UNIT 2 DATA TYPES IN 'C'

Structure

2.0 INTRODUCTION

Most modern programming languages are to an extent typed; i.e., they admit only of particular, pre-declared types of data or variables in their programs, with restrictions on the kinds of operations permissible on a particular type. In Pascal, for example, program variables may be only one of Integer, or Real, or Character, or Boolean, or else of a user-defined type. Where languages differ one from another is in the degree of typing. Thus Pascal is a strongly typed language: it does not even permit doubtful operations such as the addition of a character value to a real quantity.

Typing helps eliminate errors that might arise in inadvertently operating with unrelated variables. Another benefit of typing is that it helps the compiler allot the appropriate amount of space for each program variable: one byte for a character, two for an integer, four for a real, etc. C too is a typed language, but we shall find that it is not as strongly typed is Pascal is. Thus it provides the programmer with a far greater degree of flexibility than does Pascal. At the same time it must not be forgotten that with this greater power inherent in the language, the C programmer shoulders all the more responsibility for writing code that shall be robust and reliable in all circumstances. Note: Interestingly enough, B and BCPL, the ancestors of C, were typeless languages.

2.1 OBJECTIVES

Programs work with data. This Unit deals with the properties of data that C programs can use.

At the end of this unit you will be able to:

- see that C divides data into two large categories, with many subtypes: integer numbers and numbers with a fractional part

- state what the data types are, and what their ranges, how they are declared and how they are used

- see how simple MACRO definitions are created, and know the rules for naming identifiers.

2.2 VARIABLES OF TYPE int

C program variables and constants are of four types: char, int, float and double. [Note: Contrast this with Pascal which has in addition a type called boolean; we will see later how C very cleverly manages to do without variables explicitly declared to be of this type .] Before an identifier can be used in a C program its type must be explicitly declared. (The correspondence here is with the var statement of Pascal.) Here's a declaratory statement of C:

```
int apples;
```

This statement declares the programmer's intent that apples will store a signed integer value, i.e. apples may be either a positive or a negative integer within the range set for variables of type int. Now this range can be very much machine dependent; it depends, among other things on the word size of your machine. For most compilers for the IBM PC ints are stored in two consecutive bytes, and are restricted to the range [-32768, 32767]. Compare this with the VAX or the Macintosh, where ints are,4-byte signed integers in the range [-2147483648, 2147483647]. In declaring apples to be an int you are telling the compiler how much space to allocate for its storage. You have also made an assumption of the range in which you expect its value to lie.

In contrast to Pascal, in C it is possible and in fact usual to both declare a variable's type and, where needed, define its value in the same statement:

```
int salary = 5000;
```

It is not correct to assume that a variable which has only been declared e.g.:

```
int volts;      /* volts is unpredictable */
```

but has not been defined, i.e. assigned a value, automatically gets the value 0. In fact its value may be anything but 0! Note that the thousands, millions or billions place values are never separated by commas, when constants are defined in C programs.

Let's 1(x)k at Program 2.1, and its output. The program adds two ints x and y, and prints their sum, z. Don't worry if you can't understand everything about the program just yet. Only remember that the printf () can be. used to print numbers just as easily as it prints strings. To print an int x the following printf () will do:

```
printf ("The value of x is: %d\n", x);
```

The %d signifies that x is to be printed as a decimal integer.

```
/* Program 2.1 */
#include <stdio.h
main ( )
            {
             int x = 5, y = 7, z;
             z = x + y;
             printf ("The value of x is: %d\n", x);
             printf ("The value of y is: %d\n", y);
             printf ("Their sum, z, is: %d\n", z);
            }
```

The output of Program 2.1 is appended below.

```
/* Program 2.1 : Output: */
The value of x is: 5
The value of y is: 7
Their sum, z, is: 12
```

To understand the very great importance of using variables in a computation only after having assigned them values, execute the following program and determine its output on your computer:

```
/* Program 2.2 */
#include <stdio.h
main ( )
            {
                    int x, y, z; /* x, y and z are undefined. */
                    /* contd. */


                    z = x + y ;
                    printf ("The value of x is: %d\n", x);
                    printf ("The value of y is: %d\n", y);
                    printf ("Their sum, z, is: %d\n", z);
                }
```

On our machine, a VAX 11/780 from Digital Equipment Corporation, U.S.A. the output was:

```
/* Program 2.2: Output: */
The value of x is: 2146744409
The value of y is: 2146744417
Their sum, z, is: -1478470
```

Now, looking at the output of this program, could you possibly have predicted the values x, y and z would get? Moral: Never assume a variable has a meaningful value, unless you give it one.

