Introduction to C using lcc-win

jacob navia

Contents

1 Introduction to C

This book supposes you have the lcc-win compiler system installed. You will need a compiler anyway, and lcc-win 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.de

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.

But before we start, just a quick answer to the question:

1.1 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-win has software that was written many years ago, by many people, among others by Dennis Ritchie, the creator of the language itself. 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. 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.

C is not a functional language but you can do functional programming with it if you feel like. See the "Illinois FP" language implementations in C, and many other functional programming languages that are coded in C.

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 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.[1]

As languages come and go, C remains. It was at the heart of the UNIX operating system development in the seventies, 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. Today, the linux kernel is written completely in C together with many other operating systems, window systems, and many other applications.

1.2 Program organization

A program in C is written in one or several text files called source modules. Each of those modules is composed of functions, i.e. smaller pieces of code that accomplish some task, 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.

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. Each compilation unit defines a name space, i.e. a scope. Within this name space each name is unique and defines only one object.

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

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 flavours: system-wide, furnished with lcc-win, and private, specific to the application you are building.

Each module has in general one or several functions, i.e. pieces of code that accomplish some task, reading some data, performing some calculations, or organizing several other functions into some bigger aggregate. There is no distinction between functions and procedures in C. A procedure is a function of return type void.

A function has a parameter list, a body, and possibly a return value. The body can contain declarations for local variables, i.e. variables activated when execution reaches the function body. The body is a series of expressions separated by semicolons. Each statement can be an arithmetic operation, an assignment, a function call, or a compound statement, i.e. a statement that contains another set of statements.

[1] Lisp and scheme, two list oriented languages featured automatic garbage collection for decades. APL and other interpreters offered this feature too. Lcc-win offers you the garbage collector developed by Hans Boehm.

1.3 Hello

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

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’’); }
```

In today's C the above program would be:

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

Note that obviously the numbers in parentheses are not part of the program text but are in there so that I can refer to each line of the program.

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. You will notice that 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, for instance `#include "myfile.h"`

2) We define a function called "main" that returns an integer as its result, and receives no arguments (void). All programs in C have a function called main, and it is here that all programs start. The "main" function is the entry-point of the program.

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

4) We call the standard function "printf" that formats its arguments and displays them 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’’`. Character strings in C are enclosed in double quotes, and 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`. See page 50 for more on character string constants, page 54 for function call syntax.

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 file extension. You can create those text files with any editor that you want, but lcc-win proposes a

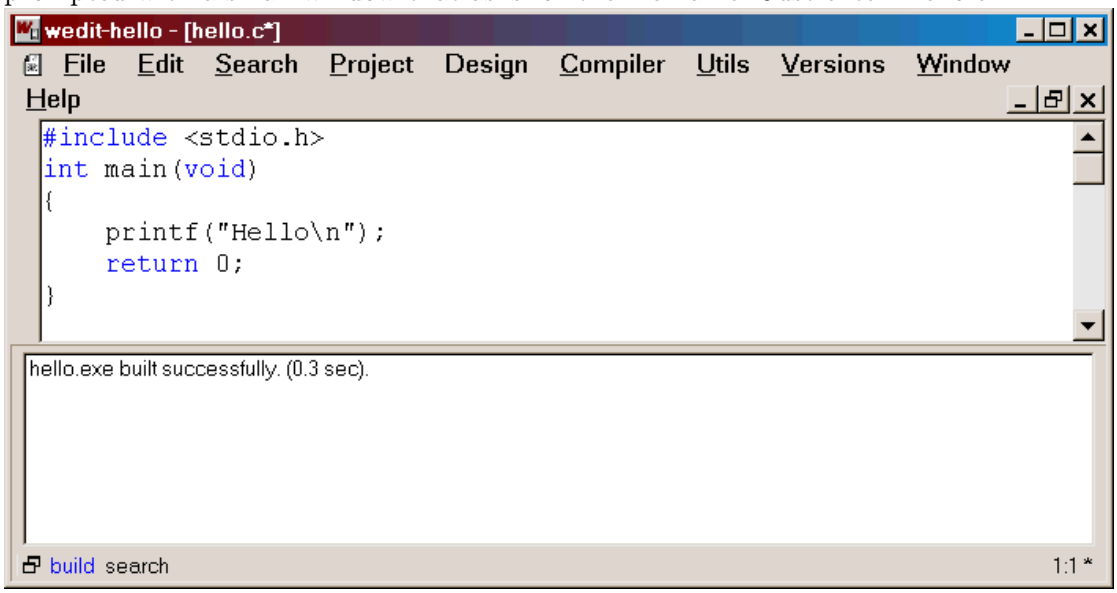
specialized editor for this task called "Wedit". This program allows you to enter the program text easily, since it is adapted to the task of displaying C source text. To make this program then, we start Wedit and enter the text of that program above.

Program input

If you know how an integrated development environment (IDE) works, you can skip this section.

When you click in the icon of lcc-win, you start a program designed to make it easy for you to type your program and to find information about it. When you start it for the first time, it will display a blank window, expecting that you tell it what it should do.

The first thing to do is to go to the "File" menu, and select the New-> File item. This will indicate to the IDE that you want to start a new program module. You get prompted with a small window that asks for the file name. Just enter "hello.c".



You will see than that a blank sheet of paper opens, where you can enter the text of the program. You should type the program as shown and pay attention to avoid any typing mistake. Remember: the machine doesn't understand anything. If you forget a quote, or any special sign it will not work and the compiler will spit error messages that can be confusing. Check that you type exactly what you see above.

Once this is done, you can compile, and link-edit your program by just clicking in the compile menu or pressing F9.[2]

[2]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-win went wrong and Wedit can't find the compiler executable, or many other reasons. If you see an error message

To run the program, you use the "execute" option in the "Compiler" menu (or you type 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 the IDE (Wedit) about it using the "Executable stats" option in the "Utils" menu. We get the following display.

We see at the first line of the bottom panel, that the program executable is called: `h:\lcc\projects\hello.exe`.
We open a command shell window, and type the command:

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

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

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

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

What are "function parameters" ?

When you have a function like:

```
int fn(int a) { ... }
```

the argument (named a) is copied into a storage area reserved by the compiler for the functions arguments. Note that the function fn will use only a copy, not the original value. For instance:

```
int fn1(int a)
{
    a = a+7;
    return a;
}
int fn2(void)
```

please do not panic, and try to correct the error the message is pointing you to.

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

```
{
    int b = 7;
    fn1(b);
    return b;
}
```

The fn2 function will always return 7, because function fn1 works with a copy of b, not with b itself. This is known as passing arguments by value. This rule will not be used for arrays, in standard C. When you see a statement like:

```
printf("Hello\n");
```

it means that the address of the first element is passed to "printf", not a copy of the whole character array. This is of course more efficient than making a copy, but there is no free lunch. The cost is that the array can be modified by the function you are calling. More about this later.

Console mode programs and windows programs

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

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

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

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

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

1.4 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, that has the same name as the source file but a .obj extension under windows, or a .o extension under linux. 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.

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

  The resulting file is called hello.i. The `i` means intermediate file.

- The front end of the compiler proper processes the resulting text. Its task is to generate a series of intermediate code statements. 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.

- The code generator takes those intermediate instructions and emits assembler instructions from them. Assembly code can be generated with the `lcc -S hello.c` command. The generated assembly file will be called hello.asm. The generated file contains a listing of the C source and the corresponding translation into assembly language.

- The assembler takes those assembly instructions and emits the encodings that the integrated circuit can understand, and packages those encodings in a file called object file that under Windows has an `.obj` extension, and under Unix a `o, 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 make this easier, the IDE will do all of this with the F9 function key.

1.4.1 The run time environment

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

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

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

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

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

The printf function deposits characters in the input buffer of the terminal emulation program, that makes the necessary bits change color using the current font, and at the exact position needed to display each glyph. Windows calls the graphic drivers in your graphic card that control the video output of the machine with those bits to change. The bits change before your hand has had the time to move a millimeter. Graphic drivers are fast today, and in no time they return to windows that returns control to the printf function.

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

Your hand is still near the return key.

We have the following phases in this process:

- Design-time. We wrote the program first.
- Compile-time. We compiled our design.
- Run-time. The compiled instructions are started and the machine executes what we told it to do.

We wrote the program first

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

We compiled our design

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

Run time

The operating system loads the prepared sequence of instructions from the disk into main memory, and passes control to the entry point. This is done in several steps. First the main executable file is loaded and then all the libraries the program needs. When everything has been mapped in memory, and all the references in each part have been resolved, the OS calls the initialization procedures of each loaded library. If everything goes well, the OS gives control to the program entry point.

1.5 An overview of the standard libraries

You remember that we stressed that in our hello.c program you should include the stdio.h system header file. OK, but how do you know which header file you need?

You have to know which header declares which functions. These headers and the associated library functions are found in all C99 compliant compilers.

Header	Purpose
assert.h	Diagnostics for debugging help.
complex.h	Complex numbers definitions.
ctype.h	Character classification (isalpha, islower, isdigit)
errno.h	Error codes set by the library functions
fenv.h	Floating point environment. Functions concerning the precision of the calculations, exception handling, and related items. See page 228
float.h	Characteristics of floating types (float, double, long double, qfloat). See page 216
inttypes.h	Characteristics of integer types
iso646.h	Alternative spellings for some keywords. If you prefer writing the operator `&&` as `and`, use this header.
limits.h	Size of integer types.
locale.h	Formatting of currency values using local conventions.
math.h	Mathematical functions.
setjmp.h	Non local jumps, i.e. jumps that can go past function boundaries. See page 87.
signal.h	Signal handling. See page 196.

stdbool.h	Boolean type and values
stddef.h	Defines macros and types that are of general use in a program. NULL, offsetof, ptrdiff_t, size_t, and several others.
stdint.h	Portable integer types of specific widths.
stdio.h	Standard input and output.
stdlib.h	Standard library functions.
string.h	String handling. Here are defined all functions that deal with the standard representation of strings as used in C. See "Traditional string representation in C" on page 138.
stdarg.h	Functions with variable number of arguments are described here. See page 55.
time.h	Time related functions.See page 165.
tgmath.h	Type-generic math functions
wchar.h	Extended multibyte/wide character utilities
wctype.h	Wide character classification and mapping utilities

The "stdheaders.h" include file

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

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

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

1.5.1 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 to access its arguments.

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

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

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

1. We include again stdio.h

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

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

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

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

   `argv[1]` The first argument, i.e. "foo".

   `argv[2]` The second argument, i.e. "bar".

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

3. 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];`

4. We use the "for" construct, i.e. an iteration. See the explanations page 23.

5. 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. In the character string passed there are two directives a `%d` and a `%s`. The first

one 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.

6. We finish the scope of the for statement with a closing brace. This means, the iteration body ends here.

Now we are ready to run this program. Suppose that we have saved the text of the program in the file "args.c". We do the following: [3]

```
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:

```
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
```

The name of the program changed from "args" to args.exe", its full name. We can even write:

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

Now the full path is displayed. 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.

[3] We use the toolsdir >www for compiling 32 bits programs. If you want to use the 64 bit tools use `lcc64` and `lcclnk64`.

Implementation details

The arguments are retrieved from the operating system by the code that calls 'main'. Some operating systems provide a specific interface for doing this; others will pass the arguments to the startup. Since C can run in circuit boards where there is no operating system at all, in those systems the 'main' function will be defined as always int main(void).

1.6 Iteration constructs

We introduced informally the "for" construct above, but a more general introduction to loops is necessary to understand the code that will follow. There are three iteration constructs in C: "for", "do", and "while".

1.6.1 for

The "for" construct has

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

The general form is then:

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

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

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

```
i=1
outer i is 1
i=0
i=1
```

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

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

        for (int i = 0;i<2;i++) {        3
            printf("i=%d\n",i);          4
        }                                5
    }                                    6
    return 0;                            7
}
```

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

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

Note that for each new scope, the identifiers of the same name are shadowed by the new ones, as you would normally expect in C. When you declare variables in the first part of the for expression, note that you can add statements and declarations, but after the first declaration, only declarations should follow. For instance, if you have:

```
struct f {int a,b};
struct f StructF;
...
for (StructF.a = 6,int i=0; i<10;i++)
is allowed, but NOT
for (int i=0, StructF.a = 67; i<10; i++) // Syntax error
```

1.6.2 while

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

The general form is:

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

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

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

1.6.3 do

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

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

1.6.4 break and continue

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 start of the statement block. The loop continues normally, only the statements between the continue keyword and the end of the loop are ignored.

1.7 Types

A machine has no concept of type, everything is just a sequence of bits, and any operation with those sequences of bits can be done, even if it is not meaningful at all, for example adding two addresses, or multiplying the contents of two character strings.

A high level programming language however, enforces the concept of types of data. Operations are allowed between compatible types and not between any data whatsoever. It is possible to add two integers, or an integer and a floating point number, and even an integer and a complex number. It is not possible to add an integer to a function or to a character string, the operation has no meaning for those types.

An operation implies always compatible types between the operands or a conversion from two incompatible types to make them compatible. It is not possible to multiply a number with a character string but is possible to transform the contents of a character string into a number and then do a multiplication. These conversions can be done automatically by the compiler (for instance the conversion between integers and floating point data) or explicitely specified by the programmer through a cast or a function call.

In C, all data must be associated with a specific type before it can be used. All variables must be declared to be of a known type before any operation with them is attempted since to be able to generate code the compiler must know the type of each operand. C is statically typed.

C allows the programmer to define new types based on the previously defined ones. This means that the type system in C is static, i.e. known at compile time, but extensible since you can add new types.

This is in contrast to dynamic typing, where no declarations are needed since the language associates types and data during the run time. Dynamic typing is much more flexible, but this flexibility has a price: the run time system must constantly check the types of the operands for each operation to see if they are compatible. This run-time checking slows down the program considerably.

In C there is absolutely no run time checking in most operations, since the compiler is able to check everything during the compilation, which accelerates the execution of the program, and allows the compiler to discover a lot of errors during the compilation instead of crashing at run time when an operation with incompatible types is attempted.

1.7.1 What is a type?

A first tentative, definition for what a type is, could be "a type is a definition of the format of a sequence of storage bits". It gives the meaning of the data stored in memory. If we say that the object a is an int, it means that the bits stored at that location are to be understood as a natural number that is built by consecutive additions of powers of two. If we say that the type of a is a double, it means that the bits are to be understood as the IEEE 754 standard sequences of bits representing a double precision floating point value.

A second, more refined definition would encompass the first but add the notion of "concept" behind a type. For instance in some machines the type `size_t` has exactly the same bits as an unsigned long, yet, it is a different type. The difference is that we store sizes in `size_t` objects, and not some arbitrary integer. The type is associated with the concept of size. We use types to convey a concept to the reader of the program.

A wider definition is that a type is also a set of operations available on it. Numeric types define the four operations, boolean data defines logical operations, etc. Some people would say that it is the set of operations that defines a type [4].

[4] What does the standard writes about types? In §6.2.5 it writes:

> The meaning of a value stored in an object or returned by a function is determined by the type of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.)

The base of C's type hierarchy are machine types, i.e. the types that the integrated circuit understands. C has abstracted from the myriad of machine types some types like 'int' or 'double' that are almost universally present in all processors. There are many machine types that C doesn't natively support, for instance some processors support BCD coded data but that data is accessible only through special libraries. C makes an abstraction of the many machine types present in many processors, selecting only some of them and ignoring others.

It can be argued why a type makes its way into the language and why another doesn't. For instance the most universal type always present in all binary machines is the boolean type (one or zero). Still, it was ignored by the language until the C99 standard incorporated it as a native type [5].

Functions have a type too. The type of a function is determined by the type of its return value, and all its arguments. The type of a function is its interface with the outside world: its inputs (arguments) and its outputs (return value).

Each type can have an associated pointer type: for int we have int pointer, for double we have double pointer, etc. We can have also pointers that point to an unspecified object. They are written as void *, i.e. pointers to void [6].

Types in C can be incomplete, i.e. they can exist as types but nothing is known about them, neither their size nor their bit-layout. They are useful for encapsulating data into entities that are known only to certain parts of the programor for partially defining types that can be fully defined later in the program. Example:

```
struct MyData;
```

Nothing is known about the internal structure of `MyData`. In most cases the module handles out pointers to those incomplete structures.

This construct is there to allow you to implement a strong barrier between each module that uses those types since all users of those hidden types can't allocate them (their size is unknown) or access the internal structure since it is unknown.

Information hiding is a design principle that stresses separation and modularity in software construction by avoiding different modules to depend too much on each other. In this specific case, an incomplete type allows the structure of a hidden type to evolve freely without affecting at all the other parts of the program that remain tied only to the specified functional interface.

1.7.2 Types classification

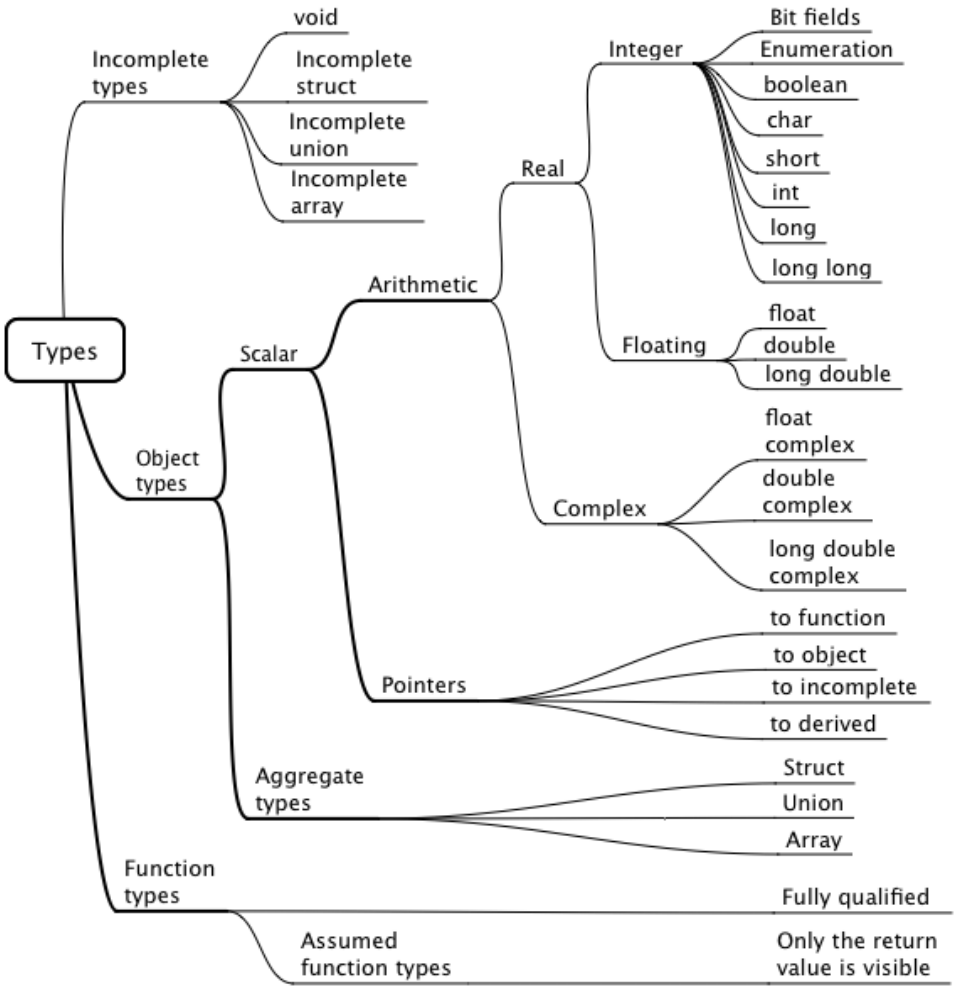
This type classification is based on the classification published by Plauger and Brody, slightly modified.

Like many other definitions in the standard I am unable to figure out anything from this sentence. Sorry.

Does this definition imply a format specification (meaning of a value)? Or it is simply a recursive definition with an infinite loop? A type is: *... the type of the expression used...*

[5] Using the operator overloading feature of lcc-win you can define user defined types that will behave almost as if they were native types. This is not a feature of the C language as such but it is present in most programming languages

[6] There is obviously no `void` object. A pointer to void is a special pointer that can point to **any** object.



The schema can be understood as follows:

A C type can be either a function type, an incomplete type or an object type. Function types can be either fully specified, i.e. we have a prototype available, or partially specified with unknown arguments but a known return value.

Incomplete types are unspecified and it is assumed that they will be specified elsewhere, except for the void type that is an incomplete type that can't be further specified. They have several uses that are explained in depth in 1.7.6.

Object types can be either scalar or aggregate types. Aggregate types are built from the scalar types: structures, unions and arrays. Scalar types are of two kinds: arithmetic or pointer types. Pointer types can point to scalar or composite types, to functions or to incomplete types.

Arithmetic types have two kinds: integer types and floating types. The integer types are bit fields, enumerations, and the types bool, char, short, int, long and long long, all with signed or unsigned types, except the boolean type that hasn't any signed form and belongs to the unsigned types. The char type has not only signed

and unsigned flavors but in some more esoteric classifications has a third flavor "plain char", different as a type from an unsigned or a signed char. We do not need to go into that hair splitting here.

Floating types are either real or complex, with both of them appearing in three flavors: float, double and long double.

1.7.3 Integer types

The language doesn't specify exactly how big each integer type must be, but it has some requirements as to the minimum values a type must be able to contain, hence its size. The char type must be at least 8 bits, the int type must be at least 16 bits, and the long type must be at least 32 bits. How big each integer type actually is, is defined in the standard header limits.h for each implementation.

1.7.4 Floating types

Floating types are discussed in more detail later. Here we will just retain that they can represent integer and non integer quantities, and in general, their dynamic range is bigger that integers of the same size. They have two parts: a mantissa and an exponent.

As a result, there are some values that can't be expressed in floating point, for instance 1/3 or 1/10. This comes as a surprise for many people, so it is better to underscore this fact here. More explanations for this later on.

Floating point arithmetic is approximate, and many mathematical laws that we take for granted like a + b - a is equal to b do not apply in many cases to floating point math.

1.7.5 Compatible types

There are types that share the same underlying representation. For instance, in lcc-win for the Intel platform, in 32 bits, long and int are compatible. In the version of lcc-linux for 64 bits however, long is 64 bits and int is 32 bits, they are no longer compatible types.

In that version long is compatible with the long long type.

Plauger and Brody give the following definition for when two types are compatible types:

- Both types are the same.
- Both are pointer types, with the same type qualifiers, that point to compatible types.
- Both are array types whose elements have compatible types. If both specify repetition counts, the repetition counts are equal.
- Both are function types whose return types are compatible. If both specify types for their parameters, both declare the same number of parameters (including ellipses) and the types of corresponding parameters are compatible. Otherwise, at least one does not specify types for its parameters. If the other

specifies types for its parameters, it specifies only a fixed number of parameters and does not specify parameters of type float or of any integer types that change when promoted.

- Both are structure, union, or enumeration types that are declared in different translation units with the same member names. Structure members are declared in the same order. Structure and union members whose names match are declared with compatible types. Enumeration constants whose names match have the same values.

1.7.6 Incomplete types

An incomplete type is missing some part of the declaration. For instance

```
struct SomeType;
```

We know now that `SomeType` is a struct, but since the contents aren't specified, we can't use directly that type. The use of this is precisely to avoid using the type: encapsulation. Many times you want to publish some interface but you do not want people using the structure, allocating a structure, or doing anything else but pass those structure to your functions. In those situations, an opaque type is a good thing to have.

Note that the opaque (incomplete) types are much more protected than "protected" members in C++. If you do not hand out the header file containing the type definitions nobody ever will be able to see how that type is built, unless (of course) they disassemble the generated code.

1.7.7 Qualified types

All types we have seen up to now are unqualified. To each unqualified type corresponds one or more qualified type that adds the keywords const, restrict, and volatile.

The const keyword means that the programmer specifies that the value of the object of this type is read only. Assignments to const objects is an error. The restrict keyword applies to pointer types and it means that there is no alias for this object, i.e. that the pointer is the only pointer to this object within this function or local scope. For instance:

```
void f(int * restrict p,int * restrict q)
{
    while (*q) {
        *p = *q; // Copy
        p++;     // Increment
        q++;
    }
}
```

During the execution of this function, the restrict keyword ensures that when the object pointed to by p is accessed, this doesn't access also the object pointed by q.

This keyword enables the compiler to perform optimizations based on the fact that p and q point to different objects.

The volatile keyword means that the object qualified in this way can change by means not known to the program and that it must be treated specially by the compiler. The compiler should follow strictly the rules of the language, and no optimizations are allowed for this object. This means that the object should be stored in memory for instance, and not in a register. This keyword is useful for describing a variable that is shared by several threads or processes. A static volatile object is a good model for a memory mapped I/O register, i.e. a memory location that is used to read data coming from an external source.

1.7.8 Casting

The programmer can at any time change the type associated with a piece of data by making a "cast" operation. For instance if you have:

```
float f = 67.8f;
```

you can do

```
double d = (double)f;
```

The `"(double)''` means that the data in f should be converted into an equivalent data using the double precision representation. We will come back to types when we speak again about casts later (page 69).

1.7.9 The basic types

The basic types of the language come wired in when you start the compiler. They are the different representations of numbers that the underlying printed circuit understands, i.e. has hardware dedicated to performing arithmetic operations with them. Sometimes the lack of hardware support can be simulated in software (for floating point numbers), even if most CPUs now support floating point.

Table-1.2 shows the sizes in the 32 bit implementation of lcc-win of the basic types of ANSI-C.

Lcc-win offers you other types of numbers, shown in Table-1.3. To use them you should include the corresponding header file, they are not "built in" into the compiler. They are built using a property of this compiler that allows you to define your own kind of numbers and their operations. This is called operator overloading and will be explained further down.

Under lcc-win 64 bits, the sizes of the standard types change a bit. See Table-1.4

The C standard defines the minimum sizes that all types must have in all implementations. See Table-1.5

1.8 Declarations and definitions

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

Table 1.2: Standard type sizes in lcc-win

Type	Size bytes	Description
_Bool	1	Logical type, can be either zero or one. Include `<stdbool.h>` to use them.
char	1	Character or small integer type. Comes in two flavours: signed or unsigned.
short	2	Integer or unicode character stored in 16 bits. Signed or unsigned.
int	4	Integer stored in 32 bits. Signed or unsigned.
long	4 or 8	Identical to int under windows 32 bit and in windows-64. In Unix 64 bit versions it is 64 bits
pointer	4 or 8	All pointers are the same size in lcc-win. Under Unix 64 it is 64 bits.
long long	8	Integer stored in 64 bits. Signed or unsigned.
float	4	Floating-point single precision. (Around 7 digits)
double	8	Floating-point double precision. (Approx. 15 digits)
long double	12	Floating point extended precision (Approx 19 digits)
float _Complex	24	Complex number
double _Complex	24	
long double _Complex [7]	24	

Table 1.3: Increased precision numbers in lcc-win

Type	Header	Size (bytes)	Description
qfloat	qfloat.h	56	352 bits floating point
bignum	bignum.h	variable	Extended precision number
int128	int128.h	16	128 bit signed integer type

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

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

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

A definition tells the compiler to allocate storage for the identifier. For instance,

Table 1.4: Type sizes for lcc-win 64 bits

Type	Size	Comment
bool		
char		
int,long		
long long		Same as in the 32 bit version In the linux and AIX 64 bit versions, long is 64 bits.
pointer	8	All 64 bit versions have this type size
long double	16	This type could not be maintained at 12 bytes since it would misalign the stack, that must be aligned at 8 byte boundaries.
float, double		Same as in the 32 bit version The double type uses the SSE registers in the Intel architecture version

Table 1.5: Minimum size of standard types

Type	Minimum size
char	8 bits
short	16 bits
int	16 bits
long	32 bits
long long	64 bits

when we defined the function main above, storage for the code generated by the compiler was created, and an entry in the program's symbol table was done. In the same way, when we wrote: `int count`, above, the compiler made space in the local variables area of the function to hold an integer.

And now the central point: You can declare a variable many times in your program, but there must be only one place where you define it. Note that a definition is also a declaration, because when you define some variable, automatically the compiler knows what it is, of course. For instance if you write: `double balance;` even if the compiler has never seen the identifier balance before, after this definition it knows it is a double precision number.

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.

1.8.1 Variable declaration

A variable is declared with `<type> <identifier> ;` like

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

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

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

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

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

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

```
int a = 7;
```

or

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

but the values you assign must be compile time constants, i.e. values that the compiler can figure out when doing its job. See "constant expressions", page 68

Pointers are declared using an asterisk:

```
int *pInt;
```

This means that `a` will contain the machine address of some unspecified integer. Remember: this pointer will contain garbage until it is initialized:

```
int sum;
int *pInt = &sum;
```

Now the `pInt` variable contains the machine address of `sum`.

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 surely somehow confusing, specially for beginners. Use this multiple declarations when all declared identifiers are of the same type and put pointers in separate lines.

The syntax of C declarations has been criticized for being quite obscure. This is true; there is no point in negating an evident weakness. In his book “Deep C secrets” Peter van der Linden writes a simple algorithm to read them. He proposes (chapter 3) the following:[8]

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

The Precedence Rule for Understanding C Declarations.

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

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

2.A : Parentheses grouping together parts of a declaration

2.B: The postfix operators:

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

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

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

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

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

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

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

Now let’s see this:

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

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

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

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

1.8.2 Function declarations

A declaration of a function specifies:

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

The general form is:

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

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

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

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

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

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

The ellipsis means "some more arguments".

Why are function declarations important?

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

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

and in another module write:

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

without any problems.

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

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

1.8.3 Function definitions

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

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

1.8.4 Scope of identifiers

The scope of an identifier is the extent of the program where the identifier is active, i.e. where in the program you can use it. There are three kinds of identifier scopes:

1) File scope. An identifier with file scope can be used anywhere from the its definition till the end of the file where it is declared.

2) Block scope. The identifier is visible within a block of code enclosed in curly braces ‘‘ and ’’.

3) Function prototype scope. This scope is concerned with the identifiers that are used within function prototypes. For instance

```
void myFunction(int arg1);
```

the identifier `'arg1'` is within prototype scope and disappears immediately after the prototype is parsed. We could add a fourth scope for function labels, that are visible anywhere within a function without any restriction of block scopes.

1.8.5 Linkage and duration of objects

The linkage of an object is whether it is visible outside the current compilation unit or not. Objects that are marked as external or appear at global scope without the keyword ‘static’ are visible outside the current compilation unit. Note that an identifier that doesn't refer to an object can have global scope but not be visible outside the current compilation unit. Enumerations, for instance, even if they have global scope are not “exported” to other modules.

In general, we have the public objects, visible in all modules of the program, and the private ones, marked with the keyword ‘static’ and visible only in the current compilation unit. The duration of an object means the time when this object is active. It can be:

1) Permanent. The object is always there, and lives from the start to the end of the program. It can maybe even live after the program has finished. They are declared at global scope. The initial value is either explicitly given by the program such as in: `int p = 78;` or implicitly defined as zero as in `int p;`.

2) Transitory. The object starts its life when the program enters the scope where it lives, and disappears after the program leaves the scope. Automatic variables are transitory objects. The initial value is undefined and in most compilers it consists of whatever values were stored previously at that memory location.

3) Allocated. The object starts its life as the result of one of the allocation functions like `malloc`, or `GC_malloc`, and it ends its life when its storage is reclaimed, either explicitly because the programs calls the ‘free’ function or because the garbage collector determines that its storage is no longer used. The initial value depends on the allocation function: malloc returns uninitialized memory, calloc and `GC_malloc` zeroes the memory before returning it.

1.8.6 Variable definition

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

```
int a;
```

or

```
int a = 67;
```

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

1.8.7 Statement syntax

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

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

in C, the if statement requires a parentheses

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

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

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

This means that the compiler generates code for assigning the value of z to x, then it compares this value with 13, and if the relationship holds, the program will set z to zero. This construct is considered harmful however, because it is very easy to mix the assignment (=) and the equality (==) operations.

1.9 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 ‘)’
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-win 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.[9] 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.

[9] You will probably see another display in your computer if you are using a recent version of lcc-win. 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-win offers you such a tool, and you can debug your program by just pressing F5 in the IDE.

Summary:

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

1.10 Input and output

In the Unix operating system, where the C language was designed, one of its central concepts is the "FILE" generalization. Devices as varied as serial devices, disks, and what the user types in his/her keyboard are abstracted under the same concept: a FILE as a sequence of bytes to handle.

The FILE structure is a special opaque structure defined in `<stdio.h>`. Contrary to other opaque structures, its definition is exposed in stdio.h, but actually its fields are never directly used.[10]

We have two kinds of input/output: direct operations and formatted operations. In direct operations, we just enter data from or to the device without any further processing. In formatted operations, the data is assumed to be of a certain type, and it is formatted before being sent to the device.

As in many other languages, to perform some operation with a file you have to setup a connection to it, by first "opening" it in some way, then you can do input/output to it, and then, eventually, you close the connection with the device by "closing" it.

Table-1.6 shows a short overview of the functions that use files.

[10] An "opaque" structure is a structure whose definition is hidden. Normally, we have just a pointer to it, but nowhere the actual definition.

Table 1.6: File operations

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
getchar	Reads a character from standard input
feof	True when current position is at the end of the file
ferror	True when error reading from the device
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
fgetc	Reads one character from a stream
fscanf	Reads data from a file using a given data format
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
remove	Erases a file
rename	Renames a file.
rewind	Repositions the file pointer to the beginning of a file.
setbuf	Controls file buffering.
tmpnam	Returns a temporary file name
ungetc	Pushes a character back into a file.
unlink	Erases a file

1.10.1 Predefined devices

To establish a connection with a device we open the file that represents it. There are three devices always open that represent the basic devices that the language assumes:

1. The standard input device, or "stdin". This device is normally associated with the keyboard at the start of the program.

2. The standard output device, or "stdout". This device is normally associated with the computer screen in text mode.[11]

3. The standard error device or "stderr" that in most cases is also associated with the computer screen.

[11]The text mode window is often called "Dos window" even if it has nothing to do with the old MSDOS operating system. It is a window with black background by default, where you can see only text, no graphics. Most of the examples following use this window. To start it you just go to "Start", then "Run" and type the name of the command shell: "cmd.exe"

Other devices that can be added or deleted from the set of devices the program can communicate with. The maximum value for the number of devices that can be simultaneously connected is given by the macro FOPEN_MAX, defined in stdio.h.

Under some systems you do not have these devices available or they are available but not visible. Some of those systems allow you to open a "command line" window that acts like a primitive system console with a text mode interface. When you use those command line windows your program has these standard devices available.

1.10.2 The typical sequence of operations

To establish a connection with a device we use the "fopen" function, that returns a pointer to a newly allocated FILE structure, or NULL if the connection with the device fails, for whatever reason. Once the connection established we use fwrite/fread to send or receive data from/to the device. When we do not need the connection any more we break the connection by "closing" the file.

```
#include <stdio.h>
int main(int argc,char *argv[])
{
    unsigned char buffer[2048];
    unsigned int byteswritten, bytesread;
    // Error checking suppressed for clarity
    FILE *f = fopen(argv[1],"r");
    FILE *out = fopen(argv[2],"w");
    bytesread = fread(buffer,1,sizeof(buffer),f);
    byteswritten = fwrite(buffer,1,sizeof(bufffer),out);
    fclose(f);
    fclose(out);
    return 0;
}
```

In this hypothetical program we establish a connection for reading from a device named in the first argument (`argv[1]`). We connect to another device named in the second argument, we read from the first device, we write into the second device and then we close both.

1.10.3 Examples

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. To make more concrete the general descriptions about input/output from the preceding sections we present here a compilable, complete example.

The example is a function that will read a 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],"rb");       (6)
    c = fgetc(infile);                  (7)
    while (c != EOF) {                  (8)
        count++;                        (9)
        c = fgetc(infile);             (10)
    }
    printf("%d\n",count);              (11)
    return 0;
}
```

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

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

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

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

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

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

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

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

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

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

11. If we arrive here, it means that we have hit EOF , the end of the file. We print our count in the screen and exit the program returning zero, i.e. all is OK. By convention, a program returns zero when no errors happened, and an error code, when something happens 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: 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[])
{
    size_t count=0; // chars read
    FILE *infile;
    int c;

    if (argc < 2) { (2)
        printf("Usage: countchars <file name>\n");
        exit(EXIT_FAILURE); (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.Note that we do not use the integer constant 1 directly, but rather use the predefined constants EXIT_SUCCESS (defined as 0) or EXIT_FAILURE (defined as 1) in stdlib.h. In other operating systems or environments, the numeric value of EXIT_FAILURE could be different. By using those predefined constants we keep our code portable from one implementation to the other.

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[])
```

```
{
    size_t count=0; // chars read
    FILE *infile;
    int c;

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

We try again:

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

Well this error checking works. But let's look again at the logic of this program. Suppose we have an empty file. Will our program work?

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

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

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

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

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

Why?

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

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

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

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

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

H:\lcc\examples> dir countchars.c

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

Yes, 493 bytes instead of 492 before, since we have added a "b" to the arguments of fopen! Still, we read the docs about file handling, and we try to see if there are no hidden bugs in our program. After a while, an obvious fact appears: we have opened a file, but we never closed it, i.e. we never break the connection between the program, and the file it is reading. We correct this, and at the same time add some commentaries to make the purpose of the program clear.

```
/*-------------------------------------------------------
 Module:        H:\LCC\EXAMPLES\countchars.c
 Author:        Jacob
 Project:       Tutorial examples
 State:         Finished
 Creation Date: July 2000
 Description:   This program opens the given file, and
                prints the number of characters in it.
-------------------------------------------------------*/
```

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

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

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

Summary:

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

1.10.4 Other input/output functions

The current position

Each open file descriptor has a *current position*, i.e. the position in the data stream where the next write or read operation will be done. To know where the file pointer is,

use the ftell function. To set the current position to some specific index use the fseek function. Here is an example of the usage of those functions. We write a function that will return the length of a given file. The algorithm is very simple: 1) Set the file pointer at the end of the file 2) Read the position of the file cursor. This is the size of the file. Easy isn't it?

```
size_t FileLength(char *FileName)    (1)
{
    FILE *f = fopen(FileName,"rb");  (2)
    size_t result;
    if (f == NULL)
        return -3;
    if (fseek(f,0,SEEK_END)) {       (3)
        fclose(f);
        return -2;
    }
    result = ftell(f);               (4)
    fclose(f);
    return result;                   (5)
}
```

1. We use the type dedicated for sizes within the language: `size_t` as the return value of our function. This type is translated by the implementation into the integer size that can hold the biggest size supported. In lcc-win's case this is an unsigned int. In 64 bit systems is normally an unsigned long long.

