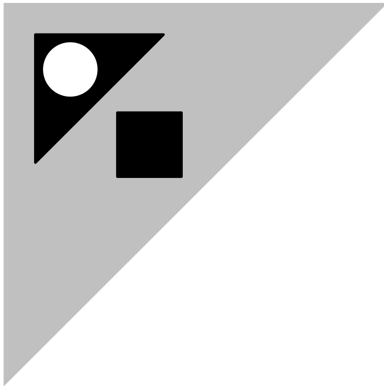

Module 812

Program Development in C

© Deakin University

Aim

After working through this module you should be able to understand the basic characteristics of the C programming language, and how programs are created in C.

Learning objectives

After working through this module you should be able to:

1. Describe the compilation process in C
2. Explain the purpose of memory models in C
3. Identify the major components of C programs
4. Appreciate the role of libraries and header files in C
5. Understand the significance of Prototyping in C
6. Define the structure and use of Makefiles & Project files in C
7. Distinguish between errors and warnings when compiling and linking in C
8. Identify the steps involved in creating a C program

Content

Program development in C

Structure of C programs

Separate compilation

Prototyping

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

References & resources

Borland International, *Turbo C/C++ User's Guide*

Kernighan, B. W. and Ritchie, D. M., *The C Programming Language* (2nd ed.), Prentice-Hall, 1988

Kernighan, B. W. and Plauger, P. J., *The Elements of Programming Style*, (2nd ed.), Mc-Graw-Hill, 1978

McConnell, S., *Code Complete*, Microsoft Press, 1993

Introduction

C and C++ are popular languages for software development because of their simplicity of expression, the compactness of the code, and the wide range of applicability. C allows the programmer a wide range of operations from high level down to a very low level, approaching that of assembly language. Programming in C is a tremendous asset in those areas where you may want to use assembly language but would rather keep it a simple to write and easy to maintain program. It has been said that a program written in C will pay a premium of a 50 to 100% increase in runtime because no language is as compact or fast as assembler. However, the time saved in coding can be tremendous, making it the most desirable language for many programming chores. In addition, since most programs spend 90 percent of their operating time in only 10 percent or less of the code, it is possible to write a program in C then rewrite a small portion of the code in assembler and approach the execution speed of the same program if it were written entirely in assembler.

It is important to note, however, that the C programmer takes on a great deal of responsibility because it is very easy to write a program that destroys itself due to the sort of small errors that a Pascal compiler would flag and call a fatal error. A C compiler, on the other hand, may only issue warnings over the same faults.

Origin of C

The C language was designed by Dennis Ritchie of AT&T's Bell Laboratories. It was intended for the development of complex systems like operating systems. (The most notable such system is UNIX, initially designed by Ken Thompson at AT&T in 1969, but largely written in the early 1970s by Dennis Ritchie using C.) The primary goal of C was therefore operating *efficiency*, and this remains one of its hallmarks.

The C programming language was fully defined in the text by Kernighan and Ritchie (*The C programming Language*), and this was the ‘standard’ used by all C programmers until a few years ago. Many commercial C compilers, for instance, advertised themselves as ‘K&R compatible’.The ANSI standard for C was approved in December 1989 and has become the official standard for programming in C (the equivalent standard for C++ has just been completed). Since the ANSI standard adds many things to the language which were not a part of the Kernighan and Ritchie definition, and changes a few others, the two definitions are not absolutely compatible and even some experienced C programmers may not know all the newer constructs added to the language by the ANSI-C standard. The ANSI-C standard continues to be revised, and the current version is 2.1 (linked to ANSI-C++).

Commercial C compilers now indicate their ANSI-C compatibility; compatibility with ‘K&R’ is maintained by the development of the ANSI standard as a superset of Kernighan and Ritchie's original definition.

The C language is rightly famed for its portability, both between implementations and across platforms. Nevertheless, there are definite differences between C compilers – as you will find whenever you try to move code from one compiler to another. Most of the differences become apparent when you use extensions such a screen I/O or file handling, but even these can be minimised by careful choice of programming method.

