitmap?

A portion of a graphic image that does not change the contents of the screen is termed "transparent." The DrawIcon function can create an image that contains transparent portions. It is also possible to obtain this functionality using the BitBlt function; however, there are some additional steps involved.

The first step is to obtain the contents of the area where the bitmap will be drawn and to store this background image in a memory display context (DC). Mask out the area of the background that corresponds to the nontransparent portion of the image bitmap and mask out all transparent pixels from the image bitmap. Use the XOR raster operation to merge the image bitmap into the background bitmap. Finally, use the BitBlt function to move the merged image to the destination DC.

The following nine steps describe a process used to draw transparent bitmaps:

- Create a DC to hold the image bitmap.
- Select the image bitmap into the DC.
- Create a memory DC to hold the final image. This is the destination DC.
- Copy the portion of the screen that will be covered by the image into the destination DC.
- Create an "AND mask" that contains the mask of the colors to draw (the nontransparent portions of the image). To do this, perform the following three steps:
- Set the background color of the image DC to the color that will be transparent in the image.
- Create a monochrome DC.
- BitBlt the image into the monochrome DC. This will create an AND mask of the bitmap by setting pixels that match the background color to white (1), and setting all other pixels to black (0).
- Use BitBlt with the SRCAND raster operation code to copy the AND mask onto the destination DC. Use BitBlt with the SRCAND raster operation code to copy the inverse of the AND mask onto the image DC. Use BitBlt with the SRCPAINT raster operation code to copy the image DC onto the destination DC. Use BitBlt to copy the contents of the destination DC to the appropriate portion of the screen.

The following code is a function that demonstrates the preceding steps:

```
void DrawTransparentBitmap(HDC hdc, HBITMAP hBitmap, short xStart,
                           short yStart, COLORREF cTransparentColor)
{
BITMAP     bm;
COLORREF   cColor;
HBITMAP    bmAndBack, bmAndObject, bmAndMem, bmSave;
HBITMAP    bmBackOld, bmObjectOld, bmMemOld, bmSaveOld;
HDC        hdcMem, hdcBack, hdcObject, hdcTemp, hdcSave;
POINT      ptSize;

hdcTemp = CreateCompatibleDC(hdc);
SelectObject(hdcTemp, hBitmap);   // Select the bitmap

GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
ptSize.x = bm.bmWidth;            // Get width of bitmap
ptSize.y = bm.bmHeight;           // Get height of bitmap
DPtoLP(hdcTemp, &ptSize, 1);      // Convert from device

                                  // to logical points
```

```
  // Create some DCs to hold temporary data.
    hdcBack   = CreateCompatibleDC(hdc);
    hdcObject = CreateCompatibleDC(hdc);
    hdcMem    = CreateCompatibleDC(hdc);
    hdcSave   = CreateCompatibleDC(hdc);

  // Create a bitmap for each DC. DCs are required for a number of
  // GDI functions.

    // Monochrome DC
    bmAndBack   = CreateBitmap(ptSize.x, ptSize.y, 1, 1, NULL);

  // Monochrome DC
    bmAndObject = CreateBitmap(ptSize.x, ptSize.y, 1, 1, NULL);

    bmAndMem    = CreateCompatibleBitmap(hdc, ptSize.x, ptSize.y);
    bmSave      = CreateCompatibleBitmap(hdc, ptSize.x, ptSize.y);

      // Each DC must select a bitmap object to store pixel data.
    bmBackOld   = SelectObject(hdcBack, bmAndBack);
    bmObjectOld = SelectObject(hdcObject, bmAndObject);
    bmMemOld    = SelectObject(hdcMem, bmAndMem);
    bmSaveOld   = SelectObject(hdcSave, bmSave);

      // Set proper mapping mode.
    SetMapMode(hdcTemp, GetMapMode(hdc));

      // Save the bitmap sent here, because it will be overwritten.
    BitBlt(hdcSave, 0, 0, ptSize.x, ptSize.y, hdcTemp, 0, 0, SRCCOPY);

      // Set the background color of the source DC to the color. contained in the parts of
      // the bitmap that should be transparent
    cColor = SetBkColor(hdcTemp, cTransparentColor);

      // Create the object mask for the bitmap by performing a BitBlt
      // from the source bitmap to a monochrome bitmap.
    BitBlt(hdcObject, 0, 0, ptSize.x, ptSize.y, hdcTemp, 0, 0,SRCCOPY);

      // Set the background color of the source DC back to the original color.
    SetBkColor(hdcTemp, cColor);

      // Create the inverse of the object mask.
    BitBlt(hdcBack, 0, 0, ptSize.x, ptSize.y, hdcObject, 0,
0,NOTSRCCOPY);

      // Copy the background of the main DC to the destination.
    BitBlt(hdcMem, 0, 0, ptSize.x, ptSize.y, hdc, xStart,
yStart,SRCCOPY);

      // Mask out the places where the bitmap will be placed.
    BitBlt(hdcMem, 0, 0, ptSize.x, ptSize.y, hdcObject, 0, 0, SRCAND);

      // Mask out the transparent colored pixels on the bitmap.
    BitBlt(hdcTemp, 0, 0, ptSize.x, ptSize.y, hdcBack, 0, 0, SRCAND);

      // XOR the bitmap with the background on the destination DC.
    BitBlt(hdcMem, 0, 0, ptSize.x, ptSize.y, hdcTemp, 0, 0, SRCPAINT);

      // Copy the destination to the screen.
```

```
   BitBlt(hdc, xStart, yStart, ptSize.x, ptSize.y, hdcMem, 0,
0,SRCCOPY);

   // Place the original bitmap back into the bitmap sent here.
   BitBlt(hdcTemp, 0, 0, ptSize.x, ptSize.y, hdcSave, 0, 0, SRCCOPY);

   // Delete the memory bitmaps.
   DeleteObject(SelectObject(hdcBack, bmBackOld));
   DeleteObject(SelectObject(hdcObject, bmObjectOld));
   DeleteObject(SelectObject(hdcMem, bmMemOld));
   DeleteObject(SelectObject(hdcSave, bmSaveOld));

   // Delete the memory DCs.
   DeleteDC(hdcMem);
   DeleteDC(hdcBack);
   DeleteDC(hdcObject);
   DeleteDC(hdcSave);
   DeleteDC(hdcTemp);
   }
```

