A quick introduction to the C language

By

Jacob Navia

Table of Contents

Introduction

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 features. The compiler will not do

[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.

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.

Organization of C programs

A program in C 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] Functions are organized in modules clustered together in source files. 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 definitions using the #include 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.

[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.

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.

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 library 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.

[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”

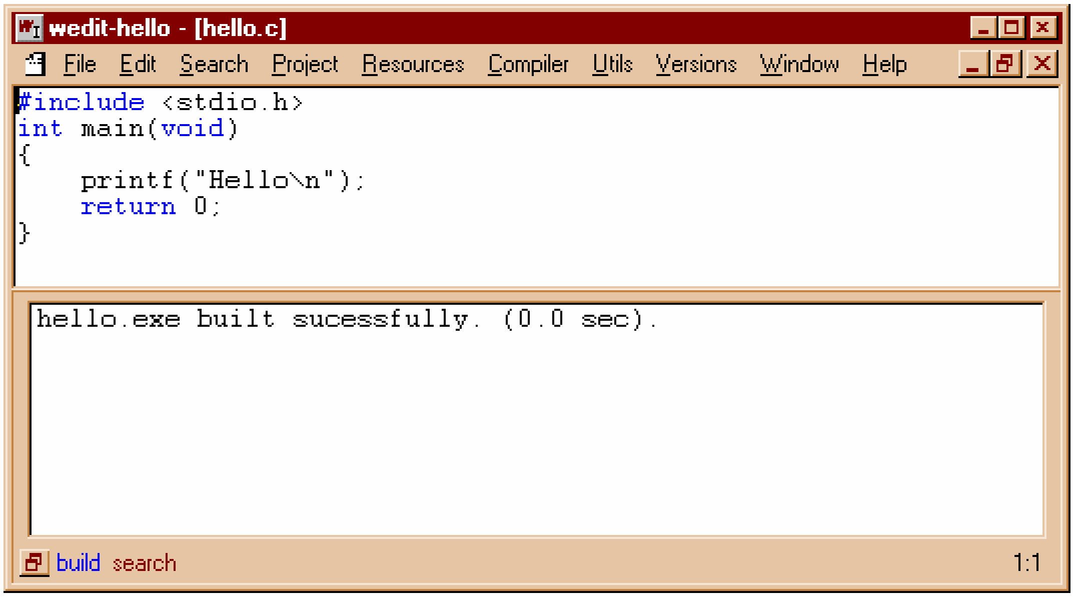
[14] This is one of the two possible definitions of the “main” function. Later we will see the other one.

[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.

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.

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]


wedit-hello - [hello.c]
File Edit Search Project Resources Compiler Utils Versions Window Help
build search
1:1

```
#include <stdio.h>
int main(void)
{
    printf("Hello\n");
    return 0;
}
```

```
hello.exe built sucessfully. (0.0 sec).
```

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

[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

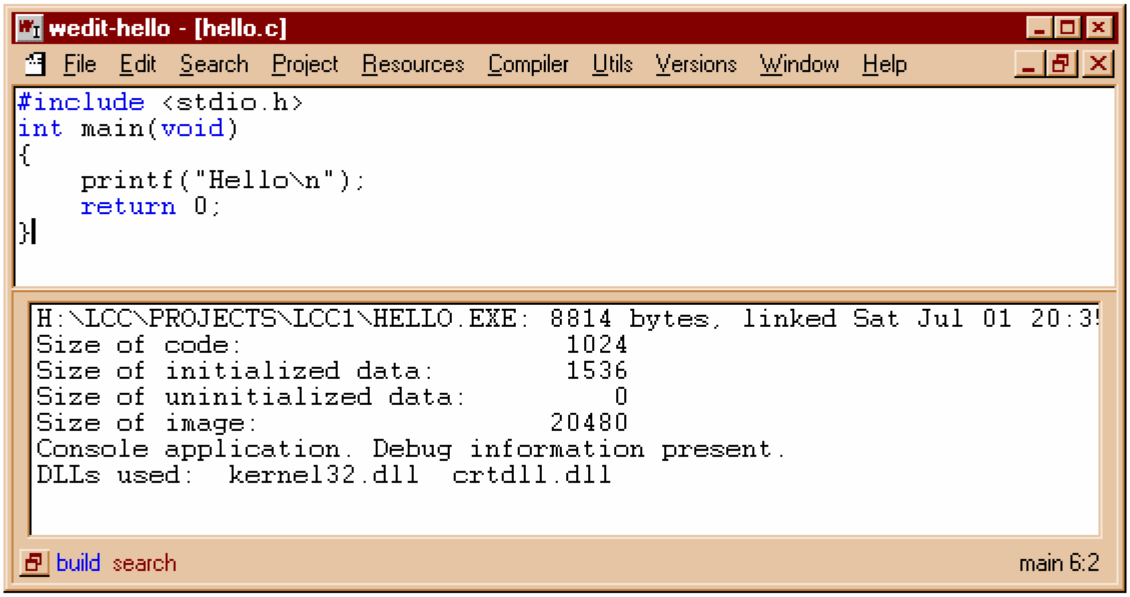
[17] If this doesn’t work or you receive warnings, you have an installation problem (unless you made a typing mistake). Or maybe I have a bug. When writing mail to me do not send messages like: “It doesn’t work”. Those messages are a nuisance since I can’t possibly know what is wrong if you do not tell 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 executables, 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.

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.



```
#include <stdio.h>
int main(void)
{
    printf("Hello\n");
    return 0;
}
```

```
H:\LCC\PROJECTS\LCC1\HELLO.EXE: 8814 bytes, linked Sat Jul 01 20:3
Size of code:                    1024
Size of initialized data:        1536
Size of uninitialized data:         0
Size of image:                  20480
Console application. Debug information present.
DLLs used:  kernel32.dll  crtdll.dll
```

We see at the first line of the bottom panel, that the program executable is called:

```
h:\lcc\projects\hello.exe
```

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:\>
```

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.

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.

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,[18] 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.[19]

[18] This has nothing to do with object oriented programming of course!

[19] 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 front end of the compiler proper processes the resulting text. Its task is to generate a series of intermediate code statements.[20] The code generator that emits assembler instructions from it processes these.[21]
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.

Summary of section Hello.

- The #include pre-processor directive allow us to include standard headers[22] in our programs to use their definitions.
- C programs begin in a function called "main"
- A function is a set of C statement that performs a single task.
- Functions can receive arguments and can return a result.

The basic data types of the C language

The C language supposes a set of data representations (types), which are available to the programmer without he/she having to define anything. These are the primitive types. They are in essence all numbers.

Numbers can be represented as integers, floating point or complex, corresponding to the set of integers, reals or complex numbers in math. Of course we are speaking here of numbers as they can fit in a computer, so obviously any infinities are excluded.

Integers:

The integers come in different lengths:

- char
- short
- int
- long
- long long

All this integer types can be either signed (the most significant bit is used for the sign) or unsigned (the number is always a positive number).

- **Char** type: it is a small integer that is often represented in 8 bits. It must be big enough to contain all characters of the character set used in the

[20] 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.
[21] Assembly code can be generated with the `lcc-S hello.c` command, and the generated assembly file will be called hello.asm.
[22] We will see that #include applies to user defined include headers too.

language.[23] Under lcc-win32 the signed chars go from –128 to +127, and the unsigned from 0 to 255. This set of integers is used to represent characters or plain text in character arrays called "strings". The character string "Hello" is made of the integers 72 (H) 101 (e) 108 (l) 108 (l) 111 (o). The integer used is chosen according to the ASCII character set. Strings are finished with a non-existing character, zero, that signals the end of the string.

- **Short** type. It is an integer that has more bits than char, and is often represented with 16 bits. Under lcc-win32 signed shorts go from `–32768` to `32767`, and unsigned shorts from `0` to `65535`. Shorts are used to represent characters sometimes, when you use the UNICODE character set.
- **Int** type: It is an integer that is represented in the native bus width of the machine. In the case of a Pentium machine, the width of the bus is 32 (it is a 32 bit processor) and the "int" type is represented in 32 bits. Other machines have different widths (under a 80286 processor the native width was of 16 bits). Under lcc-win32 signed integers go from `–2147483648` to `2147483647`, and unsigned int from `0` to `4294967295`.
- **long** type: It is a type that must be longer than a short, and is very often represented with the same length as an int. This is the case with lcc-win32: long and int are the same.
- **long long** type: It is a type that should be longer than long, and in the case of lcc-win32 has a 64 bit width.[24]

Floating point types:

These are numbers represented in a special format called "floating point". They are:

- float type: Uses 32 bits to represent all real numbers from approximately `1.1754943e-38` to `3.402823e+38`. The precision is 6 digits only.
- double type: Uses 64 bits to represent all real numbers from approximately `2.22507e-308` to `1.79769e+308`. The precision is 15 digits.
- long double type: Uses more than 64 bits to represent all real numbers. Under lcc-win32 they go from approximately `3.36210e-4932` to `1.18973e+4932`. The precision is 18 digits.[25]
- Lcc-win32 implements an extension of floating point types called "qfloat", that has a precision of 100 digits. It uses a 352 bit representation.

[23] All characters are represented within the machine as numbers. For instance the character 'A' is represented using the ASCII convention as 65.

[24] I will put the range of long longs in a footnote to avoid scaring the casual readers away with such a huge number. The signed long longs go from `–9223372036854775808` to `9223372036854775807`, the unsigned from `0` to `18446744073709551615`.

[25] Not all compilers implement the long double type. Specifically, the compiler of Microsoft Corp "MSVC" represents long double with the same width as double. The long long type is called __int64 within that system, instead of its standard name.

Logical types

The _Bool type can contain either 0 or 1. Within lcc-win32 this numbers are stored in 8 bits instead of only one since the machine can't address bits individually.

Complex Types

There are three complex types: float _Complex, double _Complex and long double _Complex. This numbers are composed of a pair of values stored in the precision of float, double, or long double.

Derived types

From these primitive types you can derive any combination of types by:

- Building an **array**, i.e. you store several units of a basic type one after the other in memory.
- Building a **structure**, i.e. you store a set of values of different types in a contiguous part of memory and use them as a unit.
- Building a **union**, i.e. you store a set of different types in the same memory locations, always using one of them at a time.
- Building a **function** type, i.e. you describe a set of arguments and a result from this arguments
- Building a **pointer** type, i.e. you represent a value by its machine address.

All this is discussed later in greater detail. This is intended just to give you a quick overview of the available primitive types.

An overview of the standard libraries

The documentation of lcc-win32, specifically the user manual, has the exhaustive list of all libraries provided by the system. The on-line help system has a detailed description of each one of those. It would be a waste to repeat all that detail here, so only a broad outline will be given:[26]

Header	**Purpose**
stdio.h	Standard input and output. Here you will find the definitions of the FILE structure, used for all input and output, the definitions of all functions that deal with file opening, closing, etc. The famous printf function is defined here too, together with sprintf, fprintf, and all the related family of functions.
math.h	Mathematical functions. sin,cos,atan, log, exp, etc. We have here • Trigonometry (sin, cos, tan, atan, etc). • Rounding (ceil, floor) • logarithms (log, exp, log10, etc) • Square and cubic roots (sqrt, cbrt) • Useful constants like pi, e, etc.

[26] 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.

stdlib.h	Standard library functions. We have here: ● abort (abnormal termination of the program) ● exit (normal termination) ● atoi, itoa (text to integer conversion) ● malloc,calloc,free (dynamic memory module) ● rand, srand (random numbers) ● putenv, getenv (environment management) ● qsort (sorting) ● strtod, strtol (conversion from string to double/long) ● sleep (suspend execution for a certain period of time)
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. We have: • strcmp (compare strings) • strlen (get the length of a string) • strcpy (copy a string into another) • strcat (concatenate a string to another) • strstr (find a substring in a string) • memset (set a RAM region to a character) • memcpy (copy memory) • memmove (copy memory taking caring of overlapping region)
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.

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.[27]

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

[27] 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 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:
    - int argc. It contains the number of arguments passed to the program plus one.
    - char *argv[] This is an array of pointers to characters[28] 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”.
1. 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”.[29]
2. 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

[28] 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.

[29] 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];

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. When this is no longer the case, we stop.
- 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. The counter of the loop starts at zero and goes up to one less than the size of the array, since we number each position from zero up. In C, array indexes for an array of size n elements always start at zero and runs until n-1.[30]

3. We use again printf to print something in the screen. This time, we pass to printf the following arguments:
   ```
   "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.

4. We finish the scope of the for statement with a closing brace. This means, the iteration definition ends here.
5. The return 0 statement, finishes the execution of the main function, and therefore exits to the operating system returning zero to the environment, as the result of the execution.

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:

[30] 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.

```
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:[31]

```
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.

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”.

The **“for”** construct has

- an initialization part, i.e. code that will be always executed before the loop begins,

[31] 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.

- 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
- 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
     }
```

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
     }
```

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.

The "continue" and "break" keywords apply to the innermost loop only. What happens if you want to exit a loop in a higher level?

Consider this code:

```
for (i=0; i<100;i++) {
     for (j=0; j<10;j++) {
          if (condition)
               // exit the loop for i somehow
     }
}
```

This can be accomplished by the following code:

```
for (i=0; i<100;i++) {
     for (j=0; j<10;j++) {
          if (condition)
               goto label;
     }
}
label:
```

The "**goto**" instruction accepts as argument a label, and continues execution at that label.

Many things have been said about the "goto", whether is considered harmful or not, whether it should be in the language or not. For instance the above example could have been written like this:

```
for (i=0; i<100;i++) {
     for (j=0; j<10 && condition;j++) {
          // some code here
     }
     if (condition)
          break;
}
```

This re-arrangement provokes a test in the "test" part of the second for statement, and provokes the exit from the innermost loop. Then, we test the condition again, and if it is true, we issue a break again.

Quite complicated just to avoid a goto statement.

Do not do this. If you see that your code is clearer with a goto statement, don't be impressed by whatever the others say and just do it!

The goto statement is a part of the C language and lcc-win32 will compile it without any problems.

Note too that the usage of labels is restricted in C, much more restricted than in Fortran for instance. You can't take the address of a label, or put a lot of labels in a table, like in Fortran.

Labels are written with an identifier followed by ":", anywhere in the body of a function. They do not have to be declared before they are used.

Summary.

- Functions receive arguments and return a result in the general case. The type of the result is declared before its name in a function declaration or definition.
- The “main” function can receive arguments from its calling environment.
- We have to declare the type of each identifier to the compiler before we can use it.

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 an 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.[32]

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

[32] 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.

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.[33]

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

[33] 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 = (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:

```
int *a;
```

This means that the pointer "a" will contain the machine address of some unspecified integer. [34]

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 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"[35] 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".

[34] 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".

[35] Deep C secrets. Peter van der Linden ISBN 0-13-177429-8

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 rule 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:

```
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.

Function declaration

A declaration of a function specifies:

- The return type of the function
- Its name
- The types of each argument

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”.[36]

Why function declarations are 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 debuggers 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

[36] The interface for using functions with a variable number of arguments is described in the standard header file “stdarg.h”. See too functions with variable number of arguments

introduce that into lcc-win32, because ignoring the function prototype is always an error. But, for compatibility reasons I haven't done it yet.[37]

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;
}
```

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
```

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 `)'
```

[37] 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.

```
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.[38]

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.

[38] 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…

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 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 of Errors

- 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.

Reading from a file

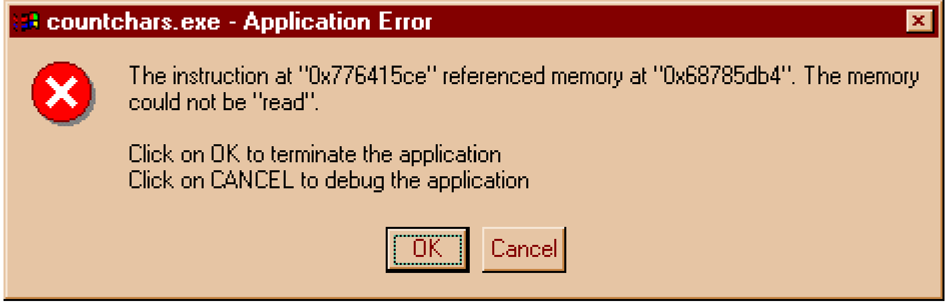
For a beginner, it is very important that the basic libraries for reading and writing to a stream, and the mathematical functions are well known. Here is an example of a function that will read a text file, counting the number of characters that appear in the file.

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. The main arguments will not be explained again.
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.[39]
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.

[39] 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.


countchars.exe - Application Error

The instruction at "0x776415ce" referenced memory at "0x68785db4". The memory could not be "read".

Click on OK to terminate the application
Click on CANCEL to debug the application

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
```

We get the following box that many of you have already seen several times:[40]

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;
```

[40] This is the display under Windows NT. In other systems like Linux for instance, you will get a “Bus error” message.

```
     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>
```

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
```

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?

Well 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, several 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 for people that were used to write their C programs expecting only ONE char as line separator, so 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], “r**b**“);, 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".

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:

1. 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.
2. 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.
3. At the beginning of each procedure, try to add a standard comment describing the purpose of the procedure, inputs/outputs, error handling etc.[41]
4. At the beginning of each module try to put a general comment describing what this module does, the main functions etc.

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

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.

Summary of Reading from a file.

- 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.

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 dictionary 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.

Statements

Expression	Meaning and value of result	Example
Identifier	The value associated with that identifier. (See Identifiers)	id
constant	The value defined with this constant (See Constants).	
	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}
Array [index]	Access the position “index” of the given array. Indexes start at zero (See array)	Table[45]
Array[i1][i2]	Access the n dimensional array using the indexes i1, i2, … *in*	Table[34][23] This access the 35th line, 24th position of Table
fn (args)	Call the function “fn” and pass it the comma separated argument list “args”.(Function Call)	printf(“%d“,5)
fn (arg, ...)	See function with variable number of arguments	
(*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[42] the value of the right hand side of the equals sign. (Assignment)	a = 45
expression++	Equivalent to expression = expression + 1. Increment expression after using its value. (See Postfix).	a = i++
expression--	Equivalent to expression = expression – 1. Decrement expression after using its value. (See Postfix).	a = i–
++expression	Equivalent to expression = expression+1. Increment expression before using its value. (see Postfix)	a = ++I
--expression	Equivalent to Expression = expression – 1. Decrement expression before using it. (See Postfix)	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.	*pData
- expression	Subtract expression from zero, i.e. change the	-a

[42] 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”.

	sign.	
~ 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.	sizeof(a)
(type) expr	Change the type of expression to the given type. This is called “cast”.	(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.	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	a&8
expr1 ^ expr2	Bitwise XOR expr1 with expr2	a^b
expr1 | expr2	Bitwise OR expr1 with expr2	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.	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.	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.	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	a=7,b=8 The result of this is 8

Declarations[43]

Expression	Meaning	Example
type identifier;	Identifier will have the specified type within this scope. See declarations.	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 a reference to a single object of the given type. References must be initialized immediately after their declaration. See[44]	int &ra = a;
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.	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.	register int f;
extern type id	The definition of the identifier is in another module.	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;
		};

[43] Lcc-win32 doesn't yet implement the keyword restrict.