2. We open the file. Note that we open it in binary mode, because ftell and fseek will NOT work with files opened in *text* mode, where the sequence `\r\n` is translated into only one character.

3. The `fseek`function will position the file pointer. The first argument is the file descriptor pointer; the second is the distance from either the beginning, the current position or the end of the file. In our example we use the position from the end of the file, SEEK_END. If the function fails for some reason we return -2.

4. We call the `ftell`function to retrieve the current position. Note that that function returns -1 if there was an error. We do not test for this result since if that function failed, we return the error code without any further processing.

5. Since all this functions return a 32 bit integer, files bigger than 2GB can't be measured using this functions. lcc-win provides some 64 bit file primitives, and the Windows operating system provides a full set of 64 bit file primitives.

1.10.5 File buffering

A file can be either unbuffered (all characters and input output operations happen immediately) or buffered, meaning that characters are accumulated in a buffer and

transmitted from or to the device as a block. Obviously buffering input output has the advantage of efficiency, since fewer I/O operations are performed.

Buffering can be either *fully buffered*, when i/o is done when the buffer is full only, or *line buffered*, when I/O is done only when the new line character is found in the data. Normally, the files associated with the keyboard and the screen are line buffered, and disk files are fully buffered.

You can force the system to empty the buffer with the function fflush. The amount of buffering can be changed with the setbuf and setvbuf functions.

The setbuf function allows you to setup your own buffer instead of the default one. This can be sometimes better when you know you need to read/write large chunks of the file, for instance:

```
#include <stdio.h>
unsigned char mybuf[BUFSIZ]; // BUFSIZ defined in stdio.h
int main(void)
{
    setbuf(stdout,mybuf);
}
```

Note that the buffer is declared global. Be careful about the scope of the buffer you pass to setbuf. You can use a buffer with local scope, but you must be aware that before leaving the scope where the buffer is declared the file must be closed. If not, the program will crash when the fclose function tries to flush the contents of the buffer that is no longer available. The fclose function will be called automatically at program exit, if any open files exist at that time.

The setvbuf function allows you to change the mode of the buffering (either line, full or none) and pass a buffer of a different size than BUFSIZ, a constant defined in stdio.h.

Error conditions

What is the proper way of finding an end of file condition while reading from a file?

Try to read the file, and if it fails, see if the reason it failed was *end of file*. You find this using the feof function. There are two reasons why a read from a file can fail. The first one is that there is nothing more to read, the end of the data set has been reached and this means that the current position is at the end of the file.

The second reason is that there is a hardware error that makes any reading impossible: the disk drive has a bad spot and refuses to give any data back, or the device underlying this file is a network connection and your internet service provider has problems with the router connected to your machine, or whatever. There are endless reasons why hardware can fail.

You can find out which the reason is responsible for this using the feof and ferror functions.

1.11 Commenting the source code

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

two forms: Two slashes // introduce a commentary that will last until the end of the line. No space should be present between the first slash and the second one. A slash and an asterisk /* introduce a commentary that can span several lines and is only terminated by an asterisk and a slash, */. The same rule as above is valid here too: no space should appear between the slash and the asterisk, and between the asterisk and the slash to be valid comment delimiters. Examples:

```
// This is a one-line commentary. Here /* are ignored anyway.
/* This is a commentary that can span several lines.
        Note that here the two slashes // are ignored too */
```

This is very simple, but the difficulty is not in the syntax of commentaries, of course, but in their content. There are several rules to keep in mind: Always keep the commentaries current with the code that they are supposed to comment. There is nothing more frustrating than to discover that the commentary was actually misleading you, because it wasn't updated when the code below changed, and actually instead of helping you to understand the code it contributes further to make it more obscure. Do not comment what you are doing but why. For instance:

```
record++; // increment record by one
```

This comment doesn't tell anything the C code doesn't tell us anyway.

```
record++;  //Pass to next record.
// The boundary tests are done at
// the beginning of the loop above
```

This comment brings useful information to the reader.

At the beginning of each procedure, try to add a standard comment describing the purpose of the procedure, inputs/outputs, error handling etc.[12]

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

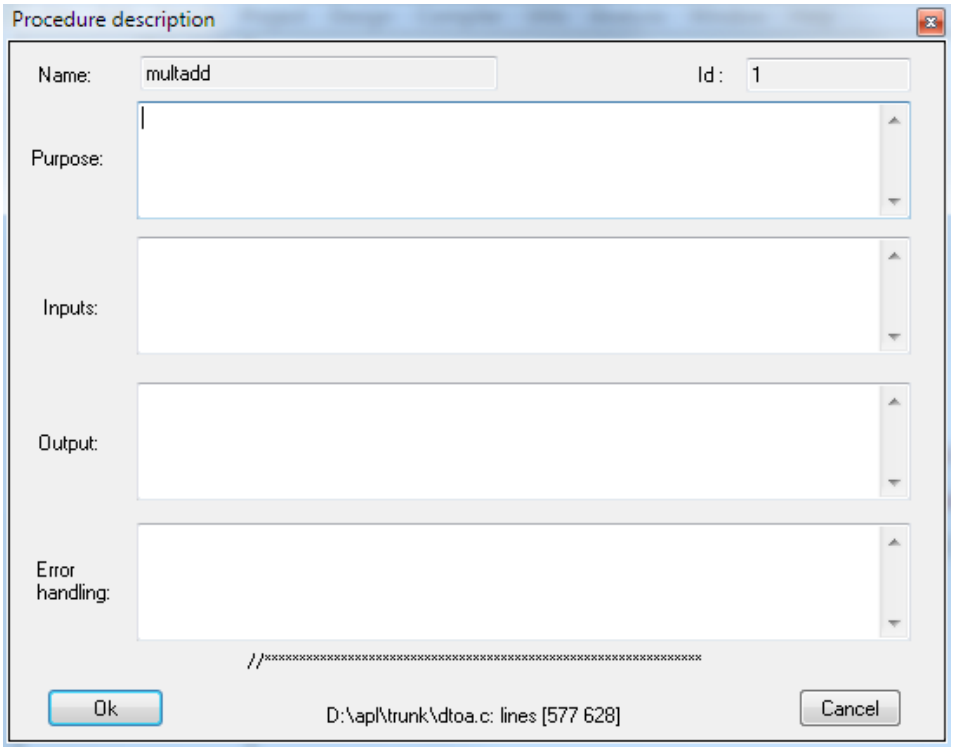
Note that you yourself will be the first guy to debug the code you write. Commentaries will help you understand again that hairy stuff you did several months ago, when in a hurry.

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

1.11.1 Describing a function

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

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



There are several fields that you should fill:

1. Purpose. This should explain what this function does, and how it does it.
2. Inputs: Here you should explain how the interface of this function is designed: the arguments of the function and global variables used if any.
3. Outputs. Here you should explain the return value of the function, and any globals that are left modified.
4. Error handling. Here you should explain the error return, and the behavior of the function in case of an error occurring within its body.

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

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

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

1.11.2 Describing a file

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

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

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

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

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

1.12 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, these words have a special meaning: "id", meaning an identifier, "type" meaning some arbitrary type and "expr" meaning some arbitrary C expression.

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

1.12.1 Statements

Expression	Meaning	Example
`identifier`	The value associated with that identifier. (see page 65.)	`id`
constant	The value defined with this constant (see page 67.).	
	Integer or unsigned integer constant.	`45 45U`
	long integer or unsigned long integer constant	`45L 45UL`
	long long integer or unsigned long long integer constant	`45LL 45ULL`
	Floating constant	`45.9`
	float constant	`45.9f`
	long double constant	`45.9L`
	qfloat constant	`45.9Q`
	character constant or wide character constant enclosed in single quotes	`'A' L'A'`
	String literal or wide character string literal enclosed in double quotes	`"Hello" L"Hello"`
`{constants}`	Define tables or structure data. Each comma separated item is an item in the table or structure.	`int tab[]={1,67}`
Prefixed integer constants	Uses different numerical bases.	
	octal constant (base 8) introduced with a leading zero	`055` (45 in base 8)
	Hexadecimal constant introduced with 0x	`0x2d` (45 in base 16)
	Binary constant introduced with 0b. This is an lcc-win extension.	`0b101101` (45 in binary)
`Array[index]`	Access the position "index" of the given array. Indexes start at zero (see page 72.)	`Table[45]`
`Array[i1][i2]`	Access the n dimensional array using the indexes i1, i2, ... in..See "Arrays." on page 72.	`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". see "Function calls" on page 75.	`sqrt(4.9);`
`fn (arg ...)`,	Function with variable number of arguments	

Table 1.7 – Continued

Expression	Meaning	Example
`(*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 the value of the right hand side of the equals sign. See "Assignment." on page 79	`a = 45`
`expression++`	Equivalent to expression = expression + 1. Increment expression after using its value. See "Postfix" on page 80.	`a = i++`
`expression--`	Equivalent to expression = expression – 1. Decrement expression after using its value. see "Postfix" on page 80.	a = i–
`++expression`	Equivalent to expression = expression+1. Increment expression before using its value.	a = ++i
`--expression`	Equivalent to Expression = expression – 1. Decrement expression before using it.	a = –i
`& object`	Return the machine address of object. The type of the result is a pointer to the given object.	i
`* pointer`	Access the contents at the machine address stored in the pointer. .See "Indirection" on page 72.	*pData
`- expression`	Subtract expression from zero, i.e. change the sign.	-a
`~ expression`	Bitwise complement expression. Change all 1 bits to 0 and all 0 bits to 1.	a
`! expression`	Negate expression: if expression is zero, !expression becomes one, if expression is different than zero, it becomes zero.	`!a`
`sizeof(expr)`	Return the size in bytes of expr. .see "Sizeof." on page 60.	sizeof(a)

Table 1.7 – Continued

Expression	Meaning	Example
`(type) expr`	Change the type of expression to the given type. This is called a "cast". The expression can be a literal expression enclosed in braces, as in a structure initialization. See page 69.	`(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. .	`a-b`
`expr1 << expr2`	Shift left expr1 expr2 bits.	`a << b`
`expr1 >> expr2`	Shift right expr1 expr2 bits.	`a >> b`
`expr1 < expr2`	1 if expr1 is smaller than expr2, zero otherwise	`a < b`
`expr1 <= expr2`	1 if expr1 is smaller or equal than expr2, zero otherwise	`a <= b`
`expr1 >= expr2`	1 if expr1 is greater or equal than expr2, zero otherwise	`a >= b`
`expr1 > expr2`	1 if expr2 is greater than expr2, zero otherwise	`a > b`
`expr1 == expr2`	1 if expr1 is equal to expr2, zero otherwise	`a == b`
`expr1 != expr2`	1 if expr1 is different from expr2, zero otherwise	`a != b`
`expr1 & expr2`	Bitwise AND expr1 with expr2. See "Bitwise operators" on page 67.	`a&8`
`expr1 ^ expr2`	Bitwise XOR expr1 with expr2. See Bitwise operators on page 67.	`a^b`
`expr1 | expr2`	Bitwise OR expr1 with expr2. See "Bitwise operators" on page 67.	`a|16`
`expr1 && expr2`	Evaluate expr1. If its result is zero, stop evaluating the whole expression and set the result of the whole expression to zero. If not, continue evaluating expr2. The result of the expression is the logical AND of the results of evaluating each expression. See "Logical operators" on page 66.	`a < 5 && a > 0` This will be 1 if "a" is between 1 to 4. If a >= 5 the second test is not performed.

Table 1.7 – Continued

Expression	Meaning	Example
`expr1 || expr2`	Evaluate expr1. If the result is one, stop evaluating the whole expression and set the result of the expression to 1. If not, continue evaluating expr2. The result of the expression is the logical OR of the results of each expression. See "Logical operators" on page 66.	`a == 5 ||a == 3` This will be 1 if either a is 5 or 3
`expr ? v1:v2`	If expr evaluates to non-zero (true), return v1, otherwise return v2. see "Conditional operator." on page 58.	`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 mod expr1 and store the result in expr	`a %= 6`
`expr += expr1`	Add expr1 with expr and store the result in expr	`a += 6`
`expr -= expr1`	Subtract expr1 from expr and store the result in expr	`a -= 76`
`expr <<= expr1`	Shift left expr by expr1 bits and store the result in expr	`a <<= 6`
`expr >>= expr1`	Shift right expr by expr1 bits and store the result in expr	`a >>= 7`
`expr &= expr1`	Bitwise and expr with expr1 and store the result in expr	`a &= 32`
`expr ^= expr1`	Bitwise xor expr with expr1 and store the result in expr	`a ^= 64`
`expr |= expr1`	Bitwise or expr with expr1 and store the result in expr.	`a |= 128`
`expr, expr1`	Evaluate expr, then expr1 and return the result of evaluating the last expression, in this case expr1. .See page 69	`a=7,b=8` Result of this is 8
;	Null statement	;

1.12.2 Declarations

Declarations	Meaning	Example
`type id;`	Identifier will have the specified type within this scope. In a local scope its value is undetermined. In a global scope, its initial value is zero, at program start.	`int a;`
`type * id;`	Identifier will be a pointer to objects of the given type. You add an asterisk for each level of indirection. A pointer to a pointer needs two asterisks, etc.	`int *pa;` pa will be a pointer to int
`type id[int expr]`	Identifier will be an array of "int 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 *ptrTab[56*2];` Array of 112 int pointers.
`typedef old new`	Define a new type-name for the old type. see "Typedef." on page 59.	`typedef unsigned uint;`
`register id;`	Try to store the identifier in a machine register. The type of identifier will be equivalent to signed integer if not explicitly specified. see "Register." on page 59.	`register f;`
`extern type id;`	The definition of the identifier is in another module. No space is reserved.	`extern int frequency;`
`static type id`	Make the definition of identifier not accessible from other modules.	`static int f;`
`struct id { declarations }`	Define a compound type composed of the list of fields enclosed within the curly braces.	`struct coord { int x; int y; };`
`type id:n`	Within a structure field declaration, declare "id" as a sequence of n bits of type "type". See "Bit fields" on page 63.	`unsigned n:4` n is an unsigned int of 4 bits
`union id { declarations };`	Reserve storage for the biggest of the declared types and store all of them in the same place. see "Union." on page 59.	`union dd { double d; int id[2]; };`
`enum id { enum list };`	Define an enumeration of comma-separated identifiers assigning them some integer value. see "Enum." on page 60.	`enum color {red,green,blue};`

`const type id;`	Declare that the given identifier can't be changed (assigned to) within this scope. see "Const." on page 60.	`const int a;`
`type * restrict`	This pointer has no other pointers that point to the same data.	`char * restrict p;`
`volatile type identifier`	Declare that the given object changes in ways unknown to the implementation. The compiler will not store this variable in a register, even if optimizations are turned on.	`volatile int hardware_clock;`
`unsigned int-type`	When applied to integer types do not use the sign bit. see "Unsigned." on page 63.	`unsigned char a;`
`type id(args);`	Declare the prototype for the given function. The arguments are a comma separated list. see "Prototypes." on page 60.	`double sqrt(double);`
`type(*id) (arguments);`	Declare a function pointer called "id" with the given return type and arguments list	void (*fn)(int)
`label:`	Declare a label.	`lab1:`
`type fn(args) {` ... statements ... `}`	Definition of a function with return type `type` and arguments `args` .	`int add1(int x) { return x+1;}`
`inline`	This is a qualifier that applies to functions. If present, it can be understood by the compiler as a specification to generate the fastest function call possible, generally by means of replicating the function body at each call site.	`double inline overPi(double a) {return a/3.14159;}`
`main`	This is the entry point of each program. There should be a single function called main in each conforming C program. There are two possible interfaces for main: without arguments, and with arguments.	`int main(void);` `int main(int argc, char *argv[])`

1.12.3 Pre-processor

Declarations	Meaning	Example
`// commentary`	Double slashes introduce comments up to the end of the line.see "Comments" on page 65.	`// comment`

`/*commentary */`	Slash star introduces a commentary until the sequence star slash */ is seen. see "Comments" on page 65.	`/* comment */`
`defined (id)`	If the given identifier is `#defined`, return 1, else return 0.	`#if defined(max)`
`#define id text`	Replace all appearances of the given identifier (`id` here) by the corresponding expression. See "Preprocessor commands" on page 176.	`#define TAX 6`
`#define macro(a,b)`	Define a macro with n arguments. When used, the arguments are lexically replaced within the macro. See page 177	`#define max(a,b) ((a)<(b)? (b):(a))`
`#ifdef id`	If the given identifier is defined (using `#define`) include the following lines. Else skip them. See page 178.	`#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`
`#warning "text''`	Writes the text of a warning message. This is an extension of lcc-win but other compilers (for instance gcc) support it too.	`#warning "MACHINE undefined"`
`#error "text''`	Writes an error message	`#error "M undefined"`
`#file "foo.c"`	Set the file name	`#file "ff.c''`
`#line nn`	Set the line number to nn	`#line 56`
`#include <fns.h>`	Insert the contents of the given file from the standard include directory into the program text at this position.	`#include <stdio.h>`

`#include "fns.h"`	Insert the contents of the named file starting the search from the current directory.	`#include "foo.h"`
`##` `#token`	Token concatenation Make a string with a token. Only valid within macro declarations	`a##b` → `ab` `#foo` → `"foo"`

`#pragma`
 	
`_Pragma(string)`
	Special compiler directives
	
	Special compiler directives
		`#pragma optimize(on)`
		`_Pragma( "optimize (on)");`

`#undef id`	Erase from the pre-processor tables the given identifier.	`#undef TA`
\	If a \ appears at the end of a line just before the newline character, the line and the following line will be joined by the preprocessor and the \ character will be eliminated.	

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

1.12.4 Control-flow

Syntax	Description

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

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

`do { ... statements ... } while (condition);`	Execute the statements in the block, and afterwards test if condition is true. If that is the case, execute the statements again.See "do" on page 14.
`for(init;test;incr) { ... statements ... }`	Execute unconditionally the expressions in the init statement. Then evaluate the test expression, and if evaluates to true, execute the statements in the block following the for. At the end of each iteration execute the incr statements and evaluate the test code again. See "for" on page 13.
`switch (expression) { case int-constant: statements ... break; default: statements }`	Evaluate the given expression. Use the resulting value to test if it matches any of the integer expressions defined in each of the 'case' constructs. If the comparison succeeds, execute the statements in sequence beginning with that case statement. If the evaluation of expression produces a value that doesn't match any of the cases and a "default" case was specified, execute the default case statements in sequence. See "Switch statement." on page 65.
`goto label;`	Transfer control unconditionally to the given label.
`continue`	Within the scope of a for/do/while loop statement, continue with the next iteration of the loop, skipping all statements until the end of the loop.See "Break and continue statements" on page 64.
`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.See page 62.

1.12.5 Extensions of lcc-win

Syntax	Description
`t operator token(args) { statements }`	Redefine one of the operators like +, * or others so that instead of issuing an error, this function is called instead. See page 206
`type & id = expr;`	Identifier will be a reference to a single object of the given type. References must be initialized immediately after their declaration.

`int fn(int a,int b=0)`	Default function arguments. If the argument is not given in a call, the compiler will fill it with the specified compile time constant
`int overloaded f(int)` `int overloaded f(char*)`	Generic functions. These functions have several types of arguments but the same name.

2 A closer view

Let's go in-depth for each of the terms described succintely in the table above. The table gives a compressed view of C. Now let's see some of the details.

2.1 Identifiers.

An "identifier" is actually a name. It names either an action, a piece of data, an enumeration of several possible states, etc. The C language uses the following rules concerning names:

- The letters allowed are A-Z, a-z and the underscore character '_'.
- Digits are allowed, but no identifier starts with a digit.
- Lower case and upper case letters are considered different
- lcc-win has 255 characters for the maximum length of a name. The standard guarantees 31 significant characters for an external identifier, 63 for an internal one. If you use overly long names, your code may not work in other environments.

Identifiers are the vocabulary of your software. When you create them, give a mnemonic that speaks about the data stored at that location. Here are some rules that you may want to follow:

- Most of the time, construct identifiers from full words, joined either by underscore or implicitly separated by capitalization. For example we would use `list_element` or `ListElement`. A variant of this rule is "camel-casing": the first word is in lower case, the second starts with upper case. In this example we would have `listElement`.
- Use abbreviations sparingly, for words you use frequently within a package. As always, be consistent if you do so.
- Identifier names should grow longer as their scope grows longer: Identifiers local to a function can have much shorter names than those used throughout a package.
- Identifiers containing a double underscore (" `__` ") or beginning with an underscore and an upper-case letter are reserved by the compiler, and should

therefore not be used by programmers. To be on the safe side it is best to avoid the use of all identifiers beginning with an underscore. [1]

2.1.1 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:

- an object.
- a function
- a tag or a member of a structure, union or enum
- a typedef
- a label

For each different entity that an identifier designates, the identifier can be used (is visible) only within a region of a program called its scope. There are four kinds of scopes in C.

The file scope is built from all identifiers declared outside any block or parameter declaration, it is the outermost scope, where global variables and functions are declared.

A function scope is given only to label identifiers.

The block scope is built from all identifiers that are defined within the block. A block scope can nest other blocks.

The function prototype scope is the list of parameters of a function. Identifiers declared within this scope are visible only within it. Let's see a concrete example of this:

```
static int Counter = 780;       // file scope
extern void fn(int Counter); // function prototype scope
void function(int newValue, int Counter) // Block scope
{
    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); <----
        }
```

[1] Microsoft has developed a large set of rules, mostly very reasonable ones here: http://msdn.microsoft.com /library/default.asp?url=/library/en-us/cpgenref/ html/cpconNamingGuidelines.asp

```
        if (i == 4)
            goto label;
    }
}
```

At the point indicated by the arrow, the poor "Counter" identifier has a busy life:

- It was bound to an integer object with file scope
- Then it had another incarnation within the function prototype scope
- Then, it was bound to the variables of the function 'setCounter' as a parameter
- That definition was again "shadowed" by a new definition in an inner block, as a local variable.

The value of "Counter" at the arrow is 78. When that scope is finished its value will be the value of the parameter called Counter, within the function "function".

When the function definition finishes, the file scope is again the current scope, and "Counter" reverts to its value of 780.

The "linkage" of an identifier refers to the visibility to other modules. Basically, all identifiers that appear at a global scope (file scope) and refer to some object are visible from other modules, unless you explicitly declare otherwise by using the "static" keyword.

Problems can appear if you first declare an identifier as static, and later on, you define it as external. For instance:

```
static void foo(void);
```

and several hundred lines below you declare:

```
void foo(void) {  ...  }
```

Which one should the compiler use? static or not static? That is the question...

Lcc-win 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.

2.2 Constants

2.2.1 Evaluation of constants

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

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

This will also put one in the variable d:

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

Why?

The expression `60 || 1+1/0` is evaluated from left to right. It is a boolean expression, and its value is 1 if the first expression is different from zero, or the value of the second expression if the value of the first one is zero. Since 60 is not zero, we stop immediately without evaluating the second expression, what is fortunate since the second one contains an error...

Constant expressions

The standard defines constant expressions as follows:

> A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constant expressions can have values of the following type:

- Arithmetic. Any arithmetic operations are allowed. If floating point is used the precision of the calculations should be at least the same as in the run time environment.
- A null pointer constant.
- The address of an object with global scope. Optionally an integer offset can be added to the address.

Since constant expressions are calculated during compilation, even inefficient algorithms are useful since the execution time is not affected. For instance Hallvard B Furuseth proposed[2] a set of clever macros to calculate the logarithm base 2 of a number during compilation:

```
/*
 * Return (v ? floor(log2(v)) : 0) when 0 <= v < 1<<[8, 16, 32, 64].
 * Inefficient algorithm, intended for compile-time constants.
 */
#define LOG2_8BIT(v)  (8 - 90/(((v)/4+14)|1) - 2/((v)/2+1))
#define LOG2_16BIT(v) (8*((v)>255) + LOG2_8BIT((v) >>8*((v)>255)))
#define LOG2_32BIT(v) \
    (16*((v)>65535L) + LOG2_16BIT((v)*1L >>16*((v)>65535L)))
#define LOG2_64BIT(v)\
    (32*((v)/2L>>31 > 0) \
     + LOG2_32BIT((v)*1L >>16*((v)/2L>>31 > 0) \
                         >>16*((v)/2L>>31 > 0)))
```

Clever isn't it?

So much clever that I have been unable to understand how they work [3]. I just tested this with the following program:

[2] In a message to the comp.lang.c discussion group posted on June 28th 2006, 4:37 pm. You can find the original message in
`https://groups.google.com/group/comp.lang.c/msg/706324f25e4a60b0?hl=en&`

[3] Thomas Richter explained them to me. He said (in comp.lang.c):

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    printf("LOG2_32BIT(35986)=%ld\n",LOG2_32BIT(35986));
    printf("log2(35986.0)=%g\n",log2(35986.0));
}
OUTPUT:
LOG2_32BIT(35986)=15
log2(35986.0)=15.1351
```

What is also interesting is that lcc-win receives from the preprocessor the result of the macro expansion. Here is it, for your amusement [4]:

```
#line 17 "tlog2.c"
int main(void)
{
printf("LOG2_32BIT(35986)=%ld\n",(16*((35986)>65535L) +
(8*(((35986)*1L >>16*((35986)>65535L))>255) +(8 - 90/
(((((35986)*1L >>16*((35986)>65535L)) >>8*(((35986)*
1L >>16*((35986)>65535L))>255))/4+14)|1) - 2/((((35986)*
1L >>16*((35986)>65535L)) >>8*(((35986)*1L >>16*((35986)
>65535L))>255))/2+1)))));
}
```

The compiler calculates all those operations during compilation, and outputs the 15, that is stuffed into a register as an argument for the printf call. Instead of calling an expensive floating point library function you get the result with no run time penalty.

2.2.2 Integer constants

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

`LOG2_8BIT` is a rational function approximation of the log - it is not identical to the log and gives false results for x = 0, or for x > 255, but only then. That is, it matches the output of log(x) by properly tweaking the coefficients.

`LOG2_16` simply uses the functional equation of the log, namely

$log(x * b) = log(b) + log(x)$

and in this case, b = 256. The same goes for `LOG2_32` and `LOG2_64` which simply extend the game to 64 bit by factoring more and more powers out.

[4] Of course that is a single huge line that I had to cut in several places to make it fit the text

Constant	Description
`12345`	integer constant, decimal
`0777`	octal for 511 decimal
`0xF98A`	hexa for 63882 decimal. Result type is unsigned
`12345L`	long integer constant
`2634455LL`	long long integer constant
`2634455i64`	long long integer, Microsoft notation
`5488UL`	unsigned long constant 5488
`548ULL`	unsigned long long constant 548

2.2.3 Floating constants

For floating constants, the convention is either to use a decimal point (1230.0) or scientific notation (in the form of 1.23e3). They can have the suffix 'F' (or 'f') to mean that they are float constants, and not double constants as it is implicitly assumed when they have no suffix.

A suffix of "l" or "L" means long double constant. A suffix of "q" or "Q" means a qfloat. The default format (without any suffix) is double. This default is important since it can be the source of bugs that are very difficult to find. For instance:

```
long double d = 1e800;
```

The dynamic range of a long double is big enough to accept this value, but since the programmer has forgotten the L the number will be read in double precision. Since a double precision number can't hold this value, the result is that the initialization will not work at all: a random value will be stored into "d".

2.2.4 Character string constants

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

```
L"abc"
```

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

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

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

Note that strings and numbers are completely different data types. Even if a string contains only digits, it will never be recognized as a number by the compiler: `"162"` is a string, and to convert it to a number you must explicitly write code to do the transformation. [5]

[5] Using operator overloading you can add new meanings to the normal arithmetic operators. You can then, "add" strings, what is a very bad idea for expressing string concatenation. String concatenation is not a similar operation to addition, since it is not commutative: `"abc"+"def"`, → `"abcdef"`, but `"def"+"abc"` → `"defabc"`

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

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

Another way, introduced with C99 is:

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

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

```
char *a = "abc";
char *b = "abc";
```

Both a and b will point to the SAME string, and if either is modified the other will not retain the original value [6].

2.2.5 Character abbreviations

Within a string constant, the following abbreviations are recognized:

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

[6] Other compilers are different. GCC, for instance, makes the program crash if a character string constant is modified.

2.3 Arrays

Here are various examples of 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
char letters[] = {'C', '-', '-'};
```

Note that the last array "letters" is NOT a zero terminated character string but an array of 3 positions that is not terminated by a zero byte.

Multidimensional arrays are indexed like this:

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

A table of 2 rows and three columns is declared. Then, we assign 7 to the second row, third column. (Remember: arrays indexes start with zero). Note that when you index a two dimensional array with only one index you obtain a pointer to the start of the indicated row.

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

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

Arrays in C are stored in row-major order, i.e. the array is a contiguous piece of memory and the rows of the array are stored one after the other. The individual array members are accessed with a simple formula:

```
x[i][j] == *(x+i*n+j)
```

where n is the row size of the array x. It is evident from this formula that the compiler treats differently a two dimensional array from a one dimensional one, because it needs one more piece of information to access the two dimensional one: the size of each row, "n" in this case.

How does the compiler know this value?

From the declaration of the array of course. When the compiler parses a declaration like

```
int tab[5][6];
```

The last number (6) is the size of each row, and is all the information the compiler needs when compiling each array access. Since arrays are passed to functions as pointers, when you pass a two dimensional array it is necessary to pass this information too, for instance by leaving empty the number of rows but passing the number of columns, like this:

```
int tab[][6]
```

This is the standard way of passing two dimensional arrays to a function. For example:

```
#include <stdio.h>
int tab[2][3] = {1,2,3,4,5,6};
// Note the declaration of the array parameter
int fn(int array[][3])
{
    printf("%d\n",array[1][1]);
}
int main(void)
{
    fn(tab);
}
```

Arrays can be fixed, i.e. their dimensions are determined at compile time, or they can be dynamic, i.e. their dimensions are determined at run time.

For dynamic arrays, we have to do a two stage process to allocate the storage that we need for them, in contrast to one dimensional arrays where we need just a single allocation.

For instance, here is the code we would write to allocate dynamically an array of integers of 3 rows and 4 columns:

```
int ** result = malloc(3*sizeof(int *));
for (int i = 0; i<3;i++) {
    result[i] = malloc(4*sizeof(int));
}
```

Of course in a real program we would have always tested the result value of malloc for failure.

We see that we allocate an array of pointers first, that is equal to the number of rows we will need. Then, we fill each row with an allocation of space for the number of columns in the array times the size of the object we are storing in the array, in this example an integer.

It is important to distinguish the difference between dynamically allocated and compile-time fixed arrays. The row major order formula does not work with dynamic arrays, only with arrays whose dimensions are known during the compilation.

From the above discussion we see too that we need always an array of pointers as big as the number of rows in the array, something we do not need in the case of arrays with known dimensions

Obviously, if you want the best of the two alternatives you can allocate a single block of memory for the two dimensional array, and instead of using the array notation you use the pointer notation (the array formula above) yourself to access the array, eliminating any need for increased storage. You would allocate the two dimensional array like this:

```
int *result = malloc(sizeof(int) * NbOfRows * NbOfColumns);
```

and you would access the array like this:

```
result[row*NbOfColumns+column];
```

Note that you have to keep the dimensions of the array separately.

The array of pointers can be cumbersome but it gives us more flexibility. We can easily add a new row to the array, i.e. between the row 1 and 2. We just need to add a pointer after the first one, and move upward the other pointers. We do not have to move the data at all.

Not so with the shortcut that we described above. There, we have to move the data itself to make space for an extra row. In the case of arrays defined at compile time it is impossible to do anything since the dimensions of the array are "compiled in" at each access by the compiler.

2.3.1 Variable length arrays.

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

```
int Function(int n)
{
    int table[n];

}
```

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

2.3.2 Array initialization

To fill an array you could theoretically write a function like this:

```
int array[10];
void arrayinit(void)
{
    array[0] = 12;
    array[1] = 13;
    array[2] = 765;
    // and so on until array[9]
}
```

To avoid unnecessary typing, and speed up the program, you can ask the compiler to do the same operation at compile time by typing:

```
int array[10] = {12,13,765,123,5,0,0,21,78,1};
```

This has exactly the same result, but instead of doing the initialization at run time, it will be done during the compilation, producing a bit image of the array that will be loaded in the executable.

Obviously this is an improvement, but still, suppose you have an array of 20 positions, where all positions are zero excepting the 18th, that has the value 1.

Obviously you can write:

```
int array[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0};
```

This is quite error prone, specially if instead of 20 positions you have 200. For this situations, the standard language provides the following syntax:

```
int array[] = {[17] = 1};
```

This is much shorter, allowing you to initialize sparse arrays easily.

2.3.3 Compound literals

You can use a construct similar to a cast expression to indicate the type of a composite constant literal. For instance:

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

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

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

```
Person __998815544ss = { "Mary Smith'', 38};
        process(&__998815544ss);
```

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

2.4 Function calls

```
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. Obviously some conversions may be done to the result of evaluating those expressions. For instance if we call the function 'sqrt' that expects a double precision number but we find a call like this `c = sqrt(42);` The compiler will convert the integer 42 (the actual argument) into a double precision number before passing it to the 'sqrt' function.

Sometimes the conversion cannot be done. A call of `sqrt("Jane");` is an error since there is no conversion possible from a character string into a double precision number.

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.

The syntax for function pointers is the same as for a normal function call. It is not needed to dereference a pointer to a function. For instance

```
int main(void)
{
     int (*fn)(int);

     // fn is initialized somewhere here
     (*fn)(7);
     fn(7);
}
```

Both calls will work. If the called function has a prototype, the arguments are implicitly converted to the types of the corresponding parameters when possible. When this conversion fails, lcc-win issues an error and compilation fails. Other compilers may have different behavior. When a function has no prototype or when a function has a variable length argument list, for each argument the default argument promotions apply. The integer promotions are applied to each argument, and float arguments are passed as double.

2.4.1 Prototypes.

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

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

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

then, the call

```
fn();
```

will be accepted. If you then in file2.c you declare another prototype
`void fn(int);`
and then you use:
`fn(6);`
the compiler cannot see this, and the program will be in error, crashing mysteriously at run time. This kind of errors can be avoided if you always declare the prototypes in a header file that will be included by all files that use that function. Do not declare prototypes in a source file if the function is an external one.

2.4.2 Functions with variable number of arguments.

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

```
#include <stdarg.h>

int va_add(int numberOfArgs, ...)
{
        va_list ap;
        int n = numberOfArgs;
        int sum = 0;

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

We would call this function with

```
va_add(4,987,876,567,9556);
```

or

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

Implementation details

Under 32 bit systems (linux or windows) the variable arguments area is just a starting point in the stack area. When you do a `va_start(ap)`, the system makes to ap pointer point to the start of the arguments, just after the return address.

Later, when you retrieve something from the variable argument list, this pointer is incremented by the size of the argument just being passed in, and rounded to point to the next. This is quite simple and works in many systems.

Other systems, specially windows 64 bits or Linux 64 bits need a much more complicated schema since arguments are not passed in the stack but in predetermined registers. This forces the compiler to save all possible registers in a stack area, and retrieve the argument s from there. The issue is further complicated because some arguments are passed in some register sets (integer arguments for instance are passed in a different set as floating point arguments), and the compiler should keep pointers to different stack areas.

2.4.3 stdcall

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

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

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

In 64 bit systems (64 bit windows, and non windows systems like AIX, or Linux) lcc-win doesn't use this calling convention. The symbol `_stdcall` is accepted but ignored.

2.4.4 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-win 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''`.

2.5 Assignment.

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

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

Within an assignment there is the concept of "L-value", i.e. any assignable object. You can't, for instance, write: `5 = 8;`. The constant 5 can't be assigned to. It is not an "L-value", the "L" comes from the left hand side of the equals sign of course. In the same vein we speak of LHS and RHS as abbreviations for left hand side and right hand side of the equals sign in an assignment.

The rules for type compatibility in assignment are also used when examining the arguments to a function. When you have a function prototyped like this:

```
void fn(TYPE1 t1);
TYPE2 t2;
...
        fn(t2);
```

The same rules apply as if you had written: `t1 = t2;`

2.6 The four operations

This should be simple, everyone should be able to understand what a*b represents. There are some subtleties to remember however.

2.6.1 Integer division

When the types of both operands to a division are one of the integer types, the division performed is "integer" division by truncating the result towards zero. Here are some examples:

```
#include <stdio.h>
int main(void)
{
        printf("5/6=%d\n",5/6);
        printf("6/5=%d\n",6/5);
        printf("-6/5=%d, (-6)/5=%d\n",-6/5,(-6)/5);
        printf("(-23)/6=%d\n",(-23)/6);
}
```

The output of this program is:

```
5/6=0
6/5=1-6/5=-1, (-6)/5=-1
(-23)/6=-3
```

2.6.2 Overflow

All four arithmetic operations can produce an overflow. For signed integer types, the behavior is completely unspecified and it is considered an error by the standard. Floating point types (double or float for instance) should set the overflow flag and this flag can be queried by the program using the floating point exception functions.

Most modern computers can distinguish between two types of overflow conditions:

1. A computation produces a carry, i.e. the result is larger than what the destination can hold

2. The sign of the result is not what you would expect from the signs of the operands, a true overflow.

Both cases unless treated by the program will produce incorrect results. Historically, integer overflow has never been treated by the language with the attention it deserves. Everything is done to make programs run fast, but much less is done to make programs run correctly, giving as a result programs that can return wrong results at an amazing speed.

lcc-win is the only C compiler that gives the user access to the overflow flag, in a similar (and efficient) way that the programmer has access to the floating point exception flags. The built-in function

```
int _overflow(void);
```

This function will return 1 when the overflow flag is set, zero otherwise. To use this feature separate your operations in small steps, and call this pseudo function when needed. Instead of `c = (b+a)/(b*b+56);` write

```
c1 = b+a;,
if (_overflow()) goto ovfl;,
c2 = b*b;,
if (_overflow()) goto ovfl;c = c1/(c2+56);
```

This can become VERY boring, so it is better to give a command line argument to the compiler, that will generate the appropiate assembly to test each operation. The operations monitored are signed ones, unsigned operations wrap around.

2.6.3 Postfix

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

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

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

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

The integer a and the array element at the 234th 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.NOTE: Only one postfix expression is allowed for a given variable within a statement. For instance the expression:

```
i++ = i++;
```

is illegal C and will never work the same from one compiler to the next, or even within the same compiler system will change depending whether you turn optimizations on or off, for instance. The same applies for the decrement operator:

```
i-- = i--;
```

is also illegal. Note that this holds even if the expression is much more complicated than this:

```
i++ = MyTable[i--].Field->table[i++];
```

is completely illegal C.

2.7 Conditional operator

The general form of this operator is:

```
expression1 ? expression2 : expression3
```

The first operand of the conditional expression (expression1) is evaluated first. The second operand (expression2) 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. These expressions can be nested.