The following is an example of how the DrawTransparentBitmap function might be called:

```
   DrawTransparentBitmap(hdc,         // The destination DC.

                         hBitmap,     // The bitmap to be drawn.
                         xPos,        // X coordinate.
                         yPos,        // Y coordinate.
                         0x00FFFFFF); // The color for transparent
                                      // pixels (white, in this
                                      // example).
```

2.21.5 How do I draw a gradient background?

```
/*  DrawBackgroundPattern()                    *
    Purpose: This function draws a gradient pattern that transitions between blue and black.
    This is similar to the background used in Microsoft setup programs.  */
   void DrawBackgroundPattern(HWND hWnd)
   {
   HDC hDC = GetDC(hWnd);  // Get the DC for the window
   RECT rectFill;          // Rectangle for filling band
   RECT rectClient;        // Rectangle for entire client area
   float fStep;            // How large is each band?
   HBRUSH hBrush;
   int iOnBand;  // Loop index

     // How large is the area you need to fill?
   GetClientRect(hWnd, &rectClient);

     // Determine how large each band should be in order to cover the client with 256
     // bands (one for every color intensity level)
   fStep = (float)rectClient.bottom / 256.0f;

     // Start filling bands
   for (iOnBand = 0; iOnBand < 256; iOnBand++) {
          // Set the location of the current band
         SetRect(&rectFill,
            0,                          // Upper left X
            (int)(iOnBand * fStep),     // Upper left Y
            rectClient.right+1,         // Lower right X
            (int)((iOnBand+1) * fStep)); // Lower right Y

          // Create a brush with the appropriate color for this band
```

```
      hBrush = CreateSolidBrush(RGB(0, 0, (255 - iOnBand)));
    // Fill the rectangle
      FillRect(hDC, &rectFill, hBrush);

    // Get rid of the brush you created
      DeleteObject(hBrush);
    }
     // Give back the DC
   ReleaseDC(hWnd, hDC);
}
```

2.21.6 How do I calculate print margins?

An application can determine printer margins as follows:

Calculate the left and top margins

- Determine the upper left corner of the printable area calling GetDeviceCaps() with the PHYSICALOFFSETX and PHYSICALOFFSETY indices. For example:

```
// Init our pt struct in case escape not supported
pt.x = 0; pt.y = 0;
// Locate the upper left corner of the printable area
pt.x = GetDeviceCaps(hPrnDC, PHYSICALOFFSETX);
pt.y = GetDeviceCaps(hPrnDC, PHYSICALOFFSETY);
```

Determine the number of pixels required to yield the desired margin (x and y offsets) by calling GetDeviceCaps() using the LOGPIXELSX and LOGPIXELSY flags.

```
/* Figure out how much you need to offset output to produce the left and top margins for the
output in the printer. Note the use of the "max" macro. It is possible that you are asking for
margins that are not possible on this printer. For example, the HP LaserJet has a 0.25"
unprintable area so we cannot get margins of 0.1". */
xOffset = max (0, GetDeviceCaps (hPrnDC, LOGPIXELSX) *
                      nInchesWeWant - pt.x);
yOffset = max (0, GetDeviceCaps (hPrnDC, LOGPIXELSY) *
                      nInchesWeWant - pt.y);
/* When doing all the output, you can either offset it by the above values or call SetViewportOrg()
to set the point (0,0) at the margin offset you calculated.*/
SetViewportOrg (hPrnDC, xOffset, yOffset); //all other output here
```

- Calculate the bottom and right margins. Obtain the total size of the physical page (including printable and unprintable areas) calling GetDeviceCaps() with the PHYSICALWIDTH and PHYSICALHEIGHT indices in Windows NT.
- Determine the number of pixels required to yield the desired right and bottom margins by calling GetDeviceCaps using the LOGPIXELSX and LOGPIXELSY flags.
- Calculate the size of the printable area with GetDeviceCaps() using the HORZRES and VERTRES flags. The following code fragment illustrates steps a through c:

```
// Get the size of the printable area
pt.x = GetDeviceCaps(hPrnDC, PHYSICALWIDTH);
pt.y = GetDeviceCaps(hPrnDC, PHYSICALHEIGHT);

xOffsetOfRightMargin = xOffset +
                       GetDeviceCaps (hPrnDC, HORZRES) -
                       pt.x -
                       GetDeviceCaps (hPrnDC, LOGPIXELSX) *
                       wInchesWeWant;

yOffsetOfBottomMargin = yOffset +
                        GetDeviceCaps (hPrnDC, VERTRES) -
                        pt.y -
```

```
                              GetDeviceCaps (hPrnDC, LOGPIXELSY) *
    wInchesWeWant;
```

NOTE: Now, you can clip all output to the rectangle bounded by xOffset, yOffset, xOffsetOf-RightMargin, and yOffsetOfBottomMargin.

2.21.7 How do I calculate the bounding rectangle of a string of text?

```
/* Get a bounding rectangle for a string of text output to a specified coordinate in a DC using the
currently selected font NOTE: The reference DC must have a True Type font selected */
BOOL GetTextBoundingRect(HDC    hDC,   // Reference DC
                         int    x,     // X-Coordinate
                         int    y,     // Y-Coordinate
                        LPSTR  lpStr, // The text string to evaluate
                         DWORD  dwLen, // The length of the string
                         LPRECT lprc)  // Holds bounding rectangle
{
      LPPOINT lpPoints;
      LPBYTE lpTypes;
      int i, iNumPts;

     // Draw the text into a path
      BeginPath(hDC);
      i = SetBkMode(hDC, TRANSPARENT);
      TextOut(hDC, x, y, lpStr, dwLen);
      SetBkMode(hDC, i);
      EndPath(hDC);

     // How many points are in the path
      iNumPts = GetPath(hDC, NULL, NULL, 0);
      if (iNumPts == -1) return FALSE;

      // Allocate room for the points
      lpPoints = (LPPOINT)GlobalAlloc(GPTR, sizeof(POINT) * iNumPts);
      if (!lpPoints) return FALSE;

     // Allocate room for the point types
      lpTypes = GlobalAlloc(GPTR, iNumPts);
      if (!lpTypes) {
          GlobalFree(lpPoints);
          return FALSE;
      }

          // Get the points and types from the current path
      iNumPts = GetPath(hDC, lpPoints, lpTypes, iNumPts);

          // More error checking
      if (iNumPts == -1) {
          GlobalFree(lpTypes);
          GlobalFree(lpPoints);
          return FALSE;
      }

          // Intialize the rectangle
      SetRect(lprc, 0xFFFFF, 0xFFFFF, 0, 0);
}
```