Using the Modules

The most effective way to use the Modules in this series (812 to 819) is while sitting in front of your computer with the C compiler active. The aim is to help you gain experience with your own C/C++ compiler, in addition to teaching you proper use of C and C++. It is assumed that you will display each example program on the monitor and read the accompanying text, which will describe any new constructs introduced in the example program. As you study the program in the text, and understand the new constructs, you can compile and execute the program. Where appropriate you should experiment by changing the program contents, re-compiling the source code, and observing the results of running the modified program. It is also assumed that you are familiar with the Turbo Pascal Integrated Development Environment (IDE). The Turbo C IDE is largely identical as an editing environment, but the compiling/linking process is somewhat different, reflecting the different development sequences of the two languages; these are discussed in detail in the next section.

Note also that you may not find it necessary to compile and execute every program. At the end of each example program on the disk, in comments, you will find the expected results of executing that program. If a construct is easy to understand you may choose not to run that program, relying instead upon the included results to show you the output. This is left to your own judgement.

Objective 1: After working through this section you should be able to describe the compilation process in C

At a superficial level, creating C programs using a microcomputer-based C implementation (such as the Borland and Turbo C++ compilers from Borland International) is very similar to the same process in Pascal – if you are using a single source code file. But C was really designed as a production language for creating large applications and operating systems. This will inevitably lead to complex structures of interlinked source code files. These are assumed to be independently compiled to create object files, which are then linked at a later date to create an executable (program) file that can be loaded into memory and run. For major applications the management of this collection of source code and object code files is the responsibility of the programmer. All commercial C implementations support the *make* process; some (such as Turbo C) have other methods of controlling the development process. We will examine these in a later section of this Module.

[Note that we will sometimes talk about C and C++ compilers as if they were separate, but today practically all implementations support both languages, so we are really discussing C/C++ compilers.]

Figure 1 gives an overview of the program development sequence in C (and C++). In a functioning system, of course, the process of development would be iterative, particularly during debugging. All source code files are created as text files in an editor, such as that in the Turbo C IDE. There is no requirement for each source file in a multi-file development project to be cross-referenced in each source file, as in the {$I include file}compiler directive in Turbo Pascal. This cross-referencing is done with the make file and the linker. Source code files can, however, access program libraries *via* one or more #include statements at the top of the 'main' source file (the file containing the single *main* function that all C programs must have).

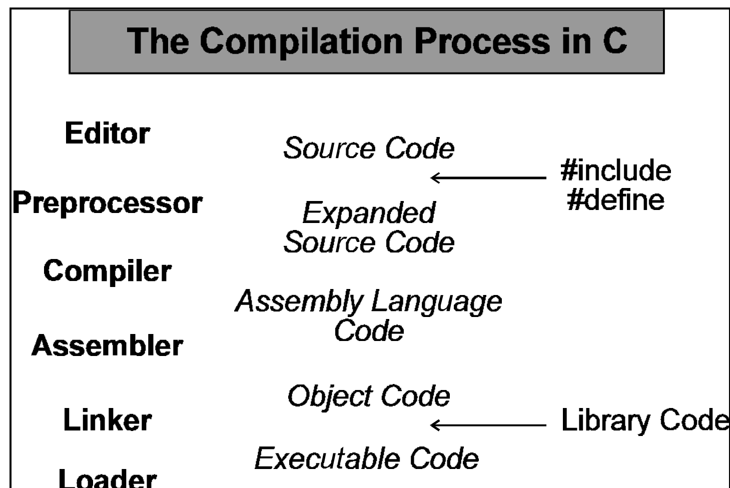

Figure 1: The Development Process in C

When such a file is compiled, it is first passed to a *preprocessor* (built into the compiler) which incorporates into the file all the required function headers from the requested libraries. In this way we are able to use libraries provided by the compiler vendor, as well as create our own. (Libraries are covered in a later section of this Module.)

