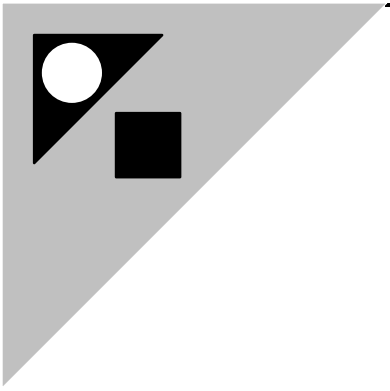

Module 813

Fundamental Programming Structures in C

M. Campbell

Aim

After working through this module you should be able to create, compile and run a simple C program.

Learning objectives

After working through this module you should be able to:

1. Identify the structural elements and layout of C source code.
2. Recall the rules for constructing C identifiers and using data types.
3. Use assignment statements and operators to store and manipulate data in a C program.
4. Identify and apply the C control structures.
5. Describe the techniques for creating program modules in C.
6. Create C functions.
7. Make use of local and global parameters in functions.
8. Create recursive functions in C.
9. Use define and macro statements to simplify code development.

Content

Program structure and layout.
Identifiers and C data types.
Assignment and Conversion statements.
Arithmetic and logic operators.
Control structures.
Creating modules in C.
Functions.
Parameters and scoping in C functions.
Recursion in C.
Using define and macro statements.

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

Turbo C/C++ Manuals.

Turbo C/C++ MS DOS compiler.

Introduction to Module 813

In this module you will be introduced to the fundamental structures used to build a C program. In the previous module (Module 812: Program Development in C) you discovered the general structure of a C program and identified the various components needed to compile the source code into an executable file. This module describes the essential program elements that are common to all C programs. Future modules will discuss specific concepts that are needed by some programs to deal with more complex problems.

The concepts presented in this module provide the basic elements needed to construct a working program. In completing this, and subsequent modules, you should make use of the knowledge gained in the modules of previous units. ASNS1101 (Introduction to Computing and Numerical Processes) provided a general introduction to computer systems and application software. ASCO1102 (Data Structures and Language Design) introduced the class of computer languages called third generation languages (3GL's), the concepts of data and file structures, and completed the study of the Pascal programming language. ASCO2202 (Language Design and Computer Hardware) developed an understanding of the architecture of machines and the compilation process. Finally, ASCO3301 (Software Engineering) developed some models of program design and construction, particularly for large projects.

It is a strong belief in computing that the best method for learning a new language is practice. Your previous studies have prepared you in terms of programming principles and program construction. In this, and future modules, you should attempt to write as many programs as possible to develop a good understanding of C. A requirement of at least one new program (designed, coded, compiled and debugged) each day should be a reasonable and attainable goal. The programs do not need to be complex. You should make sure that each program makes use of only one new concept. Over time you will develop a familiarity with the language constructs that will eventually lead to more complex programs. C is a 3GL and therefore has many constructs in common with other 3GLs such as Pascal. You will find that many programs can be converted into C quite easily. It is only a matter of identifying the differences in command syntax. There are times, however, when this conversion process is difficult, or it produces code which is more complex than the original. In these instances direct conversion is not possible and a redesign of the module is necessary.

These notes make use of a number of program examples. Each example is provided on a floppy disk which is part of the unit materials. You should load, compile and run these programs to discover how they

work. You can also use these examples as prototypes for the design of your own programs. In order to conserve paper, the programs have been written with a minimum of documentation. Your programs would normally be developed with more substantial documentation.

Objective 1 After working through this module you should be able to identify the structural elements and layout of C source code.

A C program

The best way to get started with C is to look at a program. Consider the sample program [TRIVIAL.C]:

```
void main()
{
}
```

This is the simplest possible C program. There is no way to simplify this program or leave anything out. Unfortunately, the program doesn't actually *do* anything. The word *main* is very important, and must appear once (and only once) in every C program. This is the point where execution is begun when the program is run. It does not have to be the first statement in the program but it must exist as the entry point. The use of the term *main* is actually a function definition. Functions must return a value and the term *void* is used to denote that the main program does not return a value.

Following the "main" program name is a pair of parentheses, which are an indication to the compiler that this is a function. As you will see, all C programs consist essentially of a collection of functions. The two *braces* are used to define the limits of the program itself: in this case, the main function. The actual program statements go between the two braces, and in this case there are no statements because the program does absolutely nothing. You can compile and run this program, but since it has no executable statements, it does nothing. Keep in mind, however, that it *is* a valid C program.

A program that does something

For a much more interesting program, examine the program [WRTSOME.C]:

```
#include "stdio.h"
void main()
{
   printf("This is a line of text to output.");
}
```

It is the same as the previous program except that it has one executable statement between the braces. The executable statement is another, pre-defined function. Once again, we will not worry about what a function is, but only how to use this one. In order to output text to the monitor, the text is put within the function parentheses and bounded by quotation marks. The definition of this function is defined in the STDIO library (the Standard Input/Output library). To compile and link this

program the *#include* statement is required. The end result is that whatever is included between the quotation marks in the printf function will be displayed on the monitor when the program is run. Notice the semi-colon at the end of the line. C uses a semi-colon as a statement terminator, so the semi-colon is required as a signal to the compiler that this line is complete. This program is also executable, so you can compile and run it to see if it does what you think it should.

Program output

Look at the following program [WRTMORE.C] for an example of more output and another small but important concept:

```
#include "stdio.h"
void main()
{
    printf("This is a line of text to output.\n");
    printf("And this is another ");
    printf("line of text.\n\n");
    printf("This is a third line.\n");
}
/* This is a comment ignored by the compiler */
```

You will see that there are four program statements in this program, each one being a *printf* function statement. The top line will be executed first, then the next, and so on, until the fourth line is complete. The statements are executed in order from top to bottom. Notice the character near the end of the first line, namely the backslash (\). The backslash is used in the printf statement to indicate a special control character is following. In this case, the 'n' indicates that a *newline* is requested. This is an indication to return the cursor to the left side of the monitor and move down one line. It is commonly referred to as a carriage return/line feed. You can put a newline character at any place within the text; you could even put it in the middle of a word and split the word between two lines. The C compiler considers the combination of the backslash and letter n as a single character.

A complete description of this program is now possible. The first printf outputs a line of text and returns the carriage. The second printf outputs a line but does not return the carriage so the third line is appended to that of the second, then followed by two carriage returns, resulting in a blank line. Finally the fourth printf outputs a line followed by a carriage return and the program is complete. Compile and run this program to see if it does what you expect it to do. It would be a good idea at this time for you to experiment by adding additional lines of printout to see if you understand how the statements really work.

Adding comments

Observe the next program [COMMENTS.C] for an example of how comments can be added to a C program:

```
/* This is the first comment line */
#include "stdio.h"
void main() /* This comment is ignored by the compiler */
{
   printf("We are looking at how comments are ");
               /* A comment is
                  allowed to be
                  continued on
                  another line */
   printf("used in C.\n");
}
/* One more comment for effect */
```

Comments are added to make a program more readable but the compiler must ignore them. The slash/star (/* .. */) combination is used in C for comment *delimiters*. Please note that the program does not illustrate good commenting practice, but is intended to illustrate where comments can go in a program. It is a very sloppy looking program.

The first slash star combination introduces the first comment and the star slash at the end of the first line terminates this comment. Note that this comment is prior to the beginning of the program illustrating that a comment can precede the program itself. Good programming practice would include a comment prior to the program with a short introductory description of the program. The next comment is after the main() program entry point and prior to the opening brace for the program code itself. The third comment starts after the first executable statement and continues for four lines. This is perfectly legal because a comment can continue for as many lines as desired until it is terminated. Note carefully that if anything were included in the blank spaces to the left of the three continuation lines of the comment, it would be part of the comment and would not be compiled. The last comment is located following the completion of the program, illustrating that comments can go nearly anywhere in a C program.

Experiment with this program by adding comments in other places to see what will happen. Comment out one of the printf statements by putting comment delimiters both before and after it and see that it does not get printed out.

Comments are very important in any programming language because you will soon forget what you did and why you did it. It will be much easier to modify or fix a well commented program a year from now than one with few or no comments. You will very quickly develop your own personal style of commenting. Some compilers allow you to nest comments which can be very handy if you need to comment out a section of code during debugging; check your compiler documentation for the availability of this feature with your particular compiler.