2.21.8 How do I close an open menu?

To cancel an active menu send a WM_CANCELMODE message to the window that owns it.

When you send the WM_CANCELMODE message to the owner window of an active menu, the window exits the menu mode. You can use this technique to force the abandonment of active TrackPopupMenu or TrackPopupMenuEx.

2.21.9 How do I center a dialog box in the screen?

To center a dialog box on the screen before it is visible, add the following lines to the processing of the WM_INITDIALOG message:

```
{
RECT rc;

GetWindowRect(hDlg, &rc);

SetWindowPos(hDlg, NULL,
   ((GetSystemMetrics(SM_CXSCREEN) - (rc.right - rc.left)) / 2),
   ((GetSystemMetrics(SM_CYSCREEN) - (rc.bottom - rc.top)) / 2),
   0, 0, SWP_NOSIZE | SWP_NOACTIVATE);
}
```

This code centers the dialog horizontally and vertically.

2.21.10 How do I create non-rectangular windows?

In previous versions of Windows and Windows NT, it was possible to create only rectangular windows. To simulate a non-rectangular window required a lot of work on the application developer's part. Besides handling all drawing for the window, the application was required to perform hit-testing and force underlying windows to repaint as necessary to refresh the "transparent" portions of the window.

Windows 95 and Windows NT version 3.51 greatly simplify this by providing the SetWindowRgn function. An application can now create a region with any desired shape and use SetWindowRgn to set this as the clipping region for the window. Subsequent painting and mouse messages are limited to this region, and Windows automatically updates underlying windows that show through the non-rectangular window. The application need only paint the window as desired.

For more information on using SetWindowRgn, see the Win32 API documentation.

2.21.11 How do I implement a non-blinking caret?

Although Windows is designed to blink the caret at a specified interval, a timer function and SetCaretBlinkTime() can be used to prevent Windows from turning the caret off by following these three steps:

- Call `SetCaretBlinkTime(10000)`, which instructs Windows to blink the caret every 10,000 milliseconds (10 seconds). This results in a "round-trip" time of 20 seconds to go from OFF to ON and back to OFF (or vice versa).
- Create a timer, using SetTimer(), specifying a timer procedure and a 5,000 millisecond interval between timer ticks.
- In the timer procedure, call SetCaretBlinkTime(10000). This resets the timer in Windows that controls the caret blink.

When an application implements this procedure, Windows never removes the caret from the screen, and the caret does not blink.

2.21.12 How do I create a title window (splash screen)?

A "splash" screen (title screen) can be used by an application to display important information about a program. The splash screen can also be used as an activity indicator during application startup.

A splash screen is usually used when it is known that an application will take more than a second or two to display its first UI elements. The splash screen gives the user visual feedback that the application is starting. If a splash screen is not used, the user may assume that the start command was not run properly, and the user may try to start the application again.

To display a splash window, create an overlapped window. In the WndProc for this window, intercept the WM_NCCALCSIZE message and return NULL (instead of the DefWindowProc function), which prevents Microsoft Windows from redefining the window's maximum default full-screen client area size to accommodate borders and caption bar.

This window can be created before the main application initializes. The main application can send messages to this window to display its initialization information or a bitmap.