Check Your Progress 1

Question 1: Execute the programs below, with the indicated arithmetic operations, to determine their out put:

```
/* Program 2.3 */
#include <stdio.h
main ( )
            {
             int x = 70, y = 15, z;
             printf ("x = %d,\n", x);
             printl'("y = %d,\n", y);
             z = x - y;
             printf ("Their difference x - y is: %d\n", z);
             z = x * y;
             printf ("Their product x * y is: %d\n", z);
             z = x / y;
             printf ("The quotient x / y is: %d\n", y);
            }
```

In particular, determine the values you obtain for the quotient x / y when:

```
x = 60, y = 15;
x = 70, y = 15;
x = 75, y = 15;
x = 80, y = 15;
```

Can you explain your results? What do you think is happening here?

Question 2: Execute Program 2.4 below for the stated values of x and y. What mathematical operation could the % symbol in the statement:

```
z = x % y;
```

signify?

```
/* Program 2.4 */
#include <stdio.h
main ( )
    {
            int x = 60, y = 15, z;
            printf ("x = %d,\n", x);
            printf ("y = %d,\n", y);
            z = x % y       ;       /*What does the % operator do ? */
            printf ("z is: %d\n', z),
            x = 70;
            y = 15;
            z = x % y;
            printf ("z is: %d\n", z);
            /* contd. */
            x = 75;
            y = 15;
            z = x % y;
            printf ("z is, %d\n", z);
            x = 80;
            y = 15;
            z = x % y;
            printf ("z is: %d\n", z);
        }
```

2.2.1 Range Modifiers for int Variables:

On occasions you may need to work with strictly non-negative integers, or with integers in a shorter or longer interval than the default for ints. The following types of range modifying declarations are possible:

(I) unsigned

usage: unsigned int stadium_seats;

This declaration "liberates" the sign bit, and makes the entire word (including the freed sign bit) available for the storage of non-negative integers. [Note: The sign bit_the leftmost bit of a memory word _ determines the sign of the contents of the word; when it's set to 1, the value stored in the remaining bits is negative. Most architectures use two's complement arith- metic, in which the sign bit is "weighted", i.e. it has an associated place value which is nega- tive. Thus on a 16-bit machine its value is -215, or -32,768. So a 16-bit signed number such as 10000000 00111111 would have the value 20 + 21 + 22+ 23 +24 + 25 -215 = -32,705. As an unsigned integer this string of bits would have the value 32831.] On PCs the unsigned decla- ration allows for int variables the range [0, 65535] and is useful when one deals with quan- tities which are known beforehand to be both large and non-negative, e.g. memory addresses, a stadium's seating capacity, etc.

Just as %d in-the printf ( ) prints decimal int values, %u is used to output unsigned ints, as the program below illustrates. Execute the program and determine its output; also examine the effect changing the % u format conversion specifiers to %d in the printf Os. Can you explain your results?

```
/* Program 2.5 */
#include <stdio.h
main ( )
        {
            unsigned int stadium_seats, tickets_sold, standing_viewers;
            stadium_seats = 40000;
            tickets_sold = 50000;
```

```
                                standing_viewers = tickets_sold - stadium_seats;
                                printf ("Tickets sold: %u\n", tickets_sold);
                                printf'("Scats available: %u\n", stidium_scats),
                                printf ("There could be a stajnpede because\,n");
                                printl'("there may be nearly %u standees at the match.\n", standing_viewers);
                        }
```

(II) short

usage: short int friends;

The short int declaration may be useful in instances where an integer variable is known beforehand to be small. The declaration above ensures that the range of friends will not ex- ceed that of ints, but on some computers the range may be shorter (e.g. -128 through 127); friends may be accommodated in a byte, thus saving memory. There.was a time in the olden days of computing,,when main memory was an expensive resource, that programmers tried by such declarations and other stratagems to optimise core usage to the extent possible. (The VAX computer uses two bytes to store short ints, half the amount it uses for ints; but for present-day PCs, with memory cheap and plentiful, most compiler writers make no distnc- tion between ints and short ints.)

The %d specification in the printf ( ) is also used to output short ints.