Good formatting style

Here is an example [GOODFORM.C] of a well formatted program:

```
#include "stdio.h"
void main()  /* Main program starts here */
{
    printf("Good form ");
    printf       ("can aid in ");
    printf              ("understanding a program.\n");
    printf("And bad form ");
    printf       ("can make a program ");
    printf              ("unreadable.\n");
}
```

Even though it is very short and therefore does very little, it is very easy to see at a glance what it does. With the experience you have already gained in this module, you should be able to very quickly grasp the meaning of the program in its entirety. Your C compiler ignores all extra spaces and all carriage returns giving you considerable freedom concerning how you format your program. Indenting and adding spaces is entirely up to you and is a matter of personal taste. Observe, by contrast, the next program [UGLYFORM.C]; how long would it take you to discover what this program will do?

```
#include "stdio.h"
void main()  /* Main program starts here */{printf("Good
form ");printf
("can aid in ");printf("understanding a program.\n")
;printf("And bad form ");printf("can make a program ");
printf("unreadable.\n");}
```

It doesn't matter to the compiler which format style you use, but it will matter to you when you try to debug your program. Don't get too worried about formatting style yet; you will have plenty of time to develop a style of your own as you learn the language. Be observant of styles as you see C programs in magazines, books, and other publications.

Exercise 1

Write a program to display your name on the monitor.

Modify the program to display your address and phone number on separate lines by adding two additional "printf" statements.

Objective 2 After working through this module you should be able to recall the rules for constructing C identifiers and using data types.

Identifiers

Before you can do anything in any language, you must at least know how you name an *identifier*. An identifier is applied to any variable, function, data definition, etc. In C an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline character, and the remainder being made up of combination of letters of the alphabet, numeric digits, or the underline. Two rules must be kept in mind when naming identifiers:

1. The case of alphabetic characters is significant. Using INDEX for a variable is **not** the same as using index and neither of them is the same as using InDeX for a variable. All three refer to different variables.

2. Up to 31 characters can be used and will be considered significant. If more than 31 are used, they will be ignored by the compiler. You can reduce the maximum number used to something less than 31 if you desire via a compiler option. You should not do this for the duration of your study of this tutorial as you could get some odd compiler diagnostics.

Using underlines

Even though the underline can be used as part of a variable name, it seems not to be used very much by experienced C programmers. It adds greatly to the readability of a program to use descriptive names for variables and it would be to your advantage to do so. Pascal programmers tend to use long descriptive names, but most C programmers tend to use short cryptic names. Most of the example programs in this unit use very short names for that reason.

Working with data

The basic data structure used in C programs is a number. C provides for two types of numbers - integers and floating point. Computer operations that involve text (characters, words or strings) still manipulate numbers.

Each data item used by a program must have a *data type*. The basic data types provided by C are shown in Table 1. Each data type represents a different kind of number. You must choose an appropriate type for the kind of number you need in the variable declaration section of the program.

Data Type	Data Range	Description
int	-32,768 to 32,767	integer value
long int	-2,147,483,648 to 2,147,483,647	integer value with an extended range
short int	-32,768 to 32,767	exactly the same as *int*
unsigned int	0 to 65,535	integer value
char	0 to 255	character data
float	3.4×10^{-38} to $3.4\times10^{+38}$	floating point number
double	1.7×10^{-308} to $1.7\times10^{+308}$	double precision floating point number

Table 1

Integer data types

The next example [ONEINT.C] shows how to work with data in a C program.

```
#include "stdio.h"
main()
{
int index;
    index = 13;
    printf("The value of the index is %d\n",index);
    index = 27;
    printf("The value of the index is %d\n",index);
    index = 10;
    printf("The value of the index is %d\n",index);
}
```

The first new thing we encounter is the line containing int index, which is used to define an integer variable named index. The *int* is a reserved word in C, and can therefore not be used for anything else. It defines a variable that can have a value from -32768 to 32767 on most microcomputer implementations of C. The variable name index can be any name that follows the rules for an identifier and is not one of the reserved words for C. The final character on the line, the semi-colon, is the statement terminator used in C.

We will see later that additional integers could also be defined on the same line, but we will not complicate the present situation. Observing the main body of the program, you will notice that there are three statements that assign a value to the variable index, but only one at a time. The first one assigns the value of 13 to index, and its value is printed out. (We will see how shortly.) Later, the value of 27 is assigned to index, and finally 10 is assigned to it, each value being printed out. It should be intuitively clear that index is indeed a variable

and can store many different values. Please note that the words 'printed out' are usually used to mean 'displayed on the monitor'. You will find that in many cases experienced programmers take this liberty, probably due to the printf function being used for monitor display.

Much more will be covered at a later time on all aspects of input and output formatting. A reasonably good grasp of this topic is necessary in order to understand the following lessons. It is not necessary to understand everything about output formatting at this time, only a fair understanding of the basics.

Additional data types

To illustrate how some additional data types can be used, look at the next program [MORTYPES.C]. Once again we have defined a few integer type variables which you should be fairly familiar with by now, but we have added two new types, the *char*, and the *float*.

```
/* This file introduces additional data types                    */
void main()
{
int a,b,c;   /* -32768 to 32767 with no decimal point            */
char x,y,z;  /* 0 to 255 with no decimal point                   */
float num,toy,thing;   /* 10E-38 to 10E+38 with
                          decimal point                          */
   a = b = c = -27;
   x = y = z = 'A';
   num = toy = thing = 3.6792;
   a = y;       /* a is now 65 (character A)                     */
   x = b;       /* x will now be a funny number                  */
   num = b;     /* num will now be -27.00                        */
   a = toy;     /* a will now be 3                               */
}
```

The char type of data is nearly the same as the integer except that it can only be assigned values between zero and 255, since it is stored in only one byte of memory. The char type of data is usually used for ASCII data, more commonly known as text. The text you are reading was originally written on a computer with a word processor that stored the words in the computer one character per byte. By contrast, the integer data type is stored in two bytes of computer memory on most microcomputers.

Data type mixing

It would be profitable at this time to discuss the way C handles the two types char and int. Most functions in C that are designed to operate with integer type variables will work equally well with character type variables because they are a form of an integer variable. Those functions, when called on to use a char type variable, will actually promote the char data into integer data before using it. For this reason, it is possible to mix char and int type variables in nearly any way you

desire. The compiler will not get confused, but you might! It is good not to rely on this too much, but to carefully use only the proper types of data where they should be used.

The second new data type is the float type of data, commonly called floating point data. This is a data type which usually has a very large range and a large number of significant digits, so a large number of computer words (four in Turbo C) are required to store it. The float data type has a decimal point associated with it and has an allowable range (in Turbo C) of 3.4×10^{-38} to $3.4\times10^{+38}$, with 7 significant digits.

Using data types

The first three lines of the program assign values to all nine of the defined variables so we can manipulate some of the data between the different types. Since, as mentioned above, a char data type is in reality an integer data type, no special considerations need be taken to promote a char to an int, and a char type data field can be assigned to an "int" variable. When going the other way, there is no standard, so you may simply get garbage if the value of the integer variable is outside the range of the "char" type variable. It will translate correctly if the value is within the range of -128 to 127.

The third line illustrates the simplicity of translating an integer into a float: simply assign it the new value and the system will do the proper conversion. When going the other way however, there is an added complication. Since there may be a fractional part of the floating point number, the system must decide what to do with it. By definition, it will truncate it.

This program produces no output, and we haven't covered a way to print out char and float type variables yet, so you can't really get in to this program and play with the results, but the next program will cover this.

Variable types

The next program [LOTTYPES.C] contains every standard simple data type available in the programming language C. There are other types, notably the compound types (i.e. arrays and structures) that we will cover in due time.

```
#include "stdio.h"
void main()
{
int a;           /* simple integer type                    */
long int b;      /* long integer type                      */
short int c;     /* short integer type                     */
unsigned int d;  /* unsigned integer type                  */
char e;          /* character type                         */
float f;         /* floating point type                    */
```

```
double g;        /* double precision floating point            */

   a = 1023;
   b = 2222;
   c = 123;
   d = 1234;
   e = 'X';
   f = 3.14159;
   g = 3.1415926535898;
   printf("a = %d\n",a);     /* decimal output                */
   printf("a = %o\n",a);     /* octal output                  */
   printf("a = %x\n",a);     /* hexadecimal output            */
   printf("b = %ld\n",b);    /* decimal long output           */
   printf("c = %d\n",c);     /* decimal short output          */
   printf("d = %u\n",d);     /* unsigned output               */
   printf("e = %c\n",e);     /* character output              */
   printf("f = %f\n",f);     /* floating output               */
   printf("g = %f\n",g);     /* double float output           */
   printf("\n");
   printf("a = %d\n",a);     /* simple int output             */
   printf("a = %7d\n",a);    /* use a field width of 7        */
   printf("a = %-7d\n",a);   /* left justify width = 7        */
   printf("\n");
   printf("f = %f\n",f);     /* simple float output           */
   printf("f = %12f\n",f);   /* use field width of 12         */
   printf("f = %12.3f\n",f); /* use 3 decimal places          */
   printf("f = %12.5f\n",f); /* use 5 decimal places          */
   printf("f = %-12.5f\n",f);/* left justify in field  */
}
```

First we define a simple int, followed by a long int which has a range of -2147483648 to 2147483647, and a short int which has a range that is identical to that for the int variable, namely -32768 to 32767 (in most compilers). The unsigned int is next and is defined as the same size as the int but with no sign. The unsigned int will usually therefore cover a range of 0 to 65535. It should be pointed out that when the long, short, or unsigned is desired, the actual word int is optional and is left out by most experienced programmers.