The following code shows one way to implement a splash screen window:

```
#define SZ_INIT    TEXT("Initializing application...")
#define SZ_LOAD    TEXT("Loading resources...       ")
#define SZ_CLOSE   TEXT("Closing splash window...")
#define SZ_SPLASH  TEXT("Splash window")

#define ID_TIMER_CLOSE0x1111
#define ID_TIMER_INIT0x1112
#define ID_TIMER_LOAD0x1113
#define ID_TIMER_DONE0x1114

ATOM MyRegisterClass(HINSTANCE hInstance);
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam);

HINSTANCE hInst = NULL;
TCHAR SplashWndClass[28];
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
                                  MSG msg;

lstrcpy(SplashWndClass,TEXT("SplashWindow"));
              MyRegisterClass(hInstance);
     // Perform application initialization:
```

```
     if (!InitInstance (hInstance, nCmdShow))
     {
          return FALSE;
     }

     // Main message loop:
     while (GetMessage(&msg, NULL, 0, 0))
     {
          TranslateMessage(&msg);
             DispatchMessage(&msg);
     }

     return msg.wParam;
}

//
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    This function and its use is only necessary if you want
//    this code to be compatible with Win32 systems prior to the
//    'RegisterClassEx' function that was added to Windows 95. It is
//    important to call this function so that the application will
//    get 'well formed' small icons associated with it.

ATOM MyRegisterClass(HINSTANCE hInstance)
{
     WNDCLASSEX wcex;

     wcex.cbSize             = sizeof(WNDCLASSEX);
     wcex.style              = CS_HREDRAW | CS_VREDRAW;
     wcex.lpfnWndProc        = (WNDPROC)WndProc;
     wcex.cbClsExtra         = 0;
     wcex.cbWndExtra         = 0;
     wcex.hInstance          = hInstance;
     wcex.hIcon              = NULL;
     wcex.hCursor            = LoadCursor(NULL, IDC_ARROW);
     wcex.hbrBackground      = (HBRUSH)(COLOR_WINDOW+1);
     wcex.lpszMenuName       = NULL;
     wcex.lpszClassName      = SplashWndClass;
     wcex.hIconSm            = NULL;

     return RegisterClassEx(&wcex);
}

//
//    FUNCTION: InitInstance(HANDLE, int)
//
//    PURPOSE: Saves instance handle and creates main window
//
//    COMMENTS:
//
//    In this function, we save the instance handle in a global
//    variable and create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
```

```
   HWND hWnd;
   RECT rect;
   int  splashwidth  = 350;
   int  splashheight = 350;

   hInst = hInstance; // Store instance handle in this global variable

   SystemParametersInfo(SPI_GETWORKAREA, 0, (LPVOID) &rect, 0);

   hWnd = CreateWindowEx(WS_EX_TOOLWINDOW,
                         SplashWndClass,
                         NULL,
                         WS_OVERLAPPED,
                         (rect.right - rect.left - splashwidth)/2,
                         (rect.bottom - rect.top - splashheight)/2,
                splashwidth,
                         splashheight,
                         NULL,
                         NULL,
                         hInstance,
                         NULL);
   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//
//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND- Process the application menu
//  WM_PAINT- Paint the main window
//  WM_DESTROY- Post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     switch (message)
     {
          case WM_NCCALCSIZE:  //CAPTURE THIS MESSAGE AND RETURN NULL
               return NULL;
          case WM_CREATE:
               SetTimer(hWnd, ID_TIMER_INIT, 1000, NULL);
               SetTimer(hWnd, ID_TIMER_LOAD, 2000, NULL);
               SetTimer(hWnd, ID_TIMER_DONE, 4000, NULL);
               SetTimer(hWnd, ID_TIMER_CLOSE, 5000, NULL);
               break;
             case WM_PAINT:
          {
               PAINTSTRUCT ps  = { 0 };
               RECT rc  = { 0 };
               HDC hDC = BeginPaint(hWnd, &ps);
```

```
                GetClientRect(hWnd, &rc);
                InflateRect(&rc, -2,-2);
                Rectangle(hDC, rc.left, rc.top, rc.right, rc.bottom);
                InflateRect(&rc, -15,-15);
                HFONT hFont = CreateFont(-35,-35, 0, 0,0,0,0,
                                0,0,0,0,0,0,TEXT("Arial"));
                HFONT hOldFont = (HFONT) SelectObject(hDC, hFont);
                DrawText(hDC, SZ_SPLASH, lstrlen(SZ_SPLASH),
                     &rc, DT_WORDBREAK);
                SelectObject(hDC, hOldFont);
                EndPaint(hWnd, &ps);
            }
            break;
               case WM_DESTROY:
                 PostQuitMessage(0);
                 break;
               case WM_TIMER:
               {
                    HDC hDC = GetDC(hWnd);
                    RECT rc = { 0 };
                    GetClientRect(hWnd, &rc);
                    KillTimer(hWnd, wParam);
                    switch (wParam)
                    {
                         case ID_TIMER_CLOSE:
                              DestroyWindow(hWnd);
                              break;
                         case ID_TIMER_INIT:
                              TextOut(hDC, rc.right-200, rc.bottom-20,
SZ_INIT, lstrlen(SZ_INIT));
                              break;
                         case ID_TIMER_LOAD:
                              TextOut(hDC, rc.right-200, rc.bottom-20,
SZ_LOAD, lstrlen(SZ_LOAD));
                              break;
                         case ID_TIMER_DONE:
                              TextOut(hDC, rc.right-200, rc.bottom-20,
SZ_CLOSE, lstrlen(SZ_CLOSE));
                              break;
                    }
                    ReleaseDC(hWnd, hDC);
               }
               break;
               default:
                 return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

2.21.13 How do I append text to an edit control?

Windows-based applications often use edit controls to display text. These applications sometimes need to append text to the end of an edit control instead of replacing the existing text. There are two different ways to do this in Windows:

Use the EM_SETSEL and EM_REPLACESEL messages.

Use the EM_SETSEL message with the clipboard functions to append text to the edit control's buffer.

The EM_SETSEL message can be used to place a selected range of text in a Windows edit control. If the starting and ending positions of the range are set to the same position, no selection is made and a caret can be placed at that position. To place a caret at the end of the text in a Windows edit control and set the focus to the edit control, do the following:

```
HWND hEdit = GetDlgItem (hDlg, ID_EDIT);
int ndx = GetWindowTextLength (hEdit);
SetFocus (hEdit);
SendMessage (hEdit, EM_SETSEL, (WPARAM)ndx, (LPARAM)ndx);
```

Once the caret is placed at end in the edit control, you can use the EM_REPLACESEL to append text to the edit control. An application sends an EM_REPLACESEL message to replace the current selection in an edit control with the text specified by the lpszReplace (lParam) parameter. Because there is no current selection, the replacement text is inserted at the current caret location. This example sets the selection to the end of the edit control and inserts the text in the buffer:

```
        SendMessage (hEdit, EM_SETSEL, (WPARAM)ndx, (LPARAM)ndx);
        SendMessage (hEdit, EM_REPLACESEL, 0, (LPARAM) ((LPSTR)
szBuffer));
```

Another way to insert text into an edit control is to use the Windows clipboard. If the application has the clipboard open or finds it convenient to open the clipboard, and copies the text into the clipboard, then it can send the WM_PASTE message to the edit control to append text. Of course, any data that was in the clipboard will be lost.

Before sending the WM_PASTE message, the caret must be placed at the end of the edit control text using the EM_SETSEL message. Below is "pseudo" code that shows how to implement this method:

```
OpenClipBoard () ;
EmptyClipBoard() ;
SetClipBoardData() ;