The augmented source code is then passed to the compiler where syntax and logic errors are identified. In C these can produce either error messages or warnings, and most compilers allow the experienced programmer to control which warnings are to be triggered; errors are always displayed. The error and warning process is also covered in a later section of this Module.

If the file is compiled without errors, it is converted by the compiler to assembler code, and the built-in assembler then creates a separate .OBJ file. The point of these separate steps is that this file does not need to be re-compiled unless the source code (or the code in a library that it accesses) changes.

The linker is then invoked to link together all the .OBJ files, including the relevant library files, to form an executable file. The linker will detect errors such as mis-spelling the name of a library function in the source code, or using the incorrect number or type of parameters for a function. Finally, if the program has been successfully linked, the loader can be called from within the IDE to execute the program.

Whilst users of the IDE do not need to interfere directly in this process, it is inherent in the organisation of C that creators of larger applications will eventually take more direct control of development sequence. This is made possible by the use of make and project files, which are covered in a later section of this Module. By doing so, the programmer can greatly reduce the amount of unnecessary recompilation of program modules in complex development projects. This is achieved by using the 'intelligence' of the linker to re-compile only those modules that have been modified since the last compiler pass.

Exercise 1

Describe the stages involved in creating a stand-alone C application in an integrated development environment.

Objective 2: After working through this section you should be able to explain the purpose of memory models in C

One of the distinctive differences between C and Pascal is in the area of memory usage. In Turbo Pascal, for instance, each program is usually limited to 64K of static data. In most C implementations, on the other hand, the programmer is able to select from a variety of *memory models*. These control the way in which physical memory (RAM) is utilised by the program. There may be occasions, for example, when an application is quite small in terms of the amount of code, but needs to manipulate large volumes of data in RAM. Conversely, some applications (such as simulations) involve very large amounts of code but handle small amounts of data. Flexible ways of optimising the use of available memory are clearly desirable, and most C compilers offer several memory models to achieve this.

The major determinant of memory availability is the size of the pointers used to access memory locations. In Turbo C almost all pointers are declared as *near* (16 bits per pointer, the default) or *far* (32 bits per pointer). Whilst near pointers simplify memory access in the segmented memory of Intel processors by allowing direct arithmetic on pointers, they limit accessible memory to 64 kilobytes (2^{16}). Far pointers can be used to access multiple code segments, each 64K, up to 1 megabyte (2^{32}), by using segment and offset addressing. The cost is that the programmer cannot use the simple pointer arithmetic that is possible with near pointers.

By using combinations of these pointers for data and code, Turbo C is able to compile programs using any of the six memory models shown in Table 1.

Memory Model	Pointers used	Available Memory
Tiny	Near for code, Near for data	64K code+data/stack
Small	Near for code, Near for data	64K code, 64K data/stack
Medium	Far for code, Near for data	1 M code, 64K data/stack
Compact	Near for code, Far for data	64K code, 1 M data/stack
Large	Far for code, Far for data	1 M code, 1 M data/stack
Huge	Far for code, Far for data	1 M code, 1 M data/stack (inc. static > 64K)

Table 1: Turbo C Memory Models

The default model is Small, which is effective for the majority of applications. The Tiny model is specifically designed for the production of TSR (memory-resident) programs, which must fit into one code segment and be compiled as .COM rather than .EXE files. The

remaining models are selectable in the compilation process, through the IDE or in the make file. Careful use of alternative memory models is one the hallmarks of thoughtful C programming.

Exercise 2

Explain how and why all C compilers allow the programmer to control the way in which an application accesses memory.

Objective 3: After working through this section you should be able to identify the major components of C programs

Before we turn to the language constructs of C in the next few Modules, we should examine the basic structure of a C program. In particular, we can contrast its structure with the more familiar structure of Pascal programs. As we have discussed in Module 811, this is a convenient stage in our sequence of steps for coming to terms with a new language, based upon an understanding of the development process of the kind presented earlier in this Module. Once we understand how programs are structured at a 'global' level we can develop our knowledge of the language construct-by-construct, comparing the new language with the old. Table 2 summarises the general structure of Pascal and C programs.