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

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

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

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

2.8 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.

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

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

2.8.1 Should we use the register keyword?

The register keyword is no longer really necessary with the improvement of the compiler technology. In most cases, the compiler can figure out better than the user which variables should be stored in registers and which variables should be stored in memory. Lcc-win tries to honor the register keyword, and it will follow your advice, but other compilers will ignore it and use their own schema. In general you can't rely that the register keyword means that the variable is not stored in memory.

2.9 Sizeof

The result of sizeof is an unsigned constant integer calculated at compile time. For instance `sizeof(int)` will yield under lcc-win 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.

For example:

```
int fn(int size)
{
    int tab[size];
}
```

Here the compiler can't know the size of tab in advance.

The maximum size of an object for an implementation is given by the macro `SIZE_MAX`, defined in limits.h. Lcc-win defines this as 4GB, but in 32 bit systems the actual maximum will be much lower than that. In 64 bit operating systems, a 32 bit program running an emulation layer can have all the addressing space of 4GB.

2.10 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 make `a` zero, `b` will be 1, and `c` will be 2. You can change this with `enum {a=10,b=25,c=76};`

2.10.1 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.

Implementation details

Lcc-win considers that when you declare:
`static const int myVar = 56;`
it is allowed to replace everywhere in your code the variable `myVar` with 56.

2.11 Goto

This statement transfers control to the given label. Many scientific papers have been written about this feature, and many people will tell you that writing a goto statement is a sin. I am agnostic. If you need to use it, use it. Only, do not abuse it.

Note that labels are always associated with statements. You can't write:

```
                if (condition) {
                        if (another_condition) {
                                ...
                                goto lab;
                        }
lab:                    // WRONG!
        }
```

In this case the label is not associated with any statement, this is an error. You can correct this by just adding an empty statement:

```
if (condition) {
        if (another_condition) {
                        ...
                        goto lab;
        }
lab:
        ; // empty statement
}
```

Now, the label is associated with a statement.

A goto statement can jump in the middle of a block, skipping the initialization of local variables. This is a very bad idea. For instance:

```
        if (condition)
                goto label;
        {
                int m = 0;
                ...
label:
        }
```

In this case, the jump will skip the initialization of the variable m. A very bad idea, since m can now contain any value.

2.12 Break and continue statements

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

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

is equivalent to:

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

The continue statement can be represented in a similar way:

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

is equivalent to:

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

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

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

2.13 Return

A return statement terminates the function that is currently executing, and returns (hence the name) to the function that called the current function.

2.13.1 Two types of return statements

Since a function can return a value or not, we have then, two types of return statements:

`return;`

or

`return expression;`

A return with no value is used in functions that have the return type void, i.e. they do not return any value. Functions that have a return type other than void must use the second form of the return statement. It is a serious error to use an empty return statement within a function that expects a returned value. The compiler will warn about this.

The type of the return will be enforced by the compiler that will generate any necessary conversion. For instance:

```
double fn(void)
{
    int a;
    // ...
    return a;
}
```

The compiler will generate a conversion from integer to double to convert a to double precision.

There is one exception to this rule. If the function main() does not return any value and control reaches the end of main(), the compiler will supply automatically a value of zero.

2.13.2 Returning a structure

When the return type of a function is a structure, and that structure is big enough to make it impossible to return the value in the machine registers, the compiler passes the address of a temporary storage to the function where the function writes its result. This is done automatically and does not need any special intervention from the user.

Other compilers may have different schematas for returning a structure. Under Windows, lcc-win uses the guidelines proposed by Microsoft. Non Microsoft compilers under windows may use different schematas.

2.13.3 Never return a pointer to a local variable

Suppose a function like this:

```
double *fn(void)
{
    double m;
    // ...
    return &m;
}
```

This is a serious error that will be flagged by the lcc-win compiler, but other compilers may be less explicit. The problem here is that the variable "m" lives in the activation record of the procedure "fn". Once the procedure returns, all storage associated with it is freed, including all storage where local variables are stored. The net effect of this return is to return a bad address to the calling function. The address is bad since the storage is no longer valid. Any access to it can provoke a trap, or (worst) can give wrong values when accessed.

2.13.4 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 2n for the values of that type. For instance, a signed short goes from –32768 to 32767, an unsigned short goes from zero to 65535 (216). See the standard include file `<limits.h>` for the ranges of signed and unsigned integer types.

2.14 Null statements

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

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

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

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

2.15 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. An example for this is the following:

```
switch (ch) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
        vowelCount++;
        break;
}
```

Here we have the same action for 5 cases, and we use to our advantage this feature.

There is a reserved word "default" that contains the case for all other values that do not appear explicitly in the switch. It is a good practice to always add this keyword to all switch statements and figure out what to do when the input doesn't match any of the expected values. If the input value doesn't match any of the enumerated cases and there is no default statement, no code will be executed and execution continues after the switch.

Conceptually, the switch statement above is equivalent to:

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

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

- Switch expressions must be of integer type. The "if" form doesn't have this limitation.
- In the case of a sizeable number of cases, the compiler will optimize the search in a switch statement to avoid comparisons. This can be quite difficult to do manually with "if"s.
- Cases of type other than int, or ranges of values can't be specified with the switch statement, contrary to other languages like Pascal that allows a range here.

Switch statements can be nested to any level (i.e. you can write a whole switch within a case statement), but this makes the code unreadable and is not recommended.

2.16 Logical operators

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

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

Example:

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

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

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

If we have the expression:

`result = expr1 && expr2;`

this is equivalent to the following C code:

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

In a similar way, we can say that the expression

`result = expr1 || expr2;`

is equivalent to:

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

2.17 Bitwise operators

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

The operation of each of them is as follows:

1. The `&` (AND) operator yields a 1 bit if both arguments are 1. Otherwise it yields a 0.
2. The `^` (exclusive or) operator yields 1 if one argument is 1 and the other is zero, i.e. it yields 1 if their arguments are different. Otherwise it yields zero
3. The `|` (or) operator yields 1 if either of its arguments is a 1. Otherwise it yields a zero.

We can use for those operators the following truth table.

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

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

`0 && fn(67);`

the function call will never be executed. If we write

`0 & fn(67);`

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

2.18 Shift operators

Shifts are performed between two operands of integer type. The type of the result is determined by the type of the left operand. Supposing 32 bit integers the operation

```
int a = 4976;
int b = a << 3;
```

consists of:

```
   1 0011 0111 0000      (4976)
1001 1011 1000 0000     (39808)
```

This is the same than 4976 * 8 = 39808, since 8 is `1<<3`. A shift to the left is equal to multiplying by 2, a shift to the right is equivalent to dividing by two. Obviously if you make shifts bigger than the number of bits in the integer you are shifting you get zero.

This snippet:

```
#include <stdio.h>
int main(void)
{
    int a = 4977;
    int b = -4977;
    int i;

    for (i=0; i<5;i++) {
        printf("4977 << %d:  %8d (0x%08x)\n",i,a << i,a << i);
        printf("-4977<< %d:  %8d (0x%08x)\n",i,b << i,b << i);
    }
}
```

produces the following output:

```
4977 << 0:      4977 (0x00001371)
-4977<< 0:     -4977 (0xffffec8f)
4977 << 1:      9954 (0x000026e2)
-4977<< 1:     -9954 (0xffffd91e)
4977 << 2:     19908 (0x00004dc4)
-4977<< 2:    -19908 (0xffffb23c)
4977 << 3:     39816 (0x00009b88)
-4977<< 3:    -39816 (0xffff6478)
4977 << 4:     79632 (0x00013710)
-4977<< 4:    -79632 (0xfffec8f0)
```

We have shifted a nibble (4 bits) We see it when we compare the last two lines with the first ones.

The standard specifies that a right shift with a negative number is implementation defined. It can be that in another machine you would get different results.

Right shifts are obviously very similar to left shifts. In the Intel/Amd family of machines there is a different operation code for signed or unsigned right shift, one filling the shifted bits with zero (unsigned right shift) and the other extending the sign bit.

2.19 Address-of operator

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

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

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

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

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

2.20 Indirection

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

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

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

        *p = 0;
        return 1;
}
```

We have never assigned to p an object where it should point to. We are using a dangling pointer.

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

`0xfffa5a5a.`

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

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

This kind of problem is one of the most common bugs in C. Forgetting to initialize a pointer is something that you can never afford to do. Another error is initializing a pointer within a conditional expression:

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

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

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

2.21 Sequential expressions

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

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

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

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

`fn(cd=6,78);;`

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

2.22 Casts

A cast expression specifies the conversion of a value from one type to another. For instance, a common need is to convert double precision numbers into integers. This is specified like this:

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

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

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

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

2.22.1 When to use casts

A case when casts are necessary occurs when passing arguments to a variadic function. Since the type of the arguments can't be known by the compiler, it is necessary to cast a value to its exact expected type (double to float for example), so that the arguments are converted to the exact types the variadic function expects. For instance:

```
float f;
printf("%Lg\n",(long double)f);
```

The printf function expects a long double (format Lg). We need to convert our float f into a long double to match the expectations of printf. If the cast is eliminated, the promoted value for a float when passed to a variadic function (or to a function without prototype) is double. In that case printf would receive a double and would expect a long double, resulting in a run time error or in bad output.

Another use of casts is to avoid integer division when that is not desired:

```
int a,b;
double c;
...
c = a/b; // Invokes integer division.
c = a/(double)b; // Invokes floating point division.
```

In the first case, integer division truncates the result before converting it to double precision. In the second case, double precision division is performed without any truncation.

2.22.2 When not to use casts

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

Many people add casts to get rid of compiler warnings. A cast tells essentially to the compiler "I know what I am doing. Do not complain". But this can make bugs more difficult to catch. For instance lcc-win warns when a narrowing conversion is done since the narrowed value could be greater than the capacity of the receiving type.

```
char c;
long m;
c = m; // Possible loss of data.
```

It is easy to get rid of that warning with a cast. But is this correct?

Some people in the C community say that casts are almost never necessary, are a bad programming practice, etc. For instance instead of:

```
void *p;
int c = *((char *)p);
```

they would write:

```
void *p;
char *cp = p;
int c = *cp;
```

This is more of a matter of aesthetic. Personally I would avoid the temporary variable since it introduces a new name, and complicates what is otherwise a simple expression.

2.23 Selection

A structure can have several different fields. The operators `.` and `->` select from a variable of structure type one of the fields it contains. For instance given:

```
struct example {
        int amount;
        double debt_ratio;
```

```
};
struct example Exm;
struct example *pExm;
```

you can select the field `debt_ratio` using `Exm.debt_ratio`. If you have a pointer to a structure instead of the structure itself, you use `pExm->debt_ratio`. This leads to an interesting question: Why having two operators for selection?

It is obvious that the compiler can figure out that a pointer needs to be dereferenced, and could generate the code as well if we would always write a point, as in many other languages. This distinction has historical reasons. In a discussion about this in comp.lang.c Chris Torek, one of the maintainers of the gcc C library wrote:

The "true need" for separate `.` and `->` operators went away sometime around 1978 or 1979, around the time the original K&R white book came out. Before then, Dennis' early compilers accepted things like this:

```
struct { char a, b; };
int x 12345; /* yes, no "=" sign */

main() {
    printf("%d is made up of the bytes %d and %d\n", x,
        (x.a) & 0377, (x.b) & 0377);
}
```

(in fact, in an even-earlier version of the compiler, the syntax was `struct (` rather than `struct {`. The syntax above is what appeared in V6 Unix. I have read V6 code, but never used the V6 C compiler myself.)

Note that we have taken the "a" and "b" elements of a plain "int", not the "struct" that contains them. The "." operator works on *any* lvalue, in this early C, and all the structure member names must be unique – no other struct can have members named "a" and "b".

We can (and people did) also write things like:

```
struct rkreg { unsigned rkcsr, rkdar, rkwc; };
...
    /* read disk sector(s) */
    0777440->rkdar = addr;
    0777440->rkwc = -(bytecount / 2);
    0777440->rkcsr = RK_READ | RK_GO;
```

Note that the `->` operator works on *any* value, not just pointers.

Since this "early C" did not look at the left hand side of the `.` and `->` operators, it really did require different operators to achieve different effects. These odd aspects of C were fixed even before the very first C book came out, but – as with the "wrong" precedence for the bitwise `&` and `|` operators – the historical baggage went along for the ride.

2.24 Predefined identifiers

A very practical predefined identifier is `__func__` that allows you to obtain the name of the current function being compiled. Combined with the predefined preprocessor symbols `__LINE__` and `__FILE__` it allows a full description of the location of an error or a potential problem for logging or debug purposes.

An example of the usage of those identifiers is the macro require, that tests certain entry conditions before the body of a function:

```
#define require(constraint) \
  ((constraint) ? 1 : ConstraintFailed(__func__,#constraint,NULL))
```

For instance when we write: `require(Input >= 9)`we obtain:
`((Input >= 9) ? 1 : ConstraintFailed(__func__,"Input >= 9",NULL) );`

2.25 Precedence of the different operators.

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

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

&&	logical and	binary	5	left-to-right
||	logical or	binary	4	left-to-right
? :	conditional	binary	2	right-to-left
= += -=	assignment	binary	2	right-to-left,
*= /= %=	assignment	binary	2	right-to-left,
<<= >>=	assignment	binary	2	right-to-left,
&= ^= |=	assignment	binary	2	right-to-left,
	sequential evaluation	binary	1	left-to-right

2.26 The printf family

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

Figure 2.1: The parts of a printf specification

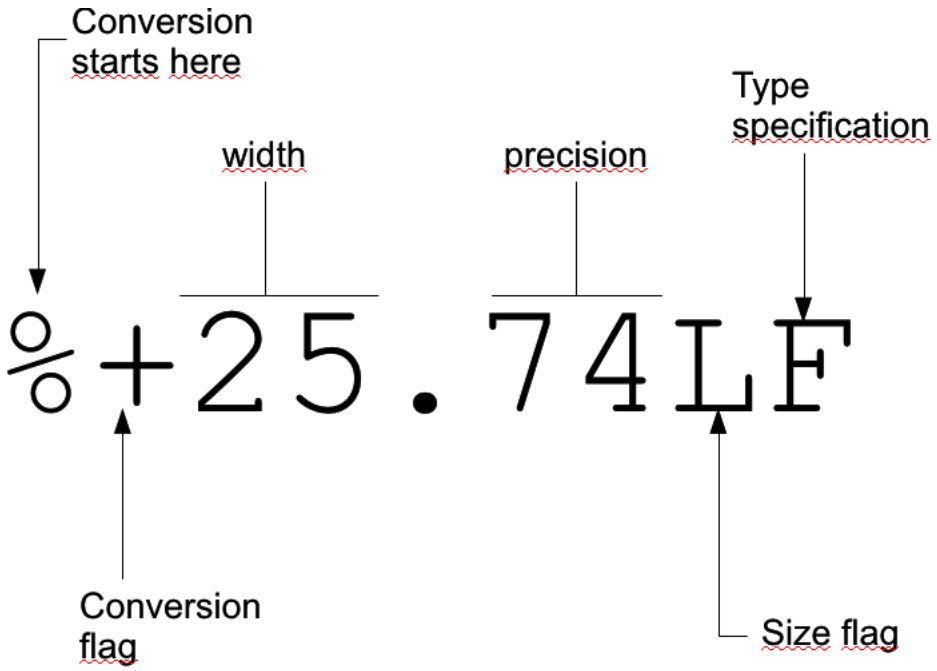


Table 2.2: Prototypes for the main functions of the printf family

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

an implicit argument "stdout", i.e. the standard output of the program, that in most cases is the console window.

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

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

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

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

2.26.1 Conversions

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

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

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

   For instance the output of this program:

   ```
   #include <stdio.h>
   int main(void)
   {
           printf("%5s\n","1234567890");
   }
   ```

   is `1234567890` and NOT `12345` as expected. If you want to specify a maximum field width you should use the precision field, not the width field.

3. An optional precision field made of a period followed by a number. This is where you specify the maximum field width.

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

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

Table 2.3: The conversion flags

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

2.26.2 The minimum field width

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

2.26.3 The precision

In floating point numbers (formats g G f e E) this indicates the number of digits to be printed after the decimal point. Used with the s format (strings) it indicates the maximum number of characters that should be used from the input string. If the precision is zero, the floating point number will not be followed by a period and rounded to the nearest integer. Table 2.26.3 shows the different size specifications, together with the formats they can be used with. Well now we can pass to the final part.

2.26.4 The conversions

Conversion	**Description**
d,i	Signed integer conversion is used and the argument is by default of type int. If the h modifier is present, the argument should be a short, if the ll modifier is present, the argument is a long long.
u	Unsigned decimal conversion. Argument should be of type unsigned int (default), unsigned short (h modifier) or unsigned long long (ll modifier).
o	Unsigned octal conversion is done. Same argument as the u format.

Table 2.5 – Continued

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

2.26.5 Scanning values

The scanf family of functions fulfills the inverse task of the printf family. They scan a character array and transform sequences of characters into values, like integers, strings or floating point values. The general format of these functions are:

Table 2.4: The size specification

Sign	Formats	Description
`l`	d,i,o,u,x, X	The letter `l` with this formats means long or unsigned long.
`l`	n	The letter `l` with the n format means long *.
`l`	c	Used with the c (character) format, it means the character string is in wide character format.
`l`	all others	No effect, is ignored
`ll`	d, i, o, u, x, X	The letters ll mean long long or unsigned long long.
`ll`	n	With this format, ll means long long *.
`h`	d, i, o, u, x, X	With this formats, h indicates short or unsigned short.
`h`	n	Means short *.
`hh`	d, i, o, u, x, X	Means char or unsigned char.
`hh`	n	Means char * or unsigned char *.
`L`	A, a, E, e, F, f, G, g,	Means that the argument is a long double. Notice that the l modifier has no effect with those formats. It is uppercase L.
`j`	d, i, o, u, x, X	Means the argument is of type `intmax_t`, i.e. the biggest integer type that an implementation offers. In the case of lcc-win this is long long.
`q`	f,g,e	Means the argument is of type qfloat (350 bits precision). This is an extension of lcc-win.
`t`	d, i, o, u, x, X	Means the argument is `ptrdiff_t`, under lcc-win int in the 32 bits version, 64 in the 64 bits version.
`Z`	e, E, g, G, f, F, A, a	Means the argument is a complex number. Output is in standard complex notation. If the alternative flag is present ('#') the output will have a lowercase 'i' instead of the standard "*I" suffix. Each of the other qualifiers that applies to the floating format will be applied to the real and to the imaginary parts of the number. Note that this a lcc-win extension.
`z`	d, i, o, u, x, X	Means the argument is `size_t`, in lcc-win unsigned int in 32 bits, unsigned long long in 64 bits.

Table 2.6: scanf directives

Type	Format
char	c
short	hd
int	d or i
long	ld
long long	lld
float	f or e
double	lf or le
long double	Lf or Le
string	s

```
scanf(format,p1,p2,p3...); // Reads characters from stdin
fscanf(file,format,p1,p2,p3,...);
sscanf(string,format,p1,p2,p3,...);
```

where format is a character string like we have seen in printf, and p1,p2,p3 are pointers to the locations where scanf will leave the result of the conversion. For instance:

```
scanf("%d",&integer);
```

and the input line
`123`
will store the integer 123 at the location of the "integer" variable.
Some things to remember when using scanf:

1. You have to provide always a format string to direct scanf.
2. For each format directive you must supply a corresponding pointer to a suitable sized location.
3. Leading blanks are ignored except for the `%c` (character) directive.
4. The "`%f`" format means float, to enter a double you should write "`%lf`".
5. Scanf ignores new lines and just skips over them. However, if we put a `\n` in the input format string, we force the system to skip a new line.
6. Calls to scanf can't be intermixed with calls to getchar, to the contrary of calls to printf and putchar.
7. When scanf fails, it stops reading at the position where it finds the first input character that doesn't correspond to the expected sequence.

If you are expecting a number and the user makes a mistake and types 465x67, scanf will leave the input pointing to the "x", that must be removed by other means. Because of this problem it is always better to read one line of input and then using sscanf in the line buffer rather than using scanf directly with user input. Here are some common errors to avoid:

```
int integer;
short shortint;
char buffer[80];
char* str = buffer;

/* providing the variable instead of a pointer to it */
sscanf("%d", integer);        /* wrong! */
sscanf("%d", &integer);       /* right */

/* providing a pointer to the wrong type
  (or a wrong format to the right pointer) */
sscanf("%d", &shortint);       /* wrong */
sscanf("%hd", &shortint);      /* right */

/* providing a pointer to a string pointer
   instead of the string pointer itself.
 (some people think "once &, always &) */
sscanf("%s", &str);     /* wrong */
sscanf("%s", str);      /* right */
```

Consider the following code:

```
#include <stdio.h>
int main(void)
{
        int i;
        char c;

        scanf("%d",&i);
        scanf("%c",&c);
        printf("%d %c\n",i,c);
}
```

Assume you type `45\n` in response to the first scanf. The 45 is copied into variable n. When the program encounters the next scanf, the remaining `\n` is quickly copied into the variable c. The fix is to put explicitly a `\n` like this: `scanf("%d\n",&i);`.

2.27 Pointers

Pointers are one of the great ideas in C, but it is one that is difficult to grasp at the beginning. All objects (integers, structures, any data) reside in RAM. Conceptually memory is organized in a linear sequence of locations, numbered from 0 upwards. Pointers allow you to pass the location of the data instead of the data itself.

To make things explicit, suppose you have some structure like this:

```
#define MAXNAME 128
struct person {
        char Name[MAXNAME];
```

```
        int Age;
        bool Sex;
        double Weight;
};
```

Instead of passing all the data to a function that works with this data, you just pass the address where it starts. What is this address? We can print it out. Consider this simple program:

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #define MAXNAME 128
4  struct person {
5          char Name[MAXNAME];
6          int Age;
7          bool Sex;
8          double Weight;
9  };
10 struct person Joe;
11 int main(void)
12 {
13        printf("0x%x + %d\n",&Joe,sizeof(struct person));
14 }
```

The address-of operator in line 13 returns the index of the memory location where the "Joe" structure starts. In my machine this prints: `0x402004 + 144`.

The memory location 0x402004 (4 202 500 in decimal) contains the start of this data, that goes up to 0x402094 (4 202 644).

When we write a function that should work with the data stored in that structure, we give it just the number 4 202 500. That means: "The data starts at 4 202 500". No copying needed, very efficient.

A pointer then, is a number that contains the machine address, i.e. the number of the memory location, where the data starts. The integer that contains a memory location is not necessarily the same as a normal "int", that can be smaller or bigger than an address. In 64 bit systems, for instance, addresses can be 64 bits wide, but "int" can remain at 32 bits. In other systems (Win 32 for instance) a pointer fits in an integer.

Pointers must be initialized before use by making them point to an object. Before initialization they contain a NULL value if they are defined at global scope, or an undefined value if they are local variables. It is always a very bad idea to use an uninitialized pointer.

Memory locations are dependent on the operating system, the amount of memory installed, and how the operating system presents memory to the programmer. Never make many assumptions about memory locations. For instance, the addresses we see now under windows 32 bit could radically change in other context, where they become 64 bit addresses. Anyway, under windows we use virtual memory, so those numbers are virtual addresses, and not really memory locations inside the circuit board. .

A pointer can store the start address of an object, but nothing says that this object continues to exist. If the object disappears, the pointers to it contain now invalid addresses, but it is up to the programmer to take care of this. An object can disappear if, for instance, its address is passed to the "free" function to release the memory. An object can disappear if its scope (i.e. the function where it was defined) ends. It is a beginner's mistake to write:

```
int *fn(int a)
{
        int a;
        ...
        return &a;
}
```

The "a" variable has a duration defined until the function "fn" exits. After that function exits, the contents of all those memory locations containing local variables are undefined, and the function is returning a pointer to memory that will be freed and recycled immediately. Of course the memory location itself will not disappear, but the contents of it will be reassigned to something else, maybe another function, maybe another local variable, nobody knows.

A pointer that is not initialized or that points to an object that has disappeared is a "dangling" pointer and it is the nightmare of any C programmer. The bugs produced by dangling pointers are very difficult to find, since they depend on whether the pointers destroy or not some other object. This programs tend to work with small inputs, but will crash mysteriously with complex and big inputs. The reason is that in a more complex environment, object recycling is done more often, what means that the memory locations referenced by the dangling pointers are more likely used by another object.

2.27.1 Operations with pointers

The most common usage of a pointer is of course the "dereferencing" operation, i.e. the operator `->` or the unary `*`. This operations consist of reading the contents of a pointer, and using the memory address thus retrieved either fetch the data stored at that place or at some displacement from that place. For instance when we use a pointer to the "person" structure above the operation:

```
struct person *pJoe = &Joe;
pJoe->weight
```

means:

1. Fetch the contents of the "pJoe" pointer.
2. Using that address, add to it sizeof(Name[MAXNAME]) + sizeof(int) + sizeof(bool)
3. Retrieve from the updated memory location a "double" value

The operation 2) is equivalent to finding the offset of the desired field within a given structure. This is often required and the language has defined the macro "offsetof" in the "stddef.h" header file for using it within user's programs.

Pointers can be used to retrieve not only a portion of the object (operation ->) but to retrieve the whole object using the "*" notation. In the example above the operation "*pJoe" would yield as its result the whole structure. This operation dereferences the pointer and retrieves the entire object it is pointing to, making it the exact opposite of the "&" (address-of) operator.

Only two kinds of arithmetic operations are possible with machine addresses: Addition or subtraction of a displacement, or subtraction of two machine addresses. No other arithmetic operators can be applied to pointers.

2.27.2 Addition or subtraction of a displacement: pointer arithmetic

Adding a displacement (an integer) to a pointer means:

Using the address stored in that pointer, find the *nth* object after it. If we have a pointer to int, and we add to it 5, we find the 5th integer after the integer whose address was stored in the pointer.

Example:

```
int d[10];
int *pint = &d[2];
```

The number stored in pint is the memory location index where the integer "d" starts, say 0x4202600. The size of each integer is 4, so the 3rd integer after 402600 starts at 0x4202608. If we want to find the fifth integer after 0x4202608, we add 20 (5*sizeof(int) = 20) and we obtain 0x420261C.

To increase a pointer by one means adding to it the size of the object it is pointing to. In the case of a pointer to integer we add 4 to 0x204600 to obtain the next integer. This is very often written with the shorthand: `pint++;` or `++pint;`.

This is a short hand for "Move the pointer to point to the next element". Obviously this is exactly the same for subtraction. Subtracting a displacement from a pointer means to point it to the *nth* element stored before the one the pointer is pointing to. Id we have a pointer to int with a value of 0x4202604, making: `p--;` meaning that we subtract 4 from the value of the pointer, to make it point to the integer stored at address 0x4202600, the previous one.

To be able to do pointer arithmetic, the compiler must know what is the underlying type of the objects. That is why you can't do pointer arithmetic with void pointers: the compiler can't know how much you need to add or subtract to get to the next element!

2.27.3 Subtraction

The subtraction of two pointers means the distance that separates two objects of the same type. This distance is not expressed in terms of memory locations but in terms of the size of the objects. For instance the distance between two consecutive objects is always one, and not the number of memory locations that separates the start addresses of the two objects.

The type of the result is an integer, but it is implementation defined exactly which (short, int, long, etc). To make things portable, the standard has defined a special typedef, `ptrdiff_t` that encapsulates this result type. Under lcc-win this is an "int" but under other versions of lcc-win (in 64 bit architectures for instance) it could be something else.

2.27.4 Relational operators

Pointers can be compared for equality with the == operator. The meaning of this operation is to find out if the contents of two pointers are the same, i.e. if they point to the same object. The other relational operators are allowed too, and the result allows to know which pointer appears before or later in the linear memory space.

Obviously, this comparisons will fail if the memory space is not linear, as is the case in segmented architectures, where a pointer belongs to a memory region or segment, and inter-segment comparisons are not meaningful.

2.27.5 Null pointers

A pointer should either contain a valid address or be empty. The "empty" value of a pointer is defined as the NULL pointer value, and it is usually zero.

Using an empty pointer is usually a bad idea since it provokes a trap immediately under lcc-win, and under most other compilers too. The address zero is specifically not mapped to real addresses to provoke an immediate crash of the program. In other environments, the address zero may exist, and a NULL pointer dereference will not provoke any trap, but just reading or writing to the address zero.

2.27.6 Pointers and arrays

Contrary to popular folklore, pointers and arrays are NOT the same thing. In some circumstances, the notation is equivalent. This leads to never ending confusion, since the language lacks a correct array type. Consider this declarations:

```
int iArray[3];
int *pArray = iArray;
```

This can lead people to think that pointers and arrays are equivalent but this is just a compiler trick: the operation being done is: `int *pArray = &iArray[0];`.

Another syntax that leads to confusion is: `char *msg = "Please enter a number";` Seeing this leads people to think that we can assign an entire array to a pointer, what is not really the case here. The assignment being done concerns the pointer that gets the address of the first element of the character array.

2.27.7 Assigning a value to a pointer

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:

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

2.27.8 References

In lcc-win 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.

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;
```

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.

2.27.9 Why pointers?

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".

2.28 setjmp and longjmp

2.28.1 General usage

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

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

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

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

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

```
}

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

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

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

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

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

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

```
int main(void)
{
```

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

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

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

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

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

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

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

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

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

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

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

2.28.2 Register variables and longjmp

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

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

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

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

```
1: 1
2: 2
3: 3
```

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

```
1: 1
2: 2
3: 2
```

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

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

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

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

2.29 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.

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;
             // Difference (minutes), moving westward, between
             // UTC and local time
        short timezone;
        unsigned short pad2;
        unsigned long lpad2;
             // Nonzero if daylight savings time is currently
```

```
            // in effect for the local time zone.
        short dstflag;
};
```

We show here a small program that displays the different time settings.

```
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include <string.h>
#ifdef _MSC_VER
/* Microsoft compilers call these functions differently */
#define strdate _strdate
#define strtime _strtime
#endif

int main(void)
{
    char tmpbuf[128], ampm[3];
    time_t ltime;
    struct _timeb tstruct;
    struct tm *today, *gmt, xmas = { 0, 0, 12, 25, 11, 113 };

    /* Display operating system-style date and time. */
    /* Windows specific functions */
#ifdef _WIN32
    strtime( tmpbuf );
    printf( "OS time:\t\t\t\t%s\n", tmpbuf );
    strdate( tmpbuf );
    printf( "OS date:\t\t\t\t%s\n", tmpbuf );
#endif

    /* 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. */
    ampm[1] = 'M'; ampm[2]=0;
    today = localtime( &ltime );
    if( today->tm_hour > 12 ) {
        ampm[0] = 'P;
```

```
        today->tm_hour -= 12;
    }
    else ampm[0]='A';
    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%d\n",
             tstruct.timezone );
    printf( "Daylight savings:\t\t\t%s\n", // See note (3) in text
             tstruct.dstflag ? "YES" : "NO" );

    /* Make time for noon on Christmas, 2013. */
    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 month of %B in the year %Y.\n",
         today );
    printf( tmpbuf );
}

OUTPUT:
OS time:                                17:53:23
OS date:                                11/30/11
Time in seconds since UTC 1/1/70:       1322693603
UNIX time and date:                     Wed Nov 30 17:53:23 2011
Coordinated universal time:             Wed Nov 30 22:53:23 2011
12-hour time:                           05:53:23 PM
Plus milliseconds:                      6
Zone difference in seconds from UTC:    -60
Daylight savings:                       NO
Christmas                               Wed Dec 25 12:00:00 2013

Today is Wednesday, day 30 month of November in the year 2011.
```

You will need some conversion functions to convert between the C time and the

windows time format:

```
#include <winbase.h>
#include <winnt.h>
#include <time.h>

void UnixTimeToFileTime(time_t t, LPFILETIME pft)
{
  long long ll;

  ll = Int32x32To64(t, 10000000) + 116444736000000000;
  pft->dwLowDateTime = (DWORD)ll;
  pft->dwHighDateTime = ll >> 32;
}
```

Once the UNIX time is converted to a FILETIME structure, other Windows time formats can be easily obtained using Windows functions such as `FileTimeToSystemTime ()`, and `FileTimeToDosDateTime ()`.

```
void UnixTimeToSystemTime(time_t t, LPSYSTEMTIME pst)
{
  FILETIME ft;

  UnixTimeToFileTime(t, &ft);
  FileTimeToSystemTime(&ft, pst);
}
```

3 Simple programs

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

3.1 strchr

Find the first occurrence of a given character in a character string. Return a pointer to the character if found, NULL otherwise.This problem is solved in the standard library by the `strchr` function. Let's write it.

The algorithm is very simple: We examine each character. If it is zero, this is the end of the string, we are done and we return NULL to indicate that the character is not there. If we find it, we stop searching and return the pointer to the character position.

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

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

3.1.1 How can strchr fail?

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

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

This is really bad news, since the calling program may not use the result immediately. It could be that the result is stored in a variable, for instance, and then used in another, completely unrelated section of the program. The program would crash without any hints of what is wrong and where was the failure. Note that this implementation of strchr will correctly accept a zero as the character to be searched. In this case it will return a pointer to the zero byte.

3.2 strlen

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

3.2.1 A straightforward implementation

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.

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.

3.2.2 An implementation by D. E. Knuth

... one of the most common programming tasks is to search through a long string of characters in order to find a particular byte value. For example strings are often represented as a sequence of nonzero bytes terminated by 0. In order to locate the end of a string quickly, we need a fast way to determine whether all eight bytes of a given word x are nonzero (because they usually are).

I discovered that quote above in the fascicle 1a in "Bitwise Tricks and Techniques" when reading D. E. Knuth's pages:
`http://www-cs-faculty.stanford.edu/~knuth/fasc1a.ps.gz`.

In that document, Knuth explains many boolean tricks, giving the mathematical background for them too, what many other books fail to do. I adapted his algorithm to C. It took me a while because of a stupid problem, but now it seems to work OK. The idea is to read 8 bytes at a time and use some boolean operations that can be done very fast in assembly to skip over most of the chain, stopping only when a zero byte appears in one of the eight bytes read. Knuth's strlen then, looks like this.

```
#include <stdio.h>
#include <string.h>
#define H 0x8080808080808080ULL
#define L 0x0101010101010101ULL
size_t myStrlen(char *s)
{
    unsigned long long t;
    char *save = s;

    while (1) {
        // This supposes that the input string is aligned
        // or that the machine doesn't trap when reading
        // a 8 byte integer at a random position like the x86
        t = *(unsigned long long *)s;
        if (H & (t - L) & ~t)
            break;
        s += sizeof(long long);
    }
    // This loop will be executed at most 7 times
    while (*s) {
        s++;
    }
    return s - save;
}

#ifdef TEST
int main(int argc,char *argv[])
{
    char *str = "The lazy fox jumped over the slow dog";

    if (argc > 1) {
        str = argv[1];
    }
    printf(
        "Strlen of '%s' is %d (%d)\n",
            str,strlen(str),myStrlen(str));
}
#endif
```

There are two issues with this code. The first is that it reads 8 bytes from a random location, what can make this code trap if used in machines that require aligned reads.

The second is that it reads some bytes beyond the end of the string, if the length of the string is not a multiple of eight.

To solve the first problem, we should read the bytes of the string until our pointer is correctly aligned, i.e. its value is a multiple of eight.

```
intptr_t i;
unsigned int n;

i = (intptr_t)s;
n = i&3;
while (n && *s)
        --n,++s;
```

This will align our pointer before the main loop starts.

To solve the second problem is impossible within the frame of the given algorithm. We can't know that we have read beyond the end of the string until after we have done it. This will never be a problem in any existing machine and in any runtime since we can't have a page boundary at an unaligned address. The only problem that could arise can come from debugging setups, where reading beyond the end of a string can be detected by special measures.

The two "magic constants" H and L can be obtained from standard values by using:

```
#include <limits.h>
#define L (ULLONG_MAX / UCHAR_MAX)
#define H (L << (CHAR_BIT - 1))
```

3.2.3 How can strlen fail?

The same problems apply that we discussed in the previous example, but in an attenuated form: only a wrong answer is returned, not an outright wrong pointer. The program will only stop at a zero byte. If (for whatever reason) the passed string does not have a zero byte this program will go on scanning memory until a zero byte is found by coincidence, or it will crash when attempting to reference inexistent memory.

3.3 ispowerOfTwo

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

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

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

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

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

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

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.

3.3.1 How can this program fail?

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

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

Why?

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

3.3.2 Write ispowerOfTwo without any loops

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

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

How does this work?

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

For instance 32 is a power of two. It is represented in binary as: `100000` 32-1 is 31 and is represented as:`011111` `32&31` is:

`100000 & 011111 ==> 0`

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

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

Negative powers of two (0.5, 0.25, 0.125, etc) could share this same property in a suitable fraction representation. 0.5 would be 0.1, 0.250 would be 0.01, 0.125 would be 0.001 etc.

This snippet and several others are neatly explained in:

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

3.4 signum

This function should return -1 if its argument is less than zero, zero for argument equal to zero, and 1 if the argument is bigger than zero. A straightforward implementation could look like this:

```
int signum1(double x)
{
        if (x < 0)
                return -1;
        else if (x == 0)
                return 0;
        else return 1;
}
```

We can rewrite that in a more incomprehensible form:

```
int signum2(double x) { return (x<0)?-1:(x==0)?0:1;}
```

Note that the second form is identical to the first in the generated code. Only in the source code there is a noticeable loss in readability.

A more interesting implementation is this one:

```
int signum3(double x) { return (x > 0) - (x < 0); }
```

All those forms are equivalent in speed and size. The only difference is that the first form is immediately comprehensible.

3.5 strlwr

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

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

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

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

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

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

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

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

3.5.1 How can this program fail?

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

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

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

3.6 paste

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

```
line 1
line2
```

and you got another file2 with this contents

```
line 10
line 11
```

you want to obtain a file

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

Note that both files can be the same.

A solution for this could be the following program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
/* We decide arbitrarily that lines longer than 32767 chars
  will make this program fail. */
#define MAXLINELEN 32767
int main(int argc,char *argv[])
{
/* We need two FILE pointers, and two line buffers to hold
   each line from each file. We receive in argc the number
   of arguments passed + 1, and in the character array
```

```
   argv[] the names of the two files */
        FILE *f1,*f2;
        char buf1[MAXLINELEN],buf2[MAXLINELEN];

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

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

3.6.1 How can this program fail?.

Well, there are obvious bugs in this program. Before reading the answer, try to see if you can see them. What is important here is that you learn how to spot bugs and that is a matter of logical thinking and a bit of effort. Solution will be in the next page. But just try to find those bugs yourself. Before that bug however we see this lines in there:

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

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

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

```
File1:
File 1: line 1
File 1: line 2
File 1: line 3
File 1: line 4
File 1: line 5
File 1: line 6
File 1: line 7
File 1: line 8

File2:
File 2: line 1
File 2: line 2
File 2: line 3
File 2: line 4
```

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

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

Is this a bug or a feature?

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

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

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

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

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

And now the answer to the other bug above.

One of the first things to notice is that the program tests with strcmp to see if two files are the same. This means that when the user passes the command line: `paste File1 filE1` our program will believe that they are different when in fact they are not. Windows is not case sensitive for file names. The right thing to do there is to compare the file names with stricmp, that ignores the differences between uppercase and lowercase.

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

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

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

`paste file1 file2 file1 file3`

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

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

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

3.7 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.

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

We declare an array of 256 integers, numbered from zero to 255. Note that in C the index origin is always zero. This array is not enclosed in any scope. Its scope then, is global, i.e. this identifier will be associated to the integer array for the current translation unit (the current file and its includes) from the point of its declaration on. Since we haven't specified otherwise, this identifier will be exported from the current module and will be visible from other modules. In another compilation unit we can then declare:

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

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

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.

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]++
```

That means: `Frequencies[count] = Frequencies[count]+1;`
i.e.; the integer at that array position is incremented, and not the count variable!