We have already covered the char and the float, which leaves only the *double*. The double usually covers a greater range than the float and has more significant digits for more precise calculations. It also requires more memory to store a value than the simple float. The double covers the range 1.7×10^{-308} to $1.7\times10^{+308}$. Note that other compounding of types can be done such as long unsigned int, unsigned char, etc; see the compiler manual for more details.

Another diversion is in order at this point. Most C compilers have no provision for floating point math, but only double floating point math. They will promote a float to a double before doing calculations and therefore only one math library will be needed. Of course, this is totally transparent to you, so you don't need to worry about it. You may think that it would be best to simply define every floating point variable as double, since they are promoted before use in any calculations, but that may not be a good idea. A float variable requires 4 bytes of storage and a double requires 8 bytes of storage, so if you have a large volume of

floating point data to store, the double will obviously require much more memory.

After defining the data types, a numerical value is assigned to each of the defined variables in order to demonstrate the means of outputting each to the monitor.

Exercises 2

Write a program which displays an integer value in a number of formats, i.e. hexadecimal, binary and octal form.

The purpose of this exercise is to help you become familiar with some of the error messages generated by the compiler. Create the following source code program and ensure that it compiles and runs correctly.

```
#include "stdio.h"
void main()
{
int a, b, c;
a = 1;
b = 2;
c = 3;
printf("Some output: %d %d %d\n", a, b, c);
```

Now introduce each of the following programming errors in turn, compile the program, and record the error messages generated.

- change the first comma in the declaration to a semi-colon
- change `printf` to `prinf`
- remove the second " in the `#include` statement
- replace the list `a, b, c` by `a, b, c, c`
- remove the semi-colon from the `printf` statement
- remove the closing brace.

Objective 3 After working through this module you should be able to use assignment statements and operators to store and manipulate data in a C program.

Assignment statements

Let's move on to examine assignment statements. In the first program [INTASIGN.C] three variables are defined for use, and the rest of the program is merely a series of illustrations of various assignments.

```
/* This program illustrates the assignment statement          */
void main()
{
int a,b,c;   /* Integer variables for examples                */
/* data assignment statements                                 */
   a = 12;
   b = 3;
/* data manipulation statements                               */
   c = a + b;       /* simple addition                        */
   c = a - b;       /* simple subtraction                     */
   c = a * b;       /* simple multiplication                  */
   c = a / b;       /* simple division                        */
   c = a % b;       /* simple modulo (remainder)              */
   c = 12*a + b/2 - a*b*2/(a*c + b*2);
   c = c/4+13*(a + b)/3 - a*b + 2*a*a;
   a = a + 1;       /* incrementing a variable                */
   b = b * 5;
   a = b = c = 20;  /* multiple assignment                    */
   a = b = c = 12*13/4;
}
```

The data assignment statements assign numerical values to a and b, and the next four lines illustrate the five basic arithmetic functions and how to use them. The fifth is the modulo operator and gives the remainder if the two variables were divided. It can only be applied to int or char type variables, and of course int extensions such as long, short, etc. Following these, there are two lines illustrating how to combine some of the variables in some complex math expressions. All of the above examples should require no comment except to say that none of the equations are meant to be particularly useful except as illustrations.

The next two expressions are perfectly acceptable as given, but we will see later in this module (objective 10) that there is another way to write these for more compact code.

This leaves us with the last two lines which may appear very strange to you. The C compiler scans the assignment statement from right to left, (which may seem a bit odd since we do not read that way), resulting in a very useful construct, namely the one given here. The compiler finds the value 20, assigns it to c, then continues to the left finding that the latest result of a calculation should be assigned to b. Thinking that the latest calculation resulted in a 20, it assigns it to b also, and continues the leftward scan assigning the value 20 to a also. This is a very useful

construct when you are initialising a group of variables. The last statement illustrates that it is possible to actually do some calculations to arrive at the value which will be assigned to all three variables.

The program has no output, so compiling and executing this program will be very uninteresting. Since you have already learnt how to display some integer results using the printf function, it would be to your advantage to add some output statements to this program to see if the various statements do what you think they should do.

This would be a good time for a preliminary definition of a rule to be followed in C. The data definitions are always given before any executable statements in any program block. This is why the variables are defined first in this program and in any C program. If you try to define a new variable after executing some statements, the compiler will issue an error.

Arithmetic Operators

The basic arithmetic operators have been introduced in the previous section. Addition (+), subtraction (-), multiplication (*) and division (/) behave in much the same way as other languages. Modular division is achieved using the % operator. The result of an arithmetic operation is determined by the data type of the receiving variable, rather than the data types of the operands. For example, in the following statement:

```
x = 12.678 * 13;
```

the result of the operation will be 164.814 if *x* is defined as a float, and 164 if *x* is defined as an int. Note that the *type casting* occurs after the calculation has been performed.

There are three constructs used in C that make no sense at all when first encountered because they are not intuitive, but they greatly increase the efficiency of the compiled code and are used extensively by experienced C programmers. Here is an example [CRYPTIC.C] of their use:

```
void main()
{
int x = 0,y = 2,z = 1025;
float a = 0.0,b = 3.14159,c = -37.234;
/* incrementing                          */
   x = x + 1;      /* This increments x                        */
   x++;            /* This increments x                        */
   ++x;            /* This increments x                        */
   z = y++;        /* z = 2, y = 3 */
   z = ++y;        /* z = 4, y = 4 */
/* decrementing                          */
   y = y - 1;      /* This decrements y                        */
   y--;            /* This decrements y                        */
   --y;            /* This decrements y                        */
   y = 3;
   z = y--;        /* z = 3, y = 2 */
   z = --y;        /* z = 1, y = 1 */
```

```
/* arithmetic operations            */
   a = a + 12;          /* This adds 12 to a              */
   a += 12;         /* This adds 12 more to a                  */
   a *= 3.2;        /* This multiplies a by 3.2                */
   a -= b;          /* This subtracts b from a                 */
   a /= 10.0;       /* This divides a by 10.0                  */
/* conditional expression           */
   a = (b >= 3.0 ? 2.0 : 10.5 );/* This expression             */
   if (b >= 3.0)       /* And this expression                  */
      a = 2.0;         /* are identical, both                  */
   else                /* will cause the same                  */
      a = 10.5;        /* result.   */
   c = (a > b?a:b);    /* c will have the max of a or b        */
   c = (a > b?b:a);    /* c will have the min of a or b        */
}
```

In this program some variables are defined and initialised in the same statements for later use. The first executable statement simply adds 1 to the value of x, and should come as no surprise to you. The next two statements also add one to the value of x, but it is not intuitive that this is what happens. It is simply by definition that this is true. Therefore, by definition of the C language, a double plus sign either before or after a variable increments that variable by 1. Additionally, if the plus signs are before the variable, the variable is incremented before it is used, and if the plus signs are after the variable, the variable is used, then incremented. In the next statement, the value of y is assigned to the variable z, then y is incremented because the plus signs are after the variable y. In the last statement of the incrementing group of example statements, the value of y is incremented then its value is assigned to the variable z.

The next group of statements illustrate decrementing a variable by one. The definition works exactly the same way for decrementing as it does for incrementing. If the minus signs are before the variable, the variable is decremented, then used, and if the minus signs are after the variable, the variable is used, then decremented.

Another useful but cryptic operator is the *arithmetic operator*. This operator is used to modify any variable by some constant value. The first statement of the arithmetic operator group of statements simply adds 12 to the value of the variable a. The second statement does the same, but once again, it is not intuitive that they are the same. Any of the four basic functions of arithmetic (+, -, *, or /) can be handled in this way, by putting the function desired in front of the equal sign and eliminating the second reference to the variable name. It should be noted that the expression on the right side of the arithmetic operator can be any valid expression; the examples are kept simple for your introduction to this new operator.

Just like the incrementing and decrementing operators, the arithmetic operator is used extensively by experienced C programmers and it would pay you well to understand it.

The conditional expression

The conditional expression is just as cryptic as the last two, but once again it can be very useful so it would pay you to understand it. It consists of three expressions within parentheses separated by a question mark and a colon. The expression prior to the question mark is evaluated to determine if it is true or false. If it is true, the expression between the question mark and the colon is evaluated, and if it is not true, the expression following the colon is evaluated. The result of the evaluation is used for the assignment. The final result is identical to that of an if statement with an else clause. This is illustrated by the second example in this group. The conditional expression has the added advantage of more compact code that will compile to fewer machine instructions in the final program.

The final two lines of this example program are given to illustrate a very compact way to assign the greater of two variables a or b to c, and to assign the lesser of the same two variables to c. Notice how efficient the code is in these two examples.