Structure	***Pascal***	***C***
'Main' section	Single main program, at the end of the source code. Execution starts after BEGIN and continues to END.	Single *main* function, anywhere in the source code. Execution starts at the first statement in this function, and continues to the last statement.
Declarations and Scope	Identifiers declared *before* all procedures or functions are global in scope; those declared *inside* procedures or functions are local in scope.	Identifiers declared *outside* any function are global in scope; those declared *inside* any function (including main) are local in scope.
Module support	Procedures and (Typed) Functions	Functions only, Typed or Untyped (void)
Parameters	Pass-by-value Pass-by-reference	Pass-by-value Pointers (predominantly) instead of pass-by-reference

Table 1: Comparative Structure of Pascal and C programs

The most important points to note are that the *main* function in C corresponds to the 'main program' in Pascal, and can be anywhere in the source code - but it is usually the first function. Consequently, there is no 'end' to a C program, as there is in Pascal, where every program ends at the END. statement, where the compiler stops processing the code. This is why, for example, the expected results for each sample program appear at the end of the file - but they have to be *in comments* or they will be treated as source code by the compiler.

The second point to note is that the executable part a C program consists entirely of declarations and functions, with the main function being defined (almost) like any other function. All functions are typed

(integer, real, etc.) but only those returning values must be declared as the correct type; other functions are declared as *void*, which is the ‘untyped type’.

Passing parameters to a function in C is similar to the same process in Pascal, but returning parameters from a function in C is handled largely by the use of pointers, and to a lesser extent, arrays.

We will examine the structure of a simple C program - and how it is compiled - in the last section of this Module.

Exercise 3

Describe the basic differences between the Pascal and C languages.

Objective 4: After working through this section you should be able to appreciate the role of libraries and header files in C

As with Pascal, all meaningful C programs need to use pre-defined functions. These may include numerical operations (sin, cos, abs etc.), screen output (printf etc.) and keyboard input (scanf etc.). In C these functions are contained in a set of libraries supplied with your compiler as binary files, usually with a .LIB extension. The numerical functions, along with other generic functions required by any implementation, are contained in a series of libraries, one for each memory model.

Implementation-specific functions, notably those dealing with input and output, are contained in another series of .LIB files provided by the compiler vendor. How many of these there are, and their organisation, varies between implementations. The scope for variation is actually controlled somewhat by the ANSI standardisation process, which defines the 'minimum' set of functions that must be supported for a compiler to be ANSI-compatible. This includes the minimal set of input/output functions (such as printf for formatted screen output, and scanf for input) that make up the *standard input/output* library (stdio). All commercial compilers go well beyond this minimum and include at least one advanced screen/keyboard control library (Turbo C has conio), and some form of sophisticated file input/output *via* streams (iostream in Turbo C). We shall examine the structure of the stdio library in more detail in the next Module, and compare it with conio.

Implementation-specific C libraries are generally made up two files:

1. The library itself, a binary file containing the compiled code for the functions and declarations that make up the library, as a .LIB file. In Turbo C it is usually placed with other library binaries, in a library sub-directory (..\lib)

2. The header file for the library, a text file containing the declarations for the functions in the library, with the extension .H. (These act as *prototypes* for the functions, a process we will consider in the next section.) In Turbo C it is usually placed with other header files, in an *include* sub-directory (..\include).

The header file contains the header for each function in the corresponding library, and has the same name as the library. In some implementations (including Turbo C) most of the supplied libraries are 'bundled' for convenience into a single binary library (one per memory model). There will be a series of header files reflecting a logical grouping of functions: complex.h for complex number operations, alloc.h for memory management, dos.h for dos interaction, and so on. However, libraries created by third-party vendors, and those created by

other programmers (and eventually perhaps by you), are more likely to follow the “one .LIB goes with one .H” pattern.