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

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

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

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

```
ABCDEFGHIJK
```

And we call:

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

```
frequencies fexample
```

and we obtain:

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

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

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

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

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

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

We test now our program with itself. We call: frequencies frequencies.c 758 chars in file I have organized the data in a table to easy the display.

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

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

3.7.1 How to sort arrays

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:

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

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

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

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:

```
typedef unsigned int size_t;
```

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

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

Argument 3 is also a `size_t`.

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

Argument 4 is a function:

```
int (*f)(const void *));
```

Well this is quite hard really. We are in the first pages of this introduction and we already have to cope with gibberish like this? We have to use recursion now. We have again to start reading this from left to right, more or less. We have a function pointer (f) that points to a function that returns an int, and that takes as arguments a void *, i.e. a pointer to some unspecified object, that can't be changed within that function (const).

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

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).

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 );
}
```

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

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

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

The structure of this example is as follows: We build a program that will sort its arguments and output the sorted result. To use qsort we define a comparison function that returns an integer, which encodes the relative lexical ordering of the two arguments passed to it. We use a library function for doing that, the stricmp function, that compares two character strings without caring about case differences. But there is quite a lot of new material in this example, and it is worth going through it in detail.

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

We define our comparison function with:
`int compare(const void *arg1,const void *arg2) { ... }`
This means that our compare function will return an int, and that takes two arguments, named arg1 and arg2, that are pointers to any object (void *).

The objects pointed to by arg1, and arg2 will not be changed within this function, i.e. they are "const".

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

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

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

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

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

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

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

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

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

To output the already sorted array we use again a "for" loop. Note that the index of the loop is declared at the initialization of the "for" construct. This is one of the new specifications of the C99 language standard, that lcc-win 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. 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.

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

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

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

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

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

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

We pass the Frequencies array, its size, the size of each element, and our comparison function. Here is the new version of our program, for your convenience. New code is in bold:

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

int Frequencies[256]; // Array of frequencies

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

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

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

```
    return 0;
}
```

We compile, link, and then we write

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

Well, sorting definitely works (you read this display line by line), but we note with dismay that All the character names are wrong!

```
  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
  Ò ( 210) = 3    Ó ( 211) = 3    Ô ( 212) = 3
... etc (20 lines elided)
```

Why?

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

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

3.7.2 Other qsort applications

Suppose you have a table of doubles, and you want to get an integer vector holding the indexes of each double precision number in a sorted vector. The solution is to make a comparison function that instead of looking at the double array to make its decision, receives the indexes vector instead, and looks in the values array. We establish two global variables:

1. a values array holding the double precision data, and
2. an indexes array holding the integer indices.

This are global variables since they should be accessible to the callback function whose interface is fixed: we can't pass more arguments to the comparison function.

We arrive then at the following program:

```
#include <stdio.h>
#include <stdlib.h>
#define N 100 // The size of the array
double values[N];
int indexes[N];
```

```
// This comparison function will use the integer pointer that it
// receives to index the global values array. Then the comparison
// is done using the double precision values found at
// those positions.
int compare(const void *pp1,const void *pp2)
{
    const int *p1 = pp1, *p2 = pp2;
    double val1 = values[*p1];
    double val2 = values[*p2];

    if (val1 > val2)
        return 1;
    else if (val1 < val2)
        return -1;
    else
        return 0;
}

int main(void)
{
    int i,r;

// We fill the array of double precision values with a
// random number for demonstration purposes. At the same
//  time we initialize our indexes array with a consecutive
// sequence of 0 <= i <= N
    for (i=0; i<N;i++) {
        r = rand();
        values[i] = (double) r;
        indexes[i] = i;
    }
// Now we call our qsort function
    qsort(indexes,N,sizeof(int),compare);
// We print now the values array, the indexes array, and the
// sorted vector.
    for (i=0; i<N; i++) {
        printf("Value %6.0f index %3d %6.0f\n",
                values[i],indexes[i],values[indexes[i]]);
    }
    return 0;
}
```

Another possibility is to make a structure containing a value and an index, and sort a table of those structures instead of sorting the values array directly, but this solution is less efficient in space terms.

3.7.3 Quicksort problems

The worst case of the quick sort algorithm is O(n2) but if you assume some randomness in the array to be sorted the worst case becomes very unlikely and everybody assumes that quicksort is O(n log n). M. D. McIlroy devised a C program that will kill most quicksort implementations. The basic insight is that, as we have seen above, the qsort routine calls a user supplied function to compare items, a specially crafted function will force qsort to go nuts!

Here are the comments that the author distributes with his software:

> Aqsort is an antiquicksort. It will drive any qsort mplementation based on quicksort into quadratic behavior, provided the implementation has these properties:
>
> 1. The implementation is single-threaded.
> 2. The pivot-choosing phase uses O(1) comparisons.
> 3. Partitioning is a contiguous phase of n-O(1) comparisons, all against the same pivot value.
> 4. No comparisons are made between items not found in the array. Comparisons may, however, involve copies of those items.
>
> **Method**
>
> Values being sorted are dichotomized into "solid" values that are known and fixed, and "gas" values that are unknown but distinct and larger than solid values. Initially all values are gas. Comparisons work as follows:
>
> - Solid-solid. Compare by value.
> - Solid-gas. Solid compares low.
> - Gas-gas. Solidify one of the operands, with a value greater than any previously solidified value. Compare afresh.
>
> During partitioning, the gas values that remain after pivot choosing will compare high, provided the pivot is solid. Then qsort will go quadratic. To force the pivot to be solid, a heuristic test identifies pivot candidates to be solidified in gas-gas comparisons.
>
> A pivot candidate is the gas item that most recently survived a comparison. This heuristic assures that the pivot gets solidified at or before the second gas-gas comparison during the partitioning phase, so that n-O(1) gas values remain.
>
> To allow for copying, we must be able to identify an operand even if it was copied from an item that has since been solidified. Hence we keep the data in fixed locations and sort pointers to them. Then qsort can move or copy the pointers at will without disturbing the underlying data.
>
> `int aqsort(int n, int *a);`
>
> Returns the count of comparisons qsort used in sorting an array of n items and fills in array a with the permutation of 0..n-1 that achieved the count.

Here is the program:

```
/* Copyright 1998, M. Douglas McIlroy. Permission is granted
to use or copy with this notice attached. */
#include <stdlib.h>
#include <assert.h>
int     *val;               /* array, solidified on the fly */
int     ncmp;               /* number of comparisons */
int     nsolid;             /* number of solid items */
int     candidate;          /* pivot candidate */
int     gas;                /* gas value = highest sorted value */
#define freeze(x) val[x] = nsolid++

int cmp(const void *px, const void *py)  /* per C standard */
{
        const int x = *(const int*)px;
        const int y = *(const int*)py;
        ncmp++;
        if(val[x]==gas && val[y]==gas)
                if(x == candidate)
                        freeze(x);
                else
                        freeze(y);
        if(val[x] == gas)
                candidate = x;
        else if(val[y] == gas)
                candidate = y;
        return val[x] - val[y];
}

int aqsort(int n, int *a)
{
        int i;
        int *ptr = malloc(n*sizeof(*ptr));
        val = a;
        gas = n-1;
        nsolid = ncmp = candidate = 0;
        for(i=0; i<n; i++) {
                ptr[i] = i;
                val[i] = gas;
        }
        qsort(ptr, n, sizeof(*ptr), cmp);
        for(i=1;i<n;i++)
                assert(val[ptr[i]]==val[ptr[i-1]]+1);
        free(ptr);
        return ncmp;
}
```

```
/* driver main program, to be linked with qsort
        usage: aqsort [-p] n
   constructs an adversarial input and reports the comparison
   count for it.  Option -p prints the adversarial input.
*/

#include <stdio.h>
#include <string.h>
int pflag;

int main(int argc, char **argv)
{
        int n, i;
        int *b;
        if(argc>1 && strcmp(argv[1],"-p") == 0) {
                pflag++;
                argc--;
                argv++;
        }
        if(argc != 2) {
                fprintf(stderr,"usage: aqsort [-p] n\n");
                exit(1);
        }
        n = atoi(argv[1]);
        b = malloc(n*sizeof(int));
        if(b == 0) {
                fprintf(stderr,"aqsort: out of space\n");
                exit(1);
        }
        i = aqsort(n, b);
        printf("n=%d count=%d\n", n, i);
        if(pflag)
                for(i=0; i<n; i++)
                        printf("%d\n", b[i]);
        exit(0);
}
```

3.8 Counting words

There is no introduction to the C language without an example like this:

"Exercise 24.C: Write a program that counts the words in a given file, and reports its result sorted by word frequency."

OK. Suppose you got one assignment like that. Suppose also, that we use the C language definition of a word, i.e. an identifier. A word is a sequence of letters that begins with an underscore or a letter and continues with the same set of characters

or digits. In principle, the solution could look like this:

1. Open the file and repeat for each character
2. If the character starts a word, scan the word and store it. Each word is stored once. If it is in the table already, the count is incremented, otherwise it is entered in the table.
3. Sort the table of words by frequency
4. Print the report.

We start with an outline of the "main" procedure. The emphasis when developing a program is to avoid getting distracted by the details and keep the main line of the program in your head. We ignore all error checking for the time being.

```
#include <stdio.h>
int main(int argc,char *argv[])
{
    FILE *f;
    int c;

    f = fopen(argv[1],"r");    // open the input file
    c = fgetc(f);              // Read the first character
    while (c != EOF) {         // Until the end of file
        if (isWordStart(c)) {  // Starts a word?
            ScanWord(c,f);     // Yes. Scan it
        }
        c = fgetc(f);          // Go on with the next character
    }
    fclose(f);                 // Done with the file
    DoReports(argv[1]);        // Print results
    return 0;                  // Return with OK.
}
```

This would do nicely. We have now just to fill the gaps. Let's start with the easy ones. A word, we said, is a sequence of `_ [A-Z] [a-z]` followed by `_ [A-Z] [a-z] [0-9]`. We write a function that returns 1 if a character is the start of an identifier (word).

```
int isWordStart(int c)
{
        if (c == '_')
                return 1;
        if (c >= 'a' && c <= 'z')
                return 1;
        if (c >= 'A' && c <= 'Z')
                return 1;
        return 0;
}
```

This function will do its job, but is not really optimal. We leave it like that for the time being. Remember: optimizations are done later, not when designing the program.

Now, we go to the more difficult task of scanning a word into the computer. The algorithm is simple: we just keep reading characters until we find a non-word char, that stops our loop. We use a local array in the stack that will hold until MAXIDLENGTH chars.

```
#define MAXIDLENGTH 512
int ScanWord(int firstchar,FILE *f)
{
    int i = 1, // index for the word buffer
            c=0;  // Character read
    char idbuf[MAXIDLENGTH+1]; // Buffer for the word

    idbuf[0] = firstchar; // We have at least one char
    c = fgetc(f);         // Read the next one
    while (isWordStart(c) || (c >= '0' && c <= '9')) {
        idbuf[i++] = c;  // Store it in the array
        if (i >= MAXIDLENGTH) { // Check for overflow!
            fprintf(stderr,
                      "Identifier too long\n");
            return 0;    // Returning zero will break the loop
                         // in the calling function
        }
        c = fgetc(f);    // Scan the next char
    }
    idbuf[i] = 0;         // Always zero terminate
    EnterWord(idbuf);     // Enter into the table
    return 1;             // OK to go on.
}
```

We hold the index into our array in the identifier "i", for index. It starts at one since we receive already the first character of a word. Note that we test with this index if we are going to overflow our local table "idbuf". We said before that error checking should be abstracted when designing the program but as any rule, that one has exceptions. If we were going to leave a lot of obvious errors in many functions around, we would need a lot of work later on to fix all those errors. Fundamental error checking like a buffer overrun should always be in our minds from the beginning, so we do it immediately. Note that this test is a very simple one.

3.8.1 The organization of the table

Now, we have no choice but to start thinking about that "EnterWord" function. All the easy work is done, we have to figure out now, an efficient organization for our word table. We have the following requirements:

1. It should provide a fast access to a word to see if a given sequence of characters is there already.

2. It should not use a lot of memory and be simple to use.

The best choice is the hash table. We use a hash table to hold all the words, and before entering something into our hash table we look if is in there already. Conceptually, we use the following structure:

Our word table is a sequence of lists of words. Each list is longer or shorter, depending on the hash function that we use and how good our hash function randomizes the input. If we use a table of 65535 positions (slots) and a good hash algorithm we divide the access time by 65535, not bad.

To enter something into our table we hash the word into an integer, and we index the slot in the table. We then compare the word with each one of the words in the list of words at that slot. If we found it, we do nothing else than increment the count of the word. If we do not find it, we add the word at the start of that slot.

Note that this requires that we define a structure to hold each word and its associated count. Since all the words are in a linked list, we could use the following structure, borrowing from the linked list representation discussed above:

```
typedef struct _WordList {
        int Count;
        struct _WordList *Next;
        char Word[];
} WORDLIST;
```

We have an integer that holds the number of times this word appears in the text, a pointer to the next word in the list, and an unspecified number of characters just following that pointer. This is a variable sized structure, since each word can hold more or less characters. Note that variable sized structures must have only one "flexible" member and it must be at the end of the definition.

Our "EnterWord" function can look like this:

```
void EnterWord(char *word)
{
    int h = hash(word); // Get the hash code for this word
    WORDLIST *wl = WordTable[h]; // Index the list at that slot
    while (wl) { // Go through the list
        if (!strcmp(wl->Word,word)) {
            wl->Count++; // Word is already in the table.
            return;      // increment the count and return
        }
        wl = wl->Next;   // Go to the next item in the list
    }
    // Here we have a new word, since it wasn't in the table.
    // Add it to the table now
    wl = NewWordList(word);
    wl->Next = WordTable[h];
```

```
    WordTable[h] = wl;
}
```

What would be a good hash function for this application?

This is a tutorial, so we keep things simple. Here is a very simple hash function:

```
int hash(char *word)
{
        int h = 0;
        while (*word) {
                h += *word;
                word++;
        }
        return h & 0xffff;
}
```

We just add up our characters. If we get a hash value of more than 65535 (the size of our table), we just take the lower 16 bits of the hash value. Easy isn't it?

3.8.2 Memory organization

We declare our word table now, like this:

```
WORDLIST *WordTable[0xffff+1];
```

Now we write the constructor for our word list structure. It should get more memory from the system to hold the new structure, and initialize its fields.

```
WORDLIST *NewWordList(char *word)
{
        int len = strlen(word);
        WORDLIST *result = more_memory(sizeof(WORDLIST)+len+1);
        result->Count = 1;
        strcpy(result->Word,word);
        return result;
}
```

We allocate more memory to hold the structure, the characters in the word, and the terminating zero. Then we copy the characters from the buffer we got, set the count to 1 since we have seen this word at least once, and return the result. Note that we do not test for failure. We rely on `more_memory` to stop the program if there isn't any more memory left, since the program can't go on if we have exhausted the machine resources.

Under windows, the implementation of the standard "malloc" function is very slow. To avoid calling "malloc" too often, we devise an intermediate structure that will hold chunks of memory, calling malloc only when each chunk is exhausted.

```
typedef struct memory {
        int used;
```

```
        int size;
        char *memory;
} MEMORY;
```

Now, we write our memory allocator:

```
#define MEM_ALLOC_SIZE 0xffff
int memoryused = 0;
void *more_memory(int siz)
{
    static MEMORY *mem;
    void *result;
    if (mem == NULL || mem->used+siz >= mem->size) {
        mem = malloc(sizeof(mem)+MEM_ALLOC_SIZE);
        if (mem == NULL) {
            fprintf(stderr,"No more memory at line %d\n",line);
            exit(EXIT_FAILURE);
        }
        mem->used = 0;
        memoryused += MEM_ALLOC_SIZE;
        mem->size = MEM_ALLOC_SIZE;
    }
    result = mem->memory+mem->used;
    mem->used += siz;
    memset(result,siz,0);
    memoryused += siz;
    return result;
}
```

We use a static pointer to a MEMORY structure to hold the location of the current memory chunk being used. Since it is static it will be initialized to NULL automatically by the compiler and will keep its value from one call to the next. We test before using it, if the chunk has enough room for the given memory size we want to allocate or if it is NULL, i.e. this is the very first word we are entering. If either of those if true, we allocate a new chunk and initialize its fields.[1]

Otherwise we have some room in our current chunk. We increase our counters and return a pointer to the position within the "memory" field where this chunk starts. We clean the memory with zeroes before returning it to the calling function. Note that we do not keep any trace of the memory we have allocated so it will be impossible to free it after we use it. This is not so bad because the operating system will free the memory after this program exists. The downside of this implementation is that we can't use this program within another one that would call our word counting routine. We have a memory leak "built-in" into our software. A way out of this is very easy though. We could just convert our mem structures into a linked list, and free the memory at the end of the program.

[1] Note that we allocate `MEM_ALLOC_SIZE` bytes. If we want to change to more or less bytes, we just change the #define line and we are done with the change.

3.8.3 Displaying the results

After all this work, we have a program that will compile when run, but is missing the essential part: showing the results to the user. Let's fix this. We need to sort the words by frequency, and display the results. We build a table of pointers to word-list structures and sort it.

But... to know how big our table should be, we need to know how many words we have entered. This can be done in two ways: Either count the number of words in the table when building the report, or count the words as we enter them.

Obviously, the second solution is simpler and requires much less effort. We just declare a global integer variable that will hold the number of words entered into the table so far:[2]

```
int words = 0;
```

We increment this counter when we enter a new word, i.e. in the function NewWordList. We will need a comparison function for the qsort library function too.

```
int comparewords(const void *w1,const void *w2)
{
        WORDLIST *pw1 = *(WORDLIST **)w1,*pw2 = *(WORDLIST **)w2;

        if (pw1->Count == pw2->Count)
                return strcmp(pw1->Word,pw2->Word);
        return pw1->Count - pw2->Count;
}
```

Note that we have implemented secondary sort key. If the counts are the same, we sort by alphabetical order within a same count.

```
void DoReports(char *filename)
{
        int i;
        int idx = 0; // Index into the resulting table

        // Print file name and number of words
        printf("%s: %d different words.\n",filename,words);

        // allocate the word-list pointer table
        WORDLIST **tab = more_memory(words*sizeof(WORDLIST *));

        // Go through the entire hash table
```

[2] Global variables like this should be used with care. Overuse of global variables leads to problems when the application grows, for instance in multi-threaded applications. When you got a lot of global variables accessed from many points of the program it becomes impossible to use threads because the danger that two threads access the same global variable at a time.

Another problem is that our global is not static, but visible through the whole program. If somewhere else somebody writes a function called "words" we are doomed. In this case and for this example the global variable solution is easier, but not as a general solution.

```
        for (i=0; i< sizeof(WordTable)/sizeof(WordTable[0]);i++) {
                WORDLIST *wl = WordTable[i];

                while (wl) {
                        // look at the list at this slot
                        tab[idx] = wl;
                        wl = wl->Next;
                        idx++;
                        if (idx >= words && wl) {
                                fprintf(stderr,"program error\n");
                                exit(EXIT_FAILURE);
                        }
                }
        }
        // Sort the table
OUTPUT:
        qsort(tab,words,sizeof(WORDLIST *),comparewords);
        // Print the results
        for (i=0; i< words;i++) {
                printf("%s %5d\n",tab[i]->Word,tab[i]->Count);
        }
}
```

We start by printing the name of the file and the number of different words found. Then, we go through our hash table, adding a pointer to the word list structure at each non-empty slot.

Note that we test for overflow of the allocated table. Since we increment the counter each time that we add a word, it would be very surprising that the count didn't match with the number of items in the table. But it is better to verify this. After filling our table for the qsort call, we call it, and then we just print the results.

3.8.4 Code review

Now that we have a bare skeleton of our program up and running, let's come back to it with a critical eye. For instance look at our "isWordStart" function. We have:

```
int isWordStart(int c)
{
        if (c == '_')
                return 1;
        if (c >= 'a' && c <= 'z')
                return 1;
        if (c >= 'A' && c <= 'Z')
                return 1;
        return 0;
}
```

A look in the "ctype.h" system header file tells us that for classifying characters we have a lot of efficient functions. We can reduce all this code to:

```
int isWordStart(int c)
{
        return c == '_' || isalpha(c);
}
```

The "isalpha" function will return 1 if the character is one of the uppercase or lowercase alphabetic characters. Always use library functions instead of writing your own. The "isalpha" function does not make any jumps like we do, but indexes a table of property bits. Much faster.

And what about error checking? Remember, we just open the file given in the command line without any test of validly. We have to fix this.

Another useful feature would be to be able to report a line number associated with our file, instead of just an error message that leaves to the user the huge task of finding where is the offending part of the input file that makes our program crash. This is not very complex. We just count the new line characters.

The output of our program is far from perfect. It would be better if we justify the columns. To do that, we have to just count the length of each word and keep a counter to the longest word we find. Another nice thing to have would be a count of how many words with 1 character we find, how many with two, etc.

3.9 Hexdump

Your task is to enhance and maintain an hexadecimal dump utility. This utility displays the values of 16 characters in a line, and in a second column, their ASCII equivalents. If there is no ASCII equivalent it will display a point instead. After 16 lines it leaves an empty line for better clarity.

The code is not very well written, as you will see. There are "magic numbers" (like this famous 16), and other, minor, problems as you will see when you try to solve the exercises. Happily, the guy who wrote the program left a good documentation that begins in the "Analysis" section.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <ctype.h>
 5 int main(int argc,char *argv[])
 6 {
 7    if (argc < 2) {
 8       fprintf(stderr,"Usage: %s <file name>\n",argv[0]);
 9       return EXIT_FAILURE;
10    }
11    FILE *file = fopen(argv[1],"rb");
12    if (file == NULL) {
13       fprintf(stderr,"Impossible to open %s for reading\n",argv[1]);
```

```
14       return EXIT_FAILURE;
15    }
16    int oneChar = fgetc(file);
17    int column = 0,line = 0;
18    char tab[16+1];
19    const char *hex = "0123456789abcdef";
20    int address = 1;
21
22    while (oneChar != EOF) {
23       if (column == 0) {
24          memset(tab,'.',16);
25          fprintf(stdout,"[%d] ",address);
26       }
27       if (isprint(oneChar)) {
28          tab[column] = oneChar;
29       }
30       fputc(hex[(oneChar >> 4)&4],stdout);
31       fputc(hex[oneChar&4],stdout);
32       fputc(' ',stdout);
33       column++;
34       if (column == 16) {
35          fputc(' ',stdout);
36          tab[column]=0;
37          fputs(tab,stdout);
38          column = 0;
39          line++;
40          if (line == 16) {
41             fputc('\n',stdout);
42             line=0;
43          }
44          fputc('\n',stdout);
45       }
46       oneChar = fgetc(file);
47       address++;
48    }
49    fclose(file);
50    address--;
51    if (column > 0 ) {
52       while (column < 16) {
53          fprintf(stdout,"   ");
54          tab[column]=' ';
55          column++;
56       }
57       tab[16]=0;
58       fprintf(stdout," %s\n[%d]\n",tab,address);
59    }
60    else fprintf(stdout,"[%d]\n",address);
```

```
61    return EXIT_SUCCESS;
62 }
```

3.9.1 Analysis

The program should be called (in its present form) like this:

```
hexdump <file name>
```

i.e. it needs at least one argument: the name of the file to dump. We test for this in line 7 and if the name is missing we issue a warning and exit with a failure value. When printing the warning we use the value stored in argv[0] as the name of the program. This is generally the case, most systems will store the name of the program in argv[0]. It could be however, that a malicious user calls this program constructing its command line for an "execv" call and leaves argv[0] empty. In that case our program would crash.

Is that possibility a real one?

Should we guard against it?

It is highly unlikely that a user that has already enough access to the machine to write (and compile) programs would bother to crash our minuscule hexdump utility.. But anyway the guard would need a tiny change only. Line 8 would need to be changed to:

```
8        if (argv[0]) fprintf(stderr,"...message",argv[0]);
```

Now we know that we have at least an argument. In line 11 we try to open the file that we should dump. Note that we use the binary form of the fopen call "rb" (read binary) to dump exactly each byte in the file.

If we can't open the file (fopen returns NULL) we print a warning message into the error output file and return a failure value.

Now we know we have an open file to dump (line 16) so we start initializing stuff for the main part of the program. We read the first character into a variable that will hold each character in the loop (line 16). We will count columns and lines, so we initialize the counters to zero (line 17). We need a table of characters to hold the ASCII equivalences of each byte (line 18). That table should be a string, so we dimension it to one character more than the required length.

We need a table of hexadecimal letters (line 19) that shouldn't be changed, it is a constant "variable". We tell the compiler this fact. And then we need to know at what position we are in the file, so we declare an "address" counter. It is initialized to one since we have already read one character in line 16.

Now we arrive at our loop. We will read and display characters until the last character of the file, i.e. until we hit the end of file condition (line 22).

If we are at the start of a line, i.e. when our "column" variable is at the start of a line we set the table of ASCII equivalences to '.' and we put out the position of the file where we are. We use a one based index for our position so that the first character is 1. But maybe that is not what the user of an hexdump utility expects, we can change that to a different address field display, see the exercises at the end.

We should put the contents of our character into the table if it is printable. We use the 'isprint' function (line 27) to determine that, and if true we store the value of our character into the table.

Then, we output the value of our character. We print in hexadecimal first the higher 4 bits, then the lower 4 bits. Since 4 bit numbers can only go from zero to 15, we index directly our "hex" table with the value of those 4 bits lines 30 and 31.

Note that we mask the bits with the value 0xf, i.e. 15. This means that we ensure that only the lower 4 bits are used. This is important to avoid making an index error when we access our "hex" table. We assume that characters are 8 bits. See exercise 6.

We separate each character with a space (line 32) update our column counter and test if we have arrived at the end of our dump line.

If that is the case we put an extra blank, finish the table with zero and print it. We bump our "line" counter, and if we have arrived at a block of 16 lines, we put an extra empty line (line 41).

We separate lines w(line 44) and read the next character (line 46).

When we arrive at the end of file the loop stops, and execution continues with line 49, where we close the file we have opened. This is not strictly necessary in this case since when a program exits all the files it has opened are close automatically in most systems, but it is better to do it since if we later want to use our dump routine as a part of a bigger software package we would leave a file open.

We adjust the address since we have counted the EOF as a character in line 50.

We are at the end of the file we output the last line, if any. If the file size is not a multiple of 16, we have already put some characters: we complete the last line with blanks. If the file size is exactly a multiple of 16 we just output our address variable to indicate to the user the exact size of the file.

3.9.2 Exercises

1. Add an optional argument so that an output file can be specified.

2. You should have noticed that between the 9th and the 10th line the output is not aligned since 10 has one more character than 9. Fix this. All lines should be aligned.

3. Add an option (call it -hexaddress) to write file addresses in hexadecimal instead of decimal as shown.

4. Add another option (call it -column:XXX) todisplay more or less text positions in a line. For instance -column:80 would fix the display to 80 columns. Adjust the number of characters displayed accordingly. Note that you should not make the number of characters less than 4 or greater than 512.

5. Add an option to display 32 bits instead of just 8 at a time.

6. What would happen if you are working in a machine where the characters are 16 bits wide? What needs to be changed in the above program?

3.10 Text processing

Text files are a widely used format for storing data. They are usually quite compact (no text processing formats like bold, italics, or other font related instructions) and they are widely portable if written in the ASCII subset of text data.

A widely used application of text files are program files. Most programming languages (and here C is not an exception) store the program in text format.

So let's see a simple application of a text manipulating program. The task at hand is to prepare a C program text to be translated into several languages. Obviously, the character string:

```
"Please enter the file name"
```

will not be readily comprehensible to a spanish user. It would be better if the program would show in Spain the character string:

```
"Entre el nombre del fichero por favor"
```

To prepare this translation, we need to extract all character strings from the program text and store them in some table. Instead of referencing directly a character string, the program will reference a certain offset from our table. In the above example the character string would be replaced by

```
StringTable[6]
```

To do this transformation we will write into the first line of our program:

```
static char *StringTable[];
```

Then, in each line where a character string appears we will replace it with an index into the string table.

```
printf("Please enter the file name");
```

will become

```
printf(StringTable[x]);
```

where "x" will be the index for that string in our table.

At the end of the file we will append the definition of our string table with:

```
static char *StringTable[] = {
...,
...,
"Please enter the file name",
...,

NULL
};
```

After some hours of work, we come with the following solution. We test a bit, and it seems to work.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <strings.h>
 4
 5 // Reads a single character constant returning the character right after
 6 static int ReadCharConstant(FILE *infile)
 7 {
 8         int c;
 9         c = fgetc(infile);
10         putchar('\'');
11         while (c != EOF && c != '\'') {
12                 putchar(c);
13                 if (c == '\\') {
14                         c = fgetc(infile);
15                         if (c == EOF)
16                                 return EOF;
17                         putchar(c);
18                 }
19                 c = fgetc(infile);
20         }
21         if (c != EOF){
22                 putchar(c);
23                 c = fgetc(infile);
24         }
25         return c;
26 }
27
28 static int ReadLongComment(FILE *infile)
29 {
30         int c;
31         putchar('/');
32         putchar('*');
33         c = fgetc(infile);
34
35         do {
36
37                 while (c != '*' && c != EOF) {
38                         putchar(c);
39                         c = fgetc(infile);
40                 }
41                 if (c == '*') {
42                         putchar(c);
43                         c = fgetc(infile);
44                 }
45         } while (c != '/' && c != EOF); /* Problem 2 */
46         if (c == '/')
47                 putchar(c);
```

```
48          return c;
49  }
50
51  static int ReadLineComment(FILE *infile)
52  {
53          int c = fgetc(infile);
54
55          putchar('/'); putchar('/');
56          while (c != EOF && c != '\n') {
57                  putchar(c);
58                  c = fgetc(infile);
59          }
60          return c;
61  }
62  static char *stringBuffer;
63  static char *stringBufferPointer;
64  static char *stringBufferEnd;
65  static size_t stringBufferSize;
66  static unsigned stringCount;
67
68  #define BUFFER_SIZE 1024
69
70  static void OutputStrings(void)
71  {
72          char *p = stringBuffer,*strPtr;
73          printf("\nstatic char *StringTable[]={\n");
74          while (p < stringBufferPointer) {
75                  printf("\t\"%s\",\n",p);
76                  p += strlen(p)+1;
77          }
78          printf("\tNULL\n};\n");
79          free(stringBuffer);
80          stringBuffer = NULL;
81  }
82  static void PutCharInBuffer(int c)
83  {
84          if (stringBufferPointer == stringBufferEnd) {
85                  size_t newSize = stringBufferSize + BUFFER_SIZE;
86                  char *tmp = realloc(stringBuffer,newSize);
87                  if (tmp == NULL) {
88                          fprintf(stderr,"Memory exhausted\n");
89                          exit(EXIT_FAILURE);
90                  }
91                  stringBuffer = tmp;
92                  stringBufferPointer = tmp+stringBufferSize;
93                  stringBufferSize += BUFFER_SIZE;
94                  stringBufferEnd = tmp + stringBufferSize;
```

```
 95         }
 96         *stringBufferPointer++ = c;
 97 }
 98
 99 static int ReadString(FILE *infile,int ignore)
100 {
101         int c;
102         if (stringBuffer == NULL) {
103                 stringBuffer = malloc(BUFFER_SIZE);
104                 if (stringBuffer == NULL)
105                         return EOF;
106                 stringBufferPointer = stringBuffer;
107                 stringBufferEnd = stringBufferPointer+BUFFER_SIZE;
108                 stringBufferSize = BUFFER_SIZE;
109         }
110         c = fgetc(infile);
111         while (c != EOF && c != '"') {
112                 if (ignore == 0)
113                         PutCharInBuffer(c);
114                 else
115                         putchar(c);
116                 if (c == '\\') {
117                         c = fgetc(infile);
118                         if (c != '\n') {
119                                 if (ignore == 0)
120                                         PutCharInBuffer(c);
121                                 else
122                                         putchar(c);
123                         }
124                 }
125                 c = fgetc(infile);
126         }
127         if (c == EOF)
128                 return EOF;
129         if (ignore == 0) {
130                 PutCharInBuffer(0);
131                 printf("StringTable[%d]",stringCount);
132         }
133         else putchar(c);
134         stringCount++;
135         return fgetc(infile);
136 }
137
138 static int ProcessChar(int c,FILE *infile)
139 {
140         switch (c) {
141                 case '\'':
```

```
142                         c = ReadCharConstant(infile);
143                         break;
144                 case '"':
145                         c = ReadString(infile,0);
146                         break;
147                 case '/':
148                         c = fgetc(infile);
149                         if (c == '*')
150                                 c = ReadLongComment(infile);
151                         else if (c == '/')
152                                 c = ReadLineComment(infile);
153                         else {
154                                 putchar(c);
155                                 c = fgetc(infile);
156                         }
157                         break;
158                 case '#':
159                         putchar(c);
160                         while (c != EOF && c != '\n') {
161                                 c = fgetc(infile);
162                                 putchar(c);
163                         }
164                         if (c == '\n')
165                                 c=fgetc(infile);
166                         break;
167                 case 'L':
168                         putchar(c);
169                         c = fgetc(infile);
170                         if (c == '"') {
171                                 putchar(c);
172                                 c = ReadString(infile,1);
173                         }
174                         break;
175                 default:
176                         putchar(c);
177                         c = fgetc(infile);
178                         break;
179         }
180         return c;
181 }
182 int main(int argc,char *argv[])
183 {
184         FILE *infile;
185
186         if (argc < 2) {
187                 fprintf(stderr,"Usage: strings <file name>\n");
188                 return EXIT_FAILURE;
```

```
189         }
190         if (!strcmp(argv[1],"-")) {
191                 infile = stdin;
192         } else {
193                 infile = fopen(argv[1],"r");
194                 if (infile == NULL) {
195                         fprintf(stderr,"Can't open %s\n",argv[1]);
196                         return EXIT_FAILURE;
197                 }
198         }
199         int c = fgetc(infile);
200         printf("static char *StringTable[];\n");
201         while (c != EOF) {
202                 c = ProcessChar(c,infile);
203         }
204         OutputStrings();
205         if (infile != stdin) fclose(infile);
206 }
```

The general structure of this program is simple. We

- Open the given file to process
- We process each character
- We are interested only in the following tokens:
  1. Char constants
  2. Comments
  3. Preprocessor directives
  4. Character strings

Why those?

1. Char constants could contain double quotes, what would lead the other parts of our programs to see strings where there aren't any. For instance:

   ```
   case'"':
   ```

   would be misunderstood as the start of a never ending string.

2. Comment processing is necessary since we should not process strings in comments. In comments we could find in constructs like " ... aren't true... ", what would trigger our character constant parsing for a never ending constant.

3. Preprocessor directives should be ignored since we do NOT want to translate

   ```
   #include "myfile.h"
   ```

Our string parsing routine stores the contents of each string in a buffer that is grown if needed, printing into standard output only the

```
StringTable[x]
```

instead of the stored string. Each string is finished with a zero, and after the last string we store additional zeroes to mark the end of the buffer.

After the whole file is processed we write the contents of the buffer in the output (written to stdout) and that was it. We have extracted the strings into a table.

3.10.1 Detailed view

main

After the command line argument processing we read a character and pass it to the `ProcessChar` routine that returns the next character. When we hit EOF we output the string table, and close the input file if it wasn't stdin.

ProcessChar

This routine switches on the given character. For each of the starting characters of the tokens we are interested in we test (if applicable) if it is the token we are looking for and act accordingly. Since we do NOT process wide character strings we just ignore them. We detect them by assuming that an uppercase 'L' followed by a double quote means a large character string. For comments we parse the two kinds of comments in C.

ReadLongComment and ReadLineComment

These routines read and echo to stdout the contents of comments found in the text file.

ReadCharConstant

This routine reads a character constant ignoring escaped characters. Since we do not do any processing of the contents we just ignore the character after a backslash, meaning that `'\0xEf'` will be parsed as `'Ef'`, what is completely wrong but doesn't actually matter.

OutputStrings

Prints the string gathered in the table `stringBuffer`. Assumes that each string is zero terminated, skipping the zero each time.

3.10.2 Analysis

Our program seems to work, but there are several corner cases that it doesn't handle at all.

For instance it is legal in C to write:

```
"String1" "String2"
```

and this will be understood as

```
"String1String2"
```

by the compiler. Our translation makes this into:

```
StringTable[0] StringTable[1]
```

what is a syntax error.

Another weak point is that a string can be present several times in our table since we do not check if the string is present before storing it in our table.

And there are many corner cases that are just ignored. For instance you can continue a single line comment with a backslash, a very bad idea of course but a legal one. We do not follow comments like these:

```
// This is a comment \
   and this line is a comment too
```

And (due to low level of testing) there could be a lot of hidden bugs in it.

But this should be a simple utility to quickly extract the strings from a file without too much manual work. We know we do not use the features it deosn't support, and it will serve our purposes well.

What is important to know is that there is always a point where we stop developing and decided that we will pass to another thing. Either because we get fed up or because our boss tell us that we should do xxx instead of continuing the development of an internal utility.

In this case we stop the first development now. See the exercises for the many ways as to how we could improve this simple program.

3.10.3 Exercises:

1. This filter can read from stdin and write to stdout. Add a command line option to specify the name of an output file. How many changes you would need to do in the code to implement that?

2. The program can store a string several times. What would be needed to avoid that? What data structure would you recommend?

3. Implement the concatenation of strings, i.e.

   ```
   "String1" "String2" --> "String1String2"
   ```

4. Seeing in the code

   ```
   printf(StringTable[21]);
   ```

   is not very easy to follow. Implement the change so that we would have instead in the output:

```
// StringTable[21]--> "Please enter the file name"
printf(StringTable[21]);
```

i.e. each line would be preceeded with one or several comment lines that describe the strings being used.

5. Add an option so that the name of the string table can be changed from "StringTable" to some other name. The reason is that a user complained that the "new" string table destroyed her program: she had a "StringTable" variable in her program!

   How could you do this change automatically?

3.11 Using containers

What I did not tell you in the preceeding chapter is that... there is a fatal error in the construction of the string translating package. The problem was highlited in the comp.lang.c (where I presented the program to discussion) by Ben Becarisse that pointed out that

```
static char *StringTable[];
```

would not compile. He said:

> That is a tentative definition of an object with incomplete type (that's a constraint violation).

The explanation was delivered by Harald van Dijk that explained:

> 6.9.2p3 says the declared type of a tentative definition with internal linkage must not be an incomplete type, but it isn't a constraint, which matters because it means compilers are not required to issue any diagnostics. And I wonder if that is really meant to apply to the declared type, rather than the composite type for the final implicit definition mentioned in p2. Compilers are already required to accept `int array[];` or `int array[20] = {1};` – without the static keyword – and they would surely need to treat the static keyword specially to reject it if present.

I had compiled the sample program with gcc under the Macintosh system, and forgot to verify under windows with lcc. The result was that lcc-win would not compile that construct.

The fix would be obviously to generate the text in a RAM buffer, and then when the program knows how many strings there are, to patch the declaration at the beginning with the number of strings so that the end product would look like:

```
static char *StringTable[23];
```

This would be very difficult to do with the current design of the program since we just write into standard output. Obviously we could modify the program to write into some file, then backtrack in the file and patch it with the right number after reading

the whole file but now the whole structure of the program comes into question. We mostly copy input to output, modifying the few lines where we find a character string...

4 Structures and unions

4.1 Structures

Structures are a contiguous piece of storage that contains several simple types, grouped as a single object. For instance, if we want to handle the two integer positions defined for each pixel in the screen we could define the following structure:

```
struct coordinates {
        int x;
        int y;
};
```

Structures are introduced with the keyword "struct" followed by their name. Then we open a scope with the curly braces, and enumerate the fields that form the structure. Fields are declared as all other declarations are done. Note that a structure declaration is just that, a declaration, and it reserves no actual storage anywhere.

After declaring a structure, we can use this new type to declare variables or other objects of this type:
`struct coordinate Coords = { 23,78};`
Here we have declared a variable called Coords, that is a structure of type coordinate, i.e. having two fields of integer type called "x" and "y". In the same statement we initialize the structure to a concrete point, the point (23,78). The compiler, when processing this declaration, will assign to the first field the first number, i.e. to the field "x" will be assigned the value 23, and to the field "y" will be assigned the number 78.

Note that the data that will initialize the structure is enclosed in curly braces. Structures can be recursive, i.e. they can contain pointers to themselves. This comes handy to define structures like lists for instance:

```
struct list {
        struct list *Next;
        int Data;
};
```

Here we have defined a structure that in its first field contains a pointer to the same structure, and in its second field contains an integer. Please note that we are defining a pointer to an identical structure, not the structure itself, what is impossible. A structure can't contain itself. Double linked list can be defined as follows:

```
struct dl_list {
        struct dl_list *Next;
        struct dl_list *Previous;
        int Data;
};
```

This list features two pointers: one forward, to the following element in the list, and one backward, to the previous element of the list.

A special declaration that can only be used in structures is the bit-field declaration. You can specify in a structure a field with a certain number of bits. That number is given as follows:

```
struct flags {
        unsigned HasBeenProcessed:1;
        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 of the start of the character string with the customer name,
- another address of the start of the name of the place where this customer lives,
- a double precision number containing the current balance,
- a `time_t` (time type) date of last transaction,
- and other bit fields for storing some flags.

```
struct mailMessage {
        MessageID ID;
        time_t date;
        char *Sender;
        char *Subject;
        char *Text;
        char *Attachements;
};
```

This one starts with another type containing the message ID, again a `time_t` to store the date, then the addresses of some character strings. The set of functions that use a certain type are the methods that you use for that type, maybe in combination with other types. There is no implicit "this" in C. Each argument to a function is explicit, and there is no predominance of anyone.