[44] This is an extension of lcc-win32 and not part of the C standard. It is widely used in other languages like C++.

type id:n	Within a structure field declaration, declare “id” as a sequence of n bits of type “type”. See bit-fields	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 Unions.	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.	enum color {
		red,green,blue
		};

const type id	Declare that the given identifier can’t be changed (assigned to) within this scope. See Const.	const int a;
unsigned int-type	When applied to integer types do not use the sign bit. See unsigned.	unsigned char a = 178;
volatile type identifier	Declare that the given object changes in ways unknown to the implementation.	volatile int hardware_clock;
type id(arg1, arg2,…)	Declare the prototype for the given function. See prototypes.	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> See Function declarations.	int add1(int x)
		{ return x+1;}

operator opname (args)
{
}
	Redefine one of the operators like +, * or others so that instead of doing the normal operation, this function is called instead. This is an extension to the C language proposed by lcc-win32 and is NOT in the language standard.	operator +(Cmp a,Cmp b) {
		… statements …
		}

inline	This is a qualifier that applies to functions. If present, it can be understood by the compiler as a specification to generate the fastet function call possible, generally by means of replicating the function body at each call site.	int inline foo(a);

Pre-processor.

// commentary	Double slashes introduce comments up to the end of the line. See Comments.	// comment
/*commentary */	Slash star introduces a commentary until the sequence star slash */ is seen. See Comments.	/* comment */
#define id expr	Replace all appearances of the given identifier by the corresponding expression. See preprocessor.	#define TAX 6
#define	Define a macro with n arguments. When used,	#define max(a,b)

mac(a,b)	the arguments are lexically replaced within the macro. See preprocessor	((a)<(b)? (b):(a))[45]
#undef id	Erase from the pre-processor tables the identifier.	#undef TAX
#include <header>	Insert the contents of the given file from the standard include directory into the program text at this position	#include <stdio.h>
#include “header“	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 preprocessor.	#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
#line nn	Set the line number to nn	#line 56
#file “foo.c”	Set the file name	#file “ff.c”
#warning "msg"	Show the indicated warning to the user. This an extension of lcc-win32	#warning "hello"
#error errmsg	Show the indicated error to the user	
#pragma instructions	Special compiler directives[46]	#pragma optimize(off)
_Pragma(str)	Special compiler directives. This is a C99 feature.	_Pragma(“optimize (off)”);
__LINE__	Replace this token by the current line number	
__FILE__	Replace this token by the current file name	
__ 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__

Control-flow

if (expression) {
}
else {
}
	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.

[45] The parentheses ensure the correct evaluation of the macro.

[46] The different pragma directives of lcc-win32 are explained in the user’s manual.

while (expression) {
… statements …
}
	If the given expression evaluates to something different than zero, execute the statements in the block. Else skip them.

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.

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.

switch (expression) {
case int-expr:
… statements
…
break;
case int-expr:
… etc …
 	
default:
… statements
…
}
	Evaluate the given expression. Use the resulting value to test if it matches any of the integer expressions defined in 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.
	
	For more details see switch.

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.
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.

Windows specific syntax

_stdcall	Use the stdcall calling convention for this function or function pointer: the called function cleans up the stack. See stdcall.	int _stdcall foo(void);
__declspec(**dllexport)**	Export this identifier in a DLL to make it visible from outside.	int __declspec(dllexport) foo(int);
__declspec**(** **dllimport**)	Import this identifier from a Dll. Note that this is needed only in data items. Functions do not need this qualifier.	
__declspec(**naked**)	Do not generate any prologue/epilogue sequence for this function. This is mainly used to write your own functions in assembly language.	
WINVER	Replace by the version number of the version of windows. This is a #defined symbol containing two bytes: the upper one is the	#if WINVER >=5 #endif

	major version, the lower one is the minor version.	
WIN32	#defined as 1	
_WIN32	#defined as 1	
__int64	#defined as long long for compatibility reasons with Microsoft's compiler.	__int64 big;

Lcc-win32 extensions[47]

Type **operator** <name>(args…)	**Operator overloading**: redefines the operator <name> for the given type. The placeholder <name> can be one of '+', '-', etc.	Type **operator**+(Type a,Type b);
Type **overloaded** fn(args…);	**Generic functions**: Allows the definition of several functions with the same name that are choosen by the compiler in function of the actual arguments being passed to them.	qfloat overloaded sqrt(qfloat);
Type &id = …;	**References:** References are pointers that must be initialized when they are declared and can't be changed afterwards.	qfloat &a = b; /* a contains the address of the qfloat structure "b" */

[47] This extensions are quite similar to the syntax used in the C++ language. The semantics change of course, since lcc-win32 remains a C compiler, not a C++ compiler.

- **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.

- **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:

`1645L`	(long)
`0xF98A`	(hexa)
`2634455LL`	(long long)
`5488UL`	(unsigned long)

  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.

  Long double constants are suffixed by an uppercase 'L', and qfloat constants use a 'Q'.

  ```
  float f = 21443f;
  long double ld = 5776.e678L;
  qfloat qf = 6644.8877e6534Q;
  ```

  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”
  ```

  To include a double quote in 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 (decimal)**

\n	New line	10
\r	Carriage return	12
\b	Backspace	8
\v	Vertical tab	11
\t	Tabulation	9
\f	Form feed	12
\e	Escape	27
\a	Bell	7

- **Array** syntax. 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
  ```

- **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.

- **Functions with variable number of arguments.**

  To access the unnamed, extra arguments you should include the <stdarg.h> include file. To access the additional arguments, you should execute the `va_start`, then, for each argument, you should 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);
```

- **Assignment**. An assignment consists of 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
                          // function pointer. Note that is not necessary to
                          // prefix the pointer with an asterisk to
                          // dereference it.
  ```

  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.
- **Postfix** 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 be 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 235th 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.

- **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.
  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.
- **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 en empty pointer (NULL), the pointer bb will receive the “next” field of the structure, otherwise bb will be set to empty.
- **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. For a detailed description of this keyword see structures .
- **Unions**. 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 union definition
- **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 do not introduce new types — they introduce new names for *existing* types.
- **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 recommendations, but it is better not to use this declaration and leave the register usage to the compiler.

- **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.
- **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};`
- **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.
- **variable lenth 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 when this function is entered.

- **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.

- **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.
- **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".
- **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. This functions will be “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 be 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.
- **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 “****”

- **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 allow 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.

- **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`”.

- **__declspec**

  This non-standard declaration (borrowed from the Microsoft compiler) introduces a feature of an identifier. Lcc-win32 supports several of them**:**

  1. **dllexport**. This instructs the compiler to emit a record for the linker that will create an executable that has a list of exported symbols where this symbol will be included. Normally this is used only in DLLs. This construct refers always to a public symbol. For example: `int __declspec(dllexport) foo(int a);` This construct declares that the symbol "foo" should be included in the exports table of the DLL being build, and can be seen by other programs.
  2. **dllimport**. This instructs the compiler to generate code for a variable that is defined in a DLL. This should be used only with *data* variables, not for functions.[48]

[48] Why this difference?
Imports dereference an address that is written to the executable by the program loader when the program starts. This is not necessary for functions since they are always dereferenced pointers. Only for data variables is this extra declaration needed then.

3. **naked**. This instructs the compiler to strip down the function to just the code you have written without any local frame. This means that this functions can't have any local variables, and normally should be written in pure assembly.

Precedence of the different operators.

Precedence	Operator
1	Parenthesis and brackets **`( ) [ ]`**
2	Structure access. Point (`.`) or indirection (**`->`**)
3	Multiply, Divide, and modulus (**`* / %)`**
4	Add and subtract **(`+ -`**)
5	Shift. (**<<** or **>>**)
6	Greater, less (**`> <`**)
7	Equal, not equal (**==** or **`!=`**)
8	Bit wise AND (**&**)
9	Bit wise exclusive OR (**^**)
10	Bit wise OR. (**|**)
11	Logical AND (&&)
12	Logical OR (**||**)
13	Conditional expression: a ? b : c;
14	Assignment (**=**)
15	Comma (**`,`**)

Definitions and identifier scope

In C, before you can use an identifier, you must declare it first. In the most common cases you declare it by writing a line with

`<type> <identifier> ';'`.

If the identifier is an array, you follow its name with the size of the array enclosed between square brackets. Here are some examples of definitions:

Definition	**Meaning**
`int a;`	Defines an integer that will be called "a". Type is "`int`", and the identifier is "a".
`long double *b[12];`	Defines an array of pointers to long double data with 12 positions. The type is "`long double *`", the name of the identifier is "b", and it is an array, since the identifier is followed by a size enclosed in square brackets.
`int (*fnPtr)(int,double);`	Defines a pointer to a function that returns an integer and receives an integer and a double as arguments. The type is `int(*)(int, double)`, and the name

	of the identifier is "fnPtr"
`char *a,*b;`	Declares two pointers to characters.
`int a,b,*c;`	Declares two integers "a" and "b", and a pointer to an integer called "c". PLEASE read very carefully the asterisk in C programs!

Definitions are just a way of naming memory locations. You have to supply the name of the memory location, i.e. how it will be named in the program, and its type, that describes to the compiler how this memory location should be accessed. When you write "`int a`" you are telling the compiler in a coded (compressed) way:

I want to name a memory location of 32 bits "a", and it should be accessed as a signed integer value. Reserve space for it in the current scope.

When you write, "char c", you tell the compiler to reserve space for an 8 bit integer that later when you write c = 'a' will be understood as: "store in those 8 bits of the character "c", the integer value corresponding to the letter "a" in the ASCII code.

The storage locations are organized in scopes. A scope is a set of memory locations and their corresponding names. They are limited by "{" and "}".[49]

Examples of scopes:

```
int fn(int arg)
{
     int a,b;

     a = arg+5;
     if (a < 23) {
          int c;
          c = a – 45;
          while (c < 0) {
               printf("%d\n",c);
               c--;
          }
     }
     int c = a + 6;
     b = c+99;
     return c-b;
}
```

In this example we have several nested scopes. The outermost scope where the function is defined is the default scope, or global scope. Then, we have a scope

[49] The `for` looping construct introduces a scope too. That's why you can write:
`for (int i=0; i<10;i++) {  /* code of the loop */ }`

that starts at line 2 and runs to line 17. Within it, we have a nested scope that starts at line 6 and runs to line 13. It contains another nested scope that begins in line 9 and ends line 12.

Note the definition in line 14. It demonstrates that you can define a variable at any moment. The variable "c" is visible (i.e. "in scope" from the line of its definition to the end of the scope where it is defined, i.e. until line 17.

Shadowing.

Consider this code:

```
int fn(int arg)
{
     int a,b;
     /* some code */
     if (a < 2) {
          int a;
          /* some more code */
     }
}
```

The second, inner scope contains a definition of a variable "a" that shadows the definition of the other "a" defined in the enclosing scope. It is impossible to access the first "a" within the inner scope, since all references to "a" will lead to the second variable.

Initializations.

When you declare a variable, it is possible to assign it immediately a value. For instance you can write:

```
     double n = 64.98877;
```

Storage for the floating point variable "n" will be immediately filled with the value given.

Here are some examples of initializations:

```
int array[] = { 1,2,3,4,5,6};
```

The variable "array" is a table with 6 integers.

```
char *message = "Please enter your name";
```

The variable "message" is a pointer to char that is filled with the address of the zero terminated character array containing the bytes given.

```
double (functionPointer)(double) = sqrt;
```

The variable "functionPointer" is a pointer to a function that returns a double and receives a double as argument. This pointer will be filled with the address of the library function "sqrt", that computes the square root.

The variables declared at a global scope will be collected by the compiler into a set of memory locations that makes the data segment of the program. Variables that are not initialized explicitly are zeroed by the program loader before the program starts.

Each variable defined at the global scope will be visible by all scopes after its definition, if it is not shadowed, of course. They are a convenient way for functions and modules to share data, but they have several disadvantages too.

The problem with global variables is that they represent an undocumented argument passed to any scope that uses them. Since they can be accessed at any time, in multithreaded programs they are a source of problems since two threads could change the value stored in a global independently of each other, producing inconsistent results.

The usage of globals should be reduced to a minimum, more or less in the same vein as the usage of "gotos" that we saw at the beginning of this tutorial.

Global variables can have two different types of visibility:

1. They can be visible to all other modules of the program. This is the default.
2. They can be visible within the file where they are defined. Thos variables are marked with the keyword "static".

Suppose in the module "input.c" you declare:

```
    int balance;
```

If you want to access the variable "balance" from module "output.c" you declare:

```
    extern int balance;
```

meaning that the definition of "balance" is somewhere else. Those declarations are normally written in a header file.

If you want a variable to remain invisible to all other modules you write:

```
static int balance;
```

This ensures that the name "balance" will not be visible to other modules. If you write in some module

```
extern int balance;
```

The linker will not find any "balance" anywhere and it will issue an error. Note however, that if you have somewhere (in the module "calcs.c") a definition like this:

double balance;

This will provoke an incredible mess. The linker will find a symbol "balance", and it will generate a program where in module input.c the variable will be seen as an integer, and in module calc.c will be seen as a double.

This will not provoke a crash, since the definition of "balance" reserves 64 bits of space, and the module "input.c" supposes an integer of 32 bits only. Of course the results will be completely wrong.

Worst is the case when the definition of "balance" is

```
int balance;
```

and somewhere else you declare:

```
extern double balance;
```

In this case, when the module calc.c accesses the variable "balance", it will overwrite the 32 bits of any data that is right afterwards of the space allocated to "balance". You will see that the value of another, completely unrelated variable changes mysteriously, without any reason.

To avoid this error you should always declare shared variables in a common header file to let the compiler warn you of any inconsistency. If "calc" and "input" share the variable "balance" they should share a common header file too, where all common variables are correctly declared and no inconsistencies can arise.

The "static" keyword applies not only to variables but to functions too. If you declare:

```
static int fn(double argument);
```

This means that the function will not be visible and can't be used from another module.

Static variables differ from other variables what initialization is concerned. Suppose this code:

```
int fn(double arg)
```

```
{
     int a = 7;

     /* some other code */
}
```

Each time the function is called, the variable "a" will have the value 7. The initialization is performed each time this function is entered.

Now, if you write:

```
int fn(double arg)
{
     static int a = 7;

     /* some other code */
}
```

The variable "a" will be initialized to 7 when the program starts, and will NOT be initialized again. This means that each time the function is called, the variable will hold the last value it had. This allows you to maintain a state from one call to the next. Especially useful are static variables when you are writing a windows procedure. More about this later.

Simple programs

Problem 1: 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 *strchr(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. We just increment our pointer. 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.

Problem 2: Return the length of a given string.
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.[50]

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.

[50] 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.

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 the next zero byte in memory.

Problem 3: 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(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[51] from 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[52].

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.

[51] 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.

[52] For a more detailed discussion, see the section Newsgroups at the end of this tutorial.

How can this program fail?

The while loop has only one condition: that n is different from zero, i.e. that n has some bits set. Since we are shifting out the bits, and shifting in always zero bits, 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 from 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 "i" 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.

Problem 4:

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.[53]

This is the library function `strlwr`. We make the transformation in-place, i.e. we transform the given string.

```
#include <ctype.h> /* needed for using isupper and tolower */

char *strlwr(char *str)
{
     char * result = str; /* needed to store return value */

     if (str == NULL)
          return NULL;
     while (*str) {
```

[53] 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.

```
          if (isupper(*str))
               *str = tolower(str);
          str++;
     }
     return result;
}
```

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`".

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 (*str)` tests if the contents of the character pointer `str` are different than zero. If this is the case, we test if the character is an upper case character using the `isupper` classification function. If it is an upper case character, we transform it into a lower case one. We increment our pointer to point to the next character, and we restart the 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.

<u>How can this program fail?</u>
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. <u>The absence of a zero byte in a string is a catastrophe for any C programmer</u>. In a tutorial this can't be too strongly emphasized!

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.[54]

```
#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) {
          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,
```

[54] Yes, code reuse is not only possible in object-oriented programming.

```
                    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. [55]

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.

[55] 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 from 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

    (  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
```

I have organized the data in a table to easy the display. 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:[56]

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

Well, this is quite an impressing proto 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:

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:[57]

```
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`

[56] The prototype is in the header file stdlib.h

[57] 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.

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>

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

void 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)
}
```

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[58] 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.

1. We include the standard header string.h, to get the definitions of string handling functions like stricmp.
2. 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”.
3. 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.
4. 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.

[58] stricmp is called strcasecmp in some UNIX systems.

At the end of 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.

5. 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.

6. 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.[59]
7. 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 project. We use cut and paste in the

[59] 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.

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:

```
#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) {
                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++;
                if (c >= ' ')
                        Frequencies[c]++;
                c = fgetc(infile);
        }
        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
  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
```

Well, sorting definitely works (you read this display line by line), but we note with dismay that 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 matched a character value. But once we sort the array, this ordering is gone, and we have lost the correspondence between each array element position 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.

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.

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 FILE starts in memory.

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 machine 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.[60]
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. [61]

References are declared in a similar way as pointers are declared:

```
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 "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;
```

[60] 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.

[61] 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 that language.

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.

Structures and unions

Structures

Structures are a contiguous piece of storage that contains several simple types, grouped as a single object.[62] 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;
};
```

[62] This has nothing to do with object oriented programming of course. The word object is used here with its generic meaning.

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, an infinite recursion would immediately appear!

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;
     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 UpdateCustomerBalance(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 monolithical 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.

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.[63]

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.

The best thing 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;
```

[63] The usage of the #pragma pack construct is explained in lcc-win32 user's manual. Those explanations will not be repeated here.

```
};
```

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.

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.[64]

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, if not, the compiler will get really confused.

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;
```

[64] 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.

```
     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. The previous value is erased, overwritten with the new data.

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;
     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.

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.

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.

Structures can be initialized to known values when the program starts. Suppose that you have:

```
typedef struct person {
     char *name;
     int age;
} PERSON;

PERSON friends[] = {
{    "John", 27 },
{    "Mary", 18},
{    "Joseph",52},
};
```

Here we initialize an array of PERSON structures. Note that each individual structure is enclosed in "{" and "}".

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.

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:[65]

- 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.