(III) unsigned short

usage: unsigned short int books;

The range of books will not exceed that of unsigned ints; it may be shorter.

(IV) long

usage: long int stars_in_galaxy;

This declaration is required when you need to use integers larger than the default for ints. On most computers long ints are 4-byte integers ranging over the interval [-2147483648, 2147483647]. When a long integer constant is assigned a value the letter L (or I) must be written immediately after the rightmost digit:

long int big_num = 123456789OL;

% ld in the printf 0 outputs long decimal ints, as you may verify by executing Program 2.6:

```
/* Program 2.6 */
#include <stdio.h
main ( )
          {
                    long int population-2000 = 123456789OL;
                    printf ("The population of this country in 2000 AD\,n");
                    printf ("will exceed %Id if we do\n", populaton_2000);
                    printf ("not take remedial steps now.\n");
          }
```

(V) unsigned long

usage: unsigned long int population_2000;

The unsigned long declaration transforms the range of long ints to the set of 4-byte non- negative integers. Here population_2000 is allowed to range over [0, 4294967295] (Let's hope that larger-sized words will not be required to store this value!) unsigned longs are output by the %lu format conversion specifier in the printf 0.

In the above declarations shorter forms are allowed: thus you may write:

```
unsigned letters;          /* instead of unsigned int letters; */
long rope;                 /* insted of long int rope; */
unsigned short note;       /* instead of unsigned short int note; */ etc.
```

Check Your Progress 2

Question 1: State the output from the execution of the following C program:

```
/* Program 2.7 */
#include <stdio.h
main ( )
            {
             unsigned cheque = 54321;
             long time = 123456789OL; /* seconds */
             printf ("I\'ve waited a long time (%ld seconds)\n", time);
             printf ("for my cheque (for Rs. %u/-), and now\n", cheque);
             printf ("I find it\'s unsigned!\n");
            }
```

Modify this program appropriately to convert the time in seconds to a value of days, hours, minutes and seconds so that it gives the following additional output on execution:

```
That was a wait of.
                                        14288 days,
                                        23 hours,
                                        31 minutes
                                        30 minutes
```

Hint: If a and b ire int variables, then a / b is the quotient and a % b the remainder after division of a by b, as you may have guessed by examining the outputs of programs 2.3 and 2.4.

Question2: You will notice that the lower limits for short,int and long are in each case one larger,in absolute value, than tlher upper lirnits. Explain why. (Hint: Recall how negative and positive integers are stored in a two's complement machine.)

As remarked above, on most current compilers for the PC the range of shorts is not different from that of ints; similarly the range of unsigned shorts is the same as that of unsigned ints.

Thus we have:

```
short <= int <= long
unsigned short <= unsigned int <= unsigned long
```

More than one variable can be declared in a single statement:

```
int apples, pears, oranges, etc;
        /* declares apples, pears,
                oranges and etc to be ints */
    unsigned long radishes, carrots, lotus_stems;
        /* radishes, carrots and lotus_stems
                are unsigned longs */
```

Consider the statement:

```
int x = 1,y = 2,z;
```

This declares x to be an int with value 1, y to be an int with value 2, and z to be an int of unpredictable value. Similarly, the statement:

```
int x, y, z = 1;
```

declares x, y and z to be ints. x and y are undefined. z has the value 1.

Octal (base 8) or hexadecimal (base 16) integers may also be assigned to int variables; octal and hexadecimal integers are used by assembly language programmers as shorthand for the sequences of binary digits that represent memory addresses or the actual contents of memory locations. (That C provides the facility for computing with octal and hexadecimal integers is a reminder of its original raison d'etre: systems programming). An octal integer is prefaced by a 0 (zero), a hexadecimal by the letters Ox or OX, for example

```
int p = 012345, q = 0x1234; /* p is in octal notation, q is in hex */
long octai_num = 01234567OL, hex_num = OX7BCDEF89L;
```

What are the decimal values of p, q, octal _ num and hex _.num ?

The printf ()can be used to output octal and hexadecimal integers just as easily as it prints decimal integers: %o prints octal ints, while %x (or %X) prints hexadecimal ints; long octal or hex ints are printed using %lo and %1x, respectively. %#o and %#x (or %#X) cause octal and hexadecimal values in the output to be preceded by 0 and by Ox (or OX) respectively. (A lowercase x outputs the alphabetical hex digits in lowercase, an uppercase X outputs them in uppercase characters.) Precisely how format control may be specified in a printf ()is described in the next Unit.