To use these implementation-specific and third-party libraries a programmer simply adds the appropriate header file (or files) as include statement(s) at the top of her program

```
#include "conio.h"
#include <strings.h>
```

and then uses any of the functions from the libraries as desired in the program. (Either form of definition - " " or < > - is allowable in Turbo C.)

Whilst this process may seem at first sight to be unnecessarily clumsy (surely every program has some type of i/o, so why separate it?) it is central to the legendary *portability* of C code. In some cases porting to another compiler or platform will require no changes at all to the source code, as any necessary changes are taken care of within the libraries. In most other cases porting will only involve changing the name of one or more libraries in the include statements: strings.h to string.h, or maths.h to math.h, for example. Even in the worst case (changing function names and/or parameters throughout the code), porting will almost always be easier than in comparable languages like Pascal.

Exercise 4

Examine and describe the library structure of your C compiler. Define the contents of the header files for the major standard libraries.

Objective 5: After working through this section you should be able to understand the significance of Prototyping in C

In the last section we introduced the term *prototyping*, and this concept is extremely important in the efficient development of C programs. Prototyping is the process of including early in the source code headers for all functions used by the program. These serve as 'templates' for the functions. Including prototypes in this manner allows the compiler to quickly check that all calls to pre-defined and user-defined functions are correctly done, using the right number and type of parameters and using the correct return type (if any). The resulting extra error checking is potentially very valuable to the programmer. This also speeds up the compilation process, particularly where a large number of external libraries are being used.

For pre-defined functions prototyping is automatically achieved by the use of the header file system described in the last section, though it is not the primary purpose of that system. For functions defined within the program it is the job of the programmer to place appropriate prototypes at the start of the program. This is a simple task that involves making a copy of each function definition and placing them immediately below the include statements. The only change that must be made is to specify the type of any parameters passed to the function. For example, the following short function is passed a single value and prints out its square:

```
void square(number)     /* function to print the square of a value
*/
int number;             /* function type is void as it returns  */
{                       /* no value                             */
int numbersquared;
numbersquared = number * number;
printf("The square of %d is %d\n", number, numbersquared);
}
```

Its prototype would be

```
void square(int number)
```

and this would appear at the start of the program file.

Attitudes to prototyping among the C community have changed over the last decade. The ANSI standards committees have moved to require the use of prototypes in all C and C++ programs. In version 2.1 of the C standard, therefore, *all* functions (apart from the *main* function) must have a prototype, and their absence will produce a compiler error. At the same time, some C programmers have objected, insisting that the use of prototyping should remain an optional programming aid. As a conscession, most compilers that allow warnings and errors to be controlled by the programmer (as described later in this Module) regard lack of prototyping as a non-fatal error that can be turned off.

Exercise 5

Define the prototypes for the following three functions:

```
print_a_value(int x)   /* function to print a single value */
{
   printf("The value of x is %d", x);
}


print_a_value(int x, float y) /* function to print two values */
{
   printf("X = %d and Y = %f", x, y);
}


float pi_it (float x)
                /* function to return the value of x times π*/
float pi = 3.1472;
{
   return (float) (x * pi)
}
```

Objective 6: After working through this section you should be able to define the structure and use of Makefiles & Project files in C

Unlike Pascal, both C and C++ were designed to be production languages in which large applications (including operating systems) could be created. Central to this process is the idea of *separate compilation*, whereby large programs are built in sections which are compiled into object code as each is completed. Once a section has been compiled the linker will use the object code for that section when other sections of the program that use functions in the compiled section are themselves compiled. If the source code for the compiled section has been modified, the linker will recompile it before compiling sections which are dependent upon it.