[65] An identifier can also represent a macro or a macro argument, but here we will assume that the pre-processor already has done its work.

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
{
     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;[66]
     }
}
```

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”.

[66] You see the infinite loop here? Tell me: why is this loop never ending? Look at the code again.

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.

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 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 monolithical, 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 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.[67]

```
void Initialize(void)
{
     for (int i = 0; i<256; i++) {
          Frequencies[i].CharacterValue = i;
     }
}
```

[67] 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.

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);[68]
     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 microprocessors, it is important to get rid of the frame of mind of twenty years ago, when saving every cycle of

[68] 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.

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",
                    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!

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.

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;
}
```

Note here the construct

```
     do { … statements … } while (condition);
```

The body of the “do” will always be executed at least once; the condition is tested at the end of the first execution.

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.[69]

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;
```

[69] 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.

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!

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) {
               fprintf(stderr,
                  "Usage: countchars <file name>\n");
               exit(1);
        }
        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 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.[70]

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 it's empty.
`diskfree`	Returns the amount of space available in a disk.

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

[70] 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.

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 "`lcc-win32`" as above, and we call the 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	45	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
strcat	Appends strings.
strchr	Find the first occurrence of character in a string
strrchr	Find the last occurrence of a character in a string
strcmp	Compares two strings
strncmp	Compare strings up to a maximum length
strnicmp	Compare strings up to a maximum length ignoring case
strcol	Compare strings using locale-specific information.
strcpy	Copy a string into another
strcspn	Find a substring in a string
strupr	Convert string to upper case
strlwr	Convert string to lower case
strerror	Get a system error message (strerror) or prints a user-supplied error message (_strerror).
strlen	Find the length of a string
strncat	Append characters of a string.
strncpy	Copy strings up to a maximum length
strpbrk	Scan strings for characters in specified character sets.
strspn	Find the first substring
strstr	Find a substring
stristr	Find a string ignoring case.
strtok	Find the next token in a string
strdup	Duplicate a string. Uses malloc.
strrev	Reverse characters in a string
strtrim	Eliminate redundant blanks from a string.
strset	Set characters in a string to a character.

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
IsCharAlphaNumeric	IsCharLower	IsCharUpper	LoadString
lstrcat	lstrcmp	lstrcmpi	lstrcpy
lstrcpyn	lstrlen	MultiByteToWideChar	OemToChar
OemToCharBuff	WideCharToMultiByte	wsprintf	wvsprintf

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.[71]

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

[71] Memory allocation problems plague also other languages like C++ that use a similar schema than C.

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.

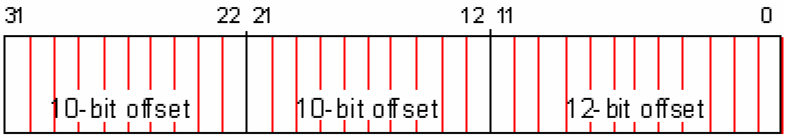
Functions for memory allocation.

`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.

Memory layout under windows.[72]

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 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.

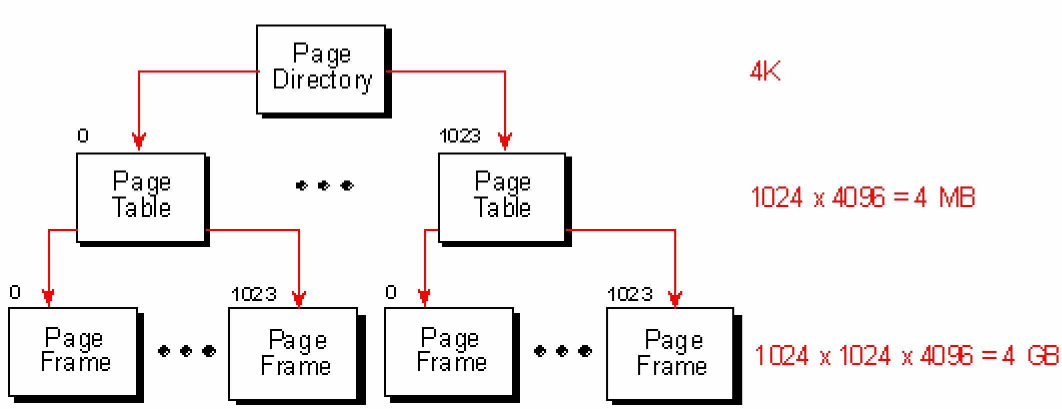


The translation goes as follows:

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.

[72] This discussion is based upon the article of Randy Kath, published in MSDN.

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.[73]

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. See inter-process communications.

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.

[73] 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.

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:

- Parameter less. 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 complicate to grasp with all those redundant parentheses around.

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.

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. [74]

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;
  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;
        short timezone; // Difference in minutes, moving westward, between
                        //  UTC and local time
```

[74] 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…

```
        unsigned short pad2;
        unsigned long lpad2;
        short dstflag; // Nonzero if daylight savings time is currently in effect
                       // for the local time zone.
        };
```

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.

1. 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.
2. 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.
3. 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 choosen , 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. [75]

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.

[75] This clock will overflow in something like 2.000 years so be prepared for windows 4.000!

`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.

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.

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;

     if (rvp == NULL) { // is the list empty?
          // 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 easies an operation like wrapping around when searching.

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 38th position in our table.

This position should contain a list of character strings that all map to the 38th 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.

Windows Programming

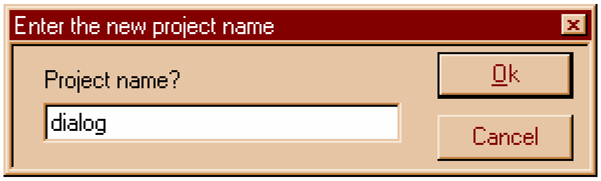
OK, up to now we have built a small program that receives all its input from a file. This is more or less easy, but a normal program will need some input from the user, input that can't be passed through command line arguments, or files. At this point, many introductory texts start explaining scanf, 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.

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.

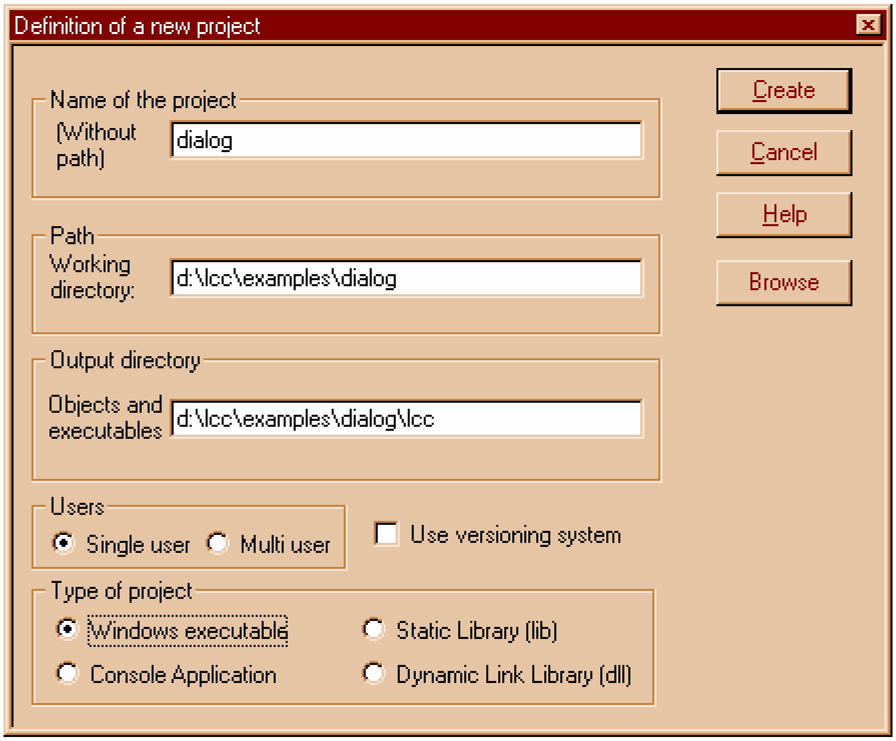
The easiest way to do this is to ask wedit to do it for you. You choose 'new project' in the project menu, give a name and a sources directory, and when the software asks you if it should generate the application skeleton for you, you answer yes.

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:


Project name? dialog

You press OK, and then we get a more complicated one, that asks quite a lot of questions.



Definition of a new project

Name of the project
(Without path) dialog

Path
Working directory: d:\lcc\examples\dialog

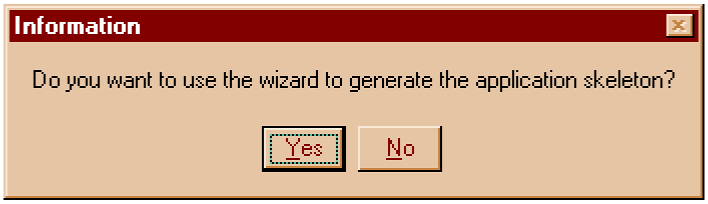
Output directory
Objects and executables d:\lcc\examples\dialog\lcc

Users
● Single user ○ Multi user

□ Use versioning system

Type of project
● Windows executable ○ Static Library (lib)
○ Console Application ○ Dynamic Link Library (dll)

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:

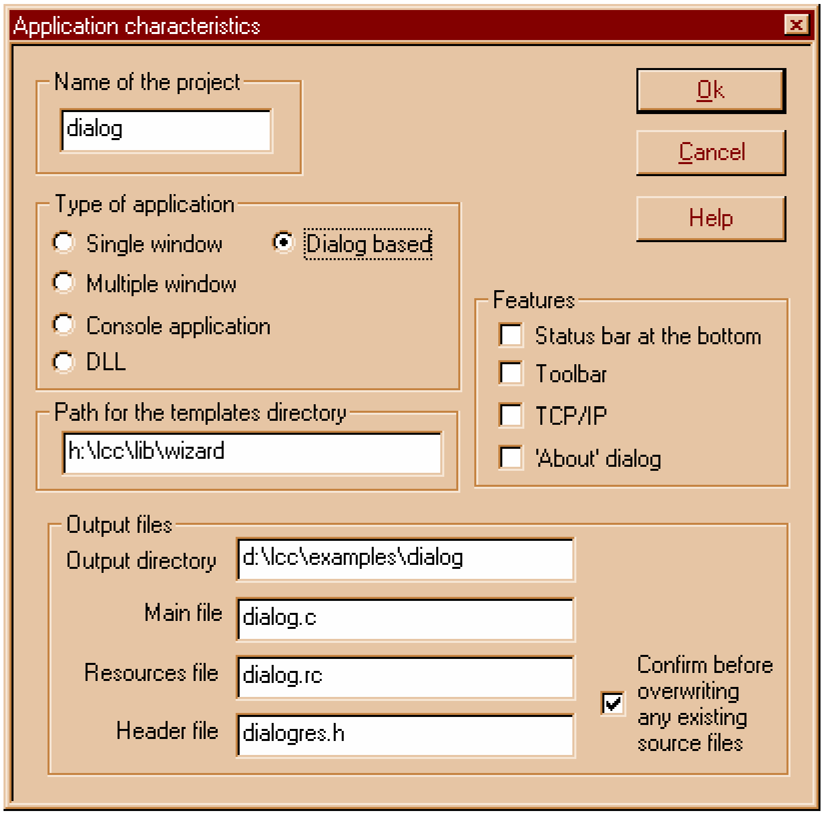


Information

Do you want to use the wizard to generate the application skeleton?

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:



Name of the project: dialog

Type of application
○ Single window ● Dialog based
○ Multiple window
○ Console application
○ DLL

Path for the templates directory: h:\lcc\lib\wizard

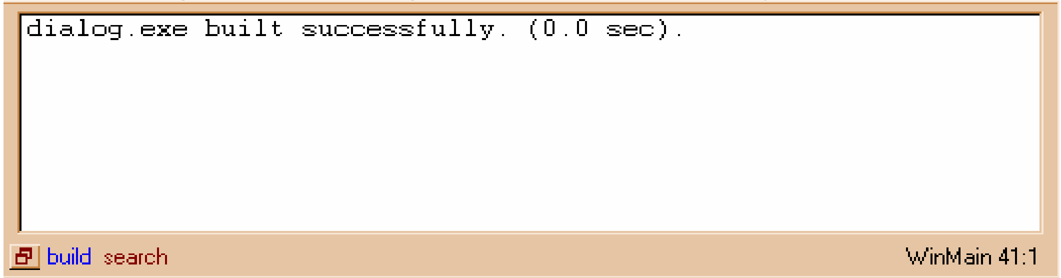
Features
□ Status bar at the bottom
□ Toolbar
□ TCP/IP
□ 'About' dialog

Output files
Output directory: d:\lcc\examples\dialog
Main file: dialog.c
Resources file: dialog.rc
Header file: dialogres.h
☑ Confirm before overwriting any existing source files

You see, the “Dialog based’ check button at the upper left is checked. Then press the OK button.

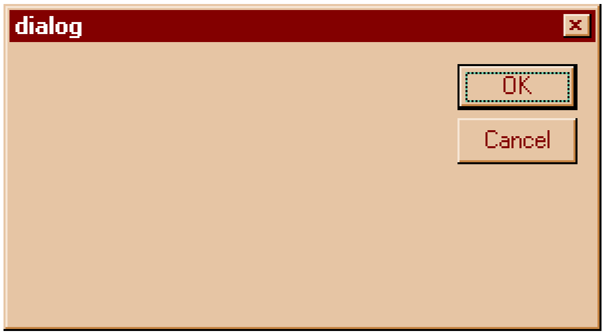
Then we get to different dialogs to configure the compiler. You leave everything with the default values, by pressing Next at each step. At the end, we obtain our desired program. For windows standards, this is a very small program: 86 lines only, including the commentaries. We will study this program 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. (0.0 sec).
```

Dialog.exe built successfully. Well, this is good news![76] 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.

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.[77]

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,
```

[76] I hope this happens to you too…

[77] 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.

```
     LPSTR lpCmdLine,
     int nCmdShow);
```

This is a function that returns an int, uses the stdcall calling convention[78] denoted by APIENTRY, and that receives (from the system) 4 parameters.

1. 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.
2. 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.
3. 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.
4. 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,
```

[78] 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.

```
                    (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[79] 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.

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[80]. We tell

[79] API means Application Programmer Interface, i.e. entry points into the windows system for use by the programmers, like you and me.

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.

[80] PLEASE never do this if you use the garbage collector!

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.

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[81], 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
```

[81] 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.

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.
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.

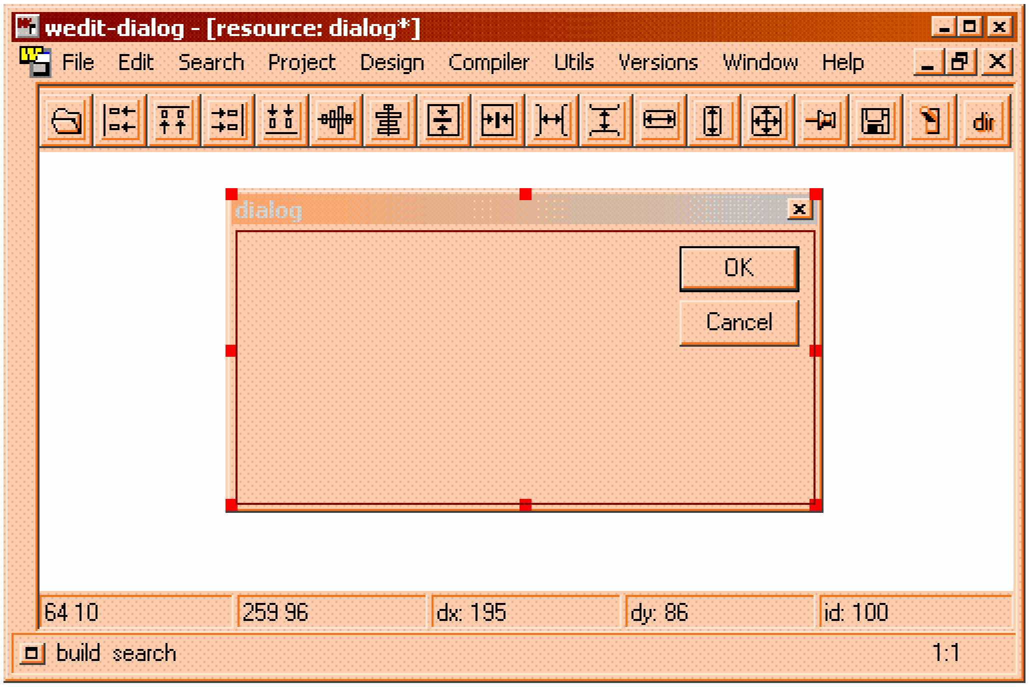
The resource compiler of lcc-win32 is explained in detail in the technical documentation, and we will not repeat that stuff again here. For our purposes it is enough to know that it is compatible with the other ones.

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.

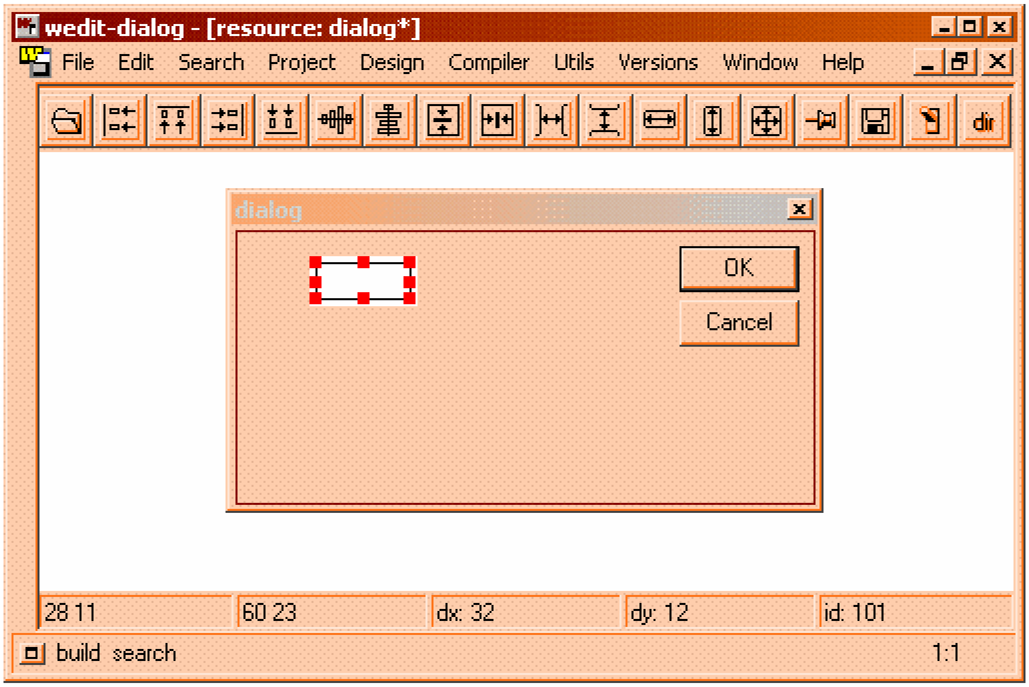
Again, we come back to the IDE, after closing our text file “dialog.rc”, and we go to the “Design” menu bar and press “Open/new”. [82]The resource editor opens up, and we see the following display:



The whole operation of the editor is quite simple: The vertical menu, that appears when you press the T button at the left 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 that, or dialog will look like this:

[82] 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”.



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 can enter this data directly in the auxiliary window in the "Name" field.

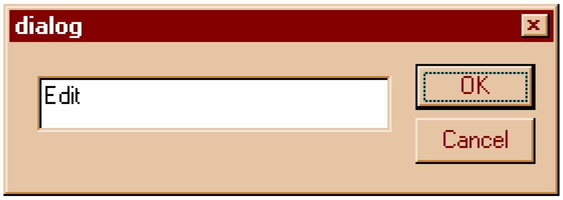
We will refer then in our program to this entry field with the name IDENTRYFIELD, maybe not a very cute name, but at least better than some bare number. The editor will write a

```
#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, the one just before the "Dr" button in the upper row of icons of Wedit's main window.