Many students of C have stated that they didn't like these three cryptic constructs and that they would simply never use them. This would be fine if they never have to read anybody else's program, or use any other programs within their own. I have found many functions that I wished to use within a program but needed a small modification to use it, requiring me to understand another person's code. It would therefore be to your advantage to learn these new constructs, and use them. They will be used in the remainder of this tutorial, so you will be constantly exposed to them.

Logic Operators

The next program [COMPARES.C] has many examples of compare statements in C. We begin by defining and initialising nine variables to use in the ensuing compare statements. This initialisation is new to you and can be used to initialise variables while they are defined.

```
void main()
{
int x = 11,y = 11,z = 11;
char a = 40,b = 40,c = 40;
float r = 12.987,s = 12.987,t = 12.987;
/* First group of compare statements   */
   if (x == y) z = -13;   /* This will set z = -13            */
   if (x > z)  a = 'A';   /* This will set a = 65             */
   if (!(x > z)) a = 'B'; /* This will change nothing         */
   if (b <= c) r = 0.0;   /* This will set r = 0.0            */
   if (r != s) t = c/2;   /* This will set t = 20             */
/* Second group of compare statements                         */
   if (x = (r != s)) z = 1000;
      /* This sets x = some positive number and z = 1000      */
   if (x = y) z = 222;    /* Sets x = y, and z = 222          */
   if (x != 0) z = 333;   /* This sets z = 333                */
```

```
   if (x) z = 444;        /* This sets z = 444                  */
/* Third group of compare statements   */
   x = y = z = 77;
   if ((x == y) && (x == 77)) z = 33;
      /* This sets z = 33                                       */
   if ((x > y) || (z > 12))   z = 22;
      /* This sets z = 22                                       */
   if (x && y && z) z = 11;  /* This sets z = 11                */
   if ((x = 1) && (y = 2) && (z = 3)) r = 12.00;
      /* This sets x = 1, y = 2, z = 3, r = 12.00               */
   if ((x == 2) && (y = 3) && (z = 4)) r = 14.56;
      /* This doesn't change anything                           */
/* Fourth group of compares                                     */
   if (x == x);  z = 27.345;  /* z always gets changed         */
   if (x != x)   z = 27.345;  /* Nothing gets changed          */
   if (x = 0)    z = 27.345; /* Sets x = 0, z unchanged        */
}
```

The first group of compare statements represents the simplest kinds of compares since they simply compare two variables. Either variable could be replaced with a constant and still be a valid compare, but two variables is the general case. The first compare checks to see if x is equal to y and it uses the double equal sign for the comparison. A single equal sign could be used here but it would have a different meaning as we will see shortly. The second comparison checks to see if x is greater than z. The third introduces the NOT operator, the exclamation, which can be used to invert the result of any logical compare. The fourth checks for ‘b less than or equal to c’, and the last checks for ‘r not

As with other programming languages, if the result of the compare is true, the statement following the if clause will be executed and the results are given in the comments. Note that ‘less than’ and ‘greater than or equal to’ are also available, but are not illustrated here.

It would be well to mention the different format used for the if statement in this example program. A carriage return is not required as a statement separator and putting the conditional clause on the same line as the if adds to the readability of the overall program. A more detailed discussion of the ‘if’ structure can be found in the next objective.

The compares in the second group are a bit more involved. Starting with the first compare, we find a rather strange looking set of conditions in the parentheses. To understand this we must understand just what a true or false is in C. A false is defined as a value of zero, and true is defined as a non-zero value. Any integer or char type of variable can be used for the result of a true/false test, or the result can be an implied integer or char.

Look at the first compare of the second group of compare statements. The expression r!= s will evaluate as a true since r was set to 0.0 above, so the result will be a non-zero value, probably 1. Even though the two

variables that are compared are float variables, the result will be of type integer. There is no explicit variable to which it will be assigned so the result of the compare is an implied integer. Finally the resulting number (1 in this case) is assigned to the integer variable x. If double equal signs were used, the phantom value, namely 1, would be compared to the value of x, but since the single equal sign is used, the value 1 is simply assigned to x, as though the statement were not in parentheses. Finally, since the result of the assignment in the parentheses was non-zero, the entire expression is evaluated as true, and z is assigned the value of 1000.

Thus we accomplished two things in this statement: we assigned x a new value, probably 1, and we assigned z the value of 1000. We covered a lot in this statement so you may wish to review it before going on. The important things to remember are the values that define true and false, and the fact that several things can be assigned in a conditional statement. The value assigned to x was probably a 1 but remember that the only requirement is that it is non-zero. The next example should help clear up some of the above in your mind. In this example, x is assigned the value of y, and since the result is 11, the condition is non-zero, which is true, and the variable z is therefore assigned 222.

The third example, in the second group, compares x to zero. If the result is true, meaning that if x is not zero, z is assigned the value of 333, which it will be. The last example in this group illustrates the same concept, since the result will be true if x is non-zero. The compare to zero is not actually needed and the result of the compare is true. The third and fourth examples of this group are therefore identical.

The third group of compares introduce some additional concepts, namely the logical AND and the logical OR. We assign the value of 77 to the three integer variables simply to get started again with some defined values. The first compare of the third group contains the new control &&, which is the logical AND. The entire statement reads, if x equals y AND if x equals 77 then the result is true. Since this is true, the variable z is set equal to 33.

The next compare in this group introduces the || operator which is the OR. The statement reads, if x is greater than y OR if z is greater than 12 then the result is true. Since z is greater than 12, it doesn't matter if x is greater than y or not, because only one of the two conditions must be true for the result to be true. The result is true, so therefore z will be assigned the value of 22.

Logical evaluation

When a compound expression is evaluated, the evaluation proceeds from left to right and as soon as the result of the outcome is assured, evaluation stops. Thus, in the case of an AND evaluation, when one of the terms evaluates to false, evaluation is discontinued because additional true terms cannot make the result ever become true. In the case of an OR evaluation, if any of the terms is found to be true, evaluation stops because it will be impossible for additional terms to cause the result to be false. In the case of additionally nested terms, the above rules will be applied to each of the nested levels.

Operator precedence

The question immediately arises of the precedence of operators: which operators are evaluated first and which last? There are many rules about this topic, which your compiler will define completely, but I would suggest that you don't worry about it at this point. Instead, use lots of parentheses to group variables, constants, and operators in a way meaningful to you. Parentheses always have the highest priority and will remove any question of which operations will be done first in any particular statements.

Going on to the next example in group three, we find three simple variables used in the conditional part of the compare. Since all three are non-zero, all three are true, and therefore the AND of the three variables are true, leading to the result being true, and z being assigned the value of 11. Note that since the variables r, s, and t are float type variables, they could not be used this way, but they could each be compared to zero and the same type of expression could be used.

Continuing on to the fourth example of the third group we find three assignment statements in the compare part of the if statement. If you understood the above discussion, you should have no difficulty understanding that the three variables are assigned their respective new values, and the result of all three are non-zero, leading to a resulting value of TRUE.

The last example of the third group contains a bit of a trick, but since we have covered it above, it is nothing new to you. Notice that the first part of the compare evaluates to FALSE. The remaining parts of the compare are not evaluated, because it is an AND and it will definitely be resolved as a FALSE because the first term is false. If the program was dependent on the value of y being set to 3 in the next part of the compare, it will fail because evaluation will cease following the FALSE found in the first term. Likewise, z will not be set to 4, and the variable r will not be changed.

Potential problem areas

The last group of compares illustrate three possibilities for getting into a bit of trouble. All three have the common result that z will not get set to the desired value, but for different reasons. In the case of the first one, the compare evaluates as true, but the semicolon following the second parentheses terminates the if clause, and the assignment statement involving z is always executed as the next statement. The if therefore has no effect because of the misplaced semicolon. The second statement is much more straightforward because x will always be equal to itself, therefore the inequality will never be true, and the entire statement will never do a thing, but is wasted effort. The last statement will always assign 0 to x and the compare will therefore always be false, never executing the conditional part of the if statement.

The conditional statement is extremely important and must be thoroughly understood to write efficient C programs. If any part of this discussion is unclear in your mind, restudy it until you are confident that you understand it thoroughly before proceeding onward. We have used an assignment statement within the conditional part of the if statements purposely for instructional reasons, but assignments are not generally used in compare statements and the compiler is warning you that you may have intended to use the double equal sign. Compile and run this program, then add some printout to see the results of some of the comparisons.

Exercise 3

The following program contains a run-time error. Locate, explain and fix the error.

```
void main()  {
int x,y;
x = 1/y; }
```

Write a C program that performs the following algorithm.

assign the value 26.87 to age
assign the value 93 to weight
assign the value 183.219 to height
calculate the height-to-weight ratio using the formula:

$$ratio = \frac{height}{weight}$$

display a heading for the results
display the results in a suitable format, for example:

```
age.....................: xx.x years
weight..................: xx kg
height..................: xx.x cm
height-to-weight ratio...: xxx.xx
```

Objective 4 After working through this module you should be able to identify and apply the C control structures.