One easy way of verifying the answer to the last question is to get printf ( )to print the numbers in decimal, octal and hexadecimal notation:

```
printf ("As an octal number, hex_num = %1o\n", x);
printf ("As a decimal number, octal_ num = %1d\n", x);
printf ("As a hexadecimal number, octal_num = %1x\n", x);
```

What happens if in the course of a computation an int or int like variable gets a value that lies outside the range for the type? C compilers give no warning. The computation proceeds unhampered. Its result is most certainly wrong. This is in contrast to Pascal where overflow causes the program to be aborted.

Check Your Progress 3

Question: Print the values of p, q, octal_num and hex_num using each of the %#o, %#10, %#Ix and %#IX format conversion characters respectively.

Question 2: Execute the following program (in which a variable x is multiplied by itself several times), and determine its output on your machine:

```
/* Program 2.8 */
#include <stdio.h
main ( )
              {
                int x = 5;
                printf ("x = %ft", x),
                x = x * x; /* now x is 5 * 5 = 25 */
                printf ("x = %d\n", x);
                x = x * x; /* now x = 25 * 25 = 625 */
                printf ("x = %d\n", x);
                x = x * x;     /* now x exceeds the limit of int on the PC */
                printf ("x = %d\,n", x);
              }
```

One statement that often confuses novice programmers is:

```
x = x * x;
```

If you have studied algebra, your immediate reaction may well be: "This can't be right, unless x is 0 or x is 1; and x is neither 0 nor 1 in the program. 1 " True; we respect your observation; however, the statement:

```
x = x * x;
```

is not an equation of algebra. It's an instruction to the computer. which in English translates to the following:

Replace x by x times x.

Or, more colloquially, after its execution:

(new value of x) is (old value of x) * (old value of x).

That's why in Program 2.8 x begins with the value 5, then is replaced by 5 * 5 (which is 25), then by 25 * 25 (which is 625), and then by 625 * 625, which is too large a value to fit inside 16 bits.

Note on ANSI C: A decimal integer constant is treated as an unsigned long if its magnitude exceeds that of signed long. An octal or hexadecimal integer that exceeds the limit of int is taken to be unsigned; if it exceeds this limit, it is taken to be long; and if it exceeds this limit it is treated as unsigned long. An integer constant is regarded as unsigned if its value is followed by the letter u or U, e.g. 0x9999u; it is regarded as unsigned long if its value is followed by u or U and 1 or L, e.g. OxFFFFFFFFul.

The system file limits.h available in ANSI C compliant compilers contains the upper and lower limits of integer types. You may #include it before main () precisely as you #Include < stdio.h :

```
#include < limits.h
```

and thereby give to your program access to the constants defined in it, e.g., the declaration:

```
long largest = LONG_MAX;
```

will initialise largest to 2147483647.

The values stated below are accepted minimum magnitudes. Larger values are permitted:

CHAR _BIT	bits in a char	8
CHAR _MAX	maximum value of char	UCHAR-MAX or SCHAR_ MAX
CHAR _MIN	minimum value of char	0 or SCHAR _ MIN
INT_MAX	maximum value of int	32767
INT_MIN	minimum value of int	-32767
LONG _MAX	maximum value of long	2147483647
LONG _MIN	minimum value of long	-2147483647
SCHAR _MAX	maximum value of signed char	127
SCHAR _MIN	minimum value of signed char	-127
SHRT _MAX	maximum value of short	32767
SHRT _MIN	minimum value of short	-32767
UCHAR _MAX	maximum value of unsigned char	255
UINT _MAX	maximum value of unsigned int	65535
ULONG _MAX	maximum value of unsigned long	4294967295
USHRT_MAX	maximum value of unsigned short	65535

End of Note]

You will agree that the programs we've been working with would be much more fun if we could supply them values we wished to compute with, while they're executing, rather than fix the values once and for all within the programs themselves, as we've been doing. What we want, in short, is to make the computer scan values for variables from the keyboard. When a value is entered, it's assigned to a designated variable. This is the value used for the variable in any subsequent computation. Program 2.9 below illustrates how this is done. It accepts two numbers that you type in, and adds them. It uses the scanf ()function, the counter- part of the printf () for the input of data. But, note one important difference: when scanf () is to read a variable x, we write

scanf (" %d", &x);

In scanf () in contrast to printf () the variable name is preceded by the ampersand, &. In the next Unit we will learn that placing the & character before a variable's name yields the memory address of the variable. Quite reasonably a variable that holds a memory address is called a pointer. It points to where the variable can be found. Variables which can store pointers are called pointer variables. Where printf () uses variable names, the scanf 0 function uses pointers.