Project files

In an integrated development environment like Turbo C's IDE this process is generally managed automatically by the compiler and linker. However, as we shall see in Module 818, there are circumstances where we will need to override this process. It is standard practice in C++, for example, to define a base class to be used in an object hierarchy in a separate header (.H) file. The implementation of the methods in the base class is then defined in a separate source code (.CPP) file. The latter can be compiled, but not linked and/or executed. If we create a program that uses this (and perhaps other) classes, we will need to tell the linker what files to use during the link, and how to link them. All Borland International's C and C++ compilers support a method for doing this within the IDE called *project* files. A project file will contain information about

- compiler, linker and library options
- the directory path of included files
- a list of all files making up the project
- any special options (eg. use of the assembler)

Makefiles

Because C was defined in an era when the development cycle was carried out by separate processors (editor, compiler, linker etc.) its carries many vestiges of its command-line based origins. The most notable of these is the use of *makefiles*, and all C/C++ compilers support the *make* process. Unfortunately, another vestige is the fact that no two makefile systems have the same syntax, so it is not uncommon to find C or C++ source code supplied with several makefiles, one for

each of the major compilers on a particular platform. Whilst this seems clumsy, it can be looked at in a positive light, as an aid to portability. Those parts of the code that are most device-dependent can be placed in one file, and alternative versions provided for each implementation. One of the makefiles' jobs is to then ensure that the correct set of files is compiled and linked.

The contents of a makefile are essentially the same as the project file described above, but it can also include file management commands, such as deleting temporary files. The aim is to tell the MAKE program - which manages the handling of files in a multi-file project - all it needs to know in order to perform its task efficiently. Each command in the makefile is designed to invoke the preprocessor, compiler, assembler or linker, with appropriate parameters (files names and paths, and/or switches). When you invoke the MAKE program it uses by default any file called makefile in the current directory, but you can override this, for example by entering ‘makefile myfile.mak’. All makefiles are text files that can be created and modified in any text editor. Comments can be included, to tell the programmer what each section is doing; these usually start with the hash (#) symbol. They occupy the rest of the line from the symbol, so both of the following makefile lines include comments. The first line is made up of a comment alone, but the second has a comment and a command:

```
# Link all files together
TCC -c -ms myprog.c    # compile the main program
```

Macros

Most makefile systems also allow *macros*, will help to automate regularly-used commands. For example, if you were compiling with the medium memory model in Turbo C, the makefile command might look like this

```
TCC -c -mm myprog.c
```

where the memory option switch is -mm, the second letter indicating the actual model. Changing this to use the large memory model would mean changing it to

```
TCC -c -ml myprog.c
```

But you would want to ensure that *all* memory model references were changed, and this might be tricky: what if you missed one? The preferable alternative is to add a macro at the beginning of the file, and use it throughout; changing the macro will therefore change all occurrences automatically. For example, we might define the macro

```
MODEL = m
```

and the command would then become

```
TCC -c -m$(MODEL) myprog.c
```

Changing globally to a different memory model will then consist simply of changing the macro to read

```
MODEL = l
```

Example

The following makefile compiles and links a program (demo.exe) using a series of separately-compiled objects. It first links these into a library file, which it deletes after it has been used. Neither the simplest possible makefile, or the most complex, it is typical of the kind of makefile needed to manage development of a reasonably complex system.

```
# Makefile for render program
CC = tcc        # compiler macro - Turbo C command line compiler
CFLAGS = -mc -ml # compiler flags for memory models
LFLAGS = -ml

# compile main progra - demo4.exe - from previously-compiled
objects

demo.exe:  demo.obj  keyboard.obj  anim.obj  world.obj  demo.lib
supp.lib userint.lib rend386.lib
$(CC) $(LFLAGS)    demo4.obj keyboard.obj anim.obj world.obj
                      demo.lib supp.lib userint.lib rend386.lib

# compile library (temporarily)
demo.lib: cursors.obj gloveptr.obj mouseptr.obj render.obj
del demo.lib
tlib demo +cursors +gloveptr +mouseptr +hdmanip +render +colormap

keyboard.obj:  keyboard.c  include\rend386.h  include\userint.h
include\plg.h         include\pointer.h         include\splits.h
include\intmath.h include\cursor.h
$(CC) $(CFLAGS) keyboard

world.obj:    world.c    include\rend386.h    include\splits.h
include\tasks.h include\plg.h include\intmath.h include\pointer.h
include\cursor.h   $(CC) $(CFLAGS) world

anim.obj: anim.c include\rend386.h include\tasks.h
$(CC) $(CFLAGS) anim

demo4.obj:    demo4.c    include\rend386.h    include\userint.h
include\plg.h        include\pointer.h        include\intmath.h
include\cursor.h
$(CC) $(CFLAGS) demo4

render.obj:   render.c   include\rend386.h   include\f3dkitd.h
include\intmath.h include\splits.h
$(CC) $(CFLAGS) render

mouseptr.obj: mouseptr.c include\rend386.h include\pointer.h
$(CC) $(CFLAGS) mouseptr

cursors.obj: cursors.c include/rend386.h include/f3dkitd.h \
include/intmath.h include/pointer.h include/cursor.h
$(CC) $(CFLAGS) cursors
```