A customer can send a mailMessage to the company, and certain functions are possible, that handle mailMessages from customers. Other mailMessages aren't from customers, and are handled differently, depending on the concrete application.

Because that's the point here: an application is a coherent set of types that performs a certain task with the computer, for instance, sending automated mailings, or invoices, or sensing the temperature of the system and acting accordingly in a multi-processing robot, or whatever. It is up to you actually.

Note that in C there is no provision or compiler support for associating methods in the structure definitions. You can, of course, make structures like this:

```
struct customer {
        int ID;
        char *Name;
        char *Address;
        double balance;
        time_t lastTransaction;
        unsigned hasACar:1;
        unsigned mailedAlready:1;
        bool (*UpdateBalance)(struct customer *Customer,
                              double newBalance);
};
```

The new field, is a function pointer that contains the address of a function that returns a Boolean result, and takes a customer and a new balance, and should (eventually) update the balance field, that isn't directly accessed by the software, other than trough this procedure pointer.

When the program starts, you assign to each structure in the creation procedure for it, the function DefaultGetBalance() that takes the right arguments and does hopefully the right thing.

This allows you the flexibility of assigning different functions to a customer for calculating his/her balance according to data that is known only at runtime. Customers with a long history of overdraws could be handled differently by the software after all. But this is no longer C, is the heart of the application.

True, there are other languages that let you specify with greater richness of rules what and how can be sub classed and inherited. C, allows you to do anything, there are no other rules here, other the ones you wish to enforce.

You can subclass a structure like this. You can store the current pointer to the procedure somewhere, and put your own procedure instead. When your procedure is called, it can either:

Do some processing before calling the original procedure Do some processing after the original procedure returns Do not call the original procedure at all and replace it entirely.

We will show a concrete example of this when we speak about windows sub classing later. Sub classing allows you to implement dynamic inheritance. This is just an example of the many ways you can program in C.

But is that flexibility really needed? Won't just

```
bool UpdateBalance(struct customer *pCustomer, double newBalance);
```

do it too?

Well it depends. Actions of the general procedure could be easy if the algorithm is simple and not too many special cases are in there. But if not, the former method, even if more complicated at first sight, is essentially simpler because it allows you greater flexibility in small manageable chunks, instead of a monolithic procedure of several hundred lines full of special case code...

Mixed strategies are possible. You leave for most customers the UpdateBalance field empty (filled with a NULL pointer), and the global UpdateBalance procedure will use that field to calculate its results only if there is a procedure there to call. True, this wastes 4 bytes per customer in most cases, since the field is mostly empty, but this is a small price to pay, the structure is probably much bigger anyway.

4.1.1 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 the actual size 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.

Normally, you should never make any assumptions about the specific size of a structure. Compilers, and lcc-win 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 to do is to always use the sizeof operator when the structure size needs to be used somewhere in the code. For instance, if you want to allocate a new piece of memory managed by the memory manager, you call it with the size of the structure.

```
GC_malloc(sizeof(struct DataPoint)*67);
```

This will allocate space for 67 structures of type "DataPoint" (as defined above). Note that we could have written

```
GC_malloc(804);
```

since we have:

```
struct DataPoint {
        struct coordinate coords;
        int Data;
};
```

We can add the sizes:

Two integers of 4 bytes for the coordinate member, makes 8 bytes, plus 4 bytes for the Data member, makes 12, that multiplies 67 to make 804 bytes. But this is very risky because of the alignment mentioned above. Besides, 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 with sizeof() instead of using hard-wired numbers. When designing a data type layout it is important to have padding issues in mind. If we define a structure like this:

```
struct s {
        char a;
        int b;
        char c;
};
```

it will use 12 bytes in most 32 bit compilers. If we rearrange the members like this:

```
struct s {
        char a;
        char c;
        int b;
};
```

it will use only 8 bytes of memory, a reduction of 33% in size!

4.1.2 Using the pragma pack feature

The pragma instruction "pack" allows you to specify how a structure will be layed out (packed) by the compiler. The syntax in lcc-win is:
`#pragma pack(<n>)`
where <n> can be one of 1, 2, 4, 8, and 16.

This compiler directive will be followed from the point where it is seen until either a new "pack" directive is seen, or the end of the compilation unit is reached.

Optionally, you can use this syntax: `#pragma pack(push,<n>)`
This means that the current value of the packing directive is stored, a new value is set, and a directive like: `#pragma pack(pop)` will restore the previous value. This syntax is compatible with the syntax used by the Microsoft compiler

Note that all this is accepted when you are defining the structure type, NOT when you are using it to define a new variable or structure member. This directives should be written in a header file, together with the definition of the structure. It is very important that the compiler sees always the same definitions across all the modules present in the program. If one module uses a packing method different than the packing method in another module, the structures will not be compatible even if they have the same definitions.

For example:

```
struct f {
        char a;
        int b;
};
struct f VAR = {1,2};
```

If we see how the structure is layed out in memory we see following assembler instructions:

```
_VAR:
        .byte   1
        .space  3
        .long   2
```

The compiler stores the byte of the character, then it leaves 3 bytes empty, and then stores the integer value. If we add a pack(1) directive we see following layout:

```
_VAR:
        .byte   1
        .long   2
```

4.1.3 Structure packing in other environments

The need to direct the compiler to use different packing schemas is apparently universal, and many compilers have developed similar packing directives.

Gcc

The gcc compiler uses an `__attribute__ ((packed))` directive to direct the compiler to pack as much data as possible in the smallest space. For different alignment than the smallest, the user can use the form with an argument:

`__attribute__ ((aligned (16))).`

Hewlett Packard

The HP compilers use

```
#pragma HP_ALIGN align_mode [ PUSH ]
#pragma HP_ALIGN [ POP ]
```

where `align_mode` can be one of "word" "natural" and many other alignment modes that work in the several machines that those compilers support.

IBM

IBM compilers also use a `#pragma pack(n)`, and an `__align` attribute for variables.

Comeau computing C

Supports also the pragma pack construct with push/pop.

Microsoft

Microsoft uses the command line option /Zp*n* (case sensitive) to set the packing to $n = 1, 2, 4, 8$, or 16. Default is 8.

4.1.4 Bit fields

A "bit field" is an unsigned or signed integer composed of some number of bits. Lcc-win will accept some other type than int for a bit field, but the real type of a bit field will be always either "int" or "unsigned int". For example, in the following structure, we have 3 bit fields, with 1, 5, and 7 bits:

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

With lcc-win the size of this structure will be 4 with no special options. With maximum packing (-Zp1 option) the size will be two. When you need to leave some space between adjacent bit fields you can use the notation: `unsigned : n;`

For example:

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

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

4.2 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-win, 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 similar to structures.

Using the definition above we can write:

```
int main(void)
{
        union intfloat ifl;
        union intfloat *pIntfl = &ifl;

        ifl.i = 2;
        pIntfl->d = 2.87;
}
```

First we assign to the integer part of the union an integer, then we assign to the double precision part a double. Unions are useful for storing structures that can have several different memory layouts. In general we have an integer that tells us which kind of data follows, then a union of several types of data. Suppose the following data structures:

```
struct fileSource {
        char *FileName;
        int LastUse;
};

struct networkSource {
        int socket;
        char *ServerName;
        int LastUse;
};

struct windowSource {
        WINDOW window;
        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-win 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.

4.3 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(EXIT_FAILURE);
```

```
    }
    infile = fopen(argv[1],"rb");
    if (infile == NULL) {
        printf("File %s doesn't exist\n",argv[1]);
        exit(EXIT_FAILURE);
    }
    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 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.

4.4 Basic data structures

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.

4.4.1 Lists

Lists are members of a more general type of objects called sequences, i.e. objects that have a natural order. You can go from a given list member to the next element, or to the previous one.

We have several types of lists, the simplest being the single-linked list, where each member contains a pointer to the next element, or NULL, if there isn't any. We can implement this structure in C like this:

```
typedef struct _list {
        struct _list *Next; // Pointer to next element
        void *Data;             // Pointer to the data element
} LIST;
```

We can use a fixed anchor as the head of the list, for instance a global variable containing a pointer to the list start.

```
LIST *Root;
```

We define the following function to add an element to the list:

```
LIST *Append(LIST **pListRoot, void *data)
{
        LIST *rvp = *pListRoot;

        if (rvp == NULL) { // is the list empty?
                // Yes. Allocate memory
                *pListRoot = rvp = GC_malloc(sizeof(LIST));
        }
        else { // find the last element
                while (rvp->Next)
                        rvp = rvp->Next;
                // Add an element at the end of the list
                rvp->Next = GC_malloc(sizeof(LIST));
                rvp = rvp->Next;
        }
        // initialize the new element
        rvp->Next = NULL;
        rvp->Data = data;
        return rvp;
}
```

This function receives a pointer to a pointer to the start of the list.

Why? If the list is empty, it needs to modify the pointer to the start of the list. We would normally call this function with:

```
newElement = Append(&Root,data);
```

Note that loop:

```
while (rvp->Next)
        rvp = rvp->Next;
```

This means that as long as the Next pointer is not NULL, we position our roving pointer (hence the name "rvp") to the next element and repeat the test. We suppose obviously that the last element of the list contains a NULL "Next" pointer. We ensure that this condition is met by initializing the `rvp->Next` field to NULL when we initialize the new element.

To access a list of n elements, we need in average to access n/2 elements.

Other functions are surely necessary. Let's see how a function that returns the nth member of a list would look like:

```
LIST *ListNth(LIST *list, int n)
{
        while (list && n-- > 0)
                list = list->Next;
        return list;
}
```

Note that this function finds the nth element beginning with the given element, which may or may not be equal to the root of the list. If there isn't any nth element, this function returns NULL.

If this function is given a negative n, it will return the same element that was passed to it. Given a NULL list pointer it will return NULL.

Other functions are necessary. Let's look at Insert.

```
LIST *Insert(LIST *list,LIST *element)
{
    LIST *tmp;

    if (list == NULL)
        return NULL;
    if (list == element)
        return list;
    tmp = list->Next;
    list->Next = element;
    if (element) {
        element->Next = tmp;
    }
    return list;
}
```

We test for different error conditions. The first and most obvious is that "list" is NULL. We just return NULL. If we are asked to insert the same element to itself, i.e. "list" and "element" are the same object, their addresses are identical, we refuse. This is an error in most cases, but maybe you would need a circular element list of one element. In that case just eliminate this test.

Note that `Insert(list ,NULL);` will effectively cut the list at the given element, since all elements after the given one would be inaccessible.

Many other functions are possible and surely necessary. They are not very difficult to write, the data structure is quite simple.

Double linked lists have two pointers, hence their name: a Next pointer, and a Previous pointer, that points to the preceding list element. Our data structure would look like this:

```
typedef struct _Dlist {
    struct _dlList *Next;
    struct _dlList *Previous;
    void *data;
} Dlist;
```

Our "Append" function above would look like:

```
LIST *AppendDl(DLLIST **pListRoot, void *data)
{
    DLLIST *rvp = *pListRoot;

    // is the list empty?
    if (rvp == NULL) {
        // Yes. Allocate memory
        *pListRoot = rvp = GC_malloc(sizeof(DLLIST));
        rvp->Previous = NULL;
    }
    else {
        // find the last element
        while (rvp->Next)
            rvp = rvp->Next;
            // Add an element at the end of the list
            rvp->Next = GC_malloc(sizeof(DLLIST));
            rvp->Next->Previous = rvp;
            rvp = rvp->Next;
    }
    // initialize the new element
    rvp->Next = NULL;
    rvp->Data = data;
    return rvp;
}
```

The Insert function would need some changes too:

```
LIST *Insert(LIST *list,LIST *element)
{
    LIST *tmp;

    if (list == NULL)
        return NULL;
    if (list == element)
        return list;
    tmp = list->Next;
    list->Next = element;
    if (element) {
        element->Next = tmp;
        element->Previous = list;
        if (tmp)
            tmp->Previous = element;
    }
    return list;
}
```

Note that we can implement a Previous function with single linked lists too. Given a pointer to the start of the list and an element of it, we can write a Previous function like this:

```
LIST *Previous(LIST *root, LIST *element)
{
    if (root == NULL )
        return NULL;
    while (root && root->Next != element)
        root = root->Next;
    return root;
}
```

Circular lists are useful too. We keep a pointer to a special member of the list to avoid infinite loops. In general we stop when we arrive at the head of the list. Wedit uses this data structure to implement a circular double linked list of text lines. In an editor, reaching the previous line by starting at the first line and searching and searching would be too slow. Wedit needs a double linked list, and a circular list makes an operation like wrapping around easier when searching.

4.4.2 Hash tables

A hash table is a table of lists. Each element in a hash table is the head of a list of element that happen to have the same hash code, or key.

To add an element into a hash table we construct from the data stored in the element a number that is specific to the data. For instance we can construct a number from character strings by just adding the characters in the string.

This number is truncated module the number of elements in the table, and used to index the hash table. We find at that slot the head of a list of strings (or other data) that maps to the same key modulus the size of the table.

To make things more specific, let's say we want a hash table of 128 elements, which will store list of strings that have the same key. Suppose then, we have the string "abc". We add the ASCII value of 'a' + 'b' + 'c' and we obtain 97+98+99 = 294. Since we have only 128 positions in our table, we divide by 128, giving 2 and a rest of 38. We use the rest, and use the 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.

4.4.3 The container library of lcc-win

Lcc-win provides in the standard distribution a container library with source code. The containers implemented are varied, and the interface is innovative. You can learn about how to implement containers, but also about error handling, library design, and many things from the code. All documentation and source code can be found at:

`http://code.google.com/p/ccl`

4.5 Fine points of structure use

- When you have a pointer to a structure and you want to access a member of it you should use the syntax: `pointer->field`
- When you have a structure OBJECT, not a pointer, you should use the syntax: `object.field`
  Beginners easily confuse this.
- 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`.
- If you have an array of structures you use: `array[index].field`
- 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.

5 Simple programs using structures

5.1 Reversing a linked list

This is an evergreen of data structures programming. In most classes you will get an exercise like this:

```
Exercise 7: Given a list "L" reverse it without moving its contents.
```

The solution for this exercise is to go through the list, relinking the "Next" pointers in the inverse order, without touching any data that the list may hold. We will use the code of the C containers library and study how it is done.

The library uses a simple structure `ListElement` whose definition runs like this:

```
typedef struct tagListElement {
    struct tagListElement *Next;
    char Data[];
} ListElement;
```

We have a pointer to the next element of the list, and a chunk of data of unspecified length. This is the basic structure that will be used to hold the list data. Besides the list element we have a header of the list, containing several fields not relevant to the task of our reverse function. We will use only the following fields from that header:

- RaiseError: Pointer to the error function.
- First: The first element of the list.
- count: The number of items in the list.
- Last: A pointer to the last element of the list.
- timestamp: A counter of the modifications done to the list.

Here is the definition of the whole structure:

```
struct tagList {
    ListInterface *VTable;   /* Methods table */
    size_t count;            /* in elements units */
    unsigned Flags;
    unsigned Modifications;  /* Changed at each modification */
    size_t ElementSize;      /* Size (in bytes) of each element */
    ListElement *Last;
    ListElement *First;      /* The list start here */
    CompareFunction Compare; /* Element comparison function */
```

```
    ErrorFunction RaiseError;/* Error function */
    ContainerHeap *Heap;
    ContainerMemoryManager *Allocator;
    DestructorFunction DestructorFn;
};
```

The interface of the reverse function is simple: it receives a list to reverse as input and returns an integer result code. A negative result means failure (with different negative numbers as error codes) and a positive number means success.

```
 1 static int Reverse(List *l)
 2 {
 3     ListElement *Previous,*Current,*Next;
 4
```

The first thing to do in a serious program is to check the validity of the received arguments, i.e. test the `preconditions` as Bertrand Meyer would say it. We test that the list handed to us is not NULL (lines 5-8) and that the list is not read-only (lines 9-12) since we are going to modify it. If anything is wrong we return an error code after calling the error function. Note that the error function is a field either in a global structure called iError (for interface Error) or is a field of the list itself. We use the global interface iError in the case that the list is NULL , or the list specific error function for the read only error. This is a powerful feature of C called function pointers that we will discuss in more detail later on.

```
 5     if (l == NULL) {
 6         iError.RaiseError("iList.Reverse",CONTAINER_ERROR_BADARG);
 7         return CONTAINER_ERROR_BADARG;
 8     }
 9     if (l->Flags & CONTAINER_READONLY) {
10         l->RaiseError("iList.Reverse",CONTAINER_ERROR_READONLY);
11         return CONTAINER_ERROR_READONLY;
12     }
```

Then, we test for special conditions, i.e. for degenerate cases. Obviously, reversing an empty list or a list with only one element is very easy: we have nothing to do. We test for that (line 13) and return success immediately if that is the case.

```
13     if (l->count < 2)
14         return 1;
```

Now, we setup our loop. We start with the first element (that must exist since the list has more than one element, line 15). Since we are going to reverse the list, the first element will be the last and the last will be the first. We setup the last one immediately since we know it in line 16. And before we start there is no previous element so we set it to NULL .

```
15     Next = l->First;
16     l->Last = l->First;
17     Previous = NULL;
```

Now we go through the whole list. We save the "Next" value, and advance it to the next element. Then, we set the value of the current element's pointer to the previous, i.e. our current will point to previous reversing the direction of the list.

```
18      while (Next) {
19          Current = Next;
20          Next = Next->Next;
21          Current->Next = Previous;
22          Previous = Current;
23      }
```

OK, we are done. We have gone through the whole list reversing pointers, and we arrive at the cleanup. We should first establish our `First` pointer, then we should ensure that our last element has the NULL marker, and that our list is marked as modified. We return a positive number meaning success.

```
24      l->First = Previous;
25      l->Last->Next = NULL;
26      l->Modifications++;
27      return 1;
28  }
```

5.1.1 Discussion

I presented the above program in the USENET discussion group comp.lang.c. The discussion was extremely interesting. I will reproduce some of the comments here:

An improvement

Ben Bacarisse wrote:

> `> 25 l->Last->Next = NULL;`
>
> This is, for me, the problem line. I don't think it's needed yet it turns the empty list into a special case.

As you can see, Ben has a bright mind, spotting the problem in my code almost immediately. Obviously you should have seen (as I didn't) that the first pass through the loop sets `l->First->Next` to NULL since `Previous` is NULL at the start, and `Current` is `First`. We can safely remove line 25.

Preconditions

Ben also started a discussion about the preconditions that I mentioned above. What is important to underscore is that there are preconditions that aren't tested in the above code, simply assumed. For instance:

- The list should be terminated with a NULL pointer.
- The list shouldn't contain any circularities.
- All the `Next` pointers should be valid pointers to list elements.

- The number of elements should be actually the value of the `count` field in the list header.

How could we test for those preconditions in our code?

Well, the first one is easy since in the list header we have a pointer to the last element. If we wanted to test that the list is NULL terminated we would write:

```
if (l != NULL && l->count > 0 && l->count->Last->Next != NULL) {
     /* Handle error: list not NULL terminated */
}
```

The second precondition is much more complicated. To test if a list has cycles in it the general algorithm runs like this:

- Establish two pointers to the start of the list and to the second element.
- loop until the second pointer is NULL
  - compare the pointers. If they are equal there is a circularity.
  - advance the first pointer *once*.
  - Advance the second pointer *twice*.
- If the second pointer reaches NULL there are no circularities.

This can be translate into a function that checks for circularities in a list:

```
int CheckForCircularities(List *l)
{
  ListElement *rvp,*rvpFast;
  rvpFast = rvp = l->First;
  while (rvpFast) {
      rvpFast = rvpFast->Next;
      if (rvp == rvpFast) {
           iError.RaiseError("Reverse", ERROR_CIRCULAR_LIST);
           return ERROR_CIRCULAR_LIST;
      }
      if (rvpFast)
          rvpFast = rvpFast->Next;
      rvp = rvp->Next;
  }
  return 0;
}
```

This function returns a negative error code if the given list is circular, or zero if it is not.

Ok, but now all these tests have a cost obviously, specially the test for circularity in a very long list. If we would verify all the implicit preconditions to our `Reverse` function it would become very slow. And we are not done yet. How do we verify that the pointer to the next element is OK?

The windows API offers a function like[1]:

[1] The windows documentation uses a Microsoft specific types for this function. I have translated them into the standard ones

```
bool IsBadReadPtr(const void *lp, size_t ucb);
```

This function can be written in almost standard C, see chapter 7.2

6 A closer look at the pre-processor

The first phase of the compilation process is the "pre-processing" phase. This consists of scanning in the program text all the preprocessor directives, i.e. lines that begin with a # character, and executing the instructions found in there before presenting the program text to the compiler.

We will interest us with just two of those instructions. The first one is the `#define` directive, that instructs the software to replace a macro by its equivalent. We have two types of macros:

Parameterless. For example:

```
#define PI 3.1415
```

Following this instruction, the preprocessor will replace all instances of the identifier PI with the text "3.11415".

Macros with arguments. For instance:

```
#define s2(a,b) ( (a*a + b*b) /2.0)
```

When the preprocessor finds a sequence like: `s2 (x, y)` it will replace it with: `(x*x + y*y)/2.0)`

The problem with that macro is that when the preprocessor finds a statement like:

```
s2(x+6.0,y-4.8);
```

it will produce :

```
(x+6.0*x+6.0 + y+6.0*y+6.0) /2.0 )
```

What will calculate completely another value:

```
(7.0*x + 7.0*y + 12.0)/2.0
```

To avoid this kind of bad surprises, it is better to enclose each argument within parentheses each time it is used:

```
#define s2(a,b) (((a)*(a) + (b)*(b))/2.0)
```

This corrects the above problem but we see immediately that the legibility of the macros suffers... quite complicated to grasp with all those redundant parentheses around.

Another problem arises when you want that the macro resembles exactly a function call and you have to include a block of statements within the body of the macro, for instance to declare a temporary variable.

```
#define s2(x,y) { int temp = x*x+y*y; x=temp+y *(temp+6);}
```

If you call it like this:

```
if (x < y) s2(x,y);
else
x = 0;
```

This will be expanded to:

```
if (x < y) { int temp = x*x+y*y; x=temp+y *(temp+6);} ;
else
x = 0;
```

This will provoke a syntax error.

To avoid this problem, you can use the do... while statement, that consumes the semicolon:

```
#define s2(x,y) do { int temp = x*x+y*y; x=temp+y *(temp+6);} \
       while(0)
```

Note the \ that continues this long line, and the absence of a semicolon at the end of the macro.

An `#undef` statement can undo the definition of a symbol. For instance:

```
#undef PI
```

will remove 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-win 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.

6.1 Preprocessor commands

The preprocessor receives its commands with lines beginning with the special character `#`. This lines can contain:

1. Macros
2. Conditional compilation instructions
3. Pragma instructions
4. The `##` operator
5. Line instructions
6. The `#include` directive

6.1.1 Preprocessor macros

The `#define` command has two forms depending on whether a left parenthesis appears immediately after the name of the macro. The first form, without parenthesis is simply the substitution of text. An example can be:

```
#define MAXBUFFERSIZE 8192
```

This means that whenever the preprocessor find the identifier MAXBUFFERSIZE in the program text, it will replace it with the character string "8192"

The second form of the preprocessor macros is the following:

```
#define add(a,b) ((a)+(b))
```

This means that when the preprocessor finds the sequence:

```
        int n = add(7,b);
```

it will replace this with:

```
int n = ((7)+(b));
```

Note that the left parenthesis MUST be written immediately after the name of the macro without any white space (blanks, tabs) in between. It is often said that white space doesn't change the meaning of a C program but that is not always true, as you can see now. If there is white space between the macro name and the left parenthesis the preprocessor will consider that this is a macro with no arguments whose body starts with a left parentheses!

If you want to delete the macro from the preprocessor table you use the `#undef <macro name>` command. This is useful for avoiding name clashes. Remember that when you define a macro, the name of the macro will take precedence before everything else since the compiler will not even see the identifier that holds the macro. This can be the origin of strange bugs like:

```
int fn(int a)
{
// some code
}
```

If you have the idea of defining a macro like this `#define fn 7987` the definition above will be transformed in

```
int 7987(int a)
{
}
```

not exactly what you would expect. This can be avoided by `#undefining` the macros that you fear could clash with other identifiers in the program.

6.2 Conditional compilation

Very often we need to write some kind of code in a special situation, and some other kind in another situation. For instance we would like to call the function "initUnix()" when we are in a UNIX environment, and do NOT call that function when in other environments. Besides we would like to erase all UNIX related instructions when compiling for a windows platform and all windows related stuff when compiling for Unix.

This is achieved with the preprocessor

```
#ifdef UNIX
lines for the Unix system
#else
lines for other (non-unix) systems.
#endif
```

This means:

If the preprocessor symbol "UNIX" is defined, include the first set of lines, else include the second set of lines. There are more sophisticated usages with the `#elif` directive:

```
#ifdef UNIX
                        Unix stuff
#elif MACINTOSH
                        Mac stuff
#elif WIN32
                        Windows stuff
#else
#error ‘‘Unknown system!’’
#endif
```

Note the `#error` directive. This directive just prints an error message and compilation fails.

The lines that are in the inactive parts of the code will be completely ignored, except (of course) preprocessor directives that tell the system when to stop. Note too that the flow of the program becomes much difficult to follow. This feature of the preprocessor can be abused, to the point that is very difficult to see which code is being actually compiled and which is not. The IDE of lcc-win provides an option for preprocessing a file and showing all inactive lines in grey color. Go to "Utils" the choose "Show `#ifdefs`" in the main menu.

Note: You can easily comment out a series of lines of program text when you enclose them in pairs of

```
#if 0
// lines to be commented out
#endif
```

6.3 The pragma directive

This directive is compiler specific, and means that what follows the `#pragma` is dependent on which compiler is running. The pragmas that lcc-wi32 uses are defined in the documentation. In general pragmas are concerned with implementation specific details, and are an advanced topic.

6.4 Token concatenation

This operator allows you to concatenate two tokens:

```
#define join(a,b) (a##b)
a = join(anne,bob)
```

When preprocessed this will produce:

```
a = (annebob)
```

This is useful for constructing variable names automatically and other more or less obscure hacks.

6.5 The # operator

This operator converts the given token into a character string. It can be used only within the body of a macro definition. After defining a macro like this:
`#define toString(Token) #Token`
an expression like `toString(MyToken)` will be translated after preprocessing into: `MyToken`

An example of its use is the following situation. We have a structure of an integer error code and a character field containing the description.

```
static struct table {
        unsigned int code;
        unsigned char *desc;
} hresultTab;
```

Then, we have a lot of error codes, defined with the preprocessor: `E_UNEXPECTED`, `E_NOTIMPL`, `E_INVALIDARG` etc. We can build a table with:

```
hresultTab Table[] = {
        {E_UNEXPECTED,"E_UNEXPECTED",}
        {E_NOTIMPL,"E_NOTIMPL",}
        ... etc
};
```

This is tedious, and there is a big probablity of making a typing mistake. A more intelligent way is:

```
#define CASE(a) {a,#a},
```

Now we can build our table like this:

```
hresultTab Table[] = {
        CASE(E_UNEXPECTED)
        CASE(E_NOTIMPL)
        ...
};
```

6.6 The include directive

This directive instructs the compiler to include the contents of the specified file as if you had simply typed the contents at that point in the source file.

This directive has two variants:

The first uses angle brackets to enclose the name of the file to include.

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

The second uses quotes to enclose the file name:

```
#include "myfile.h"
```

The first one means that the preprocessor should look for the file in the system include directory, where the files furnished by the implementation and system libraries are stored.

The second means that the preprocessor looks first in the current directory for the mentioned file. Note that the current directory starts as the one where the first source file is stored, the one that the compiler receives as a parameter when invoked. Of course, this can change later if you include a file in another directory.

For instance:

You have a file in the current directory that has an include directive like:

```
#include "myincludes/decls.h"
```

And within decls.h you have an include directive like:

```
#include "stddecls.h"
```

The preprocessor starts looking for stddecls.h in "myincludes", not in the original directory. This behavior is not mandated by the standard but it is fairly common in other compilers besides lcc-win. A third variant of this directive (seldom used) is to use a macro to define the file to be included. For instance:

```
#ifdef STANDARD_CODE
#define STDIO <stdio.h>
#else
#define STDIO "mystdio.h"
#include STDIO
```

The macro expansion must yield a correct directive of type (1) or (2).

6.7 Things to watch when using the preprocessor

- One of the most common errors is to add a semi colon after the macro:

  ```
  #define add(a,b) a+b;
  ```

  When expanded, this macro will add a semicolon into the text, with the consequence sometimes of either syntax errors or a change in the meaning of the code apparently no reason.

- Watch for side effects within macros. A macro invocation is similar to a function call, with the big difference that the arguments of the function call is evaluated once but in the macro can be evaluated several times. For instance we have the macro “square”:

```
#define square(a) (a*a)
```
If we use it like this:
```
        b = square(a++);
```
After expansion this will be converted into:
```
        b = (a++)*(a++);
```
and the variable a will be incremented twice.

- Function like macros are different from plain macros, and that difference is just that the opening parentheses must follow immediately the macro name. If you put a space between the macro name and the parentheses unexpected results will happen. For instance:
`#define SQR(x)  ((x)*(x))`
That replaces `SQR(2)` with `((2)*(2))`. OK. But
`#define SQR (x) ((x)*(x))`
That replaces `SQR(2)` with `(x) ((x)*(x))(2)` what is a syntax error!

- Each included file should only be included once to avoid repeating the definitions inside it. Some compilers (lcc-win, Microsoft, intel) provide the `#pragma once` directive, that ensures that a file will only be included once. Another way to avoid multiple inclusion is to write:

  ```
  #ifndef FILE_foo_h_included
  #define FILE_foo_h_included
  // The contents of the file go here
  #endif
  ```

  The first time the file is seen within a compilation unit the symbol is not defined. It gets defined and the contents of the file are processed. The next time the file is included the symbol is already defined and nothing from the contents of the file is added to the compilation unit.

  There are advantages and disadvantages for each method. The `#pragma` once directive is easy to use and doesn't need a new symbol in the preprocessor. The second option is portable to all compilers, and (as a side effect) allows you to disable completely an include file without changing the source code at all. Invoke the compiler with:
  `lcc -DFILE_foo_h_included somesource.c`
  and the file will never be included.

7 More advanced stuff

7.1 Using function pointers

Function pointers are one of the great ideas of C. Functions can be used as objects that can be passed around, stored in tables, and used in many ways within the language. To give an idea of the possibilities we will use a pracical example.

A very common programming problem is to recursively explore a certain part of the file system to find files that have a certain name, for instance you want to know all the files with the ".c" extension in a directory and in all subdirectories. To build such a utility you can do:

1. Build a list or table containing each file found, and return those results to the user.

2. For each file that is found, you call a user provided function that will receive the name of the file found. The user decides what does he/she want to do with the file.

Note that solution 2 includes solution 1, since the user can write a function that builds the list or table in the format he wants, instead of in a predefined format. Besides, there are many options as to what information should be provided to the user. Is he interested in the size of the file? Or in the date? Who knows. You can't know it in advance, and the most flexible solution is the best.

We can implement this by using a function pointer that will be called by the scanning function each time a file is found. We define:

```
typedef int (*callback)(char *);
```

This means, "a function pointer called callback that points to a function that returns an int and receives a char * as its argument". This function pointer will be passed to our scanning function and will return non-zero (scanning should be continued) or zero (scanning should stop).

Here is a possible implementation of this concept:

```
#include <stdio.h>
#include <windows.h>
#include <direct.h>
// Here is our callback definition
typedef int(*callback)(char *);
```

```
/*
This function has two phases. In the first, we scan for normal files and ignore any
directories that we find. For each file that matches we call the given function pointer.
The input, the char * "spec" argument should be a character string like "*.c" or "*.h".
If several specifications are needed, they should be separated by the ';' semi colon
char. For instance we can use "*.c;*.h;*.asm" to find all files that match any of those
file types. The second argument should be a function that will be called at each file
found.
*/

int ScanFiles(char *spec,callback fn)
{
    char *p,*q; // Used to parse the specs
    char dir[MAX_PATH]; // Contains the starting directory
    char fullname[MAX_PATH];// will be passed to the function
    HANDLE hdir;
    HANDLE h;
    WIN32_FIND_DATA dirdata;
    WIN32_FIND_DATA data;

    // Get the current directory so that we can always
    // come back to it after calling recursively this
    // function in another dir.
    memset(dir,0,sizeof(dir));
    getcwd(dir,sizeof(dir)-1);
    // This variable holds the current specification we are using
    q = spec;
    // First pass. We scan here only normal files, looping for each
    // of the specifications separated by ';'
    do {
        // Find the first specification
        p = strchr(q,';');
        // Cut the specification at the separator char.
        if (p)
            *p = 0;
        h = FindFirstFile(q,&data);
        if (h != INVALID_HANDLE_VALUE) {
            do {
                if (!(data.dwFileAttributes &
                              FILE_ATTRIBUTE_DIRECTORY)) {
                  // We have found a matching file.
                  // Call the user's function.
                  sprintf(fullname,
                       "%s\\%s",dir,data.cFileName);
                  if (!fn(fullname))
                      return 0;
                }
```

```
                } while (FindNextFile(h,&data));
                FindClose(h);
            }
        // Restore the input specification. It would be surprising
        // for the user of this application that we destroyed the
        // character string that was passed to this function.
            if (p)
               *p++ = ';';
               // Advance q to the next specification
            q = p;
    } while (q);
    // OK. We have done all the files in this directory.
    // Now look if there are any subdirectories in it,
    // and if we found any, recurse.
    hdir = FindFirstFile("*.*",&dirdata);
    if (hdir != INVALID_HANDLE_VALUE) {
        do {
            if (dirdata.dwFileAttributes &
                                FILE_ATTRIBUTE_DIRECTORY) {
                // This is a directory entry.
                // Ignore the ‘.’’ and ‘..’’ entries.
                if (! (dirdata.cFileName[0] == '.' &&
                      (dirdata.cFileName[1] == 0 ||
                            dirdata.cFileName[1] == '.'))) {
                // We change the current dir to the subdir
                // and recurse
                chdir(dirdata.cFileName);
                ScanFiles(spec,fn);
                // Restore current directory to the former one
                chdir(dir);
                }
            }
        } while (FindNextFile(hdir,&dirdata));
        FindClose(hdir);
    }
    return 1;
}
```

This function above could be used in a program like this:

```
static int files; // used to count the number of files seen
// This is the callback function. It will print the name of
// the file and increment a counter.
int printfile(char *fname)
{
        printf("%s\n",fname);
        files++;
```

```
        return 1;
}
// main expects a specification, and possibly a starting directory.
// If no starting directory is given, it will default to the
// current directory.
// Note that many error checks are absent to simplify the code.
// No validation is done to the result of the chdir function,
// for instance.
int main(int argc,char *argv[])
{
        char spec[MAX_PATH];
        char startdir[MAX_PATH];

        if (argc == 1) {
                printf(``scan files expects a file spec\n'');
                return 1;
        }
        memset(startdir,0,sizeof(startdir));
        memset(spec,0,sizeof(spec));
        strncpy(spec,argv[1],sizeof(spec)-1);
        if (argc > 2) {
                strcpy(startdir,argv[2]);
        }
        if (startdir[0] == 0) {
                getcwd(startdir,sizeof(startdir)-1);
                chdir(startdir);
        }
        files = 0;
        ScanFiles(spec,printfile);
        printf("%d files\n",files);
        return 0;
}
```

What is interesting about this solution, is that we use no intermediate memory to hold the results. If we have a lot of files, the size of the resulting list or table would be significant. If an application doesn't need this, it doesn't have to pay for the extra overhead.

Using the name of the file, the callback function can query any property of the file like the date of creation, the size, the owner, etc. Since the user writes that function there is no need to give several options to filter that information.One of the big advantages of C is its ability of using function pointers as first class objects that can be passed around, and used as input for other procedures. Without this feature, this application would have been difficult to write and would be a lot less flexible.

Another use of function pointers is to avoid branches. Assume the following problem:

Write a program to print all numbers from 1 to n where n integer > 0 without using any control flow statements (switch, if, goto, while, for, etc). Numbers can be

written in any order.

Solution:

The basic idea is to use a table of function pointers as a decision table. This can be done like this:

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

typedef void (*callback)(int);
void zero(int);
void greaterZero(int);
// We define a table of two functions, that represent a
// boolean  decision indexed by either zero or one.
callback callbackTable[2] = {
        zero,
        greaterZero
};

void zero(int a)
{
        exit(EXIT_SUCCESS); // This terminates recursion
}

void greaterZero(int a)
{
        printf("%d\n",a--);
        callbackTable[a>0](a); // recurse with a-1
}
int main(int argc,char *argv[])
{
        // assume correct arguments, n > 0
        greaterZero(atoi(argv[1]));
        return 0;
}
```

Error checking can be added to the above program in a similar way. This is left as an exercise for the interested reader. Of course the utility of function tables is not exhausted by this rather artificial example; Decision tables can be used as a replacement of dense switch statements, for instance. When you are very interested in execution speed, this is the fastest way of making multi-way branches. Instead of writing:

```
switch(a) {
        case 1:
                // some code
        break;
        case 2:
        break;
```

```
        ...
}
```

you could index a table of functions, each function containing the code of each case. You would construct a function table similar to the one above, and you would go to the correct "case" function using: Table[a](); The best is, of course, to use the switch syntax and avoid the overhead of a function call. Lcc-win allows you to do this with:
`#pragma density(0)`
This will modify the threshold parameter in the switch construction in the compiler so that a dense table of pointers is automatically generated for each case and all of the cases that aren't represented. This is dangerous if your code contains things like:

```
        case 1:
        ...
        case 934088:
        ...
```

In this case the table could be a very bad idea. This is a good technique for very dense switch tables only.