Experiment with the following program to see what happens if any of x or y or z exceed the limit of ints for your computer.

```
/* Program 2.9 */
#include <stdio.h
main ()
            {
             int x, y, z,
             printf'("Type in a value for int x, and press :)
             scanf ("%d", &x);
             printf ("\nx is: %d\n\n", x);
             printf ("Type in a value for int y, and press :)
             scanf ("%d", &y),
             printf ("\ny is: %d\n\n", y);
             z = x + y;
             printf ("The sum of x and y is %ft", z);
            }
```

2.3 VARIABLES OF TYPE char

Character variables are used to store single characters from the ASCII set. They're accommodated in a single byte. Character variables are declared and defined as in the statement below:

char bee = 'b', see = 'C', ccc;

This declaratory statement assigns the character constant 'b' to the char variable bee, and the character constant 'c' to the char variable named see. The char variable ccc is undefined.

Character constants are single characters. They must be enclosed in right single quotes, as in Pascal. Escape sequences may be also assigned to character variables in their usual backslash notation, with the "compound character" enclosed in right single quotes. Thus the statement:

char nextline =, '\n';

assigns the escape sequence for the newline character to the char variable nextilne. [In ANSI C a character constant is a sequence of one or more characters enclosed in single quotes. Precisely how the value of such a constant is to be interpreted is left to the implementation.]

Since character variables are accommodated in a byte, C regards chars as being a sub- range of ints, (the subrange that fits inside a byte) and each ASCII character is for all purposes equivalent to the decimal integer value of the bit picture which defines it. Thus 'A', of which the ASCII representation is 01000001, has the arithmetical value of 65 decimal. This is the decimal value of the sequence of bits 01000001, as you may easily verify. In other words, the memory representation of the char constant 'A' is indistinguishable from that of the int constant, decimal 65.

The upshot of this is that small int values may be stored in char variables, and char values may he stored in int variables! Character variables are therefore signed quantities restricted to the range [-128, 127]. However, it is a requirement of the language that the decimal equivalent of each of the printing characters be non-negative.

We are assured then that in any C implementation in which a char is stored in an 8-bit byte, the corresponding int value will always be a non-negative quantity, whatever the value of the leftmost (sign) bit may be. Now, identical bit patterns within a byte may be treated as a negative quantity by one machine, as a positive by another. For ensuring portability of programs which store non-character data in char variables the unsigned char declaration is useful: it changes the range of chars to [0, 255]. [Note :

However, the ANSI extension signed char explicitly declares a signed character type to override, if need be, a possible default representation of unsigned chars.]

One consequence of the fact that C does not distinguish between the internal representation of byte-sized ints and chars is that arithmetic operations which are allowed on ints are also allowed on chars! Thus in C, if you wish to, you may multiply one character value by another. Could you do that in Pascal? On the other hand, why would you want to? Appendix I lists the ASCII decimal equivalents of the character set of C.

Here are some variables declared as chars, and defined as escape sequences:

char newline = '\n', char single_quote = '\'';

Character constants can also be defined via their octal ASCII codes. The octal value of the character, which you may find from the table in Appendix I, is preceded by a backslash, and is enclosed in single quotes:

```
char terminal_bell = '\07';     /* 7 = octal ASCII code for beep */
char backspace = '\010';        / *10 = octal code for backspace */
```

[Note: For ANSI C Compilers character constants may be defined by hex digits instead of octals. Hex digits are preceded by x, unlike 0 in the case of octals. Thus in ANSI C:

char backspace =\x A';

is an acceptable alternative declaration to

char backspace = '\010';

Any number of digits may be written, but the value stored is undefined if the resulting character value exceeds the limit of char.]

On an ASCII machine both '\b' and '\010' are equivalent representations. Each will print the backspace character. But the latter form, the ASCII octal equivalent of '\b', will not work on an EBCDIC machine, typically an IBM mainframe, where the collating sequence of the characters (i.e., their gradation or numerical ordering) is different. In the interests of portability therefore it is preferable to write '\b' for the backspace character, rather than its octal code. Then your program will work as certifiably on an EBCDIC machine as it will on an ASCII.