For the full syntax of the makefile for a particular compiler, you should examine the compiler's documentation.

Exercise 6

Compare the development of a C program in an integrated environment with a command-line system using makefiles.

Objective 7: After working through this section you should be able to distinguish between errors and warnings when compiling and linking in C

Pascal programmers work in a world of certainty: programs either compile or they don't; statement are either correct or they're not. C programmers inhabit a more complex and sometimes confusing world. It is possible in C to have *warnings*, (*non-fatal) errors*, and *fatal errors*. Once again, this is a sign that C is neither a teaching language nor a beginner's language. Responsibility for reacting to such messages in the compiler, linker or make facility lies with the programmer.

Errors

Fatal errors are very uncommon, and usually indicate some problem with the compiler rather than the source code. *Non-fatal* errors include the syntax errors with which Pascal programmers are familiar, as well as rarer problems associated with disk and memory access. Unlike Pascal compilers, which normally stop compilation at each error, C compilers will work their way through the source code, building and displaying a list of errors. There will usually be an upper limit to the number of such errors (perhaps 30), at which point the compiler will stop and let the programmer start on the process of finding and correcting the errors that have been identified.

Other errors will occur during the linking phase. A mis-spelt function call in the main program, for instance, will not be detected until the linker attempts to bind in all the pre-defined functions in the various header files. It is only when the linker has checked for the name of the function in the main file, in any of the separately-compiled object files, and in the listed library files, that we will know that it has not been found. Only then can the error be confirmed, and the appropriate error message ("Undefined symbol function_name") generated.

Warnings

Errors provide the programmer with no options, apart from correcting them. *Warnings* in C and C++, on the other hand, offer the programmer a hint or suggestion that something may be 'wrong' (but not syntactically) with a particular piece of code. Warnings are a combination of at least two kinds of problems:

1. Things that the compiler's designers (or the ANSI-C specification) consider good C programming practice. These include indicating where a function does not return a value, and should therefore be declared as void (without a type).

2. Things that might cause run-time errors, such as assigning a value to an uninitialised pointer.

Most compilers offer the programmer the option of ignoring some or all warnings. Beyond this, some compilers (including Turbo C) allow the programmer to control which warnings will be displayed. In effect, this allows the experienced C programmer to 'tune out' those warnings that she knows about and is deliberately generating. Inadvertent problems (such as a division by zero) can still be indicated by the compiler and linker.

Exercise 7

Compare the significance of compiler/linker errors in C with compiler errors in Pascal..

Objective 8: After working through this section you should be able to identify the steps involved in creating a C program

We should now understand the general structure of C programs, and the steps involved in creating and compiling such programs. The simple program [TEMPCONV.C on the sample program disk] in Figure 3 generates a list of Centigrade and Fahrenheit temperatures and prints out a message at the freezing point of water and another at the boiling point of water. Even though you do not know the syntax and purpose of all the statements at this point, the general structure and purpose should be evident. We follow this with a Pascal program to carry out the same operations, allowing you make a simple comparison of the two languages.