There are many other uses of function tables, for instance in object oriented programming each object has a function table with the methods it supports. Since function pointers can be assigned during the run time, this allows to change the code to be run dynamically, another very useful application. We have come up with a decent solution let's see what others might do:

```
#include <stdio.h>
void print(int n)
{
        n && (print(n-1), 1) && printf("%d\n", n);
}

int main(int argc,char*argv[]) {
    print(atoi(argv[1]);
    return 0;
}
```

How does this work?

By using short circuit evaluation. The statement:

```
n && (print(n-1), 1) && printf("%d\n", n);
```

can be read (from left to right) like this: If n is zero do nothing and return. The second expression (`print(n-1),1`) calls print, then yields 1 as its result. This means that the third expression is evaluated, expression that prints the number.

7.2 Using the "signal" function

The signal function allows us to catch a signal sent to a program by invoking a previously defined function when the signal arrives. This function has many uses, and one of them is catching the terrible "Segment violation" signal.

That signal is sent to a program that is trying to access memory completely outside its allowed range. This is always an error in the program, and when this errors arise the operating system will get quite angry and will terminate the offending program on the spot.

Obviously we would like to avoid that fate, and even more, we would like to know before using a suspect pointer, if it is a pointer to a readable portion of memory or not. So, the idea of writing a function to do that arises.

The central idea of that function is then:

- Setup a signal handler that will catch the signal SIGSEGV and make a long jump to a previously established recovery point.
- Start reading from memory for the given size.
- If no problems arise that part of memory was OK to use. We return 1.
- If a problem arises, our signal handler catches it, we make the longjmp and return zero.

```
 1 /* Adapted from:
 2    http://fixunix.com/linux/337646-isbadreadptr-linux.html */
 3 #include <setjmp.h>
 4 #include <signal.h>
 5 static jmp_buf PtrTestJmpBuf;
 6 static void PtrTestHandler(int nSig)
 7 {
 8     longjmp(PtrTestJmpBuf, 1);
 9 }
10
11 int IsBadReadPtr(void* lp, size_t ObjectSize)
12 {
13     int r  = 1;
14     void (* PrevHandler)(int);
15     if (setjmp(PtrTestJmpBuf)) {
16         r = 0;
17     }
18     else {
19         volatile unsigned char c;
20         size_t i;
21         PrevHandler = signal(SIGSEGV, PtrTestHandler);
22
23         for (i = 0; i < ObjectSize; i ++)
24             c = ((unsigned char*)lp)[i];
25     }
26     signal(SIGSEGV, PrevHandler);
27
28     return r;
29 }
```

Note the `volatile` declaration in line 19. The problem there is that the variable `c` is not used at all in the program and a clever compiler could then optimize away the assignment, then the entire loop since that assignment is the only statement in the loop. Lcc-win doesn't do those optimizations (yet), but it could do that in the future.

7.2.1 Discussion

longjmp usage

There is a big discussion going on within the standards committee about calling longjmp from a signal handler. The situation is quite confusing but in any case you can be sure that you can do it safely with lcc-win. In other systems the situation is much more problematic [1].

Guard pages

Another, much more serious problems is the interference of this code with established signal handlers and guard pages. Basically, a guard page is a section of virtual memory that has been reserved. When "touched" by a program, the system sends an exception that is catched to allocated that page and continue execution. The problem is that that exception is sent only once by the system. Touching it with our code provokes a crash later on, since we do not allocate the page.

Since under windows stack pages are always allocated as guard pages, this situation could be quite frequent, and the use of IsBadReadPtr has been discouraged by official Microsoft sites like MSDN.

A more detailed discussion of these issues can be found in
`http://blogs.msdn.com/b/oldnewthing/archive/2006/09/27/773741.aspx`

[1] Heikki Kallasjoki pointed me to this fact. He wrote:
The act of calling longjmp() from within a signal handler, while possible in many circumstances/systems, is not quite completely standard. The following page summarizes the state:
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1318.htm
In particular, C89 has some verbiage to allow it (in certain cases), which has been removed from C99; and even a C90 defect report response qualifies that it is required to work only for signal handlers invoked by raise() or abort(). It does not seem to appear on the lists of signal-handler-safe functions in any standard.

8 Advanced C programming with lcc-win

The development of the C++ language has had an adverse effect in the development of C. Since C++ was designed as the "better C", C was (and is) presented as the example of "how not to do things", even if both languages retained a large common base.

The need for a simple and efficient language persists however, and C is the language of choice for many systems running today and is used as the implementation language for many new ones. However, programming in C is made more difficult than it should be because of some glaring deficiencies like its buggy string library and the lack of a common library for the most used containers like lists, stacks, and other popular data structures.

Since C++ went out to be "the better C", it is important to avoid reintroducing the whole complexities of C++ into C and keep the language as simple as it should be, but not simpler than the minimum necessary to use it without much pain.

Of course the idea of improving C is doomed according to the C++ people, that obviously will say that the solution is not to improve C but to come over to C++. This same failure forecast is to be found in some C people, that see any change of their baby as the beginning of the end of the "spirit of C".

All this developments are implemented using the lcc-win32 compiler system1. These are not just proposals but a reference implementation exists, and it is widely distributed since several years.

The main propositions developed here are:

- Operator overloading
- Garbage Collection
- Generic functions
- Default function arguments
- References

All this propositions have as a goal increasing the level of abstraction used by C programmers without unduly increasing the complexity of the implementation. All this enhancements have added only about 2 000 lines of code to the original code of the lcc-win compiler. This is extremely small, and proves that apparently difficult extensions can be inserted into an existing compiler without any code bloat. Each enhancement can be viewed separately, but their strength is only visible when they all work together.

8.1 Operator overloading

Operator overloading allows the user to define its own functions for performing the basic operations of the language for user defined data types.

8.1.1 What is 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-win 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-win 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".

Many languages today accept operator overloading. Among them Ada, C++, C#, D, Delphi, Perl, Python, Visual Basic,.Ruby, Smalltalk, Eiffel.

The purpose of this enhancement within the context of the C language is to:

- Allow the user to define new types of numbers or "numeric" objects.
- Allow a generic access to containers by allowing the user to define special 'array like' access to containers using the overloaded '[' and ']' operators.

Let's see this applications in more detail.

1. Many applications need to define special kinds of numbers. Rational arithmetic, "big" numbers, extended precision floating point come immediately to mind, and there are surely many others. For instance the Technical Report 24732 of the ISO/IEC proposes a new kind of decimal floating point, the Technical Report DTR 18037 proposes fixed point operations, etc. All of them propose changes to the language in the form of new keywords. A conceptually simpler solution is to allow a single change that would accommodate all those needs

without making C impossible to follow by adding a new keyword for every kind of number that users may need.

2. The usage of arrays in C is peculiar and quite difficult to use. Allowing users to define new kinds of array access permits to integrate many needs like bounds checking within the language without adding any special new syntax. There are several propositions about bounded strings circulating in the standardization committee, and they propose several different enhancements to the existing library, mainly by the addition of several parameters to the string functions to pass the length of the receiving strings. This is a misguided approach since it still leaves too much work to the programmer that should still take care of following the size of each string he/she uses in the program without ever making a mistake. This is asking for trouble.

Obviously counting string lengths is better done by machines. The length should be a quantity associated to the string and managed at runtime by the routines using those strings. This would be, by the way, much more efficient than searching the terminating zero each and every time a string is used.

Still, it is needed to retain the original array-like syntax for this strings or bounded buffers/arrays. It is an intuitive syntax, in use almost in all programming languages and it will allow an easier transition of existing code. Then, we need an overloading of the operator index [ ].

These are the main objectives of this syntax change. Note that it is not in the design objectives to replace normal procedures like string concatenation with an overloaded "add" operator. It is obvious too, that once this syntax is in use, such bad applications can be programmed and it is impossible to do anything about them.2

Syntax:

```
result-type operator symbol ( parameters )
```

- `result-type` is the type of the operator result.
- `symbol` is one of the operator symbols (`+ - * / << [` etc. Explained in detail later)
- The `operator` contextual keyword is only valid within this context. Outside is a normal identifier.

An exception to the above rule are the pre-increment and pre-decrement operators, that are written:

```
result-type ++operator ‘(‘ parameters ‘)’
result-type --operator ‘(‘ parameters ‘)’
```

This enhancement doesn't use any new keywords. The C99 standard explicitly forbids new keywords, and this has been respected. It remains to be seen if really an operator keyword is needed. As implemented in the reference implementation it is still possible to write:

```
int operator = 67;
```

without any problems.

The rules for using the operator identifier are as follows:

- It must appear at the global level.
- It must be preceded by a type name. |item It must be followed by one of the operator symbols, and then an opening parentheses, an argument list that can't be empty and can't be longer than 2 arguments, followed by a closing parenthesis.
- If it is followed by a ";" it is a prototype for an operator defined elsewhere. All rules applying to prototypes apply equally to this prototype.
- If it is followed by an opening brace it is the beginning of an operator definition. All rules that apply to function definition apply also here.

Note that all this rules are no longer needed if the standard accepts a new "operator" keyword. The operators that can be overloaded are:

	Description and prototype
`+`	`Type operator+(const Type arg1,const Type arg2);` The arguments aren't necessarily of the same type. Pointers can't be used for arg1 or arg2.
`-`	`Type operator-(const Type arg1,const Type arg2);` The parameters can't be pointers.
`-`	`Type operator-(const Type arg1);`
`*`	`Type operator*(const Type arg1,const Type arg2);`
`/`	`Type operator/(const Type arg1,const Type arg2);`
`==`	`int operator==(const Type arg1,const Type arg2);` The parameters can't be pointers and the result type must be an integer
`!=`	`int operator!=(const Type arg1,const Type arg2);` The parameters can't be pointers, and the result type must be an integer.
`++`	`Type operator++(Type arg1);` The parameter can't be a pointer.
`--`	`Type operator-- (Type arg1);` The parameter can't be a pointer
`++`	`Type ++operator (Type arg1);` The parameter can't be a pointer.
`--`	`Type --operatorType arg1);` The parameter can't be a pointer.
`<`	`int operator< (const Type arg1, const Type arg2);` The parameters can't be pointers. Result type is int.
`<=`	`int operator<= (const Type arg1,const Type arg2);` The parameters can't be pointers. Result type is int.
`>=`	`int operator>= (const Type arg1, const Type arg2);` The parameters can't be pointers. Result type int.
`!`	`int operator!(const Type arg1);`
`~`	`Type operator const(Type arg1);`
`%`	`Type operator%(const Type arg1,const Type arg2);`

Table 8.1 – Continued

	Description and prototype
<<	`Type operator<<(const Type arg1,const Type arg2);`
>>	`Type operator>>(const Type arg1,const Type arg2);`
=	`Type operator=(const Type1 arg1,const Type2 arg2);`
^	`Type operator^(const Type arg1,const Type arg2);`
&	`Type operator&(const Type arg1,const Type arg2);`
|	`Type operator|(const Type arg1,const Type arg2);`
[]	`Type operator[](Type table,int index);`
[]=	`Type operator[]=(Type table,int index, Type Newvalue);`
+=	`Type operator+=(Type &arg1, Type arg2);` References can be replaced by arrays of length 1.
-=	`Type operator-=(Type &arg1, Type arg2);`
*=	`Type operator*=(Type &arg1, Type arg2);`
/=	`Type operator/=(Type &arg1, Type arg2);`
<<=	`Type operator<<=(Type &arg1, Type arg2);`
>>=	`Type operator>>=(Type &arg1, Type arg2);`
()	`Type operator()(Type arg1);` The parameter can't be a pointer.
*	`Type operator*(Type arg);` Parameter can't be a pointer.

8.1.2 Rules for the arguments

At least one of the parameters for the overloaded operators must be a user defined type. They must be a structure or a union, not just a typedef for a basic type. Pointers are accepted only when the operator has no standard C counterpart for operations with pointers. Pointer multiplication is not allowed in standard C, so an overloaded multiplication operator that takes two pointers is not ambiguous. Addition of pointer and integer is well defined in C, so an operator add that would take a pointer and an integer would introduce an ambiguity in the language, and it is therefore not allowed. The same for pointer subtraction.

The result type of an operator can be any type, except for the equality and the other comparison operators that always return an integer, either one or zero.

The number of arguments are fixed for each operator (as described in the table above). It is not possible to change this and define ternary operators that would make two multiplications, for instance.

When an operator needs to modify its argument (for instance the assignment operator, or the += operator) and can't take pointers, it should take a reference to the object to be modified. This ties somehow this enhancement to the second one described further down, references.

Overloaded operators can't have default arguments.

8.1.3 Name resolution

Name resolution is the process of selecting the right operator from a list of possible candidates. You can define several functions for each operator, each taking some different input types. The compiler builds a list of the different definitions, and at each call of the operator selects from the functions that could apply the one that should be called.

There can be only one operator that applies to a given combination of input arguments, i.e. to a given signature. If at the end of the name resolution procedure more than one overloaded operator is found a fatal diagnostic is issued, and no object file is generated. It is very important that any incoherence be flagged at compile time and avoids producing a call to a function that the programmer did not intend to call.

Step one: Compare the input arguments for the list of overloaded functions for this operator without any promotion at all. If at the end of this operation one and only one match is found return success.

Step two: Compare input arguments ignoring any signed/unsigned differences. If in the implementation sizeof(int) == sizeof(long) consider long and int as equivalent types. The same if in the implementation sizeof(int) == sizeof(short). Consider the enum type as equivalent to the int type. If at the end of this operation one and only one match is found return success.

Step three: Compare input arguments ignoring all differences between numeric arguments. If at the end of this operation only one match is found return success.

Step four: If the operation is one of the comparisons operators (equal, not equal, less, less-equal greater, greater-equal) invert the operation and try to find a match. If it is found invert the order of the arguments for less, less-equal greater-equal greater, or, call the not operator for equal and not equal.

Step five: Return failure.

Assumed operator equivalences

Operator	Equivalent
equals	! different
different	! equals
less	invert: less(a,b) is greater-equal(b,a)
less-equal	invert: less-equal(a,b) is greater(b,a)
greater-equal	invert: greater-equal(a,b) is less(b,a)
greater:	invert: greater(a,b) is less-equal(b,a)
+=	binary add + assignment
-= binary minus + assignment	
...	all others

8.1.4 Differences to C++

In the C++ language, you can redefine the operators `&&` (and) `||` (or) and **,** (comma). You cannot do this in C. The reasons are very simple.

In C (as in C++), logical expressions within conditional contexts are evaluated from left to right. If, in the context of the AND operator, the first expression returns a FALSE value, the others will NOT be evaluated. This means that once the truth or

falsehood of an expression has been determined, evaluation of the expression ceases, even if some parts of the expression haven't yet been examined.

Now, if a user wanted to redefine the operator AND or the operator OR, the compiler would have to generate a function call to the user-defined function, giving it all the arguments of BOTH expressions. To make the function call, the compiler would have to evaluate them both, before passing them to the redefined operator `&&`.

Consequence: all expressions would be evaluated and expressions that rely on the normal behavior of C would not work. The same reasoning can be applied to the operator OR. It evaluates all expressions, but stops at the first that returns TRUE.

A similar problem appears with the comma operator, which evaluates in sequence all the expressions separated by the comma(s), and returns as the value of the expression the last result evaluated. When passing the arguments to the overloaded function, however, there is no guarantee that the order of evaluation will be from left to right. The C standard does not specify the order for evaluating function arguments. Therefore, this would not work.

Another difference with C++ is that here you can redefine the operator []=, i.e., the assignment to an array is a different operation than the reference of an array member. The reason is simple: the C language always distinguishes between the operator + and the operator +=, the operator * is different from the operator *=, etc. There is no reason why the operator [] should be any different.

This simple fact allows you to do things that are quite impossible for C++ programmers: You can easily distinguish between the assignment and the reference of an array, i.e., you can specialize the operation for each usage. In C++ doing this implies creating a "proxy" object, i.e., a stand-by construct that senses when the program uses it for writing or reading and acts accordingly. This proxy must be defined, created, etc., and it has to redefine all operators to be able to function. In addition, this highly complex solution is not guaranteed to work! The proxies have subtle different behaviors in many situations because they are not the objects they stand for.

In C++ the `[ ]` operator can be only defined within a class. There are no classes in C, and the `[ ]` operator is defined like any other.

8.2 Generic functions

Like an ordinary function, a generic function takes arguments, performs a series of operations, and perhaps returns useful values. An ordinary function has a single body of code that is always executed when the function is called. A generic function has a set of bodies of code of which only one is selected for execution. The selected body of code is determined by the types of the arguments to the generic function. Ordinary functions and generic functions are called with identical function-call syntax.

Generic functions were introduced into C by the C99 standard with the header tgmath.h. This introduction was severely limited to some mathematical functions but pointed to the need of having one name to remember instead of memorizing several for functions that essentially do the same thing but with slightly different data types.

With the proliferation of numeric types in C it is obvious that remembering the name of each sin function, the one for floats the one for complex, the one for long doubles, etc. uses too much memory space to be acceptable. Here I am speaking about the really important memory space: human memory space, that is far more precious now that the cheap random access memory that anyone can buy in the next supermarket. As far as I know there are no human memory extensions for sale, and an interface and a programming language is also judged by the number of things the user must learn by heart, i.e. its memory footprint.

To reduce the complexities of C interfaces two solutions are proposed: The first is the use of generic functions, i.e. functions that group several slightly different task into a single conceptual one, and default arguments, that reduce the number of arguments that the user must supply in a function call, and therefore its memory footprint. Default arguments will be explained in the next section.
Syntax:

```
result-type overloaded functionName( argument-list )
```

The same rules apply to the identifier "overloaded" as to the identifier "operator" above. It is not a keyword in this implementation to remain compatible with the standard.

The compiler will synthesize a name for this instance of the overloaded function from the name and its argument list.

If this instance of the overloaded function should be an alias for an existing function, the syntax is as follows:

```
result-type someFn( argument-list );
result-type overloaded functionName.someFn( argument-list )
```

This second form avoids any automatically generated names, what makes the code binary compatible with other compilers.

8.2.1 Usage rules

- An overloaded function must be a new identifier. It is an error to declare a function as overloaded after the compiler has seen a different prototype in the same scope.
- It is also an error to declare a function that was declared as overloaded as a normal function.
- The rules for name resolution are the same as the rules for operator overloading excepting of course the operator equivalence rule.
- Generic functions can't have default arguments.

8.3 Default arguments

Default arguments are formal parameters that are given a value (the default value) to use when the caller does not supply an actual parameter value. Of course, actual

parameter values, when given, override the default values. In many situations, some of the arguments to a function can be default values. This simplifies the interface for the function and at the same time keeps the necessary option for the user of the function, to specify some corner cases exactly.

Default arguments are used in Python, Fortran, the "Mathematica" language, Lisp, Ruby. Tcl/tk, Visual Basic.
Syntax:

```
return-type fnName(arg1,arg2,arg3=<expr>,arg4=<expr>);
```

Usage Rules

- The expressions used should be constant expressions. All mandatory arguments should come first, then, the optional arguments.
- It is an error to redefine a function that has optional arguments into another with a different list of optional arguments or with different values.
- Note that is a very bad idea to change the value of a default argument since all code that uses that function depends implicitly in the value being what is declared to be. A change in the value of the default value assigned to an argument changes all calls to it implicitly.
- The expression given as default argument will be evaluated within the context of each call. If the expression contains references to global variables their current value will be used.

8.4 References

References are pointers to specific objects. They are immediately initialized to the object they will point to, and they will never point to any other object. They are immediately dereferenced when used.

This addition is necessary for the overloaded operators that modify their arguments and can't receive pointers. References can't be reassigned, and pointer arithmetic is not possible with them.

9 Numerical programming

Computers are done for making calculations, well, at least originally that was their objective. Playing games, internet browsing, word processing, etc., came later.

The problem of storing numbers in a computer however, is that the continuum of numbers is infinite and computers are limited. Yes, we have now many times the RAM amount that we had just a few years ago, but that amount is finite and the numbers we can represent in it are just a finite approximation to the infinite mathematical continuum.

The moment we convert a problem from the domain of mathematical analysis to the range of numbers that can be handled in a machine, even in a paper and pencil "machine", we are going to necessarily introduce approximations that can lead to errors, truncation errors, rounding errors, whatever.

Suppose we want to evaluate exp(x) by using a series expansion:

$$exp(x) = 1 + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}\frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320}...O(\frac{x^n}{n!}) \qquad (9.1)$$

We have to stop somewhere. No way out. Here we get tired at the 9th term. And no matter how much effort we put into this, there will be always a truncation error. In this case the truncation error can be accurately calculated. Analyzing and estimating the error bounds is the art of numerical analysis.

Computers use bit patterns to represent any object that can be represented in a computer. In the section about structures we represented a person by a bit pattern like this:

```
structure person {
         char *Name;
         int age;
         ...
};
```

A person is surely not a bit pattern. We use the bit pattern to abstract some characteristics of the person we are interested in. Numbers aren't different. They can be represented by a bit pattern too. We can use 32 bits, what allows us to represented almost 4294967296 numbers. But obviously there are more numbers (infinitely more) than that, so any representation will always fail somewhere.

We can imagine we have a grid spaced at "strategic" points within the real numbers, and that each point in a grid is a representable machine number.

Any computation involving floating point numbers (that map onto the real-line) as its arguments can potentially produce a result that lies in-between two floating

point numbers. In that case, that number is rounded off to one of the grid-points. And this incurs round off error. Any practical numerical analyst (one who uses a computer to carry out numerical computations) must have a method of bounding, estimating and controlling both the round off error at each numerical step as well as the total round off error.

What do we want from the representation of numbers then?

1. The grid points should be as dense as possible
2. The range (the span between the smallest and the largest number) should be as wide as possible
3. The number of bits used should be as small as possible.
4. The rules of arithmetic should be mimicked as closely as possible.
5. The rules should be such that they can be implemented in hard-wired computer logic.

Note that all those requirements are completely contradictory. If we want a dense representation we have to use more bits. If we increase the range we have to thin the spacing between numbers, etc.

9.1 Floating point formats

In lcc-win we have a plethora of numeric types, ranging from the smallest single precision format, to the bignum indefinite precision numbers. All the numeric types (including the complex numbers type but excluding the bignums) are based in different floating point formats. In this formats, a number is represented by its sign, an exponent, and a fraction.

The range and precision of each subset is determined by the IEEE Standard 754 for the types float, double and long double. The qfloat and the bignum data types have their own formats.

9.1.1 Float (32 bit) format

This format uses one bit for the sign, 8 bits for the exponent, and 23 bits for the fraction.

```
struct floatFormat {
    unsigned Fraction:23;
    unsigned Exponent:8;
    unsigned sign:1;
};
```

Bits 0:22 contain the 23-bit fraction, f, with bit 0 being the least significant bit of the fraction and bit 22 being the most significant; bits 23:30 contain the 8-bit biased exponent, e, with bit 23 being the least significant bit of the biased exponent and bit

30 being the most significant; and the highest-order bit 31 contains the sign bit, s. The normalized numbers are represented using this formula:

$$(-1) \cdot sign \cdot 2^{(exponent-127)} \cdot 1.fraction \tag{9.2}$$

Here we have an exponent that uses -127 as bias. This means that a constant is added to the actual exponent so that the number is always a positive number. The value of the constant depends on the number of bits available for the exponent. In this format the bias is 127, but in other formats this number will be different.

The range of this format is from `7f7fffff` to `00800000`, in decimal 3.40282347 E+38 to 1.17549435 E-38. This numbers are defined in the standard header file <float.h> as `FLT_MAX` and `FLT_MIN`. The number of significant digits is 6, defined in float.h as `FLT_DIG`. subsectionDouble (64 bit) format This format uses one bit for the sign, 11 bits for the exponent, and 52 bits for the fraction.

```
struct doubleFormat {
    unsigned FractionLow:32;
    unsigned FractionHigh:20;
    unsigned Exponent:11;
    unsigned sign:1;
}
```

Bits 0..51 contain the 52 bit fraction f, with bit 0 the least significant and bit 51 the most significant; bits 52..62 contain the 11 bit biased exponent e, and bit 63 contains the sign bit s. The normalized numbers are represented with:

$$(-1)s \cdot 2^{exponent-1023} \cdot 1.fraction \tag{9.3}$$

The bias for the exponent in this case is -1023. The range of this format is from `7feffffff fffffff` to `00100000 00000000` in decimal from 1.7976931348623157 E+308 to 2.2250738585072014 E-308. This numbers are defined in float.h as `DBL_MAX` and `DBL_MIN` respectively. The number of significant digits is 15, defined as `DBL_DIG`.

9.1.2 Long double (80 bit) format

This format uses one bit for the sign, 15 bits for the biased exponent, and 64 bits for the fraction. Bits 0..63 contain the fraction, the bits 64..78 store the exponent, and bit 79 contains the sign.

```
struct longdoubleFormat {
    unsigned FractionLow:32;
    unsigned FractionHigh:32;
    unsigned Exponent:15;
    unsigned sign:1;
}
```

The bias for the exponent is 16383 and the formula is:

$$(-1)sign \cdot 2^{exponent-16383} \cdot 1.fraction \tag{9.4}$$

The range of this format is from 7ffe ffffffff ffffffff to 0001 80000000 00000000, or, in decimal notation, from the number 1.18973149535723176505 E+4932 to 3.36210314311209350626 E-4932. Quite enough to represent the number of atoms in the whole known universe. Those numbers are defined as `LDBL_MAX` and `LDBL_MIN` in float.h. The number of significant digits is 18, defined as `LDBL_DIG`. Note that even if the number of bits of the long double representation is 80, or ten bytes, sizeof(long double) is 12, and not 10. The reason for this is the alignment of this numbers in memory. To avoid having numbers aligned at addresses that are not multiple of four, two bytes of padding are added to each number.

9.1.3 The qfloat format

This format is specific to lcc-win and was designed by Stephen Moshier, the author of the "Cephes" mathematical library that lcc-win uses internally. Its description is as follows:

```
#define _NQ_    12
struct qfloatFormat {
    unsigned int sign;
    int exponent;
    unsigned int sign;
    int exponent;
    unsigned int mantissa[_NQ_];
} ;
```

in 64 bits machines this format has been expanded:

```
#define _NQ_    7
struct qfloatFormat {
    unsigned int sign;
    unsigned exponent;
    unsigned int sign;
    int exponent;
    unsigned long long mantissa[_NQ_];
} ;
```

This is defined in the "qfloat.h" header file.In 32 bits it provides 104 significant digits, a fraction of 352 bits, (one word is left empty for technical reasons) and a biased exponent of 32 bits. In 64 bits it provides 448 bits of fraction with roughly 132 significant digits.

9.1.4 Special numbers

All the floating point representations include two "numbers" that represent an error or NAN, and signed infinity (positive and negative infinity). The representation of NANs in the IEEE formats is as follows:

type	nan	+Infinity	-Infinity
float	7fc00000	7f800000	ff800000
double	7ff80000 00000000	7ff00000 00000000	fff00000 00000000
long double	7fffffffffff ffffffff	7fff80000000 00000000	ffff80000000 00000000

We have actually two types of NANs: quiet NANs and signalling NANs.

A Quiet NaN, when used as an operand in any floating point operation, quietly (that is without causing any trap or exception) produces another quiet NaN as the result, which, in turn, propagates. A Quiet NaN has a 1 set in the most significant bit-position in the mantissa field.

A Signaling NaN has no business inside an FPU. Its very presence means a serious error. Signaling NaNs are intended to set off an alarm the moment they are fetched as an operand of a floating point instruction. FPUs are designed to raise a trap signal when they touch a bit pattern like that. Quiet NaNs are produced, when you do things like try to divide by zero, or you pass incorrect arguments to a standard FPU function, for instance taking the square root of -1. Modern FPUs have the ability to either produce a quiet NaN, or raise a signal of some sort, when they encounter such operands on such instructions. They can be initialized to do either of the two options, in case the code runs into these situations.

9.2 Range

OK. We have seen how floating point numbers are stored in memory. To give us an idea of the range and precision of the different formats let's try this simple program. It calculates the factorial of its argument, and it is not very efficient, since it will repeat all calculations at each time.

```
#include <stdio.h>
#include <math.h>
float factf(float f)
{
        float result=1.0;

        while (f > 0) {
                result *= f;
                if (!isfinitef(f))
                        break;
                f--;
        }
        return result;
}
int main(void)
{
        float ff=1.0f,fctf = 1.0;
        while (1) {
                ff = factf(fctf);
```

```
                if (!isfinitef(ff))
                        break;
                printf("%10.0f! = %40.21g\n",fctf,ff);
                fctf++;
        }
        printf("Max factorial is %g\n",fctf-1);
        return 0;
}
```

We start with the smallest format, the float format. We test for overflow with the standard function `is_finitef`, that returns 1 if its float argument is a valid floating point number, zero otherwise. We know that our fact() function will overflow, and produce a NAN (Not A Number) after some iterations, and we rely on this to stop the infinite loop. We obtain the following output:

```
 1! =                                       1
 2! =                                       2
 3! =                                       6
 4! =                                      24
 5! =                                     120
 6! =                                     720
 7! =                                    5040
 8! =                                   40320
 9! =                                  362880
10! =                                 3628800
11! =                                39916800
12! =                               479001600
13! =                              6227020800
14! =                             87178289152
15! =                           1307674279936
16! =                          20922788478976
17! =                         355687414628352
18! =                        6402374067290112
19! =                      121645096004222980
20! =                     2432902298041581600
21! =                    51090945235216237000
22! =                  1.124000724806013e+021
23! =                 2.585201744459486e+022
24! =                  6.204483105838766e+023
25! =                 1.551120992632473e+025
26! =                  4.032915733765936e+026
27! =                  1.088886923454107e+028
28! =                 3.048883905131812e+029
29! =                 8.841760614675607e+030
30! =                 2.652529093045374e+032
31! =                 8.222838447587481e+033
32! =                  2.631308303227994e+035
```

```
        33! =                  8.6833178760213554e+036
        34! =                   2.952328838437621e+038
Max factorial is 34
```

This measures the range of numbers stored in the float format, i.e. the capacity of this format to store big numbers. We modify slightly our program by replacing the float numbers by double numbers, and we obtain this:

```
         1! =                                        1
         2! =                                        2
         3! =                                        6
         4! =                                       24
         5! =                                      120
        ...
       168! =                  2.5260757449731988e+302
       169! =                  4.2690680090047056e+304
       170! =                   7.257415615308004e+306
Max factorial is 170
```

The range of double precision floats is much greater. We can go up to 170!, quite a big number. Even greater (as expected) is the range of long doubles, where we obtain:

```
      1751 =               3.674156538617319512e+4920
      1752 =               6.437122255657543772e+4923
      1753 =               1.128427531416767423e+4927
      1754 =               1.979261890105010059e+4930
Max factorial is 1754
```

Changing the declarations of the numbers to qfloats (and increasing the printing precision) increases even more the range:

```
3207! = 2.68603536247213602472539298328381221236937795484607e+9853
3208! = 8.61680144281061236731906069037446957728096447914618e+9856
3209! = 2.76513158299792550867268657554116728734946150135801e+9860
Max factorial is 3209
```

The range increases by 4,930 orders of magnitude.

9.3 Precision

What is the precision of those numbers? We modify the first program as follows:

```
int main(void)
{
        float f=1.0f,fctf = 1.0;

        fctf = factf(34.0f);
        f = fctf+1.0f; // Add one to fctf
```

```
        if (fctf != f) { // 1+fctf is equal to fctf ???
                printf("OK\n");
        }
        else
                printf("Not ok\n");
        return 0;
}
```

We obtain the factorial of 34. We add to it 1. Then we compare if it is equal or not to the same number. Against all mathematical expectations, our program prints "Not ok". In floating point maths, 1+N = N !!!

Why this?

The density of our format makes the gaps between one number and the next one bigger and bigger as the magnitude of the numbers increases. At the extreme of the scale, almost at overflow, the density of our numbers is extremely thin. We can get an idea of the size of the gaps by writing this:

```
int main(void)
{
        float f=1.0f,fctf = 1.0;

        fctf = factf(34.0f);
        f = 1.0f;
        while (fctf == (f+fctf)) {
                f *= 10.0f;
        }
        printf("Needs: %e\n",f);
        return 0;
}
```

We get the output:

```
        Needs: 1.000000e+019
```

We see that the gap between the numbers is huge: 1e19! What are the results for double precision? We modify our program and we get:

```
        Needs: 1.000000e+019
```

What???? Why are the results of double precision identical to the floating point precision? We should find that the smallest format would yield gaps much wider than the other, more precise format! Looking at the assembler code generated by our floating point program, we notice:

```
; while (fctf == (f+fctf))
        flds    -16(%ebp) ; loads fctf in the floating point unit
        fadds   -4(%ebp)  ; adds f to the number stored in the FPU
        fcomps  -16(%ebp) ; compares the sum with fctf
```

Looking at the manuals for the pentium processor we see that the addition is done using the full FPU precision (80 bits) and not in floating point precision. Each number is loaded into the FPU and automatically converted to a 80 bits precision number.

The lcc-win compiler (as many others) uses the floating point unit to make the calculations, and even if the floating point unit can *load* floats and doubles into it, once inside the numbers are handled in 80 bit precision. The only way to avoid that is to force the compiler to store the result into memory at each time. Under lcc-win this is relatively easy, but in many other highly optimizing compilers that can be almost impossible unless they provide a compilation parameter to force it.

We modify our program like this:

```
int main(void)
{
        float f,fctf,sum;

        fctf = factf(34.0f);
        f = 1.0f;
        sum = f+fctf;
        while (fctf == sum) {
                f *= 2.0f;
                sum = fctf+f;
        }
        printf("Needs: %e\n",f);
        return 0;
}
```

Note that this modified program is mathematically equivalent to the previous one. When we run it, we obtain:

```
Needs: 1.014120e+031
```

OK, now we see that the gap between numbers using the float format is much bigger than the one with double precision.

Note that both versions of the program are mathematically equivalent but numerically completely different! Note too that the results differ by 12 orders of magnitude just by modifying slightly the calculations.

We modify our program for double precision, and now we obtain:

```
Needs: 3.777893e+022
```

The gap using double precision is much smaller (9 orders of magnitude) than with single precision. Note that there are as many IEEE754 numbers between 1.0 and 2.0 as there are between 2^{56} and 2^{57} in double format. $2^{57} - 2^{56}$ is quite a big number: 72,057,594,037,927,936. Using qfloats now, we write:

```
#include <qfloat.h>
#include <stdio.h>
int main(void)
```

```
{
        qfloat f=34,fctf;

        fctf = factq(f);
        f = fctf+1;
        if (fctf != f) {
                printf("OK\n");
        }
        else
                printf("Not ok\n");
        return 0;
}
```

This prints OK at the first try. Using the extremely precise qfloat representation we obtain gaps smaller than 1 even when the magnitude of the numbers is 10^34. This extension of lcc-win allows you to use extremely precise representation only in the places where it is needed, and revert to other formats when that precision is no longer needed.

9.4 Understanding exactly the floating point format

Let's take a concrete example: the number 178.125.