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.

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.[83]

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.
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));
```

[83] 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!

```
     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[84] 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.[85]

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.

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:

[84] You put the cursor under the WM_INITDIALOG identifier and press F1.

[85] 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.

```
     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.

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[86], 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

[86] All the notifications messages from edit fields begin with the EN_ prefix, meaning **E**dit field **N**otification.

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)
{
     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.

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.[87]

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

[87] 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.

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);
     SetFocus(GetDlgItem(hDlg,IDENTRYFIELD));
     EnableWindow(GetDlgItem(hDlg,IDOK),0);
     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.[88]

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 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:

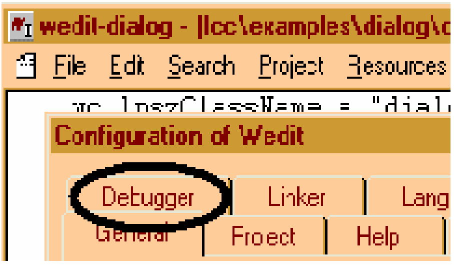
[88] Now is a good time to read the documentation for that API. It will not be repeated here.

```
     return DialogBoxParam (hinst,
          MAKEINTRESOURCE(IDD_MAINDIALOG),
          NULL, (DLGPROC) DialogFunc,
          (int)lpCmdLine);
```

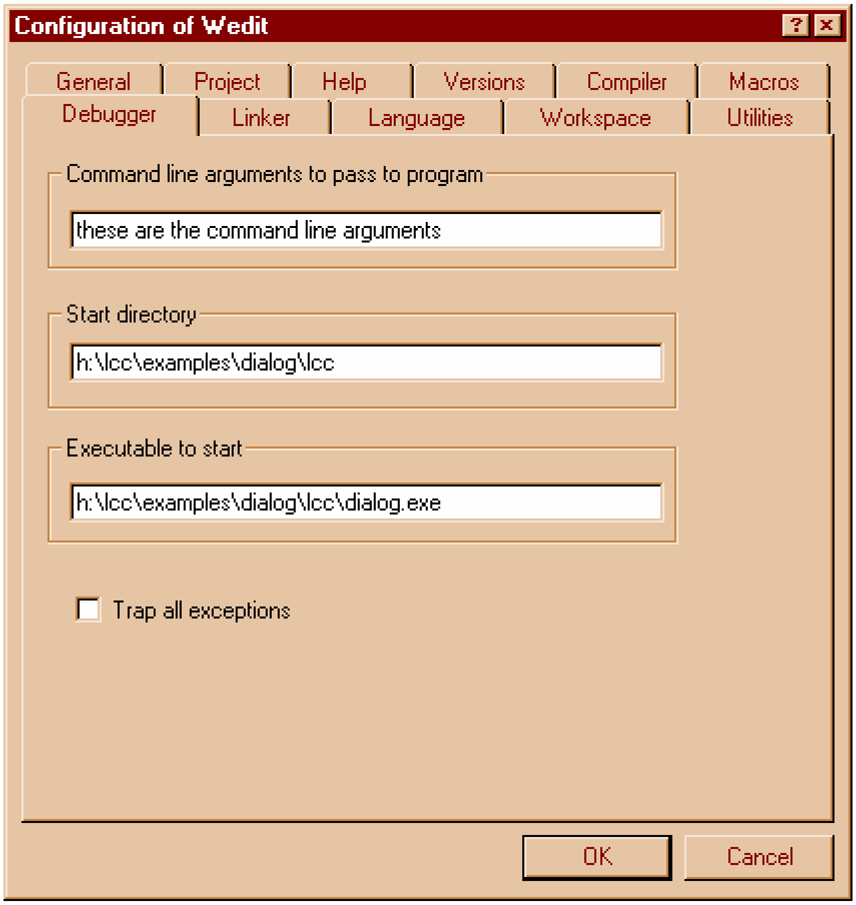
Since our dialog box is now constructed with DialogBoxParam, we receive in the lParam message parameter the same pointer that we gave to the DialogBoxParam API. Now, we have just to set that text in the caption of our dialog box and it's done. We do that (again) in our initialization procedure by adding:

```
     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:



Configuration of Wedit

Command line arguments to pass to program

these are the command line arguments

Start directory

h:\lcc\examples\dialog\lcc

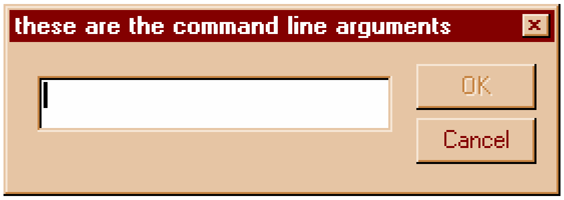
Executable to start

h:\lcc\examples\dialog\lcc\dialog.exe

□ Trap all exceptions

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:



these are the command line arguments

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 consideration 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);
```

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!

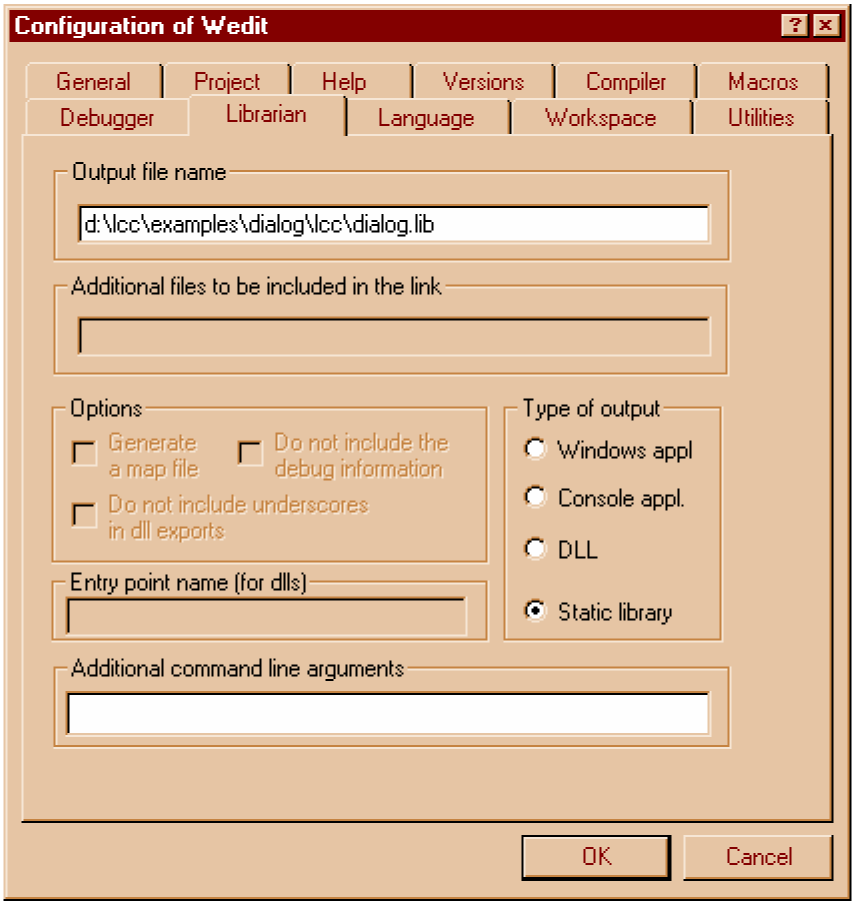
Our program needs a class registration before it can call the DialogBoxParam API. If we use the static library approach, we would have to require that the user of the library calls some initialization routine before, to allow us to register our class with windows.

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:



Output file name: d:\lcc\examples\dialog\lcc\dialog.lib

Additional files to be included in the link:

Options
□ Generate a map file
□ Do not include the debug information
□ Do not include underscores in dll exports

Entry point name (for dlls):

Type of output
○ Windows appl
○ Console appl.
○ DLL
● Static library

Additional command line arguments:

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![89]

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:

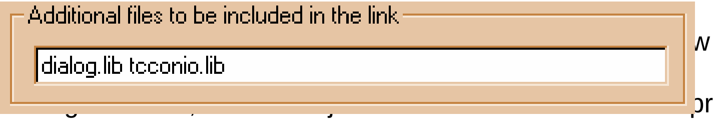
```
int GetString(char *prompt, char *destBuffer,int bufferlen);
```

and that's all.

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 console application, we should test that now.

Now, when creating the project, we ask the wizard to create a console application.[90] 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:



Additional files to be included in the link

dialog.lib tcconio.lib

[89] We introduce w we are returning that data to the calli

[90] Well, this is proposes to create the skeleton.

Another issue to remember, is the following:

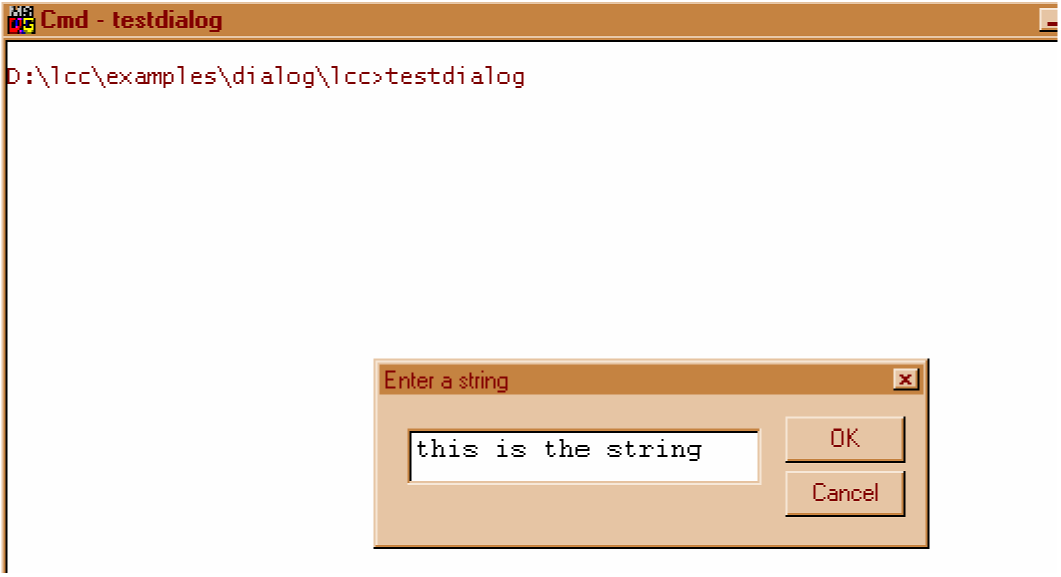
We need a resource file. Since in our library there are no resources, we have to add those resources to our test program. This is easy to do in this case: we just add the dialog.rc resource file to the project. The interface for the users of the library however is terrible. All programs that use our 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;
}
```

When we start this program from the command line, we see:



```
D:\lcc\examples\dialog\lcc>testdialog
```

this is the string

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.

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.[91]

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 hardwire 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)
```

[91] 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.

```
    {
        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.

```
#include <windows.h>
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.

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
```

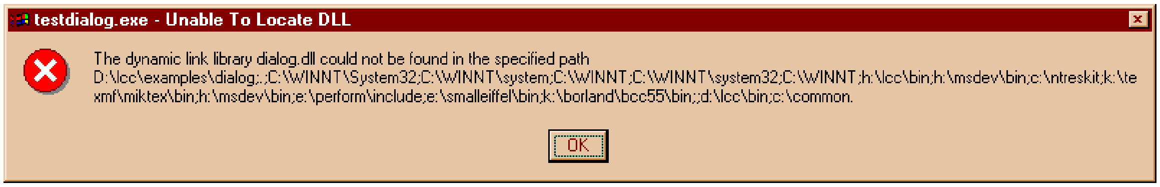
then we link with the command:

```
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:



testdialog.exe - Unable To Locate DLL

The dynamic link library dialog.dll could not be found in the specified path D:\lcc\examples\dialog;.;C:\WINNT\System32;C:\WINNT\system;C:\WINNT;C:\WINNT\system32;C:\WINNT;h:\lcc\bin;h:\msdev\bin;c:\ntreskit;k:\texmf\miktex\bin;h:\msdev\bin;e:\perform\include;e:\smalleiffel\bin;k:\borland\bcc55\bin;;d:\lcc\bin;c:\common.

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:

1. We need to include windows.h for getting the prototypes of LoadLibrary, and GetProcAddress, besides some constants like `INVALID_HANDLE_VALUE`.
2. stdio.h is necessary too, since we use fprintf
3. 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.
4. 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.
5. We test if the handle received is a correct one. If not we put up some message and we exit the program.
6. 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?
7. 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:

1. lcc-win32 like many other C compilers, adds always an underscore to the names it generates for the linker.[92]
2. 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.

[92] 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.

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.[93]

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.

A more formal approach.

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.

We have extensively used the switch statement above. This statement allows you to select several cases from the possible ones and act accordingly. It is composed from a switch expression that should evaluate to an integer result, and several pieces of code, associated to an integer value. The compiler generates code to evaluate the switch expression, and jump accordingly to the corresponding case.

In C, a case will continue with the next one, unless you finish it with a break expression, the same expression that is used to break from a loop.

```
     switch (expr) {
          case 123:
               a = p*6;
          case 251:
               a = p*9;
```

[93] 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.

```
     }
```

In this example, if expr is equal to 123, the expression `a = p*6;` will be executed, then the expression `a = p*9;` Yes, I know this is not evident, but it’s like that, and zillion lines of code are already written that use this feature. Cases can be easily merged:

```
     switch (expr) {
          case 123:
          case 834:
          case –45:
               a = p*6;
               break;
          case 251:
               a = p*9;
     }
```

Note that here a default case is missing. This is not a good practice, even if you do not want to do anything in the default case. More readable is:

```
     switch (expr) {
          case 123:
          case 834:
          case –45:
               a = p*6;
          case 251:
               a = p*9;
               break;
          default:
               break;

     }
```

Here we see that you do NOT intend to do anything in the default case.

Another 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.

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.[94] 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:

- 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.[95]

[94] The Macintosh works in the same manner, albeit with a much more primitive system.

[95] 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.

Texts started appearing explaining the surprised programmers what was WinMain and that “message pump” that was again the non-plus-ultra of modernity.[96] 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.

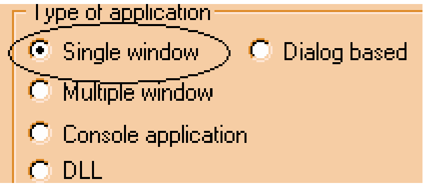
Messages are sent to window objects that are members of a class of similar objects or *window class*. The procedure SendMessage can also send messages to a window object, as if they would have been sent by the system itself. We send, for instance, the message EM_SETLIMITTEXT to an entry field to set 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.

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:



Type of application

● Single window ○ Dialog based

○ Multiple window

○ Console application

○ DLL

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

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”…

[96] 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.

bar at the bottom, a summary 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;

      hInst = hInstance;
      if (!InitApplication())
            return 0;
      hAccelTable = LoadAccelerators(hInst,MAKEINTRESOURCE(IDACCEL));
      if ((hwndMain = CreatewinexampleWndClassWnd()) == (HWND)0)
            return 0;
      CreateSBar(hwndMain,"Ready",1);
      ShowWindow(hwndMain,SW_SHOW);
      while (GetMessage(&msg,NULL,0,0)) {
            if (!TranslateAccelerator(msg.hwnd,hAccelTable,&msg)) {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
      }
      return msg.wParam;
}
```

We have the same schema that we saw before, but this time with some variations. We init 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 bits 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.[97] 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.

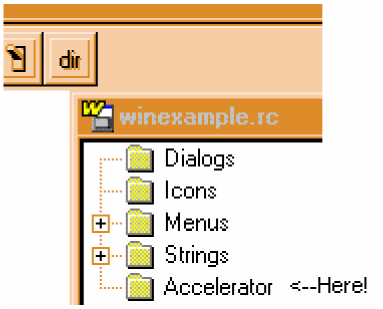
Coming back to our initialization procedure, there are some new things, besides this style setting. But this is just a matter of reading the windows documentation. No big deal. There are many introductory books that augment their volume just with the easy way of including a lot of windows documentation 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.