Repetition structures

The C programming language has several structures for looping and conditional branching. We will cover them all in this section.

The WHILE loop

The while loop continues to loop while some condition is true. When the condition becomes false, the looping is discontinued. It therefore does just what it says it does; the name of the loop being very descriptive. Look at the next program [WHILE.C] for an example of a while loop:

```
/* This is an example of a "while" loop*/
#include "stdio.h"
void main()
{
int count;
    count = 0;
    while (count < 6)  {
        printf("The value of count is %d\n",count);
        count = count + 1;
    }
}
```

We begin with a comment and the program name, then go on to define an integer variable count within the body of the program. The variable is set to zero and we come to the while loop itself. The syntax of a while loop is just as shown here. The keyword *while* is followed by an expression of something in parentheses, followed by a compound statement bracketed by braces. As long as the expression in parenthesis is true, all statements within the braces will be executed. In this case, since the variable count is incremented by one every time the statements are executed, it will eventually reach 6, the statement will not be executed, and the loop will be terminated. The program control will resume at the statement following the statements in braces.

We will cover the compare expression (the one in parentheses) soon; until then, simply accept the expressions for what you think they should do and you will probably be correct.

Several things must be pointed out regarding the while loop. First, if the variable count were initially set to any number greater than 5, the statements within the loop would not be executed at all, so it is possible to have a while loop that is never executed. Secondly, if the variable were not incremented in the loop, then in this case, the loop would never terminate, and the program would never complete. Finally, if

there is only one statement to be executed within the loop, it does not need braces but can stand alone.

The DO-WHILE loop

A variation of the while loop is illustrated in the next program. This program [DOWHILE.C] is nearly identical to the last one except that the loop begins with the reserved word *do,* followed by a compound statement in braces, then the reserved word while, and finally an expression in parentheses.

```
/* This is an example of a do-while loop                          */
#include "stdio.h"
void main()
{
int i;
   i = 0;
   do {
      printf("The value of i is now %d\n",i);
      i = i + 1;
   } while (i < 5);
}
```

The statements in the braces are executed repeatedly as long as the expression in parentheses is true. When the expression in parentheses becomes false, execution is terminated, and control passes to the statements following this statement.

Several things must be pointed out regarding this statement. Since the test is done at the end of the loop, the statements in the braces will always be executed at least once. Secondly, if i were not changed within the loop, the loop would never terminate, and hence the program would never terminate. Finally, just like for the while loop, if only one statement will be executed within the loop, no braces are required.

It should come as no surprise to you that these loops can be *nested.* That is, one loop can be included within the compound statement of another loop, and the nesting level has no limit.

The FOR loop

The for loop is really nothing new, it is simply a new way to describe the while loop. Examine the next program [FORLOOP.C] for an example of a program with a for loop; the loop consists of the reserved word *for* followed by a rather large expression in parentheses.

```
/* This is an example of a for loop */
#include "stdio.h"
void main()
{
int index;
   for(index = 0; index < 6; index = index + 1)
      printf("The value of the index is %d\n",index);
}
```

This expression is really composed of three fields separated by semi-colons. The first field contains the expression ‘index = 0’ and is an initialising field. Any expressions in this field are executed prior to the first pass through the loop. There is essentially no limit as to what can go here, but good programming practice would require it to be kept simple. Several initialising statements can be placed in this field, separated by commas.

The second field, in this case containing ‘index < 6’, is the test which is done at the beginning of each loop through the program. It can be any expression which will evaluate to a true or false. (More will be said about the actual value of true and false in the next section.) The expression contained in the third field is executed each time the loop is executed but it is not executed until after those statements in the main body of the loop are executed. This field, like the first, can also be composed of several operations separated by commas.

Following the for() expression is any single or compound statement which will be executed as the body of the loop. A compound statement is any group of valid C statements enclosed in braces. In nearly any context in C, a simple statement can be replaced by a compound statement that will be treated as if it were a single statement as far as program control goes.

You may be wondering why there are two statements available that do exactly the same thing because the while and the for loop do exactly the same thing. The while is convenient to use for a loop that you don’t have any idea how many times the loop will be executed, and the for loop is usually used in those cases when you are doing a fixed number of iterations. The for loop is also convenient because it moves all of the control information for a loop into one place, between the parentheses, rather than at both ends of the code. It is your choice as to which you would rather use.

Selection structures

The IF statement

The next program [IFELSE.C] demonstrates our first conditional branching statement, the ‘if’.

```
/* An example of the if and the if-else statements            */
#include "stdio.h"
void main()
{
int data;
   for(data = 0;data < 10; data = data + 1)   {
      if (data == 2)
         printf("Data is now equal to %d\n",data);
      if (data < 5)
         printf("Data = %d, (less than 5)\n",data);
```

```
        else
            printf("Data = %d, ( greater than 4)\n",data);
    }  /* end of for loop            */
}
```

Notice first that there is a for loop with a compound statement as its executable part containing two if statements. This is an example of how statements can be nested. It should be clear to you that each of the if statements will be executed 10 times.

Consider the first if statement. It starts with the keyword *if* followed by an expression in parentheses. If the expression is evaluated and found to be true, the single statement following the if is executed, and if false, the following statement is skipped. Here too, the single statement can be replaced by a compound statement composed of several statements bounded by braces. (The expression 'data == 2' is simply asking if the value of data is equal to 2; this will be explained in detail in the next section. Suffice for now that if 'data = 2' were used in this context, it would mean a completely different thing.)

The IF-ELSE statement

The second if is similar to the first with the addition of a new reserved word, the *else* following the first printf statement. This simply says that if the expression in the parentheses evaluates as true, the first expression is executed, otherwise the expression following the else is executed. Thus, one of the two expressions will always be executed, whereas in the first example the single expression was either executed or skipped. Both will find many uses in your C programming efforts.

The BREAK and CONTINUE statements

Now for a program [BREAKCON.C] with examples of two new statements. Notice that in the first for, there is an if statement that calls a break if xx equals 8. The break will jump out of the loop you are in and begin executing statements following the loop, effectively terminating the loop.

```
#include "stdio.h"
void main()
{
int xx;
    for(xx = 5; xx < 15; xx = xx + 1)  {
        if (xx == 8)
            break;
        printf("In the break loop, xx is now %d\n",xx);
    }
    for(xx = 5; xx < 15; xx = xx + 1)  {
        if (xx == 8)
            continue;
        printf("In the continue loop, xx is now %d\n",xx);
    }
}
```

This is a valuable statement when you need to jump out of a loop depending on the value of some results calculated in the loop. In this case, when xx reaches 8, the loop is terminated and the last value printed will be the previous value, namely 7.

The next for loop contains a continue statement which does not cause termination of the loop but jumps out of the present iteration. When the value of xx reaches 8 in this case, the program will jump to the end of the loop and continue executing the loop, effectively eliminating the printf statement during the pass through the loop when xx is eight.

The SWITCH statement

Let's move on to look at a program [SWITCH.C] containing the biggest construct we have so far come across in C - the *switch.*

```
#include "stdio.h"
void main()
{
int truck;
   for (truck = 3; truck < 13; truck = truck + 1)   {
      switch (truck)   {
         case 3 :  printf("The value is three\n");
                   break;
         case 4 :  printf("The value is four\n");
                   break;
         case 5 :
         case 6 :
         case 7 :
         case 8 :  printf("The value is between 5 & 8\n");
                   break;
         case 11   :   printf("The value is eleven\n");
                   break;
         default   :   printf("The value is undefined\n");
                   break;
      }            /*   end of switch  */
   }               /*   end of for loop                        */
}
```

It begins with the keyword *switch* followed by a variable in parentheses which is the switching variable, in this case truck. As many cases as desired are then enclosed within a pair of braces. The reserved word *case* is used to begin each case entered followed by the value of the variable, then a colon, and the statements to be executed. In this example, if the variable truck contains the value 3 during this pass of the switch statement, the printf will cause 'The value is three' to be displayed, and the break statement will cause us to jump out of the switch.

Once an entry point is found, statements will be executed until a break is found or until the program drops through the bottom of the switch braces. If the variable has the value 5, the statements will begin executing where 'case 5 :' is found, but the first statements found are where the case 8 statements are. These are executed and the break statement in the case 8 portion will direct the execution out the bottom

of the switch. The various case values can be in any order and if a value is not found, the default portion of the switch will be executed.

It should be clear that any of the above constructs can be nested within each other or placed in succession, depending on the needs of the particular programming project at hand.

Exercise 4

Write a program that writes your name on the monitor ten times. Write this program three times, once with each looping method.

Write a program that counts from one to ten, prints the values on a separate line for each, and includes a message of your choice when the count is 3 and a different message when the count is 7.