Suppose this program:

```
#include <stdio.h>
// No compiler alignment
#pragma pack(1)
// In this structure we describe a simple precision floating point
// number.
typedef union {
    float fl;
    struct {
        unsigned f:23;      // fraction part
        unsigned e:8;       // exponent part
        unsigned sign:1;    // sign
        };
} number;

// This function prints the parts of a floating point number
// in binary and decimal notation.
void pfloat(number t)
{
    printf("Sign %d, exponent %d (-127= %d), fraction: %023b\n",
                    t.sign,t.e,t.e-127,t.f);
}
int main(void)
{
```

```
        number t;

        t.fl = 178.125;
        pfloat(t);
        return 0;
}
```

This will produce the output:

```
Sign 0, exponent 134 (-127= 7), fraction: 01100100010000000000000
```

To calculate the fraction we do:

```
fraction = 01100100001 =
0 * 1/2 +
1 * 1/4 +
1 * 1/8 +
0 * 1/16+
0 * 1/32+
1 * 1/64 +
 ... +
1 * 1/1024
```

This is:
0.25+0.125+0.015625+0.0009765625 = 0.3916015625
Then, we add 1 to 0.3916015625 obtaining 1.3916015625.
This number, we multiply it by $2^7 = 128$: 1,3916015625 * 128 = 178.125.

9.5 Rounding modes

When the result of a computation does not hit directly a number in our representation grid, we have to decide which number in the grid we should use as the result. This is called rounding. We have the following rounding modes:

1. Round to the nearest grid point. This is the default setting when a program compiled with lcc-win starts executing.

2. Round upwards. Choose always the next higher grid point in the direction of positive infinity.

3. Round downwards. Choose always the next lower grid point in the direction of negative infinity.

4. Round to zero. We choose always the next grid point in the direction of zero. If the number is positive we round down, if it is negative we round up.

This rounding modes are defined in the standard header file fenv.h as:

```
/* Rounding direction macros */
#define FE_TONEAREST    0
#define FE_DOWNWARD     1
#define FE_UPWARD       2
#define FE_TOWARDZERO   3
```

You can change the default rounding precision by using the standard function `fesetround (int)` also declared in the same header file.

The rationale for this "rounding modes" is the following: To know if an algorithm is stable, change your rounding mode using the same data and run your program in all rounding modes. Are the results the same? If yes, your program is numerically stable. If not, you got a big problem and you will have to debug your algorithm.

For a total of N floating point operations you will have a rounding error of:[1]

- For round to nearest is sqrt(N) * machine epsilon
- For round up is N * machine epsilon.
- For round down is -N * machine epsilon.
- For round to zero is -N * machine epsilon if the number is positive, N * Machine Epsilon if the number is negative.

The number you actually obtain will depend of the sequence of the operations.

9.6 The machine epsilon

The standard defines for each floating point representation (float, double, long double) the difference between 1 and the least value greater than 1 that is representable in the given floating point type. In IEEE754 representation this number has an exponent value of the bias-mantissa bits, and a mantissa of zero. Another way to define this quantity in terms of the mathematical functions of the C library is:

```
DBL_EPSILON = nextafter(1.0,2.0) - 1.0;
```

For the different representations we have in the standard header <float.h>:

```
#define FLT_EPSILON 1.19209290e-07F // float
#define DBL_EPSILON 2.2204460492503131e-16 // double
#define LDBL_EPSILON 1.084202172485504434007452e-19L //long double
// qfloat epsilon truncated so that it fits in this page...
#define QFLT_EPSILON 1.09003771904865842969737513593 11 ... E-106
```

These definitions (except the qfloat part) are part of the C99 ANSI standard. For the standard types (float, double and long double) they should always exist in other compilers.

Here is a program that will find out the machine epsilon for a given floating point representation.

[1] See http://serc.iisc.ernet.in/ ghoshal/fpv.html#nodeliver, or the home page of the Siddhartha Kumar Ghoshal, Senior Scientific Officer at the Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore.

```
#include <stdio.h>
int main(void)
{
   double float_radix=2.0;
   double inverse_radix = 1.0/float_radix;
   double machine_precision = 1.0;
   double temp = 1.0 + machine_precision;

   while (temp != 1.0) {
      machine_precision *= inverse_radix;
      temp = 1.0 + machine_precision ;
      printf("%.17g\n",machine_precision);
   }
   return 0;
}
```

Exercise 2

Explain why in the above program, the value of `DBL_EPSILON` is not the last number but the number before.

Exercise 3

An alternative version of the above program is:

```
#include <stdio.h>
int main(void)
{
    volatile double oneplus = 2, epsilon = 1;
    while (1 + epsilon/2 > 1) {
        epsilon /= 2;
        oneplus = 1 + epsilon;
    }
    epsilon = oneplus - 1;
    printf("DBL_EPSILON is %g\n", epsilon);
    return 0;
}
```

Explain why this program prints `DBL_EPSILON is 0` in lcc-win.

9.7 Rounding

When in C you convert a floating point number into an integer, the result is calculated using rounding towards zero. To see this in action look at this simple program:

```
#include <stdio.h>
void fn(double a)
{
        printf("(int)(%g)=%d (int)(%g)=%d\n",a,(int)a,-a,(int)-a);
}
int main(void) {
```

```
        for (double d = -1.2; d < 2.0; d += 0.3)
                fn(d);
        return 0;
}
```

This leads to the following output (note the lack of precision: 0.3 can't be exactly represented in binary):

```
(int)(-1.5)=-1   (int)(1.5)=1
(int)(-1.2)=-1   (int)(1.2)=1
(int)(-0.9)=0    (int)(0.9)=0
(int)(-0.6)=0    (int)(0.6)=0
(int)(-0.3)=0    (int)(0.3)=0
(int)(1.11022e-016)=0    (int)(-1.11022e-016)=0
(int)(0.3)=0     (int)(-0.3)=0
(int)(0.6)=0     (int)(-0.6)=0
(int)(0.9)=0     (int)(-0.9)=0
(int)(1.2)=1     (int)(-1.2)=-1
(int)(1.5)=1     (int)(-1.5)=-1
(int)(1.8)=1     (int)(-1.8)=-1
```

To round away from zero you can use:

```
#define Round(x) ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
```

This will give nonsense results if there is an overflow. A better version would be:

```
#define Round(x) \
        ((x) < LONG_MIN-0.5 || (x) > LONG_MAX+0.5 ?\
        error() :
                ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
```

The standard library function `round()` does this too. The round functions (`roundf`, `round`, and `roundl`) round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.[2].

To round towards positive infinity you use:

```
#define RoundUp(x) ((int)(x+0.5))
```

Negative infinity is similar.

9.8 Using the floating point environment

The C standard specifies a type `fenv_t` that "refers collectively to any floating-point status flags and control modes supported by the implementation."

This environment has two special parts, the "floating point status flags" that are set by the operations the program performs, and the "control flags" that changes how the hardware does the operations, for instance the rounding mode, etc.

[2] Ansi C standard page 232

9.8.1 The status flags

Flag	Meaning
FE_DIVBYZERO	A division by zero has occurred
FE_INEXACT	The last operation had to be rounded
FE_INVALID	An invalid operation was attempted
FE_OVERFLOW	Result was too big to be represented
FE_UNDERFLOW	Result was too small to be represented

In addition to this standard flags, lcc-win provides two additional flags provided by the processor and passed by lcc-win into the environment flags.

Flag	Meaning
FE_DENORMAL	The last operation was a denormal number
FE_STACKFAULT	Overflow of the floating point stack

The denormal flag means that a loss of precision is certain, the number is too small to be represented. The stack fault flag means that lcc-win generated bad code, since all floating point operations should balance the floating point stack. If you ever test positive for this flag, do not hesitate to send me a bug report! In general, the floating point flags can be queried by the program to know if the last operation did complete without problems. Here is a small example to show you how this works:

```
/* This program tests a division by zero */
#include <fenv.h>
#include <stdio.h>
int main(void)
{
        double a=1,b=0;
        feclearexcept(FE_DIVBYZERO);
        a=a/b;
        if (fetestexcept(FE_DIVBYZERO)) {
                printf("You have divided by zero!\n");
        }
        return 0;
}
```

First we clear the flag that we are going to use using feclearexcept. Then, we perform our operation and query if the flag is set using fetestexcept. Since we know that the flags are set but not cleared, the expression could be very well be a much more complicated sequence of operations. The above code would work the same, but we would lose the possibility of knowing exactly which operation failed. This is in many cases not very important, we could very well be interested that somewhere there was a serious error, without bothering to investigate which operation was that it failed.

9.8.2 Reinitializing the floating point environment

Things can become messy, and you would like to reset everything to a known state. The standard provides the macro FE_DFL_ENV that represents the address of the

default environment, the one you started with. In lcc-win this environment is stored in the `__default_fe_env` global variable, so this macro is just:

```
#define FE_DFL_ENV  (&__default_fe_env)
```

You can reset everything to its default state with: `fesetenv(FE_DFL_ENV);` The default environment in lcc-win has the following characteristics:

1. The rounding mode is set to round to nearest.
2. The precision of the FPU is set to full precision (80 bits).
3. All exceptions are masked, i.e. the result of invalid operations is a NAN, not a trap.

9.9 Numerical stability

Suppose we have a starting point and a recurrence relation for calculating the numerical value of an integral.[3] The starting value is given by:

```
I0=[ln(x+5)]10 = ln 6 - ln 5 = 0.182322
```

The recurrence relation is given by:

```
I1 = 1/1 - 5I0
I2 = 1/2 - 5I1
I3 = 1/3 - 5I2
```

etc.

We calculate this, starting with 0.182322. We use the following program:

```
#include <stdio.h>
#include <math.h>
int main(void)
{
        float i = 0.182322;

        for (int z = 1; z<9;z++) {
                i = 1.0f/(float)z - 5.0*i;
                printf("I%-3d: %9.6g\n",z,i);
        }
        return 0;
}
```

We use single precision. Note the notation 1.0f meaning 1 in float precision.

[3] I have this example from the very good book "Numerical Mathematics and Scientific Computation" by Germund Dahlquist and Ake Björck. Available on line at:
http://www.mai.liu.se/ akbjo/NMbook.html

```
I1  :   0.08839
I2  : 0.0580499
I3  : 0.0430839
I4  : 0.0345805
I5  : 0.0270974
I6  : 0.0311798
I7  : -0.0130418
I8  :  0.190209
```

The first few numbers look correct, but I6 is bigger than I5, what after the recurrence relation should never happen. Moreover I7 is negative and later numbers are complete nonsense.

Why?

Well, because the roundoff error e in I0 is multiplied by 5 in the first iteration, then multiplied again by 5 in the next iteration so that after a few iterations the error becomes bigger than the result.

Writing this is double precision, and replacing the precalculated constant with a computation of log(6.0) - log(5.0) we get better results.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
        double i = log(6.0) - log(5.0);

        for (int z = 1; z<29;z++) {
                i = 1.0/(double)z - 5.0*i;
                printf("I%-3d: %9.6g\n",z,i);
        }
        return 0;
}
```

We get:

I1 : 0.0883922	I11 : 0.0140713	I21 : -0.0158682
I2 : 0.0580389	I12 : 0.0129767	I22 : 0.124796
I3 : 0.0431387	I13 : 0.0120398	I23 : -0.5805
I4 : 0.0343063	I14 : 0.0112295	I24 : 2.94417
I5 : 0.0284684	I15 : 0.0105192	I25 : -14.6808
I6 : 0.0243249	I16 : 0.00990385	I26 : 73.4427
I7 : 0.0212326	I17 : 0.00930427	I27 : -367.176
I8 : 0.0188369	I18 : 0.00903421	I28 : 1835.92
I9 : 0.0169265	I19 : 0.00746051	
I10 : 0.0153676	I20 : 0.0126975	

We see that now we can go up to the 19th iteration with apparently good results. We see too that the increased precision has only masked a fundamental flaw of the algorithm itself. The manner the calculation is done produces a five fold increase in the error term at each iteration. This is an example of a numerically unstable algorithm.

9.9.1 Algebra doesn't work

Because of the inherent inexactness of floating-point representations and because of the many sources of rounding inaccuracies in a floating-point computation, it happens very often that values that should be equal from a purely algebraic perspective in fact rarely will be.

Consider this program:

```
#include <stdio.h>
int main(void)
{
  union {
    double x;
    int a[2];
  } u1, u2;

  u1.x = 1.2 - 0.1;
  u2.x = 1.1;

  if (u1.x == u2.x)
    printf("1.2 - 0.1 equals 1.1\n");
  else {
    printf("1.2 - 0.1 is NOT equal to 1.1.\n");
    printf("1.2 - 0.1 = %x%x\n1.1 =       %x%x\n",
                          u1.a[1], u1.a[0], u2.a[1], u2.a[0]);
  }
}
```

The output is:

```
1.2 - 0.1 is NOT equal to 1.1.
1.2 - 0.1 = 3ff1999999999999
1.1 =       3ff199999999999a
```

The last bit is different. This is because some quantities can't be represented exactly in binary notation, in the same way that $10/3$ can't be represented exactly in decimal notation and we obtain 3.33333333.

9.9.2 Underflow

When multiplying two very small quantities, you can exceed the precision available for floating point numbers, and obtain zero. This is an «underflow» and it can be an error in some situations. Take for instance a routine that calculates the hypotenuse of some triangle:

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

double hyp(double a,double b)
```

```
{
        return sqrt(a*a + b*b);
}

int main(void)
{
    double x,y,z;

  x = 0.72887E-20;
  y = 0.2554455E-20;

  z = hyp(x,y);
  printf("%20.15f",z);
}
```

This prints:

```
   0.000000000000000
```

The multiplication of `0.72887E-20` by itself loses all precision. The same happens for `0.2554455E-20`. We have to go into qfloat precision to obtain a result different than zero:

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

qfloat hyp(qfloat a,qfloat b)
{
        return sqrtl(a*a + b*b);
}

int main(void)
{
    qfloat x,y,z;

  x = 0.72887E-20L;
  y = 0.2554455E-20L;

  z = hyp(x,y);
  printf("%25.23qe",z);
}
```

This prints:

```
+0.00000000000000000000772336636687817626456877 99
```

Exercise:
Explain the following output:

```
#include <stdio.h>
int main(void)
{
   float a=0.7;
   if(0.7>a)
      printf("???????\n");
   else
     printf("OK\n");
}
```

Output:

```
???????
```

9.10 The math library

Lcc win provides a rich set of mathematical functions, besides the standard ones. Normally most of those are provided in two or three versions: an unsuffixed one in double precision, another with the `l` suffix in long double precision, and most of the time another one in qfloat precision.

Function	Description
acos	Inverse cosine Returns the angle whose cosine is x, in the range $[0, pi]$ radians. An error occurs if $1 < \|x\|$
airy	Solution of the differential equation $$y''(x) = xy \quad (9.5)$$
acosh	Inverse hyperbolic cosine. (C99) Calculates the hyperbolic arccosine of x, in the range $[0, \infty]$. An error occurs if $x < 1$.
asin	Inverse sine. Calculates the angle whose sine is x, in the range [-pi/2, +pi/2] radians. A domain error occurs if 1 < |x|.
asinh	Calculates the inverse hyperbolic sine
atan	One-parameter inverse tangent. Calculates the angle whose tangent is x, in the range $[-\pi/2, +\pi/2]$ radians.
atan2	Two-parameter inverse tangent. Calculates the angle whose tangent is y/x, in $[-\pi, +\pi]$ radians. An error occurs if both x and y are zero.
atanh	Inverse hyperbolic tangent
bernoulli	Returns the nth bernoulli number
besselJ	Calculates the bessel function of the first order.
besselY	Bessel function of the second kind, integer order
binomial	Calculates $\binom{n}{k}$
catalan	Calulates the first 32 catalan numbers

cbrt	cube root
ceil	ceiling, the smallest integer not less than parameter
copysign(x,y)	returns the value of x with the sign of y
cos	cosine
cosh	hyperbolic cosine
cyl_besselj	Cylindrical Bessel function $$J_v(Z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\Gamma(v+k+1)k!}\left(\frac{Z}{2}\right)^{2+k+v}$$
dbesi0 dbesi1	Computes the hyperbolic Bessel function of the first kind of order zero. Computes the hyperbolic Bessel function of the first kind of order one.
ellipticE	The ellipticE function compute the incomplete elliptic integral of the second kind $$E(x;k) = \int_0^x \frac{\sqrt{1-k^2t^2}}{\sqrt{1-t^2}}dt$$
ellipticF	Calculates the incomplete elliptic integral of the first kind $$F(z\|m) = \int_0^z \frac{1}{\sqrt{1-m\sin^2(t)}}\,dt\,/; 0 < m < 1$$
ellipticK	Calculates the complete elliptic integral of the first kind. $$K(z) = F\left(\frac{\pi}{2}\|z\right)$$
erf	error function $$erf(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$
erfc	complementary error function $$1 - erf(x)$$
eulerphi	Calculates Euler's Phi function
exp2(x)	raise 2 to the power of x
expint expintN	Calculates the exponential integral Calculates the exponential integral of N,x, N integer
expm1(x)	one less than the exponential of x, ex - 1

exp	exponential e^x
fabs	absolute value
fcmp	compares two floating point numbers with epsilon accuracy
fdim(x,y)	positive difference between x and y, fmax(x-y, 0)
floor	largest integer not greater than parameter
fibonacci	Returns the nth fibonacci number ($n < 400$)
fma(x,y,z)	multiply and add, (x * y) + z
fmax(x,y)	largest value of x and y
fmin(x,y)	smallest value of x and y
fmod	floating-point remainder: x - y*(int)(x/y)
fresnelC	FresnelC calculates the Fresnel integral for $C(z) = \int_0^z \cos\left(\frac{\pi t^2}{2}\right) dt$
fresnelS	FresnelC calculates the Fresnel integral for $S(z) = \int_0^z \sin\left(\frac{\pi t^2}{2}\right) dt$
fresnel	Calculates both fresnelC and fresnelS
frexp	break floating-point number down into mantissa and exponent
HarmonicNumber	returns the nth harmonic number
hypot(x,y)	hypotenuse $\sqrt{x^2 + y^2}$
hypergeom	The hypergeom functions compute the confluent hypergeometric function for ${}_1F_1(a;b;z) = 1 + \frac{a}{b}z + \frac{a(a+1)}{b(b+1)}\frac{z^2}{2} \cdots = \sum_{k=0}^{\infty} \frac{(a)_k}{(b)_k}\frac{z^k}{k!}$
j0	The j0 functions compute the Bessel functions of the first kind of x of order zero.
j1	The j0 functions compute the Bessel functions of the first kind of x of order zero.

jv	Bessel function of noninteger order. $$J_v(z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\Gamma(k+v+1)k!}\left(\frac{z}{2}\right)^{2k+v}$$
ibell	The bell number n is the number of partitions of a set of n different elements.
ibinomial	The ibinomial function computes the binomial coefficient C(n,k) using integer arithmetic only. The binomial function does the same but using floating point
igamma	The igamma functions compute the value of the incomplete gamma function at a, of x.
ilogb	the exponent of a floating-point value, converted to an int
lambertw	Calculates the Lambert W-function, also called the omega function
laguerre	The laguerre functions compute the value of the laguerre polynomial at z and lambda. $$L_n^{\lambda} = \frac{\Gamma(n+\lambda+1)}{n!}\sum_{k=0}^{n}\frac{(-n)_k z^k}{\Gamma(k+\lambda+1)k!}$$
lerchphi	The Lerch's transcendent is defined by the following infinite series: $$\phi(z,s,w) = \sum_{n=0}^{\infty}\frac{z^n}{(n+v)^s}|z| < 1, v \neq 0, -1\cdots$$
legendre	The legendre functions compute the legendre polynomial <n> at point <x>
ldexp	scale floating-point number by exponent
lgamma	natural log of the absolute value of the gamma function
log	natural logarithm
log10	base-10 logarithm
log1p(x)	natural logarithm of 1 + x
log2	base-2 logarithm
logb	extract exponent from floating-point number
llrint	round to integer (returns long long) using current rounding mode
lrint	round to integer (returns long) using current rounding mode
llround	round to integer (returns long long)
lround	round to integer (returns long)
modf(x,p)	returns fractional part of x and stores integral part where pointer p points to
nan(s)	returns NaN, possibly using string argument

nearbyint	round floating-point number to nearest integer
nextafter(x,y)	returns next representable value after x (towards y)
nexttoward(x,y)	same as nextafter, except y is always a long double
pochhammer	The value of the pochhammer symbol. $(a)_n = \frac{\Gamma(a+n)}{\Gamma(a)} /; (\neg(-a \epsilon Z \wedge -a \geq 0 \wedge n \epsilon Z \wedge n \leq -a))$
polylog	The polylogarithm of order n is defined by the series: $Li_n(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^n}$
polyeval	Evaluates a polynomial at a given point
polyroots	Finds the roots of a polynomial
pow(x,y)	raise x to the power of y, xy
psi	The psi function is defined by $\psi(z) = \frac{d}{dz} \log \Gamma(z) = \frac{\Gamma\prime(z)}{\Gamma(z)}$
remainder(x,y)	calculates remainder, as required by IEC 60559
remquo(x,y,p)	same as remainder, but store quotient (as int) at target of pointer p
rint	round to integer (returns double) using current rounding mode
round	round to integer (returns double), rounding halfway cases away from zero
scalbln(x,n)	x * `FLT_RADIXn` (n is long)
scalbn(x,n)	x * `FLT_RADIXn` (n is int)
sin	sine
sincos	Calculates the sinus and cosinus of the given argument
sinh	hyperbolic sine
sqrt	square root
stirling1	Stirling numbers of the first kind
stirling2	Stirling numbers of the second kind
struve	Computes the Struve function Hv(z) of order v, argument z. Negative z is rejected unless v is an integer. $H_v(z) = \left(\frac{z}{2}\right)^{v+1} \sum_{k=0}^{\infty} \frac{(-1)^k}{\Gamma(k+\frac{3}{2})\Gamma(k+v+\frac{3}{2})} \left(\frac{z}{2}\right)^{2k}$
tan	tangent
tanh	hyperbolic tangent
tgamma	gamma function

trunc	truncate floating-point number
y0	The y0 functions compute the Bessel functions of the second kind of x of order zero
y1	The y1 functions compute the Bessel functions of the second kind of x of order one.
yn	The yn functions compute the Bessel functions of the second kind of x of order n.
zetac	Calculates Rieman's Zeta function - 1

10 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:

- 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.

- 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.

- The heap. Here is the space that we obtain with malloc or equivalent routines. This is 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. the run time system 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 operating 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.

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-win. Lcc-win 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 allocate memory with the `GC_malloc` function instead of allocating it with malloc. This function's result type and type of arguments is the same as [textttmalloc, so by just replacing all malloc calls with `GC_malloc` in your program you can benefit of the automatic memory manager without writing any new line of code.

The memory manager works by inspecting regularly your whole heap and stack address space, and checking if there is anywhere a reference to the memory it manages. If it doesn't find any references to a piece of memory it will mark that memory as free and recycle it. It is a very simple schema, taken to almost perfection by several years of work from the part of the authors. To use the memory manager you should add the gc.lib library to your link statement or indicate that library in the IDE in the linker configuration tab.

10.1 Functions for memory management

- `malloc` Returns a pointer to a newly allocated memory block or NULL if there is not enough memory to satisfy the request. This is a standard function.
- `free` Releases a memory block. This is a standard function.
- `calloc` Returns a pointer to a newly allocated zero-filled memory block or NULL if there is not enough memory to satisfy the request. This is a standard function.
- `realloc` Resizes a memory block preserving its contents. Returns NULL if there isn't enough memory to satisfy the request. This is a standard function.
- `alloca` Allocate a memory block in the stack that is automatically destroyed when the function where the allocation is requested exits. In some other compilers this function may be absent.
- `GC_malloc` Allocates a memory block managed by the memory manager. In some other compilers this function may not be present or if present it may be called differently.
- `GC_realloc` Like the realloc function above, but for GC managed memory.

10.2 Memory management strategies

Each program needs some workspace to work in. How this space is managed (allocated, recycled, verified) makes a memory allocation strategy. Here is a short description of some of the most popular ones.

10.2.1 Static buffers

This is the simplest strategy. You reserve a fixed memory area (buffer) at compile time, and you use that space and not a byte more during the run time of the program.

Advantages:

- It is the fastest possible memory management method since it has no run-time overhead. There is no memory allocation, nor recycling that incurs in run time costs.
- In memory starved systems (embedded systems, micro controller applications, etc) it is good to know that there is no possibility of memory fragmentation or other memory space costs associated with dynamic allocation.

Drawbacks:

- Since the amount of memory allocated to the program is fixed, it is not possible to adapt memory consumption to the actual needs of the program. The static buffers could be either over-dimensioned, wasting memory space, or not enough to hold the data needed. Since the static buffers must be patterned after the biggest possible input, they will be over-dimensioned for the average case.

- Unless programming is adapted to this strategy, it is difficult to reuse memory being used in different buffers to make space for a temporary surge in the space needs of the program.

10.3 Stack based allocation

The C standard allows for this when you write:

```
int fn(int a)
{
        char workspace[10000];
        ...
}
```

In this case, the compiler generates code that allocates 10000 bytes of storage from the stack. This is a refinement of the static buffers strategy. Under the windows operating system, the stack is 1MB in normal programs but this can be increased with a special linker option.

A variant of this strategy allows for dynamic allocation. Instead of allocating a memory block of size “siz with malloc, we can write:

```
char workspace[siz];
```

and the compiler will generate code that allocates “siz” bytes from the program stack.

Advantages:

- Very fast allocation and deallocation. To allocate a memory block only a few assembly instructions are needed. Deallocation is done without any extra cost when the function where the variables are located exits.

Drawbacks:

- There is no way to know if the allocation fails. If the stack has reached its maximum size, the application will catastrophically fail with a stack overflow exception.

- There is no way to pass this memory block to a calling function. Only functions called by the current function can see the memory allocated with this method.

- Even if the C99 standard is already several years old, some compilers do not implement this. Microsoft compilers, for instance, do not allow this type of allocation. A work-around is to use the `_alloca` function. Instead of the code above you would write:

```
char *workspace = _alloca(siz);
```

10.3.1 "Arena" based allocation

This strategy is adapted when a lot of allocations are done in a particular sequence of the program, allocations that can be released in a single block after the phase of the program where they were done finishes. The program allocates a large amount of memory called "arena", and sub-allocates it to the consuming routines needing memory. When a certain phase of the program is reached the whole chunk of memory is either marked as free or released to the operating system.

The windows operating system provides support for this strategy with the APIs CreateHeap, HeapAlloc, and others.

This strategy is adapted when a lot of allocations are done in a particular sequence of the program, allocations that can be released in a single block after the phase of the program where they were done finishes. The program allocates a large amount of memory called "arena", and sub-allocates it to the consuming routines needing memory. When a certain phase of the program is reached the whole chunk of memory is either marked as free or released to the operating system.

The windows operating system provides support for this strategy with the APIs CreateHeap, HeapAlloc, and others.

Advantages:

- Fewer calls to memory allocation/deallocation routines.
- No global fragmentation of memory.

Drawbacks:

- Since the size of the memory that will be needed is not known in advance, once an arena is full, the strategy fails or needs to be complemented with more sophisticated variations. A common solution is to make the arena a linked list of blocks, what needs a small processing overhead.
- Determining when the moment has come to release all memory is tricky unless the data processed by the program has a logical structure that adapts itself to this strategy. Since there is no way of preserving data beyond the frontier where it is released, data that is to be preserved must be copied into another location.

10.4 The malloc / free strategy

This is the strategy that is most widely used in the C language. The standard provides the functions malloc, a function that returns a pointer to an available memory block, and free, a function that returns the block to the memory pool or to the operating system. The program allocates memory as needed, keeping track of each memory block, and freeing it when no longer needed. The free function needs a pointer to the same exact location that was returned by malloc. If the pointer was incremented or decremented, and it is passed to the free function havoc ensues.

Advantages:

- It is very flexible, since the program can allocate as needed, without being imposed any other limit besides the normal limit of available memory.
- It is economic since the program doesn't grab any more memory than it actually needs.
- It is portable since it is based in functions required by the C language.

Drawbacks:

- It is very error prone. Any error will provoke obscure and difficult to track bugs that need advanced programming skills to find. And the possibilities of errors are numerous: freeing twice a memory block, passing a wrong pointer to free, forgetting to free a block, etc.
- The time used by memory allocation functions can grow to an important percentage of the total run time of the application. The complexity of the application increases with all the code needed to keep track and free the memory blocks.
- This strategy suffers from the memory fragmentation problem. After many malloc/free cycles, the memory space can be littered with many small blocks of memory, and when a request for a big block of memory arrives, the malloc system fails even if there is enough free memory to satisfy the request. Since it is impossible for the malloc system to move memory blocks around, no memory consolidation can be done.
- Another problem is aliasing, i.e. when several pointers point to the same object. It is the responsibility of the programmer to invalidate all pointers to an object that has been freed, but this can be very difficult to do in practice. If any pointer to a freed object remains in some data structure, the next time it will be used the program can catastrophically fail or return invalid results, depending on whether the block was reallocated or not.
- It can be slow. Malloc/free was a big bottleneck for performance using the Microsoft C runtime provided by the windows system for windows 95/98, for instance.

10.5 The malloc with no free strategy

This strategy uses only malloc, never freeing any memory. It is adapted to transient programs, i.e. programs that do a well defined task and then exit. It relies on the operating system to reclaim the memory used by the program.

Advantages:

- Simplified programming, since all the code needed to keep track of memory blocks disappears.
- It is fast since expensive calls to free are avoided.

Drawbacks:

- The program could use more memory than strictly needed.
- It is very difficult to incorporate software using this strategy into another program, i.e. to reuse it. This strategy can be easily converted into an arena based strategy though, since only a call to free the arena used by the program would be needed. It is even easier to convert it to a garbage collector based memory management. Just replace `malloc` by `GC_malloc` and you are done.

10.6 Automatic freeing (garbage collection).

This strategy relies upon a collector, i.e. a program that scans the stack and the global area of the application looking for pointers to its buffers. All the memory blocks that have a pointer to them, or to an inner portion of them, are marked as used, the others are considered free.

This strategy combines easy of use and reclaiming of memory in a winning combination for most applications, and it is the recommended strategy for people that do not feel like messing around in the debugger to track memory accounting bugs.

Advantages:

- Program logic is simplified and freed from the chores of keeping track of memory blocks.
- The program uses no more memory than needed since blocks no longer in use are recycled.

Drawbacks:

- It requires strict alignment of pointers in addresses multiple of four. Normally, this is ensured by the compiler, but under certain packing conditions (compilation option -Zp1) the following layout could be disastrous:

```
#pragma pack(1)
struct {
```

```
        short a;
        char *ptr;
} s;
```

The pointer "ptr" will NOT be aligned in a memory address multiple of four, and it will not be seen by the collector because the alignment directive instructs the compiler to pack structure members.

- You are supposed to store the pointers in memory accessible to the collector. If you store pointers to memory allocated by the collector in files, for instance, or in the "windows extra bytes" structure maintained by the OS, the collector will not see them and it will consider the memory they point to as free, releasing them again to the application when new requests are done.

- Whenever a full gc is done (a full scan of the stack and the heap), a noticeable stop in program activity can be perceived by the user. In normal applications this can take a bit less than a second in large memory pools. The collector tries to improve this by doing small partial collections each time a call to its allocator function is done.

- If you have only one reference to a block, the block will be retained. If you have stored somewhere a pointer to a block no longer needed, it can be very difficult indeed to find it.

- The garbage collector of lcc-win is a conservative one, i.e. if something in the stack looks like a pointer, it will be assumed that this is a pointer (fail-safe) and the memory block referenced will be retained. This means that if by chance you are working with numeric data that contains numbers that can be interpreted as valid memory addresses more memory will be retained than strictly necessary. The collector provides special APIs for allocating tables that contain no pointers and whose contents will be ignored by the collector. Use them to avoid this problems.

10.7 Mixed strategies

Obviously you can use any combination of this methods in your programs. But some methods do not mix well. For instance combining malloc/free with automatic garbage collection exposes you to more errors than using only one strategy. If you pass to free a pointer allocated with `GC_malloc` chaos will reign in your memory areas. To the contrary, the stack allocation strategy can be combined very well with all other strategies since it is specially adapted to the allocation of small buffers that make for many of the calls to the allocator functions.

10.8 A debugging implementation of malloc

Instead of using directly malloc/free here are two implementations of equivalent functions with some added safety features:

- Freeing NULL is allowed and is not an error, the same behavior of the standard "free" function.
- Double freeing is made impossible.
- Any overwrite immediately at the end of the block is checked for.
- Memory is initialized to zero.
- A count of allocated memory is kept in a global variable.

```
#include <stdlib.h>
#define MAGIC 0xFFFF
#define SIGNATURE 12345678L
/ This global variable contains the number of bytes
// allocated so far.
size_t AllocatedMemory;
// Allocation function
void *allocate(size_t size)
{
    char *r;
    int *ip = NULL;
    size += 3 * sizeof(int);
    r = malloc(size);
    if (r == NULL)
        return r;
    AllocatedMemory += size;
    ip = (int *) r;
    // At the start of the block we write the signature
    *ip++ = SIGNATURE;
    // Then we write the size of the block in bytes
    *ip++ = (int) size;
    // We zero the data space
    memset(ip, 0, size - 3*sizeof(int));
    // We write the magic number at the end of the block,
    // just behind the data
    ip = (int *) (&r[size - sizeof(int)]);
    *ip = MAGIC;
    // Return a pointer to the start of the data area
    return (r + 2 * sizeof(int));
}

// Freeing the allocated block
void release(void *pp)
{
    int *ip = NULL;
    int s;
    register char *p = pp;
```

```
    if (p == NULL) // Freeing NULL is allowed
        return;
    // The start of the block is two integers before the data.
    p -= 2 * sizeof(int);
    ip = (int *) p;
    if (*ip == SIGNATURE) {
        // Overwrite the signature so that this block can't be
        // freed again
        *ip++ = 0;
        s = *ip;
        ip = (int *) (&p[s - sizeof(int)]);
        if (*ip != MAGIC) {
            ErrorPrintf("Overwritten block size %d", s);
            return;
        }
       *ip = 0;
        AllocatedMemory -= s;
        free(p);
    }
    else {
        /* The block has been overwritten. Complain. */
        ErrorPrintf("Wrong block passed to release");
    }
}
```

The allocate function adds to the requested size space for 3 integers.

1. The first is a magic number (a signature) that allows the identification of this block as a block allocated by our allocation system.

2. The second is the size of the block. After this two numbers, the data follows.

3. The data is followed by a third number that is placed at the end of the block. Any memory overwrite of any block will overwrite probably this number first. Since the "release" function check for this, we will be able to detect when a block has been overwritten.

At any time, the user can ask for the size of total allocated memory (valid blocks in circulation) by querying the AllocatedMemory variable.

The "release function" accepts NULL (that is ignored). If the pointer passed to it is not NULL, it will check that it is a valid block, and that the signature is still there, i.e. that no memory overwrites have happened during the usage of the block.

The global variable AllocatedMemory contains the number of bytes allocated so far. This is useful for detecting memory leaks. Suppose you want to make sure a new operation you have just added to your software doesn't leak any memory. You just do:

```
int mem = AllocatedMemory;
result = newOperation(parm1,parm2);
```

```
if (mem != AllocatedMemory {
        // Here we have detected a memory leak.
}
```

10.8.1 Improving allocate/release

Of course with this simple code we have just scratched the surface of this problem. Several issues are more or less immediately visible.

- We have assumed in our code that there are no alignment problems, i.e. that we can access an integer at any memory address. This is not true for many processor, that have strict alignment requirements and need to be feeded aligned integers, with catastrophic failures if they are not. We need to align the start pointer that we return to the user of our allocation functions, and we need to align the MAGIC number at the end of the block. One portable way to do this is to make a "alignment union" like this that gives us the alignment requirement: Of course with this simple code we have just scratched the surface of this problem. Several issues are more or less immediately visible.