Note that the character constant 'a' is not the same as the string "a". (We will learn later that a string is really an array of characters, a bunch of characters stored in consecutive memory locations, the last location containing the null character; so the string "a" really contains two chars, 'a' immediately followed by '\0'). It is important to realise that the null character is not the same as the decimal digit 0, the ASCII code of which is 00110000.

Just as %d in printf () or scanf () allows us to print and read ints, %c enables the input and output of single characters which are the values of char variables. Let's look at Programs 2. 10 and 2.11 below:

```
/* Program 2. 1 0 */
#include <stdio.h
main ( )
            {
             char a = 'H', b = 'e', c 'I', d 'o', newline '\n';
             printf ("%c", a);
             printf ("%c", b);
             printf ("%c", c);
             printf ("%c", c);
             printf ("%c", d);
             printf ("%c", newline);
            }
```

The output of Program 2.10 is easily predictable (what is it?).

```
/* Program 2.11 */
#include <stdio.h
main ( )
            {
             char val_l, val-2;
             int vai_3;
             printf ("Press any of the keys a - z, then press \n");
             scanf ("%c", &val_1);
             printf ("Press another key in a - z, then press \n");
             scanf ("%c", &val_2);
             printf ("Assuming you're working on an ASCII machine,\
             \nthe chars that you typed have decimal equivalents %d and %d,\
             \nrespectively", val_l, vai_2);
             val_3 = val_1 * val_2;
             printf ("Their product is %d\n", val_3);
            }
```

Execute the above program to verify that char variables behave like one byte ints in arithmetic operations.

The %c format conversion character in a printf () outputs escape sequences, as you saw in Program 2. 10. Execute Program 2.12 if you want a little music:

```
        /* Program 2.12 */
        #include <stdio.h
        main ( )
            {
             char bell = '\007'; /* octal code of the terminal bell *1
             char x = 'Y', y = 'E', z = 'S', exclam = '!';
             printf ("Do you hear the bell ? %c%c%c%c%c%c%c",
             bell, x, bell, y, bell, z, exclam);
            }
```

Program 2.13 assigns a character value to an int variable, an integer value to a char variable, and performs a computation involving these variables. Predict the output of the program, and verify your result by executing it.

```
/* Program 2.13 */
#include <stdio.h
main ( )
        {
         int alpha = 'A', beta;
         char gamma = 122;
         beta = gamma - alpha;
         printf ("beta seen as an int is: %d\n", beta);
         printf ("beta seen as a char is: %c\n", beta);
        }
```

One character that is often required to be sensed in C programs is not strictly speaking a character at all: it's the **EOF** or End-0f-File character, and its occurrence indicates to a program that the end of terminal or file input has been reached. Because the EOF is not a character, being outside the range of chars, any program that's written to sense it must declare an int variable to store character values. As we see in Program 2.13, this is always possible to do. An int variable can accommodate all the characters assigned to it, and can also accommodate **EOF**. We'll see uses for **EOF** in later units.

The putcular ( )function (pronounced "put character"), which takes a character variable or constant as its sole argument, is often a more convenient alternative for screen output thin is the printf ( ) When this function is invoked (for which purpose you may need to #include stdio.h), the character equivalent of its argument is output on the terminal, at the current position of the cursor:

putchar (char_var);

Suppose char_var has been assigned the value 'A'. Then 'A' will be displayed where the cursor was.

Reciprocally, getchar ( ) pronounced "get character") gets a single character from the keyboard, and can assign it to a char variable. (stdio.h 'may have to be #included before getchar ( )can be used.) It has no arguments, and is typically invoked in the following way:

char_var = getchar ( );

When such a statement is encountered, the execution of the program is stayed until a key is pressed. Then char_var is assigned the character value of the key that was pressed.

Program 2.14 below illustrates the usage of these functions. Your compiler may require the inclusion of stdio.h to invoke getchar ( ) and putcliar ( ).

```
/* Program 2.14 */
#include <stdio.h
main ( )
        {
         char key_pressed;
         printf ("Type in a lowercase letter (a - z), press :");
         key_pressed = getchar ( );       /* get char from keyboard
         printf ("You pressed");
         putch,ir (key_pressed - 32);     /* put upperease char on Terminal
         putchar ('\n');
                                         /* converts to uppercase because
                                         ASCII decimal equivalent is 32
```

```
                                        less than for the corresponding
                                        lower case character. */
        }
```

Check Your Progress 4

Question 1: In Program 2.13 insert the declaration:
unsigned char delta = alpha - gamma;
and print the values of delta as an int, an unsigned int, a char and as an unsigned char quantity

Question 2: Write a program which gets a character via getchar (), and prints its ASCII decimal equivalent.