   SendMessage (hEdit, EM_SETSEL, (WPARAM)ndx, (LPARAM)ndx);
   SendMessage (hEdit, WM_PASTE,   0, 0L);
```

This "pseudo" code appends text to the end of the edit control. Note that the data in the clipboard must be in CF_TEXT format.

2.21.14 How do I spawn a process with redirected stdin and stdout?

The CreateProcess() API through the STARTUPINFO structure enables you to redirect the standard handles of a child console based process. If the dwFlags member is set to STARTF_USESTDHANDLES, then the following STARTUPINFO members specify the standard handles of the child console based process:

HANDLE hStdInput - Standard input handle of the child process.

HANDLE hStdOutput - Standard output handle of the child process.

HANDLE hStdError - Standard error handle of the child process.

You can set these handles to either a pipe handle, file handle, or any handle that can do synchronous reads and writes through the ReadFile() and WriteFile() API. The handles must be inheritable and the CreateProcess() API must specify that inheritable handles are to be inherited by the child process by specifying TRUE in the bInheritHandles parameter. If the parent process only wishes to redirect one or two standard handles, specifying GetStdHandle() for the specific handles causes the child to create the standard handle as it normally would without

redirection. For example, if the parent process only needs to redirect the standard output and error of the child process, then the hStdInput member of the STARTUPINFO structure is filled as follows:

```
hStdInput = GetStdHandle(STD_INPUT_HANDLE);
```

NOTE: Child processes that use such C run-time functions as printf() and fprintf() can behave poorly when redirected. The C run-time functions maintain separate IO buffers. When redirected, these buffers might not be flushed immediately after each IO call. As a result, the output to the redirection pipe of a printf() call or the input from a getch() call is not flushed immediately and delays, sometimes-infinite delays occur. This problem is avoided if the child process flushes the IO buffers after each call to a C run-time IO function. Only the child process can flush its C run-time IO buffers. A process can flush its C run-time IO buffers by calling the fflush() function.

NOTE: Windows 95 and Windows 98 require an extra step when you redirect the standard handles of certain child processes.

2.21.15 How to modify the width of the list of a combo box

The combo box in Windows is actually a combination of two or more controls; that's why it's called a "combo" box.

To make the combo box list wider or narrower, you need the handle of the list box control within the combo box. This task is difficult because the list box is actually a child of the desktop window (for CBS_DROPDOWN and CBS_DROPDOWNLIST styles). If it were a child of the ComboBox control, dropping down the list box would clip it to the parent, and it wouldn't display.

A combo box receives WM_CTLCOLOR messages for its component controls when they need to be painted. This allows the combo box to specify a color for these controls. The HIWORD of the lParam in this message is the type of the control. In case of the combo box, Windows sends it a WM_CTLCOLOR message with the HIWORD set to CTLCOLOR_LISTBOX when the list box control needs to be painted. The LOWORD of the lParam contains the handle of the list box control.

Once you obtain the handle to the list box control window, you can resize the control by using the MoveWindow API.

The following code sample demonstrates how to do this. This sample assumes that you have placed the combo box control in a dialog box.

```
LRESULT CALLBACK NewComboProc (HWND hWnd,   UINT message,   WPARAM
   wParam, LPARAM lParam ); // prototype for the combo box subclass proc
HANDLE hInst;                       // Current app instance.
BOOL bFirst;                        // A flag.

// Dialog procedure for the dialog containing the combo box.
BOOL CALLBACK DialogProc(HWND hDlg, UINT message, WPARAM
                                      wParam, LPARAM lParam)
{
     FARPROC lpfnNewComboProc;
     switch (message) {
          case WM_INITDIALOG:
               bFirst = TRUE; // Set flag here - see below for usage.
               // Subclass the combo box.
               lpfnOldComboProc = (FARPROC ) SetWindowLong (
                   GetDlgItem ( hDlg, IDC_COMBO1 ),
                   GWL_WNDPROC,
                   (LONG)NewComboProc );
```

```
            break;
            case WM_DESTROY:
                 (FARPROC ) SetWindowLong (GetDlgItem ( hDlg,
                             IDC_COMBO1 ),
                             GWL_WNDPROC,
                             (LONG)lpfnOldComboProc );
            break;
            default:
            break;
       }

       return FALSE;

  } // End dialog proc.

  // Combobox subclass proc.
  LRESULT CALLBACK NewComboProc (HWND hWnd,   UINT message,   WPARAM
                 wParam, LPARAM lParam );
  {
       static HWND hwndList;
       static RECT rectList;

       if ( WM_CTLCOLORLISTBOX == message ) // 32 bits has new message.
            {
            // Is this message for the list box control in the combo?
            // Do only the very first time, get the list
            // box handle and the list box rectangle.
            // Note the use of GetWindowRect, as the parent
            // of the list box is the desktop window

               if ( bFirst ) {
                  hwndList = (HWND) lParam ;     // HWND is 32 bits.
                  GetWindowRect ( hwndList, &rectList );
                             bFirst = FALSE;
               }
               // Resize listbox window cx by 50 (use your size here).
               MoveWindow ( hwndList, rectList.left, rectList.top,
               ( rectList.right - rectList.left + 50 ),
               rectList.bottom - rectList.top, TRUE );
        }
        // Call original combo box procedure to handle other combo messages.
        return CallWindowProc ( lpfnOldComboProc, hWnd, message,
                     wParam, lParam  );
     }
```

2.21.16 How do I modify environment variables permanently?

You can modify user environment variables by editing the following Registry key:

```
HKEY_CURRENT_USER\Environment
```

You can modify system environment variables by editing the following Registry key:

```
HKEY_LOCAL_MACHINE \
            SYSTEM \
 CurrentControlSet \
          Control \
   Session Manager \
       Environment
```

Note that any environment variable that needs to be expanded (for example, when you use %SYSTEM%) must be stored in the registry as a REG_EXPAND_SZ registry value. Any values of type REG_SZ will not be expanded when read from the registry.

Note that RegEdit.exe does not have a way to add REG_EXPAND_SZ. Use RegEdt32.exe when editing these values manually.

However, note that modifications to the environment variables do not result in immediate change. For example, if you start another Command Prompt after making the changes, the environment variables will reflect the previous (not the current) values. The changes do not take effect until you log off and then log back on.

To effect these changes without having to log off, broadcast a WM_SETTINGCHANGE message to all windows in the system, so that any interested applications (such as Program Manager, Task Manager, Control Panel, and so forth) can perform an update.

For example, on Windows NT, the following code fragment should propagate the changes to the environment variables used in the Command Prompt:

```
SendMessageTimeout(HWND_BROADCAST, WM_SETTINGCHANGE, 0,
 (LPARAM) "Environment", SMTO_ABORTIFHUNG,
 5000, &dwReturnValue);
```

None of the applications that ship with Windows 95, including Program Manager and the shell, respond to this message. Thus, while this article can technically be implemented on Windows 95, there is no effect except to notify third-party applications. The only method of changing global environment variables on Windows 95 is to modify the autoexec.bat file and reboot.

2.21.17 How do I add a menu item to the explorer right click menu?

When you press the right mouse button within the explorer window, there is a pop-up menu that shows you a series of options like “Explore”, “Open”, etc. To add an entry to this menu you just add your new menu item name to

```
[HKEY_CLASSES_ROOT\Directory\shell\YourItemName]
```

The value of this key should be a text string containing the menu item name.

Then, you add a subkey called “command” that specifies the command to be executed. To take a concrete and simple example: we want to start a command shell in the directory where the mouse is pointing to. We prepare a text file to add this keys using notepad or another text editor:

```
REGEDIT4
[HKEY_CLASSES_ROOT\Directory\shell\CmdHere]
@="CMD &Prompt Here"
[HKEY_CLASSES_ROOT\Directory\shell\CmdHere\command]
@="G:\\WINDOWS\\System32\\cmd.exe /k cd \"%1\""
```

We call the program “regedit” with the argument the file we just created and that is all. The two new keys will be automatically added to the registry. Note the double backward slashes! Obviously you should change the path to cmd.exe to suit your system configuration.[126]

126.I got the idea of this from http://www.kbcafe.com/articles/HowTo.Shell.pdf by by Randy Charles Morin.

2.21.18 How do I translate between dialog units and pixels?

When a application dynamically adds a child window to a dialog box, it may be necessary to align the new control with other controls that were defined in the dialog box's resource template in the RC file. Because the dialog box template defines the size and position of a control in dialog-box units rather than in screen units (pixels), the application must translate dialog-box units to screen units to align the new child window.

An application can use the following two methods to translate dialog- box units to screen units:

The MapDialogRect function provides the easier method. This function converts dialog-box units to screen units automatically.

For more details on this method, please see the documentation for the MapDialogRect function in the Microsoft Windows Software Development Kit (SDK).

Use the GetDialogBaseUnits function to retrieve the size of the dialog base units in pixels. A dialog unit in the x direction is one-fourth of the width that GetDialogBaseUnits returns. A dialog unit in the y direction is one-eighth of the height that the function returns.

For more details on this method, see the documentation for the GetDialogBaseUnits function in the Windows documentation of lcc-win32.

2.21.19 How do I translate between client coordinates to screen coordinates?

To determine the screen coordinates for the client area of a window, call the ClientToScreen function to translate the client coordinates returned by GetClientRect into screen coordinates. The following code demonstrates how to use the two functions together:

```
RECT rMyRect;

GetClientRect(hwnd, (LPRECT)&rMyRect);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.left);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.right);
```

2.21.20 When should I use critical sections and when is a mutex better?

Critical sections and mutexes provide synchronization that is very similar, except that critical sections can be used only by the threads of a single process. There are two areas to consider when choosing which method to use within a single process:

Speed. The Synchronization overview says the following about critical sections:

- Critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization.
- Critical sections use a processor-specific test and set instruction to determine mutual exclusion.

Deadlock. The Synchronization overview says the following about mutexes:

- If a thread terminates without releasing its ownership of a mutex object, the mutex is considered to be abandoned. A waiting thread can acquire ownership of an abandoned mutex, but the wait function's return value indicates that the mutex is abandoned.
- WaitForSingleObject() will return WAIT_ABANDONED for a mutex that has been abandoned. However, the resource that the mutex is protecting is left in an unknown state.
- There is no way to tell whether a critical section has been abandoned.

2.21.21 Why is my call to CreateFile failing when I use conin$ or conout$?

If you attempt to open a console input or output handle by calling the CreateFile() function with the special CONIN$ or CONOUT$ filenames, this call will return INVALID_HANDLE_VALUE if you do not use the proper sharing attributes for the fdwShareMode parameter in your CreateFile() call. Be sure to use FILE_SHARE_READ when opening "CONIN$" and FILE_SHARE_WRITE when opening "CONOUT$".

2.21.22 How to erase a file into the recycle bin?

When you erase a file within a program, the deleted file doesn't appear in the recycle bin. This can be annoying if you want to undo this operation of course.

The solution is to use the API SHFileOperation with the flag `FOF_ALLOWUNDO`. Here is a small program that does all this for you. Compile with -DTEST to have a standalone program that will accept command line arguments, or without it to have just the Recycle function that will erase a file to the recycle bin.[127]

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <direct.h>
#include <shellapi.h>
//////////////////
// Send a file to the recycle bin. Args:
//  - full pathname of file.
//  - bDelete: if TRUE, really delete file (no recycle bin)
//
int Recycle(LPCTSTR pszPath, BOOL bDelete)
{
   // Copy pathname to double-NULL-terminated string.
   //
   char buf[_MAX_PATH + 1]; // allow one more character
   SHFILEOPSTRUCT sh;
   strcpy(buf, pszPath);    // copy caller's path name
   buf[strlen(buf)+1]=0;    // need two NULLs at end

   // Set SHFILEOPSTRUCT params for delete operation
   //
```