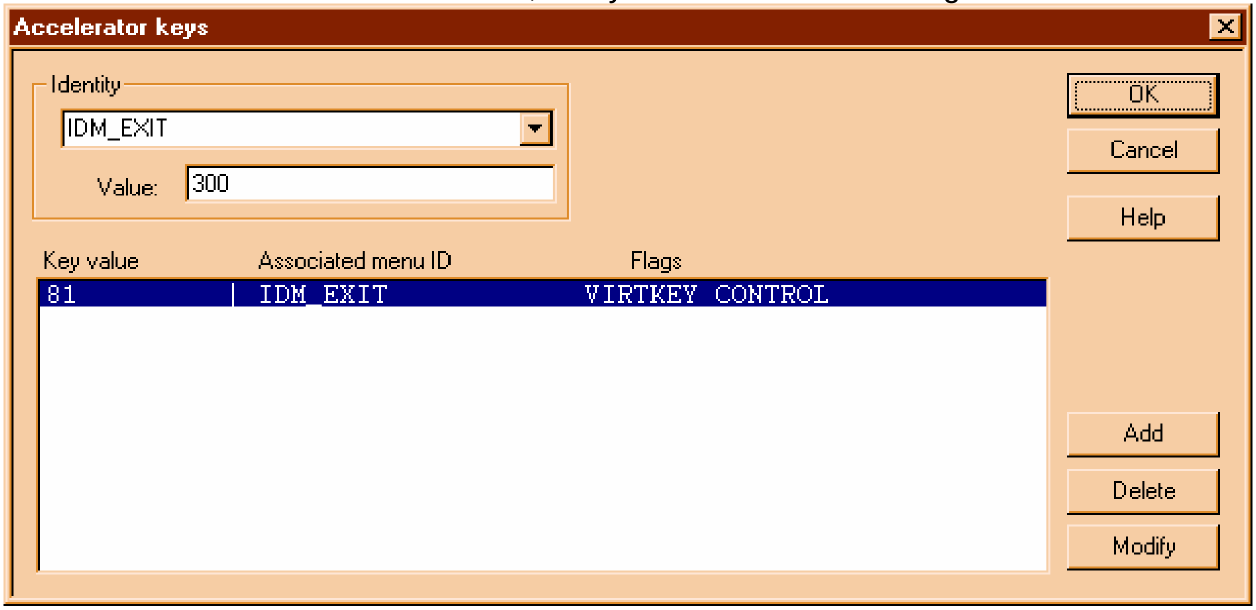
Note that not all is in the index. You have to have a feeling of where to look. Lcc-win32 comes with a help file of reasonable size to be downloaded with a standard modem. It is 13MB compressed, and it has the essentials. A more detailed documentation complete with the latest stuff is in the Software Development Kit (SDK) furnished by Microsoft. It is available at their Web site, and it has a much more sophisticated help engine.

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.



[97] 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.

You click in the “Accelerator” tree tab, and you will see the following:


Accelerator keys

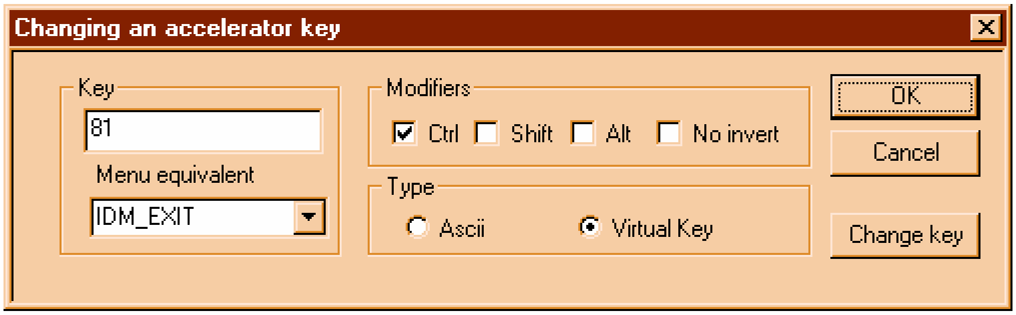
Identity: IDM_EXIT

Value: 300

Key value	Associated menu ID	Flags
81	IDM_EXIT	VIRTKEY CONTROL

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:


Changing an accelerator key

Key: 81

Menu equivalent: IDM_EXIT

Modifiers: ☑ Ctrl □ Shift □ Alt □ No invert

Type: ○ Ascii ● Virtual Key

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.[98]

Wow. Finished?

[98] 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.
-

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 pre-processor *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:

```
    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.

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.

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.[99]

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.

[99] Never forget this: local variables do NOT retain their value from one call of the function to the next one!

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 of the tree control there. This has the advantage that it will work when the user resizes the main window too.

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.

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.

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.

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:

1. 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.
2. We get the class name of the window.
3. We format the text (enclosed in quotes) and the class name in a buffer.
4. We start filling the `TV_INSERTSTRUCT`. These steps are very similar to what we did for the desktop window.
5. 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.
6. 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.

Review:

Let's look at our "BuildTree" function again and ask us:

How could this fail?

We notice immediately several things.

1. 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.
2. 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.

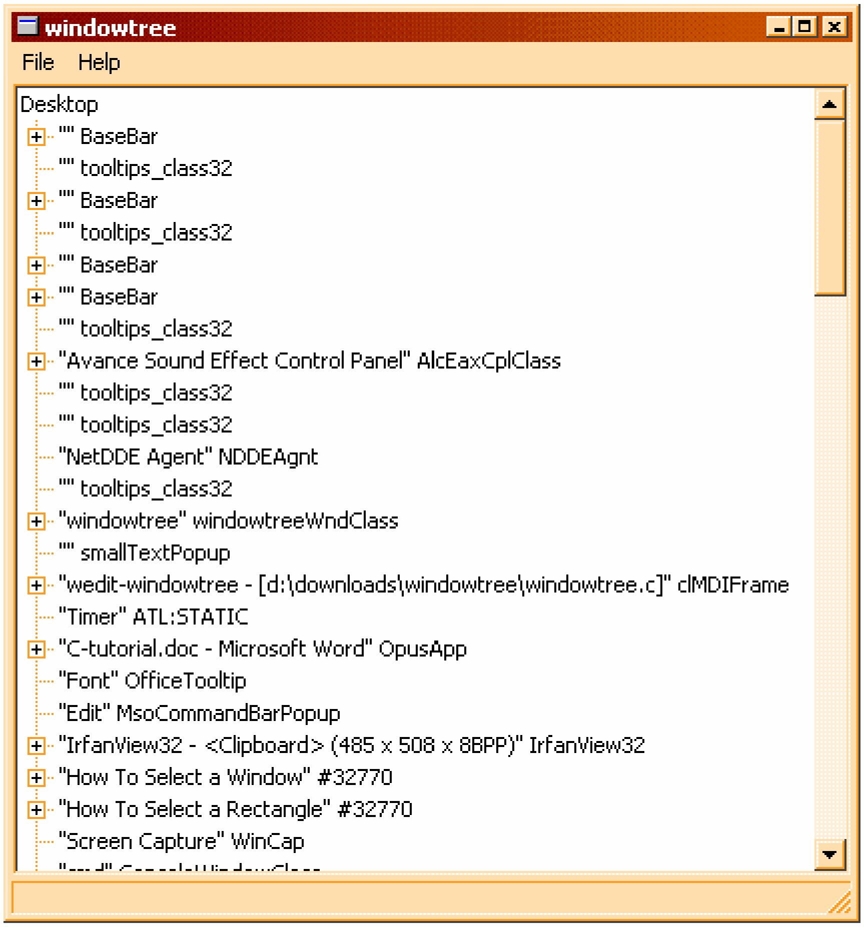
We modify the "BuildTree" procedure as follows:

```
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.



```
Desktop
  "" BaseBar
  "" tooltips_class32
  "" BaseBar
  "" tooltips_class32
  "" BaseBar
  "" BaseBar
  "" tooltips_class32
  "Avance Sound Effect Control Panel" AlcEaxCplClass
  "" tooltips_class32
  "" tooltips_class32
  "NetDDE Agent" NDDEAgnt
  "" tooltips_class32
  "windowtree" windowtreeWndClass
  "" smallTextPopup
  "wedit-windowtree - [d:\downloads\windowtree\windowtree.c]" clMDIFrame
  "Timer" ATL:STATIC
  "C-tutorial.doc - Microsoft Word" OpusApp
  "Font" OfficeTooltip
  "Edit" MsoCommandBarPopup
  "IrfanView32 - <Clipboard> (485 x 508 x 8BPP)" IrfanView32
  "How To Select a Window" #32770
  "How To Select a Rectangle" #32770
  "Screen Capture" WinCap
  "cmd" ConsoleWindowClass
```

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.

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:

1. 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.
2. Map the coordinates we received into the coordinates of the tree window.
3. Ask the tree control if there is an item under this coordinates.
4. If there is none we stop

5. 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.
6. Using that window handle we call another function that will display the info in the status bar.
7. 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.

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);
}
```

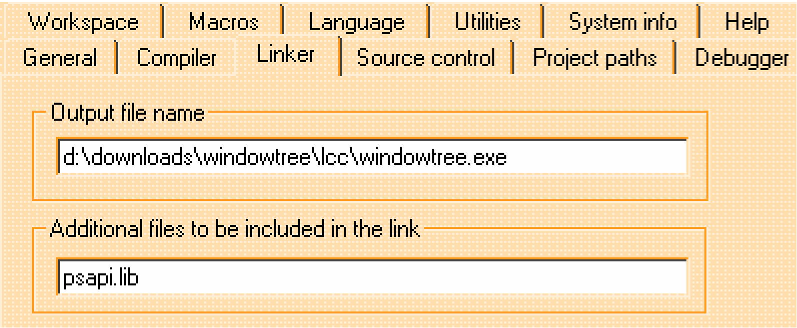
The algorithm here is as follows:

1. Query the window rectangle (in screen coordinates).
2. We get the process ID associated with this window
3. We call a subroutine for putting the name of the process executable file given its process ID.
4. We format everything into a buffer
5. 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:



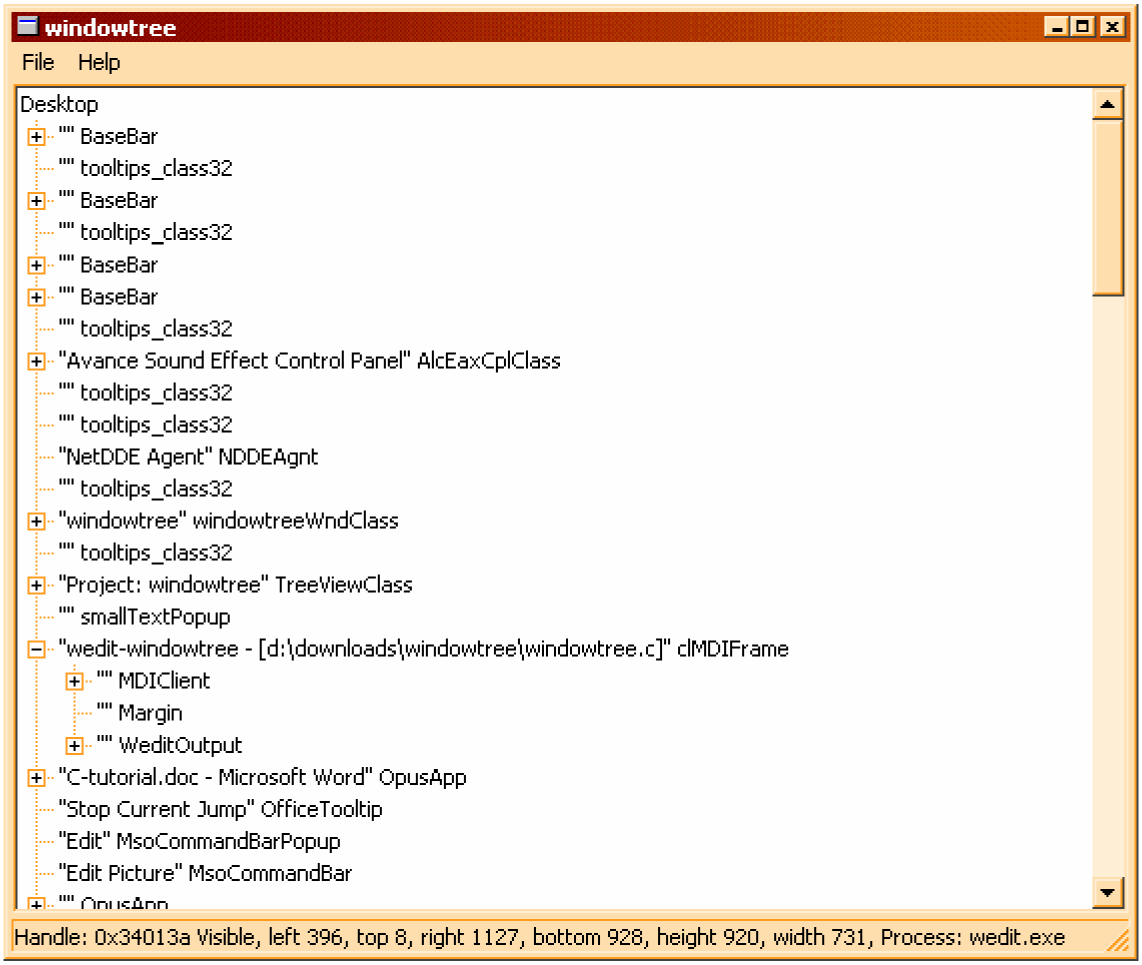
Output file name

d:\downloads\windowtree\lcc\windowtree.exe

Additional files to be included in the link

psapi.lib

And now we are done. Each time you click in an item window, the program will display the associated information in the status bar:



```
Desktop
  "" BaseBar
  "" tooltips_class32
  "" BaseBar
  "" tooltips_class32
  "" BaseBar
  "" BaseBar
  "" tooltips_class32
  "Avance Sound Effect Control Panel" AlcEaxCplClass
  "" tooltips_class32
  "" tooltips_class32
  "NetDDE Agent" NDDEAgnt
  "" tooltips_class32
  "windowtree" windowtreeWndClass
  "" tooltips_class32
  "Project: windowtree" TreeViewClass
  "" smallTextPopup
  "wedit-windowtree - [d:\downloads\windowtree\windowtree.c]" clMDIFrame
      "" MDIClient
      "" Margin
      "" WeditOutput
  "C-tutorial.doc - Microsoft Word" OpusApp
  "Stop Current Jump" OfficeTooltip
  "Edit" MsoCommandBarPopup
  "Edit Picture" MsoCommandBar
  "" OpusApp
```

```
Handle: 0x34013a Visible, left 396, top 8, right 1127, bottom 928, height 920, width 731, Process: wedit.exe
```

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.

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[100] 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;
     theta = float(i)/50;
     theta2 = (float(i)/50)-3.14;
```

[100] In page 218. Published by St Martin's Press. 1990. ISBN 0-312-06179-X (pbk)

```
     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…

1. 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.

2. 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.
3. 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.
4. A break statement “breaks” the loop.
5. 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);
}
```

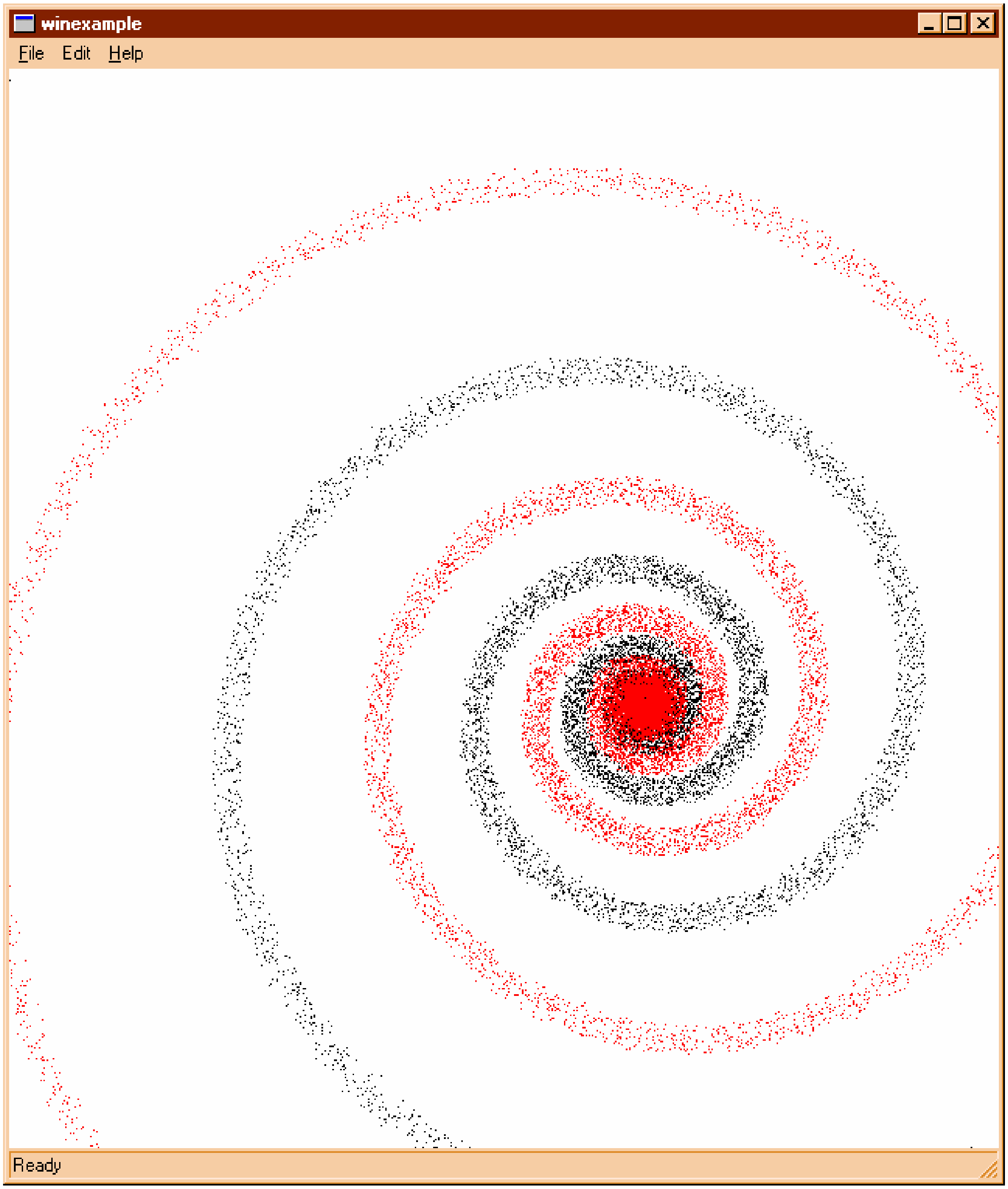
We call the windows API BeginPaint, passing it the address of a PAINTSTRUCT, a structure filled by that function that contains more information about the region that is to be painted, etc. We do not use it the information in it, because for simplicity we will repaint the whole window each time we receive 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.

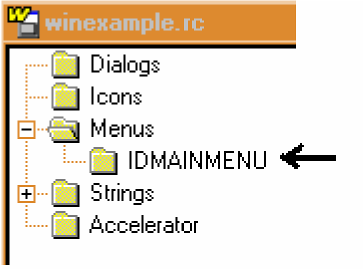
All functions that return a double result must declare their prototype to the compiler so that the right code will be generated for them. An unprototyped function returning a double will surely result in incorrect results!

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.