Write a program that will count from 1 to 12 and print the count, and its square, for each count.

```
1    1
2    4
3    9
etc.
```

Write a program that counts from 1 to 12 and prints the count and its inversion to 5 decimal places for each count. This will require a floating point number.

```
1    1.00000
2    0.50000
3    0.33333
4    0.25000
etc.
```

Write a program that will count from 1 to 100 and print only those values between 32 and 39, one to a line.

Objective 5 After working through this module you should be able to describe the techniques for creating program modules in C.

Creating Modules in C

All modules in C are functions. We have already seen that the main module of a C program is a function (called *main*). Each function must return a single value, hence all functions are defined having a particular type. The type associated with each function is the data type of the value being returned by the function.

There are times, however, when a module is required to perform a task but it is not logical to return a value. Examples of this situation include modules which print headings to the screen. C provides a data type to use in these situations. The data type is called *void*. We have used this system when defining the main program module in each of the examples considered so far.

All modules in a main program must be *prototyped*. A prototype is a model of the real thing, and when programming in C you must define a model of each function for the compiler. The compiler can then use the model to check each of your calls to the function and determine if you have used the correct number of arguments in the function call and if they are of the correct type. By using prototypes, you let the compiler do some additional error checking for you. The ANSI standard for C contains prototyping as part of its recommended standard. Every good C compiler will have prototyping available, so you should learn to use it.

Standard function libraries

Every compiler comes with some standard predefined functions which are available for your use. These are mostly input/output functions, character and string manipulation functions, and math functions. We will cover most of these in subsequent sections. Prototypes are defined for you by the compiler writer for all of the functions that are included with your compiler. A few minutes spent studying your compiler's Reference Guide will give you an insight into where the prototypes are defined for each of the functions.

In addition, most compilers have additional functions predefined that are not standard but allow the programmer to get the most out of a particular computer. In the case of the IBM-PC and compatibles, most of these functions allow the programmer to use the BIOS services available in the operating system, or to write directly to the video monitor or to any place in memory. These will not be covered in any

detail as you can study the unique aspects of the compiler on your own. One example of a library of this type is the CONIO library which is provided as part of the Turbo C package. The CONIO library provides functions that allow you to position output on the monitor, change the colour of text written to the monitor and a number of other screen (console) based operations.

Objective 6 After working through this module you should be able to create C functions.

A user-defined function

Examine the following listing [SUMSQRES.C] as an example of a C program with functions. (Actually this is not the first function we have encountered because the 'main' program we have been using all along is technically a function, as is the printf function. The printf function is a *library* function that was supplied with your compiler.)

```
#include "stdio.h"
int sum;        /* This is a global variable                  */
/* define the prototypes for the functions used in this       */
/* program.                          */
void header()          /* This function does not return       */
                       /* a value.  */
void square(int number)   /* The square function uses a       */
                       /* single integer parameter. It        */
                       /* does not return a value.            */
void ending()          /* This function does not return       */
                       /* a value.  */
void main()
{
int index;
   header(); /* This calls the function named header          */
      for (index = 1;index <= 7;index++)
         square(index);     /* Calls the square function     */
      ending();             /* Calls the ending function     */
}
/* -- Function definitions --------------------
void header()      /* This is the function named header       */
{
   sum = 0;        /* Initialise the variable "sum"           */
   printf("The header for the square program\n\n");
}
int square(int number) /* This is the square function         */
{
int numsq;
   numsq = number * number;  /* This produces the square     */
   sum += numsq;
   printf("The square of %d is %d\n", number, numsq);
}
void ending()             /* This is the ending function      */
{
   printf("\nThe sum of the squares is %d\n", sum);
}
```

The executable part of this program begins with a line that simply says header(), which is the way to call any function. The parentheses are required because the C compiler uses them to determine that it is a function call and not simply a misplaced variable. When the program comes to this line of code, the function named header is called, its statements are executed, and control returns to the statement following this call. Continuing on we come to a for loop which will be executed 7 times and which calls another function named square each time through the loop, and finally a function named ending will be called and executed. (For the moment ignore the index in the parentheses of the

call to square.) We have seen that this program therefore calls a header, makes 7 calls to square, and a call to ending. Now we need to define the functions.

Defining functions

Following the main program you will see another program that follows all of the rules set forth so far for a main program except that it is named header(). This is the function which is called from within the main program. Each of these statements are executed, and when they are all complete, control returns to the main program. The first statement sets the variable sum equal to zero because we will use it to accumulate a sum of squares. Since the variable sum is defined as an integer type variable prior to the main program, it is available to be used in any of the following functions. It is called a *global* variable, and its *scope* is the entire program and all functions. More will be said about the scope of variables at the end of this section. The next statement outputs a header message to the monitor. Program control then returns to the main program since there are no additional statements to execute in this function. It should be clear to you that the two executable lines from this function could be moved to the main program, replacing the header call, and the program would do exactly the same thing that it does as it is now written. This does not minimise the value of functions, it merely illustrates the operation of this simple function in a simple way. You will find functions to be crucial in C programming.

Passing a value to a function

Going back to the main program, and the for loop specifically, we find the new construct from the end of the last section used in the last part of the for loop, namely incrementing the index variable using index++. You should get used to seeing this, as you will see it a lot in C programs. In the call to the function square, we have an added feature, namely the variable index within the parentheses. This is an indication to the compiler that when you go to the function, you wish to take along the value of index to use in the execution of that function. Looking ahead at the function square, we find that another variable name is enclosed in its parentheses, namely the variable ‘number’. This is the name we prefer to call the variable passed to the function when we are in the function. We can call it anything we wish as long as it follows the rules of naming an identifier.

Since the function must know what type the variable is, it is defined following the function name but before the opening brace of the function itself. Thus the line containing ‘int number;’ tells the function that the value passed to it will be an integer type variable.

With all that out of the way, we now have the value of index from the main program passed to the function square, but renamed number, and available for use within the function. This is the classic style of defining function variables and has been in use since C was originally defined. A newer method is gaining in popularity due to its many benefits and will be discussed later in this section.

Following the opening brace of the function, we define another variable numsq for use only within the function itself, (more about that later) and proceed with the required calculations. We set numsq equal to the square of number, then add numsq to the current total stored in sum (remember from the last section that 'sum += numsq' is the same as 'sum = sum + numsq'). We print the number and its square, and return to the main program.

More about passing a value to a function

When we passed the value of index to the function a little more happened than meets the eye. We did not actually pass the value of index to the function, we actually passed a copy of the value. In this way the original value is protected from accidental corruption by a called function. We could have modified the variable number in any way we wished in the function square, and when we returned to the main program, index would not have been modified. We thus protect the value of a variable in the main program from being accidentally corrupted, but we cannot return a value to the main program from a function using this technique. We will find a well defined method of returning values to the main program or to any calling function when we get to arrays and another method when we get to pointers. Until then the only way you will be able to communicate back to the calling function will be with global variables. We have already hinted at global variables above, and will discuss them in detail later in this section.

Continuing in the main program, we come to the last function call, the call to ending. This call simply calls the last function which has no local variables defined. It prints out a message with the value of sum contained in it to end the program. The program ends by returning to the main program and finding nothing else to do. Compile and run this program and observe the output.

I told you a short time ago that the only way to get a value back to the main program was through use of a global variable, but there is another way which we will discuss by reference to the next program [SQUARES.C]:

```
#include "stdio.h"
/* define the prototype for the function used in this          */
/* program.                          */
int squ(int in)     /* The square function uses a single       */
```

```
                    /* integer parameter. It returns an         */
                    /* integer value.   */
void main()            /* This is the main program             */
{
int x,y;
   for(x = 0;x <= 7;x++) {
      y = squ(x);      /* Get the value of x*x                          */
      printf("The square of %d is %d\n", x, y);
   }
   for (x = 0;x <= 7;++x)
      printf("The value of %d is %d\n", x, squ(x));
}
/* -- Function definition --------------------
int squ(int in) /* Function to get the square of in            */
{
int square;
   square = in * in;
   return(square);     /* This sets squ() = square             */
}
```

In this program we see that it is simple to return a single value from a called function to the calling function. But once again, it is true that to return more than one value, we will need to study either arrays or pointers. In the main program, we define two integers and begin a for loop which will be executed 8 times. The first statement of the for loop is 'y = squ(x);' which is a new and rather strange looking construct. From past experience, we should have no trouble understanding that the squ(x) portion of the statement is a call to the squ function taking along the value of x as a variable. Looking ahead to the function itself we find that the function prefers to call the variable 'in' and it proceeds to square the value of in and call the result square.

Finally, a new kind of a statement appears, the *return* statement. The value within the parentheses is assigned to the function itself and is returned as a usable value in the main program. Thus, the function call squ(x) is assigned the value of the square and is returned to the main program such that y is then set equal to that value. If x were therefore assigned the value 4 prior to this call, y would then be set to 16 as a result of this line of code.

Another way to think of this is to consider the grouping of characters squ(x) as another variable with a value that is the square of x, and this new variable can be used any place it is legal to use a variable of its type. The values of x and y are then printed out. To illustrate that the grouping of squ(x) can be thought of as just another variable, another for loop is introduced in which the function call is placed in the print statement rather than assigning it to a new variable.