The program begins with the usual set of comments defining what the program does. This is followed by an #include statement that gives the program access to an external library (in this case the standard input/libary, stdio) in which the standard screen output function *printf* is defined. This is followed by prototypes for all the user-defined functions in the program.

```
/************************************************************       */
/* This is a temperature conversion program written in C            */
/* This program generates and displays a table of farenheit and     */
/* centigrade temperatures, and lists the freezing and boiling      */
/* of water.                                 */
/* ***********************************************************      */

#include "stdio.h"     /* access the stdio library                  */
#include "conio.h"     /* access the conio library                  */

#define FREEZING 0     /* define 'constants'                        */
#define BOILING 100

             /* prototypes          */
void hotcold(int centigrade);

void main()         /* start of (untyped) main function             */
{
             /* local declarations within main                      */
int count;          /* loop control variable                        */
int farenheit;         /* temperature in farenheit degrees          */
int centigrade;        /* temperature in centigrade degrees         */

clrscr();
printf("Centigrade to Farenheit temperature table\n\n");
for (count = -2;count <= 12;count = count + 1)
   {  centigrade = 10 * count;
    farenheit = 32 + (centigrade * 9)/5;
          /* print the results      */
    printf("  C =%4d   F =%4d  ",centigrade,farenheit);
    hotcold(centigrade);
       /* print messages at freezing and boiling points             */
   }      /* end of for loop        */
}       /* end of main function     */

void hotcold(temperature)
          /* function to display messages at 0/100                  */
```

```
int temperature;
{
   if (temperature == FREEZING)
      printf("    <= Freezing Point of Water - Brrrr!\n");
    else
    if (temperature == BOILING)
       printf("    <= Boiling Point of Water - Owwww!\n");
    else
       printf("\n");
}   /* end of function hotcold   */
/*  end of program            */
```

Figure 3: Temperature Conversion Program in C

As with most languages, good formatting in C is important. You will eventually develop your own formatting style, but this is a good way to start. Note the use of comments (text between /* and */). If you observe the for loop, you will notice that all the contents of the compound statement are indented 3 spaces to the right of the for reserved word *for*, and the closing brace is lined up under the f in for. This should be familiar from Pascal, and makes debugging easier because the overall structure becomes more obvious. You will also notice that the *printf* statements that are in the *if* statements within the big for loop are indented three additional spaces because they are part of another construct.

Figure 4 presents the same program written in Turbo Pascal.

```
PROGRAM temperature_conversion;
(* ***********************************************************      *)
(* This is a temperature conversion program written in C            *)
(* This program generates and displays a table of farenheit and     *)
(* centigrade temperatures, and lists the freezing and boiling      *)
(* of water.                                                        *)
(* *********************************************************        *)

USES
    Crt;        { access system crt unit for clrscr }

CONST           { declare global constants }
     boiling_point  = 100;
     freezing_point =   0;

VAR                 { declare global identifiers }
   count: INTEGER;
   farenheit, centigrade: REAL;

PROCEDURE hotcold(temperature: INTEGER);
          (* display messages at 0/100 *)
BEGIN
   IF (centigrade = 0)  THEN
      WRITELN('    <= Freezing Point of Water - Brrrr!')
   ELSE
      IF (centigrade = 100) THEN
         WRITELN('    <= Boiling point of Water! -  Owwww!')
      ELSE
         WRITELN;
END;

BEGIN
   ClrScr;
   WRITELN('Centigrade to Farenheit temperature table');
   WRITELN;
```

```
    FOR count:= -2 TO 12 DO
        BEGIN
          centigrade:= 10 * count;
           farenheit:= 32 + (centigrade * 9) /5;
       (* print the results  *)
      WRITE('  C = ',centigrade:5:1,'      F = ',farenheit:5:1);
      hotcold(centigrade)
     END;
           (* end of for loop  *)
END.            (* end of program   *)
```

Figure 3: Temperature Conversion Program in Pascal

The similarities between the two languages (at least at this simple level) are apparent. The differences will gradually become clearer as we examine the language in depth in the next three Modules.