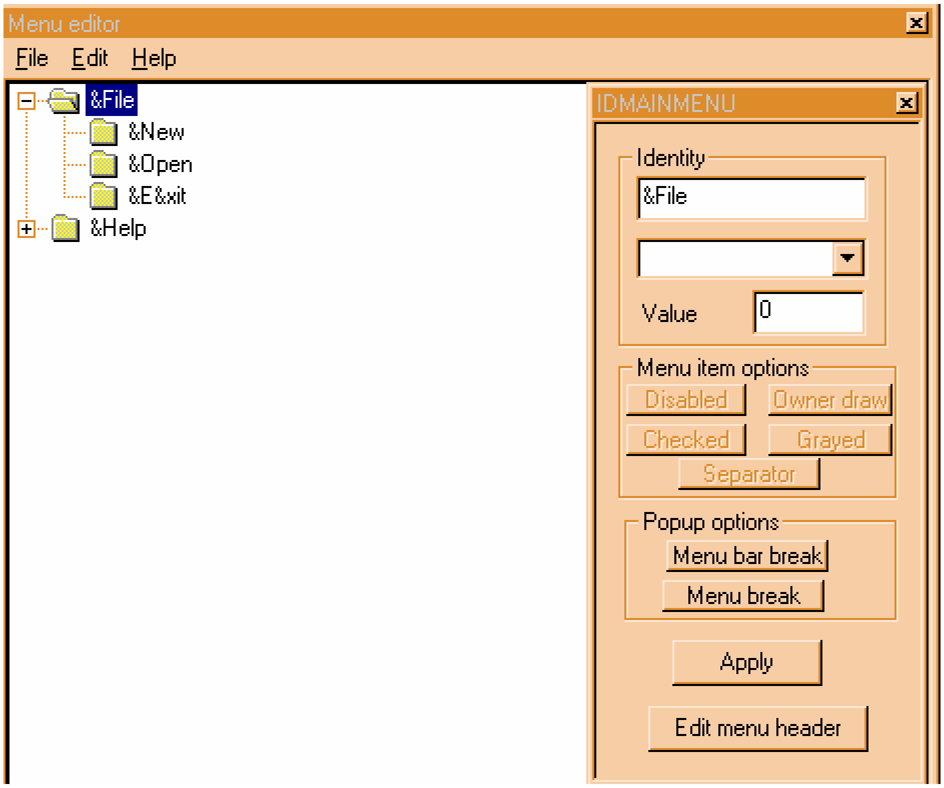
Filling the blanks

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[101]. If we open each branch of the tree in its left side, it looks like this:

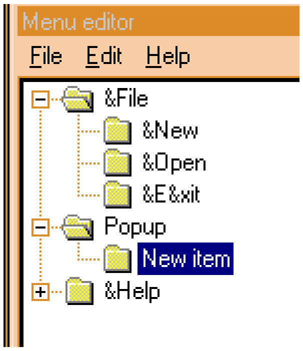


Identity: &File

Value: 0

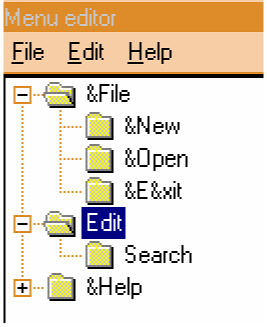
We have at the left side the tree representing our menu. Each submenu is a branch, and the items in the branch; the leaves are the items of the submenu. We select the "File" submenu and press the "insert" key. We obtain a display like this:

[101] 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.



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. [102]

OK, we change the default names to the traditional “Edit” and “Search”, to obtain this display:

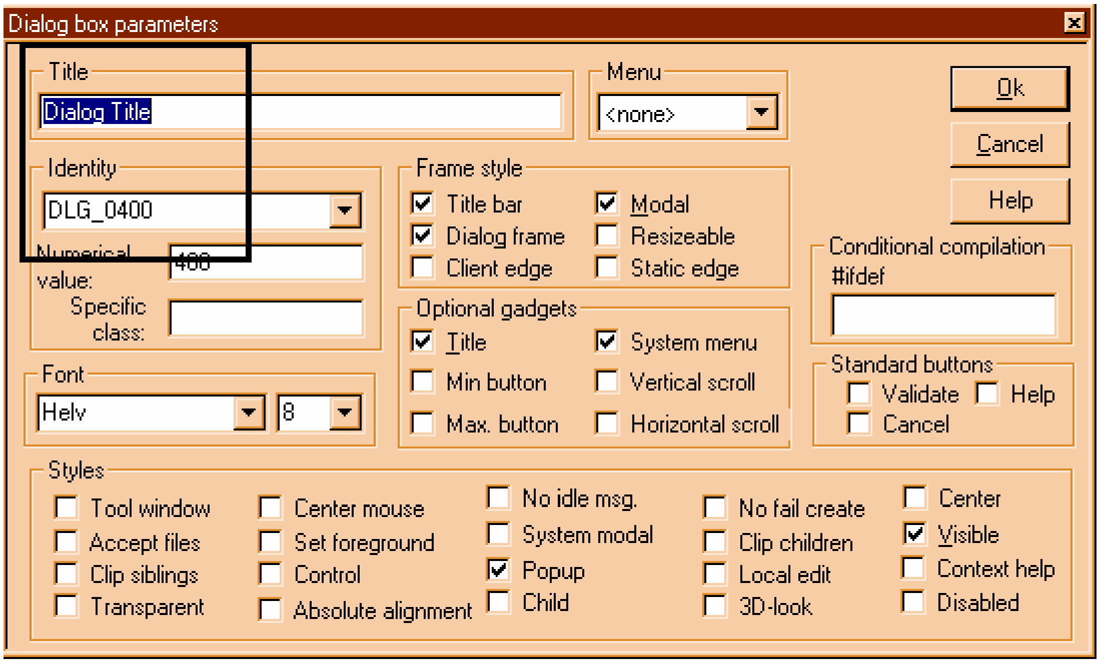


We will name the new item `IDM_SEARCH`. I am used to name all those constants starting with IDM_ from ID Menu, to separate them in my mind from IDD_ (ID Dialog).

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. 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:

[102] 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.


Dialog box parameters

Title: Dialog Title

Menu: <none>

Identity: DLG_0400

Numerical value: 400

Specific class:

Font: Helv 8

Frame style: ☑ Title bar ☑ Modal ☑ Dialog frame □ Resizeable □ Client edge □ Static edge

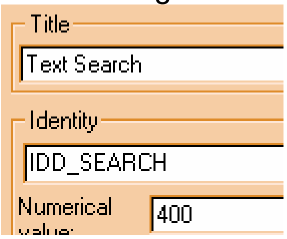
Optional gadgets: ☑ Title ☑ System menu □ Min button □ Vertical scroll □ Max. button □ Horizontal scroll

Conditional compilation #ifdef:

Standard buttons: □ Validate □ Help □ Cancel

Styles: □ Tool window □ Accept files □ Clip siblings □ Transparent □ Center mouse □ Set foreground □ Control □ Absolute alignment □ No idle msg. □ System modal ☑ Popup □ Child □ No fail create □ Clip children □ Local edit □ 3D-look □ Center ☑ Visible □ Context help □ Disabled

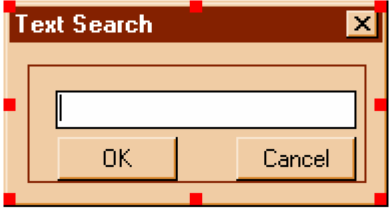
Even if this quite overwhelming, we are only interested in two things: the title of the dialog and the symbolic identifier. We leave all other things in their default state. We name the dialog IDD_SEARCH, and we give it the title “Text search”. After editing it looks like this:


Title: Text Search

Identity: IDD_SEARCH

Numerical value: 400

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 we 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 MainWndProc. 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;
}
```

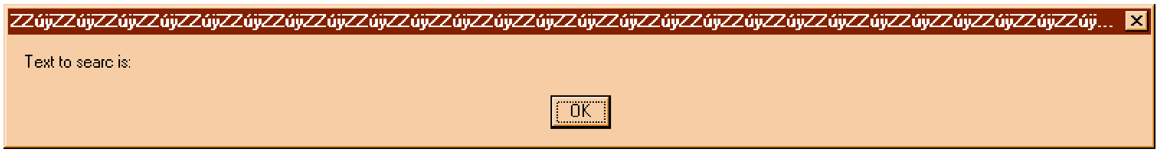
This does nothing, it doesn't even put the text in the received parameter, but what we are interested in here, is to first ensure the dialog box shows itself. Later we will refine it. I develop software like this, as you may have noticed: I try to get a working model first, a first approximation. Then 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:



ZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿZZúÿ…

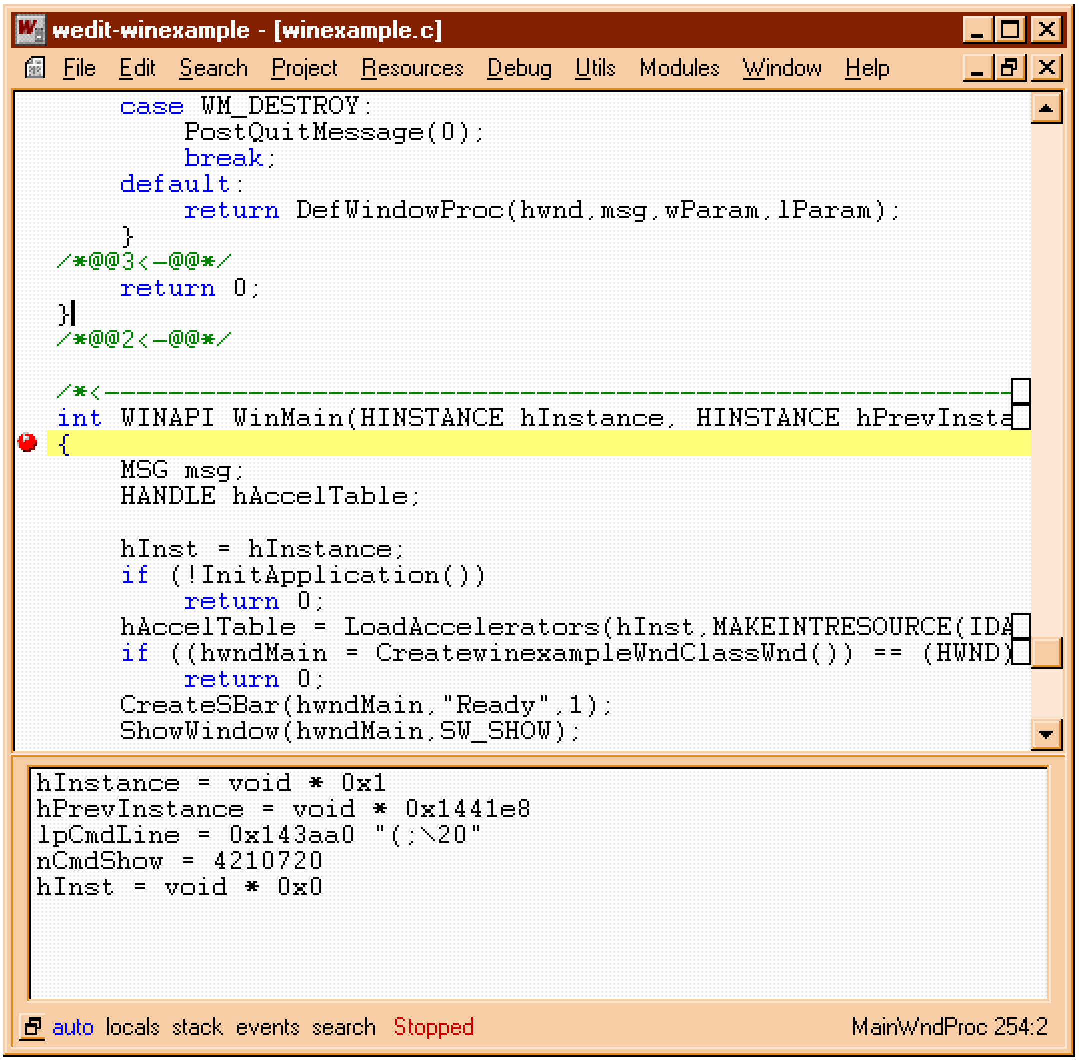
Text to searc is:

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. [103] 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.

[103] 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


wedit-winexample - [winexample.c]
File Edit Search Project Resources Debug Utils Modules Window Help
auto locals stack events search Stopped
MainWndProc 254:2

```
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd,msg,wParam,lParam);
    }
/*@@3<-@@*/
    return 0;
}
/*@@2<-@@*/

/*<-----------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInsta
{
    MSG msg;
    HANDLE hAccelTable;

    hInst = hInstance;
    if (!InitApplication())
        return 0;
    hAccelTable = LoadAccelerators(hInst,MAKEINTRESOURCE(IDA
    if ((hwndMain = CreatewinexampleWndClassWnd()) == (HWND)
        return 0;
    CreateSBar(hwndMain,"Ready",1);
    ShowWindow(hwndMain,SW_SHOW);
```

```
hInstance = void * 0x1
hPrevInstance = void * 0x1441e8
lpCmdLine = 0x143aa0 "(;\20"
nCmdShow = 4210720
hInst = void * 0x0
```

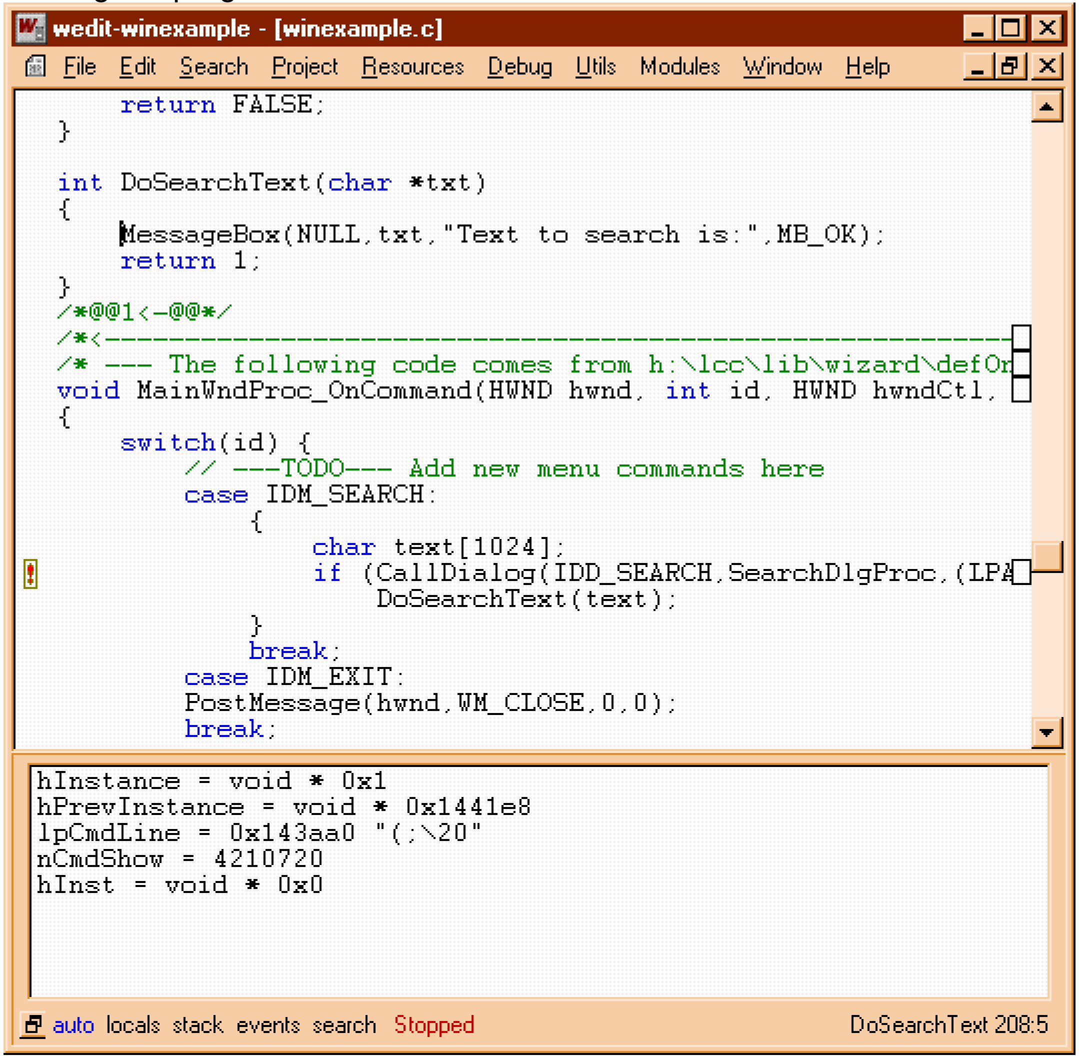
that you start it with F5 (or Debugger in the compiler menu), you can single step at the same level with F4 and trace with F8. The debugger shows you in yellow the line the program will execute next, and marks breakpoints with a special symbol at the left. Other debuggers may differ from this of course, 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.

Now, wait a minute, our window procedure that receives the message from the system is called indirectly from Windows, and we can't just follow the program blindly. If we did that, we would end up in the main loop, wasting our time.

No, we have to set a breakpoint there. We set a breakpoint when we call the dialog using the F2 accelerator. 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.

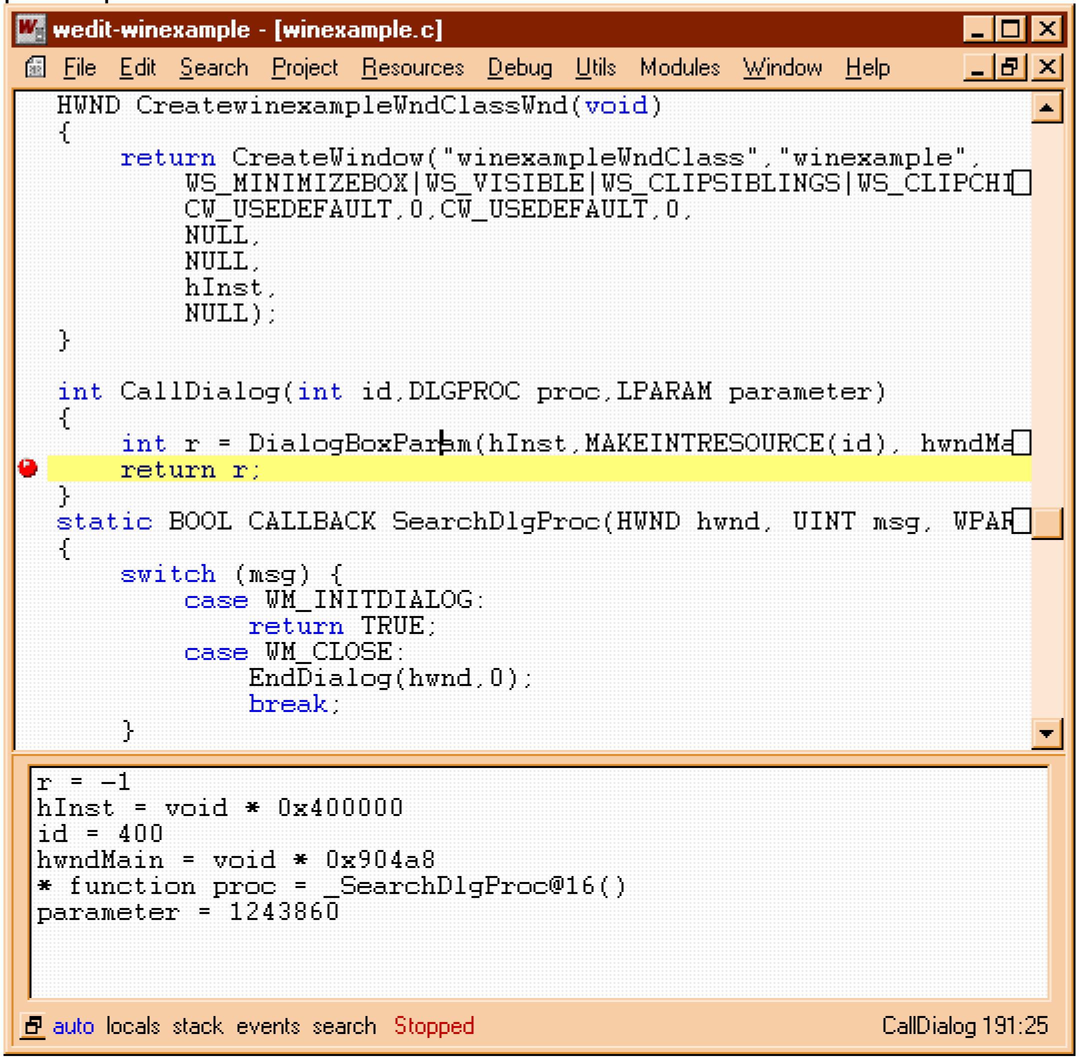

wedit-winexample - [winexample.c]
File Edit Search Project Resources Debug Utils Modules Window Help