Note on ANSI C: There are certain characters required in C programs which are not available on the keyboards of many non- ASCII computers. In ANSI C these characters can be simulated by trigraph sequences, which are sequences of three characters of the form ??x. They're treated as special characters even if they are embedded inside character strings.

Trigraph	Substitutes for
??=	#
??(	[
??)	]
??<	{
??	}
??/	\
??'	^
??!	|
??-	~

2.4 VARIABLES OF TYPE float

Integer and character data types are incapable of storing numbers with fractional parts. Depending on the precision required, C provides two variable types for computation with "f'loating point" numbers, i.e. numbers with a decimal (internally a binary) point. Such numbers are cable f'loats because the binary point can only nationally be represented in the binary- digits expansion of the number, in which it is made to "float" to the appropriate " position" for optimal precision. (You can immediately see the difficulty of imagining a binary "point" within any particular bit of a floating point word, which can contain only a 0 or a 1!) Typically, some of the leftmost bits of a floating point number store its characteristic (the positive or negative power of two to which it is raised), and the remaining its mantissa, the digits which comprise the number. In base 10, for example, if 2.3 is written as 0.0023 *103, the mantissa is 0.0023, and the characteristic (exponent) is 3.

Single precision floating point variables are declared by the float specification, for example:

```
float bank_balance = 1.234567E8;
    /*En means 10 raised to the power n */
```

The scientific notation En, where the lowercase form en is also acceptable, is optional; one may alternatively write:

```
float bank_balance = 123456700.0;
```

Floats are stored in four bytes and are accurate to about seven significant digits; on PCs their range extends over the interval [E-38, E37].

It must never be lost sight of that the floating point numbers held in a computer's memory are at best approximations to real numbers. There are two reasons for this shortcoming. First, the finite extent of the word size of any computer forces a truncation or round-off of the value to be stored; whether a storage location is two bytes wide, or four, or even eight, the value stored therein can be precise only to so many binary digits.

Second, it is inherently impossible to represent with unlimited accuracy some fractional values as a f'inite series of digits preceded by a binary or decimal point. For example

1/7 = 0. 142857142857142857 ..... ad infinitum

As long as a finite number of digits is written after the decimal point, you will be unable to accurately represent the fraction 1/7. But rational fractions that expand into an infinite series of decimals aren't the only types of floating point numbers that are impossible to store accurately in a computer. Irrational numbers such as the square root of 2 have aperiodic (non- repeating) expansions__there's no way that you can predict the next digit, as you could in the, expansion of 1/7 above, at any point in the series. Therefore it's inherently impossible to store such numbers infinitely accurately inside the machine. PI, the ratio of the circumference of any circle to its diameter, is another number that you cannot represent as a finite sequence of digits. It's not merely irrational, it's a transcendental number. (It cannot be the root of any algebraic equation with rational coefficients.) So an element of imprecision may be introduced by the very nature of the numbers to be stored.

Third, in any computation with floating point numbers, errors of round-off or truncation are necessarily introduced. For suppose you multiply two n-bit numbers; the result will in general be a 2n-bit number. If this number (2n) of bits is larger than the number of bits in the location which will hold the result, you will be forcing a large object into a small hole! Ergo, there'll be a problem! Some of it will just have to be chopped off. Therefore it is wisest to regard with a pinch of salt any number emitted by a computer as the result of a computation, that has a long string of digit-, after the decimal point. It may not be quite as accurate as it seems.

The %e, %f and %g format conversion characters are used in the scant () and printf () functions to read and print floating point numbers. %e (or %E) is Used with floating point numbers in exponent format, while %g (or %G) may be used with floats in either format. %g (or %G) in the printf () outputs a float variable either as a string of decimal numbers including a decimal point, or in exponent notation, whichever is shorter. An uppercase E or G prints an uppercase E in the output. We shall be discussing more about format control in a later Unit.) Program 2.15 below finds the average of five numbers input from the