Floating point functions

Now for an example of a function with a floating point type of return [FLOATSQ.C]. It begins by defining a global floating point variable we will use later; then in the main part of the program an integer is defined,

followed by two floating point variables, and then by two strange looking definitions.

```
#include "stdio.h"
float z;           /* This is a global variable              */
/* define the prototype for the function used in this        */
/* program.                          */
float sqr(float inval) /* The square function uses a          */
                       /* single float parameter. It         */
                       /* returns a float value.             */
void main()
{
int index;
float x, y;
   for (index = 0;index <= 7;index++){
      x = index;   /* convert int to float                   */
      y = sqr(x);  /* square x to a float variable           */
      printf("The square of %d is %10.4f\n",index,y);
   }
   for (index = 0; index <= 7;index++) {
      z = index;
      y = glsqr();
      printf("The square of %d is %10.4f\n",index,y);
   }
}
/* -- Function definition --------------------
float sqr(float inval) /* square a float, return a float      */
{
float square;
   square = inval * inval;
   return(square);
}
```

The expressions sqr() and glsqr() look like function calls - and they are. This is the proper way in C to define that a function will return a value that is not of the type int, but of some other type, in this case float. This tells the compiler that when a value is returned from either of these two functions, it will be of type float. This once again is the classic method of defining functions.

Now refer to the function sqr near the centre of the listing and you will see that the function name is preceded by the name float. This is an indication to the compiler that this function will return a value of type float to any program that calls it. The function is now compatible with the call to it. The line following the function name contains ‘float inval;’, which indicates to the compiler that the variable passed to this function from the calling program will be of type float. As is customary with all example programs, compile and run this program.

Exercise 6

Rewrite TEMPCONV.C to move the temperature calculation to a function.

Write a program that writes your name on the monitor 10 times by calling a function to do the writing. Move the called function ahead of the main function to see if the C compiler will allow it.

Objective 7 After working through this module you should be able to make use of local and global parameters in functions.

Scope and Parameters

The next program [SCOPE.C] leads us to a discussion of the scope of variables in a program. Ignore the 4 statements in lines 3 to 5 of this program for a few moments; we will discuss them later.

```
#include "stdio.h"     /* Prototypes for Input/Output          */
/* define the prototypes for the functions used in this program*/
void head1(void);      /* Prototype for head1                  */
void head2(void);      /* Prototype for head2                  */
void head3(void);      /* Prototype for head3                  */
int count;             /* This is a global variable            */
void main()
{
register int index;    /* This variable is available only      */
                       /* in main                              */
   head1();
   head2();
   head3();
/* main "for" loop of this program                             */
   for (index = 8; index > 0; index--) {
      int stuff;   /* This variable is only available         */
                   /* within these braces                     */
      for (stuff = 0; stuff <= 6; stuff++)
         printf("%d ", stuff);
      printf(" index is now %d\n", index);
   }
}
int counter;    /* This is available from this point on        */
/* -- Function definitions ---------------------
void head1(void)
{
/* This variable is only available in head1                    */
int index;
   index = 23;
   printf("The header1 value is %d\n", index);
}
void head2(void)
{
/* This variable is available only in head2                    */
/* and it displaces the global of the same name                */
int count;
   count = 53;
   printf("The header2 value is %d\n", count);
   counter = 77;
}
void head3(void)
{
   printf("The header3 value is %d\n", counter);
}
```

The first variable defined is a global variable count which is available to any function in the program since it is defined before any of the functions. In addition, it is always available because it does not come and go as the program is executed. (This will make more sense shortly.) Further down in the program, another global variable named counter is defined which is also global but is not available to the main program since it is defined following the main program. A global variable is any

variable that is defined outside of any function. Note that both of these variables are sometimes referred to as *external* variables because they are external to any functions.

Return to the main program and you will see the variable index defined as an integer. [Ignore the word register for the moment.] This variable is only available within the main program because that is where it is defined. In addition, it is an *automatic* variable, which means that it only comes into existence when the function in which it is contained is invoked, and ceases to exist when the function is finished. This really means nothing here because the main program is always in operation, even when it gives control to another function.

Another integer is defined within the for braces, namely stuff. Any pairing of braces can contain a variable definition which will be valid and available only while the program is executing statements within those braces. The variable will be an automatic variable and will cease to exist when execution leaves the braces. This is convenient to use for a loop counter or some other very localised variable.

Automatic variables

Observe the function named head1, which looks a little funny because void is used twice; the purpose of the word void will be explained shortly. The function contains a variable named index that has nothing to do with the index of the main program, except that both are automatic variables. When the program is not actually executing statements in this function, this variable named index does not even exist. When head1 is called, the variable is generated, and when head1 completes its task, the variable index is eliminated completely from existence. Keep in mind that this does not affect the variable of the same name in the main program, since it is a completely separate entity.

Automatic variables therefore are automatically generated and disposed of when needed. The important thing to remember is that from one call to a function to the next call, the value of an automatic variable is not preserved and must therefore be reinitialised.

Static variables

An additional variable type must be mentioned at this point, the *static* variable. By putting the reserved word static in front of a variable declaration within a function, the variable or variables in that declaration are static variables and will stay in existence from call to call of the particular function. By putting the same reserved word in front of an external variable, one outside of any function, it makes the variable private and not accessible to use in any other file. This implies that it is

possible to refer to external variables in other separately compiled files, and that is true. Examples of this usage will be given later.

Reusing names

Refer to the function named head2. It contains another definition of the variable named count. Even though count has already been defined as a global variable, it is perfectly all right to reuse the name in this function. It is a completely new variable that has nothing to do with the global variable of the same name, and causes the global variable to be unavailable in this function. This allows you to write programs using existing functions without worrying about what names were used for variables in the functions because there can be no conflict. You only need to worry about the variables that interface with the functions.

Register variables

Now to fulfil a promise made earlier about what a *register* variable is. A computer can keep data in a register or in memory. A register is much faster in operation than memory but there are very few registers available for the programmer to use. If there are certain variables that are used extensively in a program, you can designate that those variables are to be stored in a register if possible in order to speed up the execution of the program. Most C compilers allow you to use 2 register variables and will ignore additional requests if you request more than 2. You are usually allowed to use register variables for short int and int types along with their unsigned counterparts. You can also normally use register storage for two byte pointers - which we will study later.

Defining variables

Now for a refinement on a general rule stated earlier. When you have variables brought to a function as arguments to the function, they are defined immediately after the function name and prior to the opening brace for the program. Other variables used in the function are defined at the beginning of the function, immediately following the opening brace of the function, and before any executable statements.

Returning to lines 2, 3 and 4 in the program, we have the prototypes for the three functions contained within the program. The first *void* in each line tells the compiler that these particular functions do not return a value, so that the compiler would flag the statement ‘index = head1( );’ as an error because nothing is returned to assign to the variable index. The word void within the parentheses tells the compiler that this function requires no parameters and if a variable were included, it would be an error and the compiler would issue a warning message. If you wrote the statement ‘head1(index);’ it would be an error. This

allows you to use type checking when programming in C in much the same manner that it is used in Pascal, Modula 2, or Ada.

Line 1 tells the system to go to the include files and get the file named STDIO.H which contains the prototypes for the standard input and output functions so they can be checked for proper variable types. Don't worry about the *include* yet, it will be covered in detail later in this tutorial.

Exercise 7

Create a program that contains the following main program block. Compile and run the program and explain why the seconf *printf* statement produces different output to the other two.

```
void main()
{
   int a;
   a = 2;
   printf("%d\n", a);
   {
      int a;
      a = 3;
      printf("%d\n", a);
   }
   printf("%d\n", a);
}
```

Objective 8 After working through this module you should be able to create recursive functions in C.

Recursion

Recursion is another of those programming techniques that seems intimidating the first time you come across it, but looking at the next program [RECURSON.C] should help to take some of the mystery out of it. This is probably the simplest recursive program that it is possible to write and it is therefore actually a rather silly program, but for purposes of illustration it is fine.

```
#include "stdio.h"
/* define the prototype for the function used in this          */
/* program.                          */
void count_dn(int count);
void main()
{
int index;
   index = 8;
   count_dn(index);
}
/* -- Function definition --------------------                 */
void count_dn(int count)
{
   count--;
   printf("The value of the count is %d\n",count);
   if (count > 0)
      count_dn(count);
   printf("Now the count is %d\n",count);
}
```

Recursion is nothing more than a function that calls itself. It is therefore in a loop which must have a way of terminating. In this program the variable index is set to 8, and is used as the argument to the function count_dn. The function simply decrements the variable, prints it out in a message, and if the variable is not zero, it calls itself, where it decrements it again, prints it, etc. etc. etc. Finally, the variable will reach zero, and the function will not call itself again. Instead, it will return to the prior time it called itself, and return again, until finally it will return to the main program and will return to DOS. For purposes of understanding you can think of it as having 8 copies of the function count_dn available and it simply called all of them one at a time, keeping track of which copy it was in at any given time. That is not what actually happened, but it is a reasonable illustration for you to begin understanding what it was really doing.