```
        return FALSE;
    }

    int DoSearchText(char *txt)
    {
        MessageBox(NULL,txt,"Text to search is:",MB_OK);
        return 1;
    }
    /*@@1<-@@*/
    /*<------------------------------------------------------------
    /* --- The following code comes from h:\lcc\lib\wizard\defOn
    void MainWndProc_OnCommand(HWND hwnd, int id, HWND hwndCtl,
    {
        switch(id) {
            // ---TODO--- Add new menu commands here
            case IDM_SEARCH:
                {
                    char text[1024];
                    if (CallDialog(IDD_SEARCH,SearchDlgProc,(LPA
                        DoSearchText(text);
                }
                break;
            case IDM_EXIT:
            PostMessage(hwnd,WM_CLOSE,0,0);
            break;
```

```
hInstance = void * 0x1
hPrevInstance = void * 0x1441e8
lpCmdLine = 0x143aa0 "(;\20"
nCmdShow = 4210720
hInst = void * 0x0
```

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.


wedit-winexample - [winexample.c]
File Edit Search Project Resources Debug Utils Modules Window Help

```
HWND CreatewinexampleWndClassWnd(void)
{
    return CreateWindow("winexampleWndClass","winexample",
        WS_MINIMIZEBOX|WS_VISIBLE|WS_CLIPSIBLINGS|WS_CLIPCHI
        CW_USEDEFAULT,0,CW_USEDEFAULT,0,
        NULL,
        NULL,
        hInst,
        NULL);
}

int CallDialog(int id,DLGPROC proc,LPARAM parameter)
{
    int r = DialogBoxParam(hInst,MAKEINTRESOURCE(id), hwndMa
    return r;
}
static BOOL CALLBACK SearchDlgProc(HWND hwnd, UINT msg, WPAR
{
    switch (msg) {
        case WM_INITDIALOG:
            return TRUE;
        case WM_CLOSE:
            EndDialog(hwnd,0);
            break;
    }
```

```
r = -1
hInst = void * 0x400000
id = 400
hwndMain = void * 0x904a8
* function proc = _SearchDlgProc@16()
parameter = 1243860
```

Why –1?

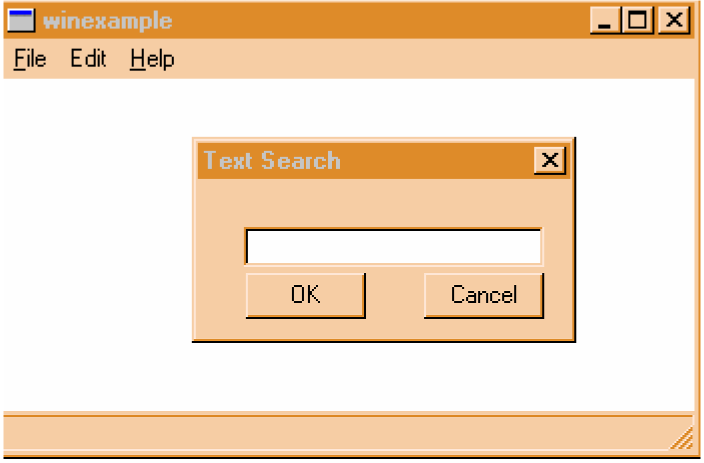
A quick look at the doc of `DialogBoxParam` tells us “If the function fails, the return value is -1.”

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.

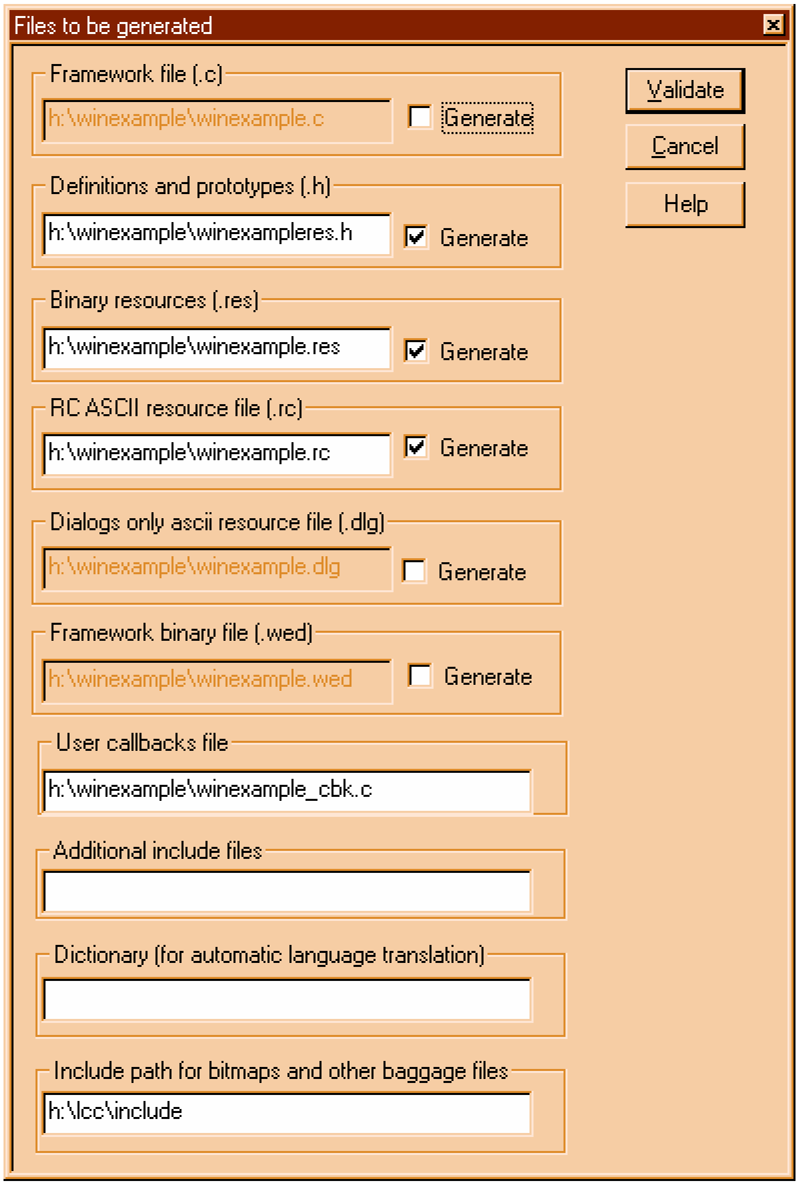
The rest is quite trivial most of the wok was done when building the DLL. Actually, the dialog box is exactly the same.[104]

[104] 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?

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:



Framework file (.c)
h:\winexample\winexample.c ☐ Generate

Definitions and prototypes (.h)
h:\winexample\winexampleres.h ☑ Generate

Binary resources (.res)
h:\winexample\winexample.res ☑ Generate

RC ASCII resource file (.rc)
h:\winexample\winexample.rc ☑ Generate

Dialogs only ascii resource file (.dlg)
h:\winexample\winexample.dlg ☐ Generate

Framework binary file (.wed)
h:\winexample\winexample.wed ☐ Generate

User callbacks file
h:\winexample\winexample_cbk.c

Additional include files

Dictionary (for automatic language translation)

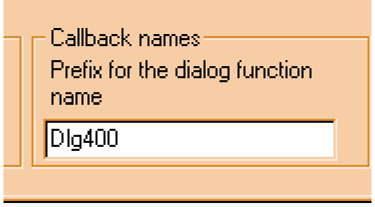
Include path for bitmaps and other baggage files
h:\lcc\include

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.[105] Choose another name like "dialogs.c" for instance.

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. At the bottom of that dialog, we find:



Callback names

Prefix for the dialog function name

Dlg400

"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.

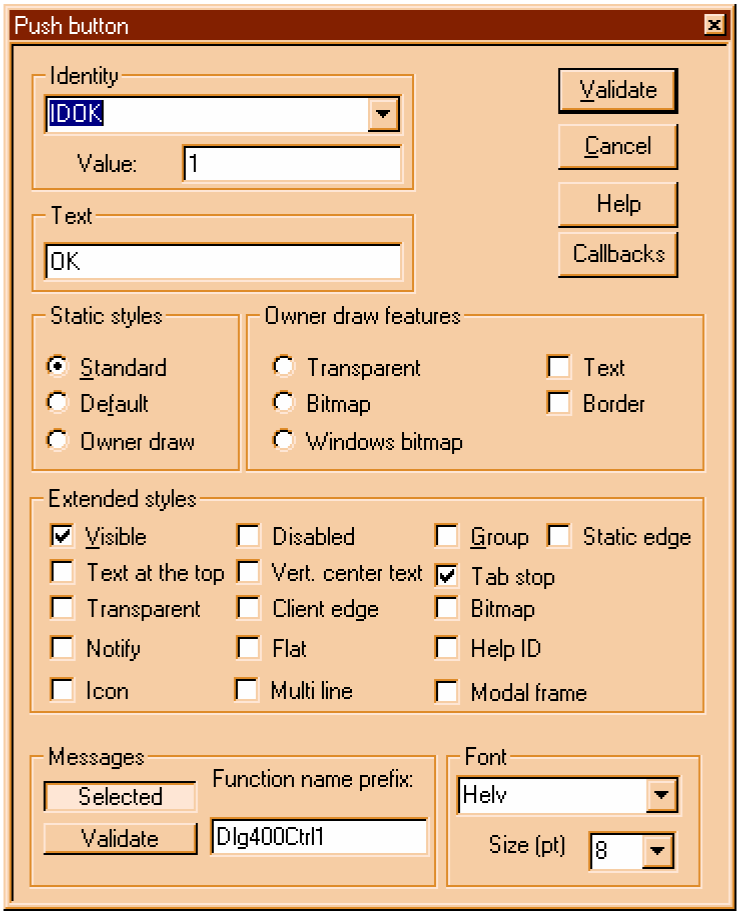
What we want to do now is to specify to the editor where are the events that interest us. For each of those events we will specify a *callback procedure* that the code generated by the editor will call when the event arrives. Basically all frameworks, no matter how sophisticated, boil down to that: a 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 for the dialog, associated with the dialog box that appears when you double click in the title, and you have the buttons properties that appear when you double click in a button.

Those dialog boxes have generally a standard or "static" 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 "messages" and appears at the bottom. That label tells us which kind of events the framework supports: window messages. There are many

[105] Newer versions of Wedit check for this. Older ones aren't so sophisticated so please take care.

events that can possibly happen of course, but the framework handles only those.



Identity: IDOK
Value: 1

Text: OK

Static styles
● Standard
○ Default
○ Owner draw

Owner draw features
○ Transparent
○ Bitmap
○ Windows bitmap
□ Text
□ Border

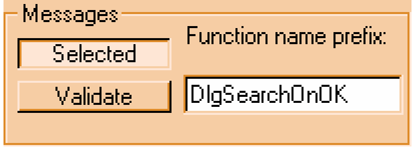
Extended styles
☑ Visible
□ Text at the top
□ Transparent
□ Notify
□ Icon
□ Disabled
□ Vert. center text
□ Client edge
□ Flat
□ Multi line
□ Group
☑ Tab stop
□ Bitmap
□ Help ID
□ Modal frame
□ Static edge

Messages
Selected
Validate
Function name prefix: Dlg400Ctrl1

Font: Helv
Size (pt) 8

We see at the bottom 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.



Messages
Selected
Validate
Function name prefix: DlgSearchOnOK

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 “DlgSearch**Init**”, and the one we attached to the OK button above, “DlgSearch**OnOkSelected**”.

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[106] and is relatively easy to modify.

[106] 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.

Understanding the wizard generated sample code

OK. Go to new, make a new project, and use all the default options for an application with a single window and a status bar. Let's see how you modify that to make it the start of *your* program.

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 MainWndProc function.[107] 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`. 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 your new identifier `IDMENU_ASK_PARAMETERS.` There you can do whatever you want to do when that menu item is clicked.

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.[108] 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.[109] 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,
          MAKEINTRESOURCE( IDD_ASK_PARAMS),
          ghwndMain,ParamsDlgProc);
```

[107] To find that easily just press F12 and click in its name in the function list.

[108] Go to "help", then click in Win32 API, get to the index and write the name of that function.

[109] Again, this is the #defined identifier of the dialog, not the dialog's title!

```
          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.

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.

Initializing the 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.

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.

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.

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.

Creating additional controls in your window without using 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.

Etc.

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.[110]
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.
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.

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_TIFF`), sound (`CF_WAVE`, `CF_RIFF`), text (`CF_TEXT`) , pen data (`CF_PENDATA`) and several others.

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,
```

[110] 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.

```
                        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.

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.

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.

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.

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.

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.

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.

Multimedia

Audio. You can use Mixers, MIDI, and waveform audio using MCI.[111]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.

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.

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.

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.

[111] **M**edia **C**ontrol **I**nterface

- Expandable null terminated strings. These strings contain variables like %PATH% that are expanded when accessed.

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 namespace, and provides users and applications with a consistent and efficient way to access and manage objects.

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

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.

Advanced C programming in lcc-win32

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.

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.

This automatic conversion of objects to their addresses is done by the compiler when a function that expects a reference is passed the whole object. For instance:

```
int fn(struct S &a);

…

struct S s;

fn(s);
```

In the call statement we pass to the function the whole structure. The compiler notices that the function expects a reference and converts internally the argument into a pointer.

Overloaded 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 after 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.

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.

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[112]
                  liBytesRemaining,
                  liMapSize,
                  liOffset;

   HANDLE hSrcFile    = INVALID_HANDLE_VALUE,
          hDstFile    = INVALID_HANDLE_VALUE,
          hSrcMap     = 0,
          hDstMap     = 0;

   BYTE * pSrc = 0,
        * pDst = 0;
```

[112] 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.

```
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
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
     liMapSize  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:

1. Open the file with CreateFile
2. Create a mapping object with CreateFileMapping using the handle you receive from CreateFile.
3. Map a portion (or all) of the file contents, i.e. create a view of it, with MapViewOfFile.

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;
}
```

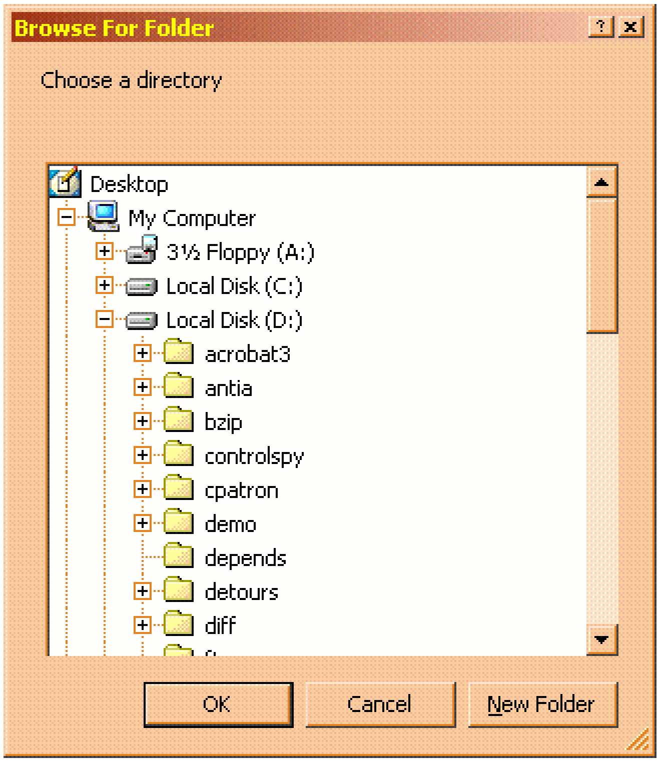
Small isn’t it?

Let’s see the gory details.

1. 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.
2. 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.
3. 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.
4. 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.

5. 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.
6. 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.
7. 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.
8. 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.[113]
9. 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.
10. 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.
11. 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.

[113] This is not really necessary. Passing a NULL value works too.



12. 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.
13. 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.
14. 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.
15. 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

Pitfalls of the C language

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!

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.

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.

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 enclose the scopes of “if” or “else” between curly braces.

5: Using the post-increment or pre-increment operators more than once in an expression. The ANSI C standard[114] 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`)

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.[115]

7: Extra Semi-colon in Macros

[114] Paragraph 6.5.2: “Expressions”: << Between the previous and the next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. >>. Then, the standards adds in a footnote: “This paragraph renders undefined statement expressions such as:

```
        i = ++i + 1;
        a[i++] = i;
```

[115] 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.

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.