127.This program is based on a similar program published in MSDN magazine 2001. In the comments for that program I found the following:
// If this code works, it was written by Paul DiLascia.
// If not, I don't know who wrote it.

```
   memset(&sh,0,sizeof(SHFILEOPSTRUCT));
   sh.fFlags |= FOF_SILENT;         // don't report progress
   sh.fFlags |= FOF_NOERRORUI;      // don't report errors
   sh.fFlags |= FOF_NOCONFIRMATION; // don't confirm delete
   sh.wFunc = FO_DELETE;            // REQUIRED: delete operation
   sh.pFrom = buf;                  // REQUIRED: which file(s)
   sh.pTo = NULL;                   // MUST be NULL
   if (bDelete) {                   // if delete requested..
      sh.fFlags &= ~FOF_ALLOWUNDO;  // ..don't use Recycle Bin
   } else {                         // otherwise..
      sh.fFlags |= FOF_ALLOWUNDO;   // ..send to Recycle Bin
   }
   return SHFileOperation(&sh);
}

#ifdef TEST
// Test program.
#include <gc.h>                      // for GC_malloc
#include <sys/stat.h>                // For stat()
// This list structure holds the names of the files given in
// the command line
typedef struct tagFileList {
     struct tagFileList *Next;
     char *Name;
} FILELIST;
// pre-declare functions
void    usage(void);
void    help(void);
char *GetCurrentDir();
char * MakeAbsolute(char *relname);
BOOL    confirm(LPCTSTR pFileName);
LPCTSTR GetErrorMsg(int err);

// global command-line switches
BOOL bPrompt=FALSE;        // prompt at each file
BOOL bQuiet=FALSE;         // don't display messages
BOOL bDisplayOnly=FALSE;   // display results only; don't erase
BOOL bZap=FALSE;           // delete (don't recycle)

// test if file exists
int fileexists(LPCTSTR pFilename)
{
   struct stat st;
   return stat(pFilename, &st)==0;
}

// Adds to the list of file a new name. Allocates memory with
// the garbage collector
FILELIST *AddToList(FILELIST *start,char *name)
{
     FILELIST *newlist;

     newlist = GC_malloc(sizeof(FILELIST));
     newlist->Name = GC_malloc(strlen(name)+1);
     strcpy(newlist->Name,name);
     if (start) {
          newlist->Next = start;
     }
     return newlist;
}
```

```
int main(int argc, TCHAR* argv[], TCHAR* envp[])
{
   // Parse command line, building list of file names.
   // Switches can come in any order.
   //
   FILELIST * files = NULL;

   for (int i=1; i<argc; i++) {
      if (argv[i][0] == '/' || argv[i][0] == '-') {
         switch(tolower(argv[i][1])) {
         case 'q':
            bQuiet=TRUE;
            break;
         case 'n':
            bDisplayOnly=TRUE;
            break;
         case 'p':
            bPrompt=TRUE;
            break;
         case 'z':
            bZap=TRUE;
            break;
         case '?':
            help();
            return 0;
         default:
            usage();
            return 0;
         }
      } else {
         // Got a file name. Make it absolute and add to list.
         files = AddToList(files,MakeAbsolute(argv[i]));
      }
   }

   if (files == NULL) {
      // No files specified: tell bozo user how to use this command.
      usage();
      return 0;
   }

   // Delete (recycle) all the files in the list
   int nDel=0;

   // loop over list of files and recycle each one
   for (; files; files = files->Next) {
      // Only recycle if file exists.
      if (fileexists(files->Name)) {

         if (!bQuiet && !bPrompt) {
            // tell user I'm recycling this file
            fprintf(stderr,"%s %s\n",
               bZap ? "Deleting" : "Recycling", files->Name);
         }

         if (!bDisplayOnly) {
            if (!bPrompt || confirm(files->Name)) {
               // Finally! Recycle the file. Use CRecycleFile.
               int err = Recycle(files->Name,bZap);
```

```
                if (err==0) {
                   nDel++;
                } else {
                   // Can't recycle: display error message
                   fprintf(stderr,"Error %d: %s", err,
                                 GetErrorMsg(err));
                }
             }
          }
       } else {
          fprintf(stderr,"File not found \"%s\"\n", files->Name);
       }
   }
   if (!bQuiet) {
      fprintf(stderr,"%d files recycled\n",nDel);
   }
   return 0;
}

void usage(void)
{
   printf("Usage: RECYCLE [/QNPZ?] file...\n");
}

void help(void)
{
   printf("Purpose:  Send one or more files to the recycle bin.\n");
   printf("Format:   RECYCLE [/Q /N /P /Z] file....\n");
   printf("          /Q(uiet)     no messages\n");
   printf("          /N(othing)   don't delete, just show files\n");
   printf("          /P(rompt)    confirm each file\n");
   printf("          /Z(ap)       really deletesame as del\n");
}

// Make a file name absolute with respect to current directory.
char *MakeAbsolute(char *relname)
{
   // Get current directory.
   char *cwd = GetCurrentDir();
   char *absname;

   if (relname[0] && relname[1] && relname[1]==':') {
      // relname is already absolute
      absname = relname;

   }
   else if (relname[0]=='\\') {
   // relname begins with \  add drive letter and colon
      memmove(relname+2,relname,strlen(relname)+1);
      relname[0] = cwd[0];
      relname[1] = cwd[1];
      absname = relname;

   } else {  // file name begins with letter:
      // relname begins with a letter  prepend cwd
     strcat(cwd,relname);
        absname = cwd;
   }
   return absname;
}
```

```
//////////////////
// Get current directory. For some reason unknown to mankind, getcwd
// returns "C:\FOO" (no \ at end) if dir is NOT root; yet it returns "C:\"
// (with \) if cwd is root. Go figure. To make the result consistent for
// appending a file name, GetCurrentDir adds the missing \ if needed.
// Result always has final \.
//////////////////
char *GetCurrentDir(void)
{
   static char dir[MAX_PATH];
   getcwd(dir, sizeof(dir));

   // Append '\' if needed
   int lastchar = strlen(dir)-1;
   if (lastchar>0 && dir[lastchar] != '\\') // if last char isn't \ ..
      strcat(dir,"\\");                 // ..add one

   return dir;
}

//////////////////
// Get user confirmation to recycle/delete a file
//////////////////
BOOL confirm(LPCTSTR pFileName)
{
   while (TRUE) {
      printf("Recycle %s (Y/N/All)? ", pFileName);
      char c = getch();
      if (c==' ') {
         printf("^C\n");
         exit(0);
      }
      printf("\n");
      switch (tolower(c)) {
      case 'a':
         bPrompt=FALSE;
         // fall through
      case 'y':
         return TRUE;
      case 'n':
         return FALSE;
      }
   }
}

//////////////////
// Get Windows system error message
//////////////////
LPCTSTR GetErrorMsg(int err)
{
   static char buf[BUFSIZ];
   buf[0]=0;

  // Only Windows could have a function this confusing to get a simple
  // error message.
   FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS,
      NULL,    // source
      err,     // error code
```

```
        0,       // language ID
        buf,     // buffer to receive message
        BUFSIZ, // size of buf
        NULL);   // arguments
    return buf;
}
#endif
```