A better explanation of what actually happened is in order. When you called the function from itself, it stored all of the variables and all of the internal flags it needed to complete the function in a block somewhere. The next time it called itself, it did the same thing, creating and storing another block of everything it needed to complete that function call. It continued making these blocks and storing them away until it reached

the last function - when it started retrieving the blocks of data, and using them to complete each function call.

The blocks were stored on an internal part of the computer called the *stack;* this is a part of memory carefully organised to store data just as described above. It is beyond the scope of this tutorial to describe the stack in detail, but it would be good for your programming experience to read some material describing the stack. A stack is used in nearly all modern computers for internal housekeeping chores.

In using recursion, you may desire to write a program with indirect recursion as opposed to the direct recursion described above. Indirect recursion would be when a function A calls the function B, which in turn calls A, etc. This is entirely permissible; the system will take care of putting the necessary things on the stack and retrieving them when needed again. There is no reason why you could not have three functions calling each other in a circle, or four, or five, etc. The C compiler will take care of all of the details for you.

The thing you must remember about recursion is that at some point, something must go to zero, or reach some predefined point, to terminate the loop. If not, you will have an infinite loop, and the stack will fill up and overflow, giving you an error and stopping the program rather abruptly.

Another example of recursion

The next program [BACKWARD.C] is another example of recursion, so let's look at it. This program is similar to the last one except that it uses a character array. Each successive call to the function named 'forward_and_backward' causes one character of the message to be printed. Additionally, each time the function ends, one of the characters is printed again, this time backwards as the string of recursive function calls is retraced.

```
#include "stdio.h"  /* Prototypes for Input/Output             */
#include "string.h" /* Prototypes for string operations       */
/* define the prototype for the function used in this         */
/* program.                        */
void forward_and_backwards(char line_of_char[],int index);
void main()
{
char line_of_char[80];
int index = 0;
   strcpy(line_of_char,"This is a string.\n");
   forward_and_backwards(line_of_char, index);
}
/* -- Function definition --------------------               */
void forward_and_backwards(char line_of_char[],int index)
{
   if (line_of_char[index]) {
      printf("%c",line_of_char[index]);
      index++;
      forward_and_backwards(line_of_char,index);
```

```
        }
        printf("%c",line_of_char[index]);
}
```

This program uses the 'modern' method of function definition and includes full prototype definitions. The modern method of function definition moves the types of the variables into the parentheses along with the variable names themselves. The final result is that the line containing the function name looks more like the corresponding line in Pascal, Modula 2, or Ada.

Exercise 8

Write a recursive function that when given an integer will calculate the sum of all the integers up to, and including, that integer. For example, given 5, the function will calculate 5 + 4 + 3 + 2 + 1.

The greatest common divisor of two positive integers is the largest integer that is a divisor of both of them. For example, 6 and 15 have 3 as their greatest common divisor. The following recursive function computes the greatest common divisor of two integers. First write a program that will test this function, and then write and test an equivalent iterative function.

```
int gcd(int p, int q)
{
   int r;
   if ((r = p % q) == 0)
      return (q);
   else
      return (gcd(q, r));
}
```

Objective 9 After working through this module you should be able to use the *#define* statement to simplify code development.

Aids to clear programming

The first example program in this section [DEFINE.C] contains your first look at some define and macro statements.

```
#include "stdio.h"

#define START  0             /* Starting point of loop         */
#define ENDING 9             /* Ending point of loop           */
#define MAX(A,B)  ((A)>(B)?(A):(B))
                     /* Max macro definition                   */
#define MIN(A,B)  ((A)>(B)?(B):(A))
                     /* Min macro definition                   */

void main()
{
int index,mn,mx;
int count = 5;
   for (index = START; index <= ENDING;index++) {
      mx = MAX(index,count);
      mn = MIN(index,count);
      printf("Max is %d and min is %d\n",mx,mn);
   }
}
```

Notice the four lines starting with the word #define. This is the way all defines and macros are defined. Before the actual compilation starts, the compiler goes through a preprocessor pass to resolve all of the defines. In the present case, it will find every place in the program where the combination START is found and it will simply replace it with the 0 since that is the definition. The compiler itself will never see the word START, so as far as the compiler is concerned, the zeros were always there. Note that if the word START appears in a text string or a comment, it will be ignored and unchanged.

It should be clear to you by now that putting the word START in your program instead of the numeral 0 is only a convenience to you and actually acts like a comment since the word START helps you to understand what the zero is used for.

In the case of a very small program, such as that before you, it doesn't really matter what you use. If, however, you had a 2000 line program before you with 27 references to the START, it would be a completely different matter. If you wanted to change all of the STARTs in the program to a new number, it would be simple to change the one #define, but difficult to find and change all of the references to it manually, and possibly disastrous if you missed one or two of the references.

In the same manner, the preprocessor will find all occurrences of the word ENDING and change them to 9, then the compiler will operate on the changed file with no knowledge that ENDING ever existed.

It is a fairly common practice in C programming to use all capital letters for a symbolic constant such as START and ENDING, and all lower case letters for variable names. You can use any method you choose since it is mostly a matter of personal taste.

When we get to the sections discussing input and output, we will need an indicator to tell us when we reach the end-of-file of an input file. Since different compilers use different numerical values for this, although most use either a zero or a minus 1, we will write the program with a define to define the EOF used by our particular compiler. If at some later date we change to a new compiler, it is a simple matter to change this one define to fix the entire program.

Macros

A macro is nothing more than another define, but since it is capable of at least appearing to perform some logical decisions or some math functions, it has a unique name. Consider line 3 of the current program for an example of a macro. In this case, any time the preprocessor finds the word MAX followed by a group in parentheses, it expects to find two terms in the parentheses and will do a replacement of the terms into the second definition. Thus the first term will replace every A in the second definition and the second term will replace every B in the second definition. When line 10 of the program is reached, index will be substituted for every A, and count will be substituted for every B. (This replacement will not take place in string literals or comments.) Remembering the cryptic construct we studied a couple of sections ago will reveal that mx will receive the maximum value of index or count.

In like manner, the MIN macro will result in mn receiving the minimum value of index or count. The results are then printed out. There are a lot of seemingly extra parentheses in the macro definition but they are not extra, they are essential. We will discuss the extra parentheses in our next program. Compile and run the program now.

Wrong macros

The next program [MACRO.C] gives us a better look at a macro and its use. The first line defines a macro named WRONG that appears to get the cube of A, and indeed it does in some cases, but it fails miserably in others. The second macro named CUBE actually does get the cube in all cases.

```
#include "stdio.h"
```

```
#define WRONG(A) A*A*A          /* Wrong macro for cube       */
#define CUBE(A) (A)*(A)*(A)     /* Right macro for cube       */
#define SQUR(A) (A)*(A)         /* Right macro for square     */
#define ADD_WRONG(A) (A)+(A)    /* Wrong macro for add        */
#define ADD_RIGHT(A) ((A)+(A))  /* Right macro for add        */
#define START 1
#define STOP  7
void main()
{
int i, offset;
   offset = 5;
   for (i = START; i <= STOP; i++) {
      printf("The square of %3d is %4d, the cube is %6d\n",
            i+offset, SQUR(i+offset), CUBE(i+offset));
      printf("The wrong of %3d is %6d\n",
            i+offset, WRONG(i+offset));
}
   printf("\nNow try the addition macro's\n");
   for (i = START;i <= STOP;i++)   {
      printf("Wrong add macro = %6d, and right = %6d\n",
            5*ADD_WRONG(i),5*ADD_RIGHT(i));
   }
}
```

Consider the program itself where the CUBE of i+offset is calculated. If i is 1, which it is the first time through, then we will be looking for the cube of 1+5 = 6, which will result in 216. When using CUBE, we group the values like this, (1+5)*(1+5)*(1+5) = 6*6*6 = 216. However, when we use WRONG, we group them as 1+5*1+5*1+5 = 1+5+5+5 = 16 which is a wrong answer. The parentheses are therefore required to properly group the variables together. It should be clear to you that either CUBE or WRONG would arrive at a correct answer for a single term replacement such as we did in the last program. The correct values of the cube and the square of the numbers are printed out as well as the wrong values for your inspection.

Inspection of line 20 will reveal that we are evaluating 5*(i) + (i) which is 6 if i is one, and in the second case 5*((i) + (i)) which is 10 if i is one. The parentheses around the entire expression ensure that the value will be evaluated correctly.

Exercise 9

1. Write a program to count from 7 to -5 by counting down. Use #define statements to define the limits. (Hint, you will need to use a decrementing variable in the third part of the for loop control.)

2. Add some print statements to MACRO.C to see the result of the erroneous addition macro.