Some Coding Tips

- **Determining which version of Windows the program is running**

```
BOOL  InWinNT() //test for NT
{
      OSVERSIONINFO osv;
      osv.dwOSVersionInfoSize=sizeof(osv);
      GetVersionEx(&osv);
      return osv.dwPlatformId==VER_PLATFORM_WIN32_NT;
}
```

- **Translating the value returned by GetLastError() into a readable string**

```
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);
}
```

- **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)

- **Getting a pointer to the stack**

To get a pointer to the stack, use the following code:

```
   int MyFunction()
   {
   int x;
   int *y = &x;
   }
```

NOTE: This pointer will not be valid when the function is exited, since the stack contents will change.

- **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
                       );
```

- **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);
}
```

- **Capturing and printing the contents of a 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 you 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);
}
```

- **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);
}
```

- **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.
}
```

- **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.

- **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!

- **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);
```

- **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);
```

- **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.

- **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;
}
```

- **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.

How to find more information. Overview of lcc-win32's documentation

The documentation of lcc-win32 comes in four files:

- **lccdoc.exe**. This file contains the following documentation:
    1. C-tutorial.doc. This document.
    2. Manual.doc. 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.
    3. 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.
    4. Mmx.doc. This small document explains how to use the MMX intrinsic functions in your programs, to access those relatively new instructions of the Pentium processors.

Other documentation is available as .hlp files shipped with the main file lccwin32.exe. Specifically:

5. **wedit.hlp**. Here are the standard library functions documentation, and the online-help for the IDE.

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

6. Win32.hlp. Here you will find the documentation of the windows API.

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.

Newsgroups

Internet newsgroups are a great way of sharing information. There is an lcc newsgroup comp.compilers.lcc. Here is a question that appeared in another interesting newsgroup: comp.std.c that shows an interesting discussion:

From: serin_d@my-deja.com
I need to write an algorithm in C, to:
1) determine the most significant set bit
2) determine the least significant set bit
in a byte/int/long whatever. I am looking for an efficient algorithm that does not involve iterating through every bit position individually.
Cheers
Serin

Many people answered, and the discussion about which algorithm to use was a very informative one.

From: pornin@bolet.ens.fr (Thomas Pornin)
Organization: Ecole Normale Superieure, Paris
Notwithstanding the problem of knowing the exact size of a type in standard C (you would have better luck with unsigned types, by the way), use a dichotomy:

```
/*
  * Returns the least significant bit in the 32-bit value stored in x
  * (return value from 0 to 31; 32 if no bit is set)
  */
int least_significant_set_bit(unsigned long x)[116]
{
        int t = 0;

        if (x & 65535UL == 0) { t += 16; x >>= 16; }
        if (x & 255UL == 0) { t += 8; x >>= 8; }
        if (x & 15UL == 0) { t += 4; x >>= 4; }
        if (x & 3UL == 0) { t += 2; x >>= 2; }
        if (x & 1UL == 0) { t += 1; x >>= 1; }
        if (!x) t ++;
        return t;
}
```

[116] 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.

For completeness: comp.std.c is about the C standard. Your question would be better addressed in comp.lang.c.[117]

Another participant posted a different version of this algorithm. Here it is:

From: "Douglas A. Gwyn" <DAGwyn@null.net>
There is no Standard C function for this (traditionally called "find first one bit"). Followup has been set accordingly. To get that thread started off, here is a scheme that you might consider:
For example assume a 64-bit word:
If the whole word is 0, return a failure-to-find indication.
Set bit location accumulator to `0` and total mask to `0xFFFFFFFFFFFFFFFF`. Mask word with `0xFFFFFFFF00000000` to see if first one bit is in the left half of the word; if so, add 32 to the bit location accumulator and update total mask by ANDing with this mask, else update total mask by ANDing with the complement (~) of this mask. (This first mask update step can be simplified by omitting the initialozations and just storing 32 or 0 and the appropriate mask.)

Mask with total mask and 0xFFFF0000FFFF0000 to see if " is in left half of whatever half was just determined; if so add 16 to accumulator and update total mask by ANDing with this mask, else update total mask by ANDing with the complement (~) of this mask.
Mask with total mask and `0xFF00FF00FF00FF00` to see if " " " add 8 " "
....
Mask with `0xAAAAAAAAAAAAAAAA` to see if is in odd # bit position; if so add 1 (last mask update is not necessary). The above can be done in a compact loop, but since you're worried about efficiency the loop should be completely unrolled and the parenthesized optimizations made. Accumulator now contains bit location (counting from right starting with 0). If you performed the final mask update, the total mask is now the isolated first one bit.

Of course, the last one bit can be found in a similar fashion.
Now that the general idea is exposed, try to find optimizations. For example, instead of masking the original word with total mask and new mask each time, update the original word by masking it with the new contribution to the total mask and don't maintain a total mask variable at all.
--
comp.lang.c.moderated - moderation address: clcm@plethora.net

This is surely an improvement over the first algorithm, since the shifts are gone.

Another answer was the following:

From: Francis Glassborow <francis.glassborow@ntlworld.com>

[117] I have to disagree with Thomas. comp.lang.c is quite boring, full of empty discussions very often. There **are** good discussions of course, but there is a lot more noise.

Subject: Re: Most significant bit algorithm
Before giving any guidelines to a solution, note that this was the wrong place to ask, you should have posted to comp.lang.c.moderated.[118]
The answers are likely to be different for the different size types. In addition, some possible 'solutions' are subject knowing the endianess of your system.
For an 8-bit byte, consider masking in the following order:

```
bits 7 & 0
bits 6 & 1
bits 5 & 2
bits 4 & 3
```

The first of these to be non-zero tells you that either one or both of the bits you want to locate have been determined. Overall that is likely to provide little advantage over mask each bit and test.
For other types, as long as you know their layout in bytes (unsigned char) use a union to map the value to an array of unsigned char. Now test the individual chars against zero. The highest and lowest non-zero byte can now be tested for the requisite bit. However if you really have a need for maximum efficiency and can sacrifice portability to this end, consider writing an assembler code routine.[119]

Another contribution was:

From: Conor O'Neill <conor.oneill@aethos.co.uk>
Organization: Speaking for myself

Least significant bit is fairly easy:

```
unsigned int least_significant_bit(unsigned int x)[120]
{
        return x & ~(x - 1);
}
```

A little bit of playing with examples written out in binary should convince you that it is correct. It even works with x == 0 (i.e., it returns 0). I don't know if there is a similar algorithm for the most significant bit.

I don’t either. But it is surely correct I tested it.
Another answer was:

From: "Peter L. Montgomery" <Peter-Lawrence.Montgomery@cwi.nl>
Organization: CWI, Amsterdam

[118] Yes, moderated groups have less noise.
[119] See the lcc-win32 solution at the end of the discussion.
[120] The lcc compiler needs to know if a given number is a power of two. It uses this construct too.

In article dkeisen@best.com () writes:
>Mathew Hendry <math@vissci.com> wrote:
>>serin_d@my-deja.com wrote:
>>: I need to write an algorithm in C, to [...] determine the least
>In 15 years of programming, I've never had to do anything
>like this. Not once.
>Sure looks to me like y'all did someone's homework for him.

I write number theoretic codes. Locating the rightmost and leftmost significant bits in a nonzero word are two of the 20 or so important primitives needed for more complicated algorithms. If a programming language supports “AND” and “OR” on bits, it should support these primitives too. Alas, few do.

For example, the left-to-right binary method of exponentiation starts with the most significant bit of a number and proceeds downward.

A more complicated example is a binary GCD (greatest common divisor). Assume uint64 is an unsigned 64-bit type:

```
uint64 GCD(uint64 x, uint64 y)
{
      int nzero;
      uint64 x1, y1;

      if (x == 0 || y == 0) return x | y;
      nzero = trailing_zero_count(x | y); /* Shared power of 2 */
      x1 = x >> trailing_zero_count(x);
      y1 = y >> trailing_zero_count(y);
      while (x1 != y1)  {     /*  Both are odd  */
            if (x1 > y1) {
                  x1 = (x1 - y1) >> trailing_zero_count(x1 - y1);
            } else {
                  y1 = (y1 - x1) >> trailing_zero_count(y1 - x1);
            }
      }
      return x1 << nzero;
}
```

This short program has five references to a function locating the least significant bit. No wonder that processors such as the Intel Pentium and Alpha 21264 have hardware instructions to locate the rightmost and leftmost significant bits in a word. Optimizing these is much more than homework.

What is the solution in lcc-win32?

Easy:

```
int getmsb(unsigned int n) /* returns position of most significant bit */
{
       return _bsf(n);
}
```

```
int getlsb(unsigned int n) /* returns position of least significant bit */
{
      return _bsr(n);
}
```

The functions _bsf (bit scan forward) and _bsr (bit scan reverse) are intrinsics. Those functions will be mapped by the compiler directly into assembly, in this case to the machine instructions BSF and BSR that all Intel compatible PCs have. This is extremely efficient, since they will just issue:

```
      bsr    eax,eax
```

The number to be searched will be placed in the eax register by the compiler, and the machine will leave the result in it. This is conceptually the same as calling a function, but takes just a few cycles, depending on the bit pattern of the number.

Appendix 1: File types

Here is a list of the extensions you will find often under windows. Note that this is not an exhaustive list. I have tried to concentrate in extensions that are related to programming.
Note too that all these are conventions. If you call your source files “source.drv”, the source will compile anyway.

Extension	Type	Description
BAT	Source code	Batch command file.
BMP	Resource	Bitmap file.
C	Source code	C definitions.
CAB	Binary	Microsoft cabinet files: several files in one.
CHM	Help file	Compiled HTML. Used by Microsoft help system.
CPP	Source code	C++ source file
CSV	Data base	Comma Separated Values file
CUR	Resource	File containing a cursor image.
DB	Data base	Paradox files
DBF	Data base	dBase files
DBG	Symbol file	Contains information about the symbols contained in system DLLs.
DEF	Resource	Definitions file containing the list of exported functions in a DLL. Used by the linker.
DLG	Resource	ASCII description of dialog resources. Generated by the resource editor.
DLL	Executable	Compiled executable code that is loaded when needed into a running process.
DOC	Documents	Microsoft Word document.
DRV	Driver	Compiled device driver.
EXE	Executable	Executable containing compiled code, resources, etc. You can inspect its contents using the pedump utility..
EXP	Exports	ASCII list of exported functions from a DLL. Used by the “buildlib” utility to build import libraries. This is specific to lcc-win32.
FNT	Font typeface	File containing a single font description.
FON	Font lib	File containing one or more fonts.
H	Source code	Header file containing C definitions.
HLP	Help file	Compiled help file. Generated with the help compiler hcw. You can inspect it with pedump, and see the help with winhlp32.exe.
HPJ	Resource	Microsoft help compiler project file
IDL	Source code	Interface Description Language file. Used by Microsoft tools
ILK	Temp file	Produced by Microsoft Linker when incremental linking is selected.
INC	Source code	Include files for the assembler or other Microsoft tools
INI	Initialization file	Contains initialization data in text form.
INF	Source code	Information file for the setup utility.
ICO	Resource	File containing an icon image.
LIB	Library file	Contains several object files. Used by the linker.
MAK	Source code	Makefile extension.
MDB	Data base	Microsoft Access files
OBJ	Compiled code	Result of the compilation of a single C file. Contains code and (possibly) debug information.
OCX	Compiled code	Visual basic DLL.
ODL	Source code	Object Description Language file. Used by Microsoft tools

PRJ	Project	Text file containing the project description. Generated by wedit.
RC	Source code	ASCII resource description.
RES	Resource	Compiled (binary) resource file. You can edit it using weditres.
RTF	Rich text	Rich text file. Can be used to compile a help file using hcw, the help compiler.
TLB	Resource	COM type library. You can see its contents with pedump.
TPL	Source code	Templates used by lcc-win32's wizard.
VBS	Source code	Visual Basic script.
WAV	Resource	Sound files
WED	Resource	Binary project description used by the resource editor. Lcc-win32 specific.
XLS	Data base	Microsoft Excel files

Appendix 2: Programs distributed with lcc-win32

The full description of these programs, together with the arguments they accept is in the user manual.

Program name	**Purpose**
lc.exe	Complation driver. Compiles and then calls the linker.
Lcc.exe	Compiler. Includes preprocessor, compiler and assembler.
Lrc.exe	Resource compiler.
Lcclnk.exe	Linker.
Wedit.exe	Integrated Development Environment (IDE)
Pedump.exe	Binary file dumper
Buildlib.exe	Makes import libraries
Lcclib.exe	Builds normal libraries from several object files.
Weditres.exe	Stand-alone resource editor.
Gc.dll	Memory manager DLL.
Make.exe	Build utility for a project.
Mc.exe	Message compiler
Wizard.exe	Project skeleton generator.
Browsegen.exe	Generates browse information
Iedit.exe	Image editor
Dynloader.dll	Dynamic code loader: loads an object file into a running program.
Bind.exe	Binds an object file to be loaded later by the dynamic loader.
Rundos.exe	Runs a console application. Used by the IDE.

Appendix 3: Useful libraries of lcc-win32

Library	**Purpose**
complex.lib	Complex number arithmetic with all standard complex functions.
bignum.lib	Arbitrary precision library
tcconio.lib	Library of console functions compatible with the old Turbo C library.
scrnsave.lib	Screen saver library, allowing you to write one.
regexp.lib	Regular expression pattern matching.
fdlibm.lib	Complete math library
weditres.lib	Resource editor run time
iostream.lib	Operator overloading for >> in a C++ compatible way.
gdbm.lib	Gnu's database library

Appendix 4: The window tree: full source code

WindowTree.c

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <string.h>
#include <psapi.h>
#include "windowtreeres.h"
#define IDTREEWINDOW 10545
HINSTANCE hInst;                // Instance handle
HWND hwndMain;          //Main window handle

LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM
lParam);

// Global Variables for the status bar control.
HWND  hWndStatusbar;

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;
}

void UpdateStatusBar(LPSTR lpszStatusString, WORD partNumber, WORD
displayFlags)
{
        SendMessage(hWndStatusbar,
                SB_SETTEXT,
                partNumber | displayFlags,
                (LPARAM)lpszStatusString);
}

void InitializeStatusBar(HWND hwndParent,int nrOfParts)
{
        const int cSpaceInBetween = 8;
        int   ptArray[40];
        RECT  rect;
        GetClientRect(hwndParent, &rect);
        ptArray[nrOfParts-1] = rect.right;
        SendMessage(hWndStatusbar,
                SB_SETPARTS,
                nrOfParts,
                (LPARAM)(LPINT)ptArray);
}


static BOOL CreateSBar(HWND hwndParent,char *initialText,int nrOfParts)
{
        hWndStatusbar = CreateStatusWindow(WS_CHILD | WS_VISIBLE |
             WS_BORDER|SBARS_SIZEGRIP,
                initialText,
                hwndParent,
                IDM_STATUSBAR);
        if(hWndStatusbar)
        {
                InitializeStatusBar(hwndParent,nrOfParts);
                return TRUE;
        }

        return FALSE;
}

static BOOL InitApplication(void)
{
        WNDCLASS wc;

        memset(&wc,0,sizeof(WNDCLASS));
        wc.style = CS_HREDRAW|CS_VREDRAW |CS_DBLCLKS ;
        wc.lpfnWndProc = (WNDPROC)MainWndProc;
        wc.hInstance = hInst;
        wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
        wc.lpszClassName = "windowtreeWndClass";
        wc.lpszMenuName = MAKEINTRESOURCE(IDMAINMENU);
        wc.hCursor = LoadCursor(NULL,IDC_ARROW);
        wc.hIcon = LoadIcon(NULL,IDI_APPLICATION);
        if (!RegisterClass(&wc))
                return 0;
        /*@@0<-@@*/
        // ---TODO--- Call module specific initialization routines here

        return 1;
}

HWND CreatewindowtreeWndClassWnd(void)
{
        return CreateWindow("windowtreeWndClass","windowtree",

        WS_MINIMIZEBOX|WS_VISIBLE|WS_CLIPSIBLINGS|WS_CLIPCHILDREN|WS_MAX
IMIZEBOX|WS_CAPTION|WS_BORDER|WS_SYSMENU|WS_THICKFRAME,
                CW_USEDEFAULT,0,CW_USEDEFAULT,0,
                NULL,
                NULL,
                hInst,
                NULL);
```

```
}

void Scan(HWND hTree,HTREEITEM hTreeParent,HWND Start)
{
        HWND hwnd = Start,hwnd1;
        TV_INSERTSTRUCT TreeCtrlItem;
        HTREEITEM htiNewNode;

        while (hwnd != NULL) {
                char bufTxt[256],bufClassName[256],Output[1024];
                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);

        }
}

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);
        Start = GetWindow(Start,GW_CHILD);
        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);
        Scan(hTree,hNewNode,Start);
        TreeView_Expand(hTree,hNewNode,TVE_EXPAND);
        SendMessage(hTree,WM_SETREDRAW,1,0);
        return 1;

}
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;
        }
}
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);
}

void SetTextInStatusBar(HWND hParent,HWND hwnd)
{
        RECT rc;
        HANDLE pid,thread;
        char info[4096];

        GetWindowRect(hwnd,&rc);
        thread = (HANDLE)GetWindowThreadProcessId(hwnd,&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,PrintProcessNameAndID((ULONG)pid));
        UpdateStatusBar(info, 0, 0);
}
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;
}

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);
}

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_NOTIFY:
                return HandleWmNotify(hwnd,wParam,lParam);
        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;
        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;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, INT nCmdShow)
{
        MSG msg;
        HANDLE hAccelTable;

        hInst = hInstance;
        if (!InitApplication())
                return 0;
        hAccelTable = LoadAccelerators(hInst,MAKEINTRESOURCE(IDACCEL));
        if ((hwndMain = CreatewindowtreeWndClassWnd()) == (HWND)0)
                return 0;
        CreateSBar(hwndMain,"",1);
        ShowWindow(hwndMain,SW_SHOW);
        PostMessage(hwndMain,WM_COMMAND,IDM_NEW,0);
        while (GetMessage(&msg,NULL,0,0)) {
                if (!TranslateAccelerator(msg.hwnd,hAccelTable,&msg)) {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }
        return msg.wParam;
}
```

WindowTreeRes.h

```
#define IDACCEL 100
#define IDM_NEW 200
#define IDM_OPEN        210
#define IDM_SAVE        220
#define IDM_SAVEAS      230
#define IDM_CLOSE       240
#define IDM_PRINT       250
#define IDM_PAGESETUP   260
#define IDM_EXIT        300
#define IDM_ABOUT       500
#define IDMAINMENU      600
#define IDAPPLICON      710
#define IDAPPLCURSOR    810
#define IDS_FILEMENU    2000
#define IDS_HELPMENU    2010
#define IDS_SYSMENU     2030
#define IDM_STATUSBAR   3000
```

```
Compiling from the command line:
lc windowtree.c psapi.lib
```