  ```
  union __altypes {
          char c;
          short s;
          int i;
          long long ll;
          long double ld;
  }       AllTypes;
  ```

- It may be a better idea to destroy all data in the released block, to avoid it being used by mistake later in the program. For instance if we set all data to zero, any usage of pointers within the released block would trap.

- We could store all allocated block addresses somewhere and make a "heap check" function that would verify all allocated blocks without waiting till they are released

- Other debugging information could be gathered, for instance the line and file name, the calling function, a time stamp, etc. This information could be helpful in investigating leaks and other problems.

All these issues can be added to this base implementation, but it would be beyond the scope of a tutorial to do it for you. Start with this ideas and build further.

11 The libraries of lcc-win

Lcc-win comes with many useful libraries, already compiled and ready to use. You should include the header file that describes the interface of the library, then include in the link command the binary library that contains the compiled code of the functions in the library.

Name	#include	Binary lib file
Containers	containers.h	libc.lib
Regular expressions	regexp.h	regexp.lib
Perl regular expressions	pcre.h	pcre.lib
Console routines	tcconio.h	tcconio.lib
Statistics	stats.h	stats.lib
SQL library	sqlite3.h	sqlite3.lib
Linear algebra	matrix.h	matrix.lib
Network utilities	netutils.h	netutils.lib
Advanced math	specialfns.h	libc.lib
Safer C library	—	libc.lib
Zlib functions	zlib.h	zlib.lib

- The regular expressions libraries The C language started under the Unix operating system, and that system provided since quite a long time a regular expressions library. For reasons unknown to me that library wasn't included in the language. Lcc-win provides it with the implementation proposed by Henry Spencer, from the university of Toronto.

  Another regular expression package is the one compatible with the PERL language. This library was written by Philippe Hazel from the University of Cambridge.

- Console formatting routines This library was developed by Daniel Guerrero Miralles and emulates the old Turbo C compiler functions for clearing the screen, positioning the cursor at some line and column in the text screen, etc. You can also change the color of the text and modify the background.

- Statistics library This library provides a very complete set of statistics functions, from the simple ones to the more complex ones. Most of it uses the functions provided by the CEPHES mathematical library, but adds a lot of other functions to it. It follows for the naming of the functions the proposal of Paul Bristow for the C++ language.

- Linear algebra library This library is based on the MESCHACH linear algebra package. It helps you solve linear systems, and many other things. It accepts also sparse matrices and complex matrices.

- Network utilities This is a small library but very useful. It has functions like "GetHttpUrl" that allow you to get any file from the internet with a minimum effort. Other things like "ping" and client/server protocols are there. See the "Network" chapter at the end of this tutorial for more information.

- Advanced math functions This library has been incorporated into the standard C library "libc.lib". They are principally the elliptic functions, Bessel functions, psi, Struve function, and others. Functions to evaluate a polynomial or find the roots of a polynomial have been added recently.

- Compression/decompression functions This is the "zlib" library, that allows to compress and decompress data to save space. It is written by Jean Loup Gailly and Mark Adler.

- Structured Query Language (SQL) This is a database library based on the public domain "sqlite" package. It is a very complete implementation of SQL, with a good interface to the C language. It comes with a documentation adapted to the windows environment, accessible through the lcc-win IDE.

- Safer C Library Microsoft proposed to replace many of the standard C functions with this library, and presented a proposal to the C standards committee.Even if it is not complete and the design is flawed, it is a step in the right direction.

We show here an example of a library usage, consult the documentation for a full description.

11.1 The regular expressions library. A "grep" clone.

A regular expression is a string that describes or matches a set of strings, according to certain syntax rules. Regular expressions are used by many text editors and utilities to search and manipulate bodies of text based on certain patterns.

The task at hand is then, to build a program that will accept a regular expression pattern, and a set of files in the form of a specification like "*.c" and will print in the standard output all lines that match the regular expression, with their line number.

We have then, two tasks:

1. The first one is to find all files that match a (possibly) ambiguous file specification like, for instance "*.c"

2. The second is to apply a regular expression to each line of the set of files. A regular expression is a program that will be interpreted by a special interpreter at run time. We have first to compile it, then we apply it to each line.

Here is then, a first approximation of "grep".

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

```
#include <string.h>
#include <direct.h>
#include <shlwapi.h>
int main(int argc,char *argv[])
{
    regexp *expression;
    struct _finddata_t fd;
    long h;
    int matches;
    char *p;
    unsigned char path[MAX_PATH*2];
    unsigned char currentdir[MAX_PATH];
    unsigned char Path[2*MAX_PATH];

    if (argc < 3) {                          //(1)
         fprintf(stderr,"Usage: grep <expression> <files>\n");
         return(-1);
    }
    expression = regcomp(argv[1]);           //(2)
    if (expression == NULL) {
        fprintf(stderr,"Incorrect reg expression %s\n",argv[1]);
        return 1;
    }
    if (argv[2][0] != '\\' && argv[2][1] != ':') {
        //(3)
        getcwd(currentdir,sizeof(currentdir)-1);
        strcpy(path,currentdir);
        strcat(path,"\\");
        strcat(path,argv[2]);
        p = strrchr(path,'\\');
        if (p)
           *p=0;
        PathCanonicalize(Path,path);
        p = strrchr(argv[2],'\\');
        if (p == NULL) {
            strcat(Path,"\\");
            p = argv[2];
        }
        strcat(Path,p);
    }
    else strcpy(Path,argv[2]);               //(3)
    h = _findfirst(Path,&fd);                //(4)
    if (h == -1) {
        fprintf(stderr,"No files match %s\n",Path);
        return -1;
    }
    matches = 0;
```

```
    do {
        p = strrchr(Path,'\\');
        if (p)
            *++p=0;
            strcat(Path,fd.name);
            matches += processfile(expression,Path);
    } while (_findnext(h,&fd) == 0);
    _findclose(h);
    if (matches)
       printf("%d matches\n",matches);
    return matches;
}
```

1. We do not do an extensive error checking in this function, to keep things simple, but a minimum should be done. We test the format of the arguments, since it is pointless to go on if this data is absent.

2. It is more efficient to compile once for all the regular expression into its internal form. If the compilation fails, this means that the regular expression was incorrect and we exit with an error.

3. Now we start a quite messy path handling stuff. We test first if the path we have received is an absolute path, i.e. a path like `c:\mydir\myprograms\...` or a path that starts at the root of the current drive like `\mydir\myprograms\...`. There is a function for this in the Windows API (PathIsRoot), but I have seen it fail for mysterious reasons, so we use a rather crude algorithm: if the second character of the path is a ':' or the first character is a '
' we have an absolute path. Note that there can't be any buffer overflow here since if we have a path of only one character, the second one will be the terminating zero of the string, that will be always there.

4. If it is not an absolute path, we have to handle the cases where we have a path in the form of `..\..\foo\*.c`. We find the current directory, using the getcwd function, and we add to it the path part of our specification, for example if we had `..\*.c` we would add to it `..\`. Then we pass this path to the PathCanonicalize function, that will resolve the embedded .. in any order they appear.

   If the path was an absolute path, we assume there are no embedded .. in it, and we just copy it to the buffer that receives the path.Of course, in a more advanced and robust program you should check for that possibility, and adding a call to PathCanonicalize is not really a big deal.

5. At last we arrive to the second part of our task. We have a (possibly) ambiguous specification like "*.c" and we want to ask the system to find out all files that match. We do that by calling the function `_findfirst` that returns either a valid "search handle", or -1 if it could not find any file that matches the specification.

6. If we found at least one file, we start looping calling `_findnext` to get the next file to process. We give to the function that will process the file two arguments: the expression that we compiled just before, and the file name to process. That function returns the number of matches, that we add to our count.

When there are no more matches we are done, we close the search handle, and exit returning the number of matches. Now, having setup the framework, here is the function that process each file.

```
static int processfile(regexp *expression,char *filename)
{
    FILE *f = fopen(filename,"r");
    int namewritten = 0;
    int matches = 0;
    int line = 1;
    char buf[8192];

    if (f == NULL)
        return 0;
    while (fgets(buf,sizeof(buf),f)) {
        if (regexec(expression,buf)) {
            matches++;
            if (!namewritten) {
                printf("%s:\n",filename);
                namewritten = 1;
            }
            printf("[%4d] %s",line,buf);
        }
        line++;
    }
    fclose(f);
    return matches;
}
```

No secrets here. We open the file, and read each line of it using fgets. To each line we apply our regular expression, and if there is a match we print that line. We print the file name before printing any matches to separate the matches from one file from the matches of the next file in the output. We always write the line number, and we loop until there are no more lines.

Note that the test for failure to open the file is maybe unnecessary because we are certain that the file exists, since the name is the result of a `_findfirst` or a `_findnext`. It could be however, that the file exists but it is not accessible, because it is being used by another program in exclusive mode, or that the access rights are such that we can't read it. It is better to avoid crashes so a simple test doesn't make our program much fatter.

11.2 Using qfloats: Some examples

Qfloats do not need any special initialization to work. You can use them anywhere you would use a double or a long double. You should include their header file

```
#include <qfloat.h>
```

before you use them. To print qfloats using printf you should add the qualifier 'q' to the `%f`, `%g` or `%e` format specifications. You specify the width of the field and the precision in the same way as you specify the width and precision of the double type. For instance: `%110.105qe` would specify a qfloat of field with 110 and 105 numbers precision. Here is an example of the factorial function using qfloats:

```
#include <qfloat.h>
#include <stdio.h>
#include <stdlib.h>
qfloat factorial(int n)
{
        qfloat r = 1.0;
        qfloat i = 1.0;

        while (n > 0) {
                r = r*i;
                i++;
                n--;
        }
        return r;
}

int main(int argc,char *argv[])
{
        if (argc < 2) {
                printf("Usage: factorial <number>\n");
                return 0;
        }
        int n = atoi(argv[1]);
        printf("Factorial %d =\n %110.105qe\n",n,factorial(n));
}
```

11.3 Using bignums: some examples

For using bignums the usage is slightly more complicated than qfloats. Bignums require also a header file:

```
#include <bignums.h>
```

and requires that before using the bignums library you specify the precision. Note that the usage of bignums is slightly more complicated since there is less compiler integration than with qfloats.

```
#include <bignums.h>
#include <stdio.h>
#include <stdlib.h>
pBignum factorial(int n)
{
        pBignum r = newBignum(1);
        pBignum i = newBignum(1);

        while (n > 0) {
                r = r*i;
                i = i+1;
                n--;
        }
        return r;
}

int main(int argc,char *argv[])
{
        pBignum b;
        char buf[8192];
        if (argc < 2) {
                printf("Usage: factorial <number>\n");
                return 0;
        }
        int n = atoi(argv[1]);
        BignumPrecision(500); // Initialization
        b = factorial(n);
        quadformat(b,buf);
        puts(buf);
}
```

The initialization function needs as input the number of 32 bit integers to use to store the numbers. Here we use 500 * 32 –> 16 000 bits.

12 Pitfalls of the C language

Look. I am not a religious person. C is not a religion for me, and this means that I see some of the pitfalls of the language. I write this so that you see them at the start, and they do not bite you.

12.1 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!

12.2 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 6, what is different from zero. Some compilers will emit a warning whenever an assignment in a conditional expression is detected.

12.3 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, the IDE of lcc-win, helps you avoid this by coloring commentaries in another color as normal program text.

12.4 Easily changed block scope.

Suppose you write the following code:

```
if  (someCondition)
        fn1();
else
        OtherFn();
```

Trying to debug your program, you add a printf statement line this:

```
if  (someCondition)
        fn1();
else
        printf(‘‘Calling OtherFn\n’’);
        OtherFn();
```

The else is not enclosed in curly braces, so only one statement will be executed. The end result is that the call to `OtherFn` is always executed, no matter what. Golden rule: ALWAYS watch out for scopes of “if” or “else” not between curly braces when adding code.

12.5 Using increment or decrement more than once in an expression.

The ANSI C standard 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. This one is invalid too:

```
i = i+++++i;
```

(in clear `i++ + ++i`)

12.6 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. Another operator precedence error is the following:

```
#include "stdio.h"

int main(void)
{
        int a,b;
        a= 10;
        a>10?b=20:b=30; // Error in this line
        printf("%d",b);
}
```

This will provoke an error. If we rewrite the above line like this:

```
a>10?b=20:(b=30);
```

the error disappears. Why?

The explanation is that the first line is parsed as:

```
((a>10)?(b=20):b)=30;
```

because of the established precedence of the operators. The assignment operator is one of the weakest binding operators in C, i.e. its precedence is very low. The correct way to write the expression above is:

```
b = (a<10)? 20 : 10;
```

12.7 Extra Semi-colon in Macros

The next code fragment illustrates a common error when using the preprocessor to define constants:

```
#define MAXVAL 10; // note the semicolon at the end
/* ... */
if( value >= MAXVAL ) break;
```

The compiler will report an error. The problem is easily spotted when the macro substitution is performed on the above line. Using the definition for MAXVAL, the substituted version reads,

```
if( value >= 10; ) break;
```

The semi-colon (`;`) in the definition was not treated as an end-of-statement indicator as expected, but was included in the definition of the macro `MAXVAL`. The substitution then results in a semi-colon being placed in the middle of the controlling expression, which yields the syntax error. Remember: the pre-processor does only a textual substitution of macros.

12.8 Watch those semicolons!

Yes, speaking about semicolons, look at this:

```
if (x[j] > 25);
        x[j] = 25;
```

The semicolon after the condition of the if statement is considered an empty statement. It changes the whole meaning of the code to this:

```
if (x[j] > 25) { }
x[j] = 25;
```

The `x[j] = 25` statement will be always executed.

12.9 Assuming pointer size is equal to integer size

Today under lcc-win 32 bits the sizeof(void *) is equal to the sizeof(int). This is a situation that changes when we start using the 64 bit machines and the 64 bit version of lcc-win where int can be 32 bits but pointers would be 64 bits. This assumption is deeply rooted in many places even under the windows API, and it will cause problems in the future. Never assume that a pointer is going to fit in an integer, if possible.

12.10 Careful with unsigned numbers

Consider this loop:

```
int i;
for (i = 5; i >= 0; i --) {
        printf("i = %d\n", i);
}
```

This will terminate after 6 iterations. This loop however, will never terminate:

```
unsigned int i;
for (i = 5; i >= 0; i --) {
        printf("i = %d\n", i);
}
```

The loop variable i will decrease to zero, but then the decrement operation will produce the unsigned number 4294967296 that is bigger than zero. The loop goes on forever.Note too that the common windows type DWORD is unsigned!

12.11 Changing constant strings

Constant strings are the literal strings that you write in your program. For instance, you write:

```
        outScreen("Please enter your name");
```

This constant string "Please enter your name" is stored (under lcc-win) in the data section of your program and can be theoretically modified. For instance suppose that the routine "outScreen" adds a `\r\n` to its input argument. This will be in almost all cases a serious problem since:

1. The compiler stores identical strings into the same place. For instance if you write

   ```
   a = "This is a string";
   b = "This is a string";
   ```

   there will be only one string in the program. The compiler will store them under the same address, and if you modify one, the others will be automatically modified too since they are all the same string.

2. If you add characters to the string (with strcat for instance) you will destroy the other strings or other data that lies beyond the end of the string you are modifying.

3. Some other compilers like gcc will store those strings in read memory marked as read only, what will lead to an exception if you try to modify this. Lcc-win doesn't do this for different reasons, but even if you do not get a trap it is a bad practice that should be avoided.

A common beginner error is:

```
char *a = "hello";
char *b = "world";
strcat(a,b);
```

In this case you are adding at the end of the space reserved for the character array "hello" another character string, destroying whatever was stored after the "a" character array.

12.12 Indefinite order of evaluation

Consider this code:

```
fn(pointer->member, pointer = &buffer[0]);
```

This will work for some compilers (gcc, lcc-win) and not for others. The order of evaluation of arguments in a function call is undefined. Keep this in mind. If you use a compiler that will evaluate the arguments from left to right you will get a trap or a nonsense result.

12.13 A local variable shadows a global one

Suppose you have a global variable, say "buf". If you declare a local variable of the same name at an inner scope, the local variable will take precedence over the global one, i.e. when you write:

```
unsigned char buf[BUFSIZ];
int fn(int a)
{
        char buf[3];
        ...
        buf[BUFSIZ-1] = 0;// Error! the local variable
                          // is accessed, not the global one
}
```

Giving the command line option "-shadows" to the compiler will generate a warning when this happens.

12.14 Careful with integer wraparound

Consider this code:

```
bool func(size_t cbSize) {
   if (cbSize < 1024) {
      // we never deal with a string trailing null
      char *buf = malloc(cbSize-1);
      memset(buf,0,cbSize-1);

      // do stuff

      free(buf);

      return true;
   } else {
      return false;
   }
}
```

Everything looks normal and perfect in the best of all worlds here. We test if the size if smaller than a specified limit, and we then allocate the new string. But... what happens if cbSize is zero???

Our call to malloc will ask for 0-1 bytes, and using 32 bit arithmetic we will get an integer wrap-around to 0xffffffff, or -1. We are asking then for a string of 4GB. The program has died in the spot.

12.15 Problems with integer casting

In general, the compiler tries to preserve the value, but casting from signed to unsigned types can produce unexpected results. Look, for instance, at this sequence:

```
char c = 0x80; //-128
//now we cast it to short
short s = (short)c;    //now s = 0xff80 still -128
//us = 0xff80, which is 65408!
unsigned short us = (unsigned short)s;
```

In general you should not try to mix signed/unsigned types in casts. Casts can occur automatically as a result of the operations performed. The operators + (addition), ~ (bitwise negation) - (minus) will cast any type shorter than an int into a signed int. If the type is larger than an integer, it will be left unchanged.

12.16 Octal numbers

Remember that numbers beginning with zero are treated as number written in base 8. The number 012 is decimal 10, not 12. This error is difficult to find because everything will compile without any error, after all, this is a legal construct.

12.17 Wrong assumptions with realloc

Consider this code:

```
if (req > len)
        realloc(p,len);
memcpy(p,mem,len);
```

This code assumes that the block p will be reallocated to the same address, what can be false. The correct way of doing this reallocation is:

```
if (req > len)
        p = realloc(p,len);
memcpy(p,mem,len);
```

This is still wrong since it assumes that realloc always returns the desired memory block. But in reality realloc could very well fail, and that should be tested:

```
if (req > len) {
        void *tmp = realloc(p,len);
        if (tmp)
                p = tmp;
        else
                fatal_error("No more memory\n");
}
memcpy(p,mem,len);
```

We suppose that the function `fatal_error` never returns.

12.18 Be careful with integer overflow

12.18.1 Overflow in calloc

The calloc function multiplies implicitely the size argument and the number of items argumentsb to yield the total number of bytes needed. Here, an overflow is possible, and not all implementations consider this possibility. Consider this code:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE_MAX 0xffffffff
int main()
{
    size_t s, n;
    void *p;

    s = 12;
    n = (SIZE_MAX / s) + 1;
    p = malloc(s * n);
    printf("s=%lu,n=%lu s*n=%lu %p %p\n", s, n, n * s, p);
}
```

We use malloc() to allocate 12 pieces of RAM, 357 913 942 bytes each. Obviously, the allocation should fail, since it is much more than windows can allocate. When we run this program however, we see the following output:

```
s=12 n=357913942 s*n=8 0x002d58c8
```

We see that s*n is not 4 294 967 304 but 8 !!!! We have passed to malloc the number 8 instead of 4GB. Imagine what will happen when we try to access that pointer that points to a memory area of just 8 bytes...

To avoid this kind of bad surprises it is better to pass the s and n variables to calloc instead, that will take care of overflow.[1]

12.19 The abs macro can yield a negative number.

Within a 32 bit system, -(-2147483548) yields the same number! This is true for the labs and the llabs functions with different numbers. In fact, this will happen with all numbers whose hexadecimal representation is 0x8000...

The addition operation in the CPU doesn't really care as to whether an integer is signed or unsigned. For 32-bit signed integers, the minimum value is `0x80000000` in decimal `-2147483648` and the maximum value is `0x7fffffff` decimal `2147483647`. Note that there is no value than can hold `2147483648`, so if you negate `(int) 0x80000000`, you get `(int)0x80000000` again. That is something to look out for, because it means `abs()` returns a negative value when fed `-2147483648`.

[1] Note that the versions of lcc-win that relied on the system supplied calloc will fail, they do not test for overflow. I rewrote that function on Apr 18th, 2006 and lcc-win does not use the system provided calloc any more.

12.20 Adding two positive numbers might make the result smaller.

If you add a 32 bit number to another 32 bit number, only the lower 32 bits of the 33 bit result will be written to the destination. The 33rd bit is the carry flag, that can't be accessed in standard C. Lcc-win offer you the `_overflow()` pseudo function that allows you to access this flag, but other compilers are less helpful. Replacing this pseudo function in straight C is very difficult, and surely an order of magnitude less efficient since the `_overflow()` "function" takes exactly one cycle... The C FAQ proposes for this problem following solution:[2]

```
int chkadd(int a, int b)
{
        if(INT_MAX - b < a) {
                fputs("int overflow\n", stderr);
                return INT_MAX;
        }
        return a + b;
}
```

The problem with the above solution is that only works for ints, and then only for `INT_MAX` overflows: if you add -20 to `INT_MIN` it will accept it.

We can generalize that with the following macros:[3]

```
#define __HALF_MAX_SIGNED(type) ((type)1 << (sizeof(type)*8-2))
#define __MAX_SIGNED(type) (__HALF_MAX_SIGNED(type) - 1 \
                                + __HALF_MAX_SIGNED(type))
#define __MIN_SIGNED(type) (-1 - __MAX_SIGNED(type))

#define __MIN(type) ((type)-1 < 1?__MIN_SIGNED(type):(type)0)
#define __MAX(type) ((type)~__MIN(type))
```

12.21 Assigning a value avoiding truncation

When you assign an integer to a short or a char, it is possible that the assignment overflows. For instance, assigning 45000 to a 16 bit short will produce a negative value:

```
#include <stdio.h>
int main(void)
{
        short j = 45000;

        printf("%d\n",j);
}
```

Output:

[2] http://c-faq.com/misc/intovf.html

[3] This macros were proposed by Felix von Leitner

```
-20536
```

To avoid this problems we can do the assignment and then test if the resulting value is equal to what it should be, but that doesn't work since if you assign a negative value to an unsigned integer the comparison will yield a wrong true result.

```
#include <stdio.h>
int main(void)
{
        unsigned int u = -45;

        if (u == -45) {
                printf("0x%x (%u) is equal to %d\n",u,u,u);
        }
}
```

Output:

```
0xffffffd3 (4294967251) is equal to -45
```

The -45 will be converted to an unsigned integer before the comparison, what yields the same value as 4294967251.

12.22 The C standard

Yes, you read correctly. Many language problems can be understood when we trace them to the source document of the language. Let's start with a problem that appears immediately when you start reading that famous document:

12.22.1 Standard word salads

One of the problems of the language is that the C standard is written in a language designed to make it incomprehensible. Each time I read something or try to understand something in that document I have to ask in the discussion group comp.std.c if I understand correctly, and more often than not I discover that I have forgotten to take into account some corrigenda, some other text that specifies some unknown side effect. This situation has been summarized by "Han from China", a pseudo that writes in the discussion group (USENET) comp.lang.c:

> The overly abstract writing has the ironic effect of making unwanted implementation differences all the more likely, as each implementor attempts to unravel the cryptic, ethereal prose of the standard. The sharp disagreements in interpretations of the C standard are evidence of its murkiness.
>
> The fact that the standard is a technical document is no excuse. I'm aware of no other technical document with such impenetrable writing. Take a look at the RFCs for examples of approachable technical writing.Appendix A of K&R2 is another good example, but that's incomplete. Many of the concepts written about in the C standard aren't difficult to

explain in plain English with clarity and precision. Obscurantism is uncalled for.

As one example of the sharp disconnect between the standard and a "good" book, check out all the recommended books and see how many of them get the whole "update" mode fsetpos / fseek / rewind / (fflush) thing right for input-to-output switching. In fact, just for fun, try counting past three the number of books that even mentions it, whether erroneously or not. Don't know what I'm talking about? You're in for a good laugh (or cringe) when you learn about the mechanics of "update" mode (reading AND writing a file).

As another example of how "precise" the standard is and how "good" and "reliable" the recommended books are, we'll turn to clause 7 of the standard, which is usually considered one of the most readable of the clauses. Right from the start, we read the following gem:

"The program shall not define any macros with names lexically identical to keywords currently defined prior to the inclusion."

Yes, not the more readable "The program shall not define..." because clearly some ANSI/ISO pedant must have made the wonderful "technical" distinction that programmers do the act of defining, not programs. Instead we get that "defined" placed way out in ambiguity territory.

You may think the sentence implies that

```
#define void int
#include <...> // and rest of standard headers
```

IS NOT allowed, whereas

```
#include <...> // and rest of standard headers
#define void int
```

is allowed. But an alternative reading of the sentence could imply that

```
#define void int
#include <...> // and rest of standard headers
```

IS allowed, whereas

```
#define void int
#include <...> // and rest of standard headers
#undef void
#define void char
```

is not allowed (since the keyword 'void' is "currently defined prior to the inclusion").

And our "good", "reliable" books aren't quite so, since the latest edition of H&S rejects both interpretations and offers the following:

"The identifiers listed in Table 2-4 are keywords in Standard C and must not be used as ordinary identifiers. They can be used as macro names since all preprocessing occurs before the recognition of these keywords." (2.6)

and

"Since the preprocessor does not distinguish reserved words from other identifiers, it is possible, in principle, to use a Standard C reserved word as the name of a preprocessor macro, but to do so is usually bad programming practice." (3.3)

Notice nothing about header inclusion? This would imply that

```
#define void int
#include <...> // and rest of standard headers
```

is perfectly legal.

12.22.2 A buffer overflow in the C standard document

Examples of sloppy programming abound, but it was for me a surprise to discover that the C standard itself propagates this same kind of "who cares?" attitude.

In the official C standard of 1999 we find the specifications of the "asctime" function, page 341:

```
char *asctime(const struct tm *timeptr)
{
        static const char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
        };
        static const char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
        };
        static char result[26];
        sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
                wday_name[timeptr->tm_wday],
                mon_name[timeptr->tm_mon],
                timeptr->tm_mday, timeptr->tm_hour,
                timeptr->tm_min, timeptr->tm_sec,
                1900 + timeptr->tm_year);
        return result;
}
```

This function is supposed to output a character string of 26 positions at most, including the terminating zero. If we count the length indicated by the format directives we arrive at 25:

- 3 for the day of the week +
- 1 space +
- 3 for the month +
- 3 for the day in the month +

- 1 space +
- 8 for the hour in hh:mm:ss format +
- 1 space +
- 4 for the year +
- 1 newline.

It makes 25 characters, and taking into account the terminating zero, the calculation seems OK.

But it is not.

The problem is that the format `%d` of the printf specification doesn't allow for a maximum size. When you write `%.3d` it means that at least 3 characters will be output, but it could be much more if, for instance, the input is bigger than 999. In that case, the buffer allocated for asctime is too small to contain the printf result, and a buffer overflow will happen. The consequences of that overflow will change from system to system, depending on what data is stored beyond that buffer, the alignment used for character strings, etc. If, for instance, a function pointer is stored just beyond the buffer the consequences could be catastrophic in the sense that it would be very difficult to trace where the error is happening.

Getting rid of buffer overflows

How much buffer space we would need to protect asctime from buffer overflows in the worst case?

This is very easy to calculate. We know that in all cases, `%d` can't output more characters than the maximum numbers of characters an integer can hold. This is `INT_MAX` and taking into account the possible negative sign we know that:

$$N = 1 + \lceil log_{10}(2^{(CHAR\_BIT \cdot sizeof(int))-1} \rceil \tag{12.1}$$

Or, to write the above equation in C:

```
Number of digits N = 1 + ceil(log10(INT_MAX));
```

For a 32 bit system this is 11, for a 64 bit system this is 21. In the asctime specification there are 5 `%d` format specifications, meaning that we have as the size for the buffer the expression: `26+5*N` bytes. In a 32 bit system this is 26+55=81.

This is a worst case oversized buffer, since we have already counted some of those digits in the original calculation, where we have allowed for 3+2+2+2+4 = 13 characters for the digits. A tighter calculation can be done like this:

1. Number of characters besides specifications (`%d` or `%s`) in the string: 6.
2. Number of `%d` specs 5
3. Total = 6+5*11 = 61 + terminating zero 62.

The correct buffer size for a 32 bit system is 62.

Buffer overflows are not inevitable.

As we have seen above, a bit of reflection and some easy calculations allows us to determine how much we need exactly. Most buffer overflows come from people not doing those calculations because "C is full of buffer overflows anyway" attitude. What is absolutely incredible however, is to find a buffer overflow in the text of the last official C standard.

An international standard should present code that is 100% right. There are so few lines of code in the standard, that making those lines "buffer overflow free" is not a daunting task by any means.

The attitude of the committee

I am not the first one to point out this problem. In a "Defect Report" filed in 2001, Clive Feather proposed to fix it. The answer of the committee was that if any of the members of the input argument was out of range this was "undefined behavior", and anything was permitted, including corrupting memory.

Corrupting memory provokes bugs almost impossible to trace. The symptoms vary, and the bug can appear and disappear almost at random, depending on the linker and what data was exactly beyond the overflowed buffer. This means that the asctime function can't be relied upon. Any small mistake like passing it an uninitialized variable will provoke no immediate crash but a crash later, in some completely unrelated program point.

Here is the deffect report of Mr Cleaver:

> `Defect Report #217`
> `Submitter: Clive Feather (UK)`
> `Submission Date: 2000-04-04`
> `Reference Document: N/A`
> `Version: 1.3`
> `Date: 2001-09-18 15:51:36`
> `Subject: asctime limits`
>
> Summary
>
> The definition of the asctime function involves a sprintf call writing into a buffer of size 26. This call will have undefined behavior if the year being represented falls outside the range [-999, 9999]. Since applications may have relied on the size of 26, this should not be corrected by allowing the implementation to generate a longer string. This is a defect because the specification is not self-consistent and does not restrict the domain of the argument.
>
> Suggested Technical Corrigendum
>
> Append to `7.23.3.1[#2]:`
>
> except that if the value of `timeptr->tm_year` is outside the range [-2899, 8099] (and thus the represented year will not fit into four characters) it is replaced by up to 4 implementation-defined characters.

And here is the answer of the committee:

> Committee Response
> From 7.1.4 paragraph 1:
> If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to non-modifiable storage when the corresponding parameter is not const-qualified) or a type (after promotion) not expected by a function with variable number of arguments, the behavior is undefined.
> Thus, asctime() may exhibit undefined behavior if any of the members of timeptr produce undefined behavior in the sample algorithm (for example, if the `timeptr->tm_wday` is outside the range 0 to 6 the function may index beyond the end of an array).
> As always, the range of undefined behavior permitted includes:
>
> - Corrupting memory
> - Aborting the program
> - Range checking the argument and returning a failure indicator (e.g., a null pointer)
> - Returning truncated results within the traditional 26 byte buffer.
>
> There is no consensus to make the suggested change or any change along this line

The committee answer is enlightening: corrupting memory is preferred to a simple modification to avoid a catastrophic crash. Anything goes.

In the last years the attitude of the committee has evolved a bit, and there is now a clause that limits the values accepted by `asctime` to avoid the buffer overflow. This will be included in the future standard to be published sometime in the future during this decade.

12.22.3 A better implementation of asctime

The implementation of asctime in a form that doesn't overflow and always reports any overflow error is trivial. Here is one of the many possible implementations. Its design goals are clear:

1. The function should never fail, even if its argument is NULL.
2. Any input outside its normal range should be flagged in the output.
3. Performance should be improved by a factor of 10.

This requirements aren't extraordinarily complex, and they are easy to implement. Here is one of the possible ways of doing this:

```
#include <time.h>
#include <stdio.h>
#include <string.h>
```

```
// Proposed change to the code. FULL ERROR CHECKING
// Performance: more than 10 times FASTER
char *asctime1(const struct tm *timeptr)
{
        static const char wday_name[8][3] = {
                "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
        };
        static const char mon_name[16][3] = {
                "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
        };
        static char result[26];
        int year;

        // Initialize the result buffer
        strcpy(result,"*** *** ** **:**:** ****\n");
        if (timeptr == NULL)
                // If timeptr is NULL we get a string of asterisks
                return result;
        // (1): Output the day of the week in 3 positions
        if (timeptr->tm_wday >= 0 && timeptr->tm_wday < 7)
                memcpy(result,wday_name[timeptr->tm_wday],3);

        // (2): Output the month in 3 positions
        if (timeptr->tm_mon >= 0 && timeptr->tm_mon < 12)  {
                memcpy(result+4,mon_name[timeptr->tm_mon],3);
        }

        // (3): Output the day of the month
        if (timeptr->tm_mday >= 1 && timeptr->tm_mday <= 31) {
                result[8] = '0' + timeptr->tm_mday/10;
                result[9] = '0' + timeptr->tm_mday%10;
        }

        // (4): Output the hour
        if (timeptr->tm_hour >= 0 && timeptr->tm_hour <= 23) {
                result[11] = '0' + timeptr->tm_hour / 10;
                result[12] = '0' + timeptr->tm_hour % 10;
        }
        // (5): Output the minute
        if (timeptr->tm_min >= 0 && timeptr->tm_min <= 59) {
                result[14] = '0'+timeptr->tm_min / 10;
                result[15] = '0'+timeptr->tm_min % 10;
        }

     // (6): Output seconds. Range is 0-60 inclusive
     //   (allows for leap seconds).
```

```
        if (timeptr->tm_sec >= 0 && timeptr->tm_sec <= 60) {
                result[17] = '0'+timeptr->tm_sec / 10;
                result[18] = '0'+timeptr->tm_sec % 10;
        }

        // (7): Output the year
        year = timeptr->tm_year+1900;
        if (year <= 9999 && year >= -999) {
                if (year < 0) {
                        result[20] = '-';
                        year = -year;
                }
                else {
                        result[20] = '0' + year/1000;
                        year %= 1000;
                }
                result[21] = '0' + year/100;
                year %= 100;
                result[22] = '0' + year/10;
                year %= 10;
                result[23] = '0'+ year;
        }
        return result;
}
```

Why is performance so much better than in the standard's version? Because there is no call to sprintf. That function is a very general function and because of that, it is very expensive. A much better and faster approach is to fill the output buffer one field at a time.

13 Bibliography

Here are some books about C. I recommend you to read them before you believe what I say about them.

- **The C programming language** Brian W Kernighan, Dennis Ritchie. (second edition)

  This was the first book about C that I got, and it is still a good read. With many exercises, it is very good start for a serious beginner.

- **C Unleashed** Richard Heathfield, Lawrence Kirby et al.

  Heavy duty book full of interesting stuff like structures, matrix arithmetic, genetic algorithms and many more. The basics are covered too, with lists, queues, double linked lists, stacks, etc.

- **Algorithms in C** Robert Sedgewick.

  I have only the part 5, graph algorithms. For that part (that covers DAGs and many others) I can say that this is a no-nonsense book, full of useful algorithms. The code is clear and well presented.

- **C, a reference manual** (Fifth edition) Samuel P Harbison and Guy L Steele Jr.

  If you are a professional that wants to get all the C language described in great detail this book is for you. It covers the whole grammar and the standard library with each part of it described in detail.

- **A retargetable C compiler: design and implementation** Chris Fraser and Dave Hanson. Addison-Wesley Professional

  This book got me started in this adventure. It is a book about compiler construction and not really about the C language but if you are interested in knowing how a compiler works this is surely the place to start.

- **C interfaces and implementations** David R. Hanson

  This is an excellent book about many subjects, like multiple precision arithmetic, lists, sets, exception handling, and many others. The implementation is in straight C and will compile without any problems in lcc-win.

- **Safer C** Les Hatton

As we have seen in the section «Pitfalls of the C language», C is quite ridden with problems. This book address how to avoid this problems and design and develop you work to avoid getting bitten by them.

- **C Traps and Pitfalls** Andrew Koenig

  Good discussions about avoiding "off by one" errors, function declarations and the always obscure relationship between arrays and pointers.

- **C Programming FAQ** Steve Summit

  C Programming FAQs contains more than 400 frequently asked questions about C, accompanied by definitive answers. Some of them are distributed with lcc-win but the book is more complete and up-to-date.

- **The Standard C Library** P.J. Plauger.

  This book shows you an implementation (with the source code) of the standard C library done by somebody that is in the standards committee, and knows what he is speaking about. One of the best ways of learning C is to read C. This will give you a lot of examples of well written C, and show you how a big project can be structured.

- **The C Standard** John Wiley and Sons.

  This is the reference book for the language. It contains the complete C standard and the rationale, explaining some fine points of the standard.

- **Secure coding in C and C++** Robert C Seacord.

  This is a very complete book about the known vulnerabilities of C programs. Very clear, and easy to read. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade Crispin Cowan, Perry Wagle, Calton Pu,Steve Beattie, and Jonathan Walpole Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology. URL:

  http://downloads.securityfocus.com/library/discex00.pdf

- **C Mathematical function Handbook** Louis Baker (1991)

  This book has provided invaluable help in implementing many functions from the "special functions" library.

Appendices

.1 Using the command line compiler

Now, integrated development environments (IDEs) have many advantages but there many situations where you want to compile some file without it.

The compilation process is organized by the compiler driver `lc`. This program calls the compiler proper, and later the linker `lcclnk`. If a resource file should be compiled, `lc` calls the resource compiler `lrc`.

The lc.exe compiler driver has the following format:

```
lc [ compiler options] files... [linker options]
lc64 [ compiler options] files... [linker options]
```

This means that you give it all the compiler options first, then the files you want to compile, then the options you want for the linker pass. Things in brackets are optional. You can just use `lc file.c` and the driver will call the compiler without any options, as well as the linker if all compilation steps happen without any error.

The `lc` program builds 32 bits executables, the `lc64` program builds 64 bits executables.

Note that you can pass to the driver an ambiguous file specification like

```
lc *.c -o prog.exe
```

Here are all the command line "switches" (arguments) that the compiler understands

Option	Meaning
-A	All warnings will be active. Repeating this (`-A -A`) increases further the warning level.
-ansic	Disallow the language extensions of lcc-win32: operator overloading and references will not be accepted.
-C	Keep comments in the preprocessed output. You should be prepared for side effects when using -C; it causes the preprocessor to treat comments as tokens in their own right. For example, macro redefinitions that were trivial when comments were replaced by a single space might become significant when comments are retained. Also, comments appearing at the start of what would be a directive line have the effect of turning that line into an ordinary source line, since the first token on the line is no longer a #.
-check	Read all the input file without generating anything. The warning level is automatically increased. This is designed for checking a source file for errors and warnings.
-D	Define the symbol following the D. Example: -DNODEBUG The symbol NODEBUG is #defined. Note that there is NO space between the D and the symbol.
-E	Generate an intermediate file with the output of the preprocessor. The output file name will be deduced from the input file name, i.e., for a compilation of foo.c you will obtain foo.i.

-E+	Like the -E option, but instead of generating a #line xxx directive, the preprocessor generates a # xxx directive. Some systems need this option, specifically some versions of gcc.
-EP	Like the -E option, but no #line directives will be generated.
-errout	Append the warning/error messages to the indicated file. Example : `errout=Myexe.err` This will append to Myexe.err all warnings and error messages.
-eN	Set the maximum error count to N. Example: `-e25` The compiler will stop after 25 errors.
-Fo<file name>	This forces the name of the output file. Normally lcc deduces that name from the name of the input file, i.e., for foo.c, foo.obj, or foo.asm, or foo.i will be generated.
-fno-inline	Disables any inling of functions. No inline expansion will be performed, even if optimizations or on.
-g2	Generate the debugging information. Two types of debug information will be generated: COFF and CodeView (NB09).
-g3	Arrange for function stack tracing. If a trap occurs, the function stack will be displayed.
-g4	Arrange for function stack and line number tracing.
-g5	Arrange for function stack, line number, and return call stack corruption tracing.
-I	Add a path to the path included, i.e., to the path the compiler follows to find the header files. Example: -Ic:\project\headers Note that there is NO space between the I and the following path.
-libcdl	Use the declarations needed to link with lcclibc.dll instead of libc.lib, the static library. Most of the declarations are the same for both except some rare exceptions like the table of the ctype.h library
-M	Print in standard output the names of all files that the preprocessor has opened when processing the given input file. If the Fo option is active, printing will be done in the file indicated by the Fo option. No object file is generated.
-M1	Print in standard output each include file recursively, indicating where it is called from, and when it is closed. This option is used in the IDE in the Show includes option.
-nw	No warnings will be emitted. Errors will be still printed.
-O	Optimize the output. This activates the peephole optimizer. Do not use this option with the -g option above.

-overflowcheck	Generates code to check for any overflow in integer addition, subtraction and multiplication. If an overflow is detected, the user function void `_overflow(char *FunctionName, int line);` is called. A default implementation that just prints the function name and the line in stderr is supplied.
-p6	Enable Pentium III instructions
-profile	Generate profiling code. At each line, the compiler will generate code to record the number of times execution passed through that line. At the end of the program, a file is generated with the name "profile.data", that contains the profiling data for the execution. This file is erased, if it exists.
-S	Generate an assembly file. The output file name will be deduced from the input file name, i.e., for a compilation of foo.c you will obtain foo.asm.
-U	Undefine the symbol following the U.
-unused	Warns about unused assignments and suppresses the dead code. Use with care
-x	Generate browse information in an .xrf file.
-z	Generate a file with the intermediate language of lcc. The name of the generated file will have a .lil extension (lccs intermediate language).
-Zp[1, 2, 4, 8, 16]	Set the default alignment in structures to one, two, four, etc. If you set it to one, this actually means no alignment at all.
File.asm	All files with a .asm extension are assumed to be files written for lccs assembler. Beware: the syntax of lccs assembler is radically different from all standard assemblers under windows.