2.22 Finding more examples and source code

John Finlay has created a web site for source code for lcc-win32. Here is what John writes about his work:

When I initially started using LCC-Win32 searching the Internet revealed no good sources of material that one could download as examples, so creating a web site dedicated to this end seemed a good idea. If I needed help obviously others would too.

The web site http://www.btinternet.com/~john.findlay1/ contains many examples that lcc-win32 users can easily compile and learn from. I have tried to place each example in an appropriate section to help in locating as the number of examples would be bewildering if there were no discrete departments. These 'departments/sections' however are not absolute as examples do not always readily fit into any specific category.

The largest section is 'Windows Programming' where you will find examples ranging from a simple 'Generic' application to far more sophisticated software. There are many other sections that may be of interest - OpenGL, DirectX, Windows Common Controls, Custom Controls, MultiMedia, Telephony, Games, etc. There is also the 'Assembler' section with texts and examples explaining and showing how to code in assembler using lcc-win32 specifically as the asm syntax used is AT&T not Intel as is with most other compilers.

2.23 Overview of lcc-win32's documentation

The documentation of lcc-win32 comes in four files:

9) **Manual.chm**. This is the user's manual, where you will find information about how the system is used, command line options, menu descriptions, how to use the debugger, etc. It explains how to build a project, how to setup the compiler, each compiler option, all that with all the details.

10) **c-library.chm**. This file contains the documentation for all functions in the C runtime library. It will be automatically invoked by the IDE when you press the F1 function key. This two files (c-library.chm and manual.chm) are distributed in "manual.exe".

11) **Lcc-win32.doc**. This is a technical description for interested users that may want to know how the system is built, how the programs that build it were designed, the options I had when writing them, etc.

The documentation of the windows API is distributed in a relatively large file called **win32hlp.exe**. This is absolutely essential, unless you know it by heart… When installed, this file will become Win32.hlp. That file is not complete however. More documentation for the new features of Win32 can be found in the **win32apidoc.exe** file, also in the lcc distribution site. When installed, that file will install:

- Shelldoc.doc. This documents the windows shell, its interfaces, function definitions, etc.
- Wininet.doc. This documents the TCP/IP subsystem for network programming.

- CommonControls.doc. This explains the new controls added to windows after 1995.

Note that Wedit will detect if the documentation is installed, and will allow you to see the documentation of any function just with pressing the F1 key. This is a nice feature, especially for beginners. Install a full version if you aren't an expert. A version without the documentation it is a pain, since you have to go fishing for that information each time you want to call an API, not a very exciting perspective.

But if you want to get serious about windows programming, you should download the Microsoft Software Development Kit (SDK) from the msdn site, and install it in your machine. Wedit will automatically recognize the msdn library or the SDK if installed and will call them instead of using the win32.hlp file.

2.24 Bibliography

Here are some books about C. I recommend you to read them before you believe what I say about them.

«C Unleashed»

Richard Heathfield, Lawrence Kirby et al.

Heavy duty book full of interesting stuff like structures, matrix arithmetic, genetic algorithms and many more. The basics are covered too, with lists, queues, double linked lists, stacks, etc.

«Algorithms in C»

Robert Sedgewick.

I have only the part 5, graph algorithms. For that part (that covers DAGs and many others) I can say that this is a no-nonsense book, full of useful algorithms. The code is clear and well presented.

«C a reference manual»

(Fifth edition) Samuel P Harbison Guy L Steele Jr.

If you are a professional that wants to get all the C language described in great detail this book is for you. It covers the whole grammar and the standard library with each part of it described in detail.

«The C programming language»

Brian W Kernighan, Dennis Ritchie. (second edition)

This was the first book about C that I got, and it is still a good read. With many exercises, it is this tutorial in a better rendering...

«A retargetable C compiler: design and implementation»

Chris Fraser and Dave Hanson

This book got me started in this adventure. It is a book about compiler construction and not really about the C language but if you are interested in knowing how lcc-win32 works this is surely the place to start.

"C interfaces and implementations"

David R. Hanson

This is an excellent book about many subjects, like multiple precision arithmetic, lists, sets, exception handling, and many others. The implementation is in straight C and will compile without any problems in lcc-win32.

“Safer C”

Les Hatton

As we have seen in the section «Pitfalls of the C language», C is quite ridden with problems. This book address how to avoid this problems and design and develop you work to avoid getting bitten by them.

“Programming Windows”

Charles Petzold

Microsoft Press

This is a very easy to read introduction to the windows API that covers in depth all aspects of windows programming. You can download an electronic version of this book at the site of the author: www.cpetzold.com.

“Windows Network Programming”

Ralph Davis

Addison Wesley

This is a very extensive introduction to this side of programming. If you want an in-depth coverage of sockets, net-bios, etc etc, here you will find it.

C Programming FAQs

Steve Summit

C Programming FAQs contains more than 400 frequently asked questions about C, accompanied by definitive answers. Some of them are distributed with lcc-win32 but the book is more complete and up-to-date.

The Standard C Library

P.J. Plauger.

This book shows you an implementation (with the source code) of the standard C library done by somebody that is in the standards committee, and knows what he is speaking about. One of the best ways of learning C is to read C. This will give you a lot of examples of well written C, and show you how a big project can be structured.

The C Standard

John Wiley and Sons.

This is the reference book for the language. It contains the complete C standard and the rationale, explaining some fine points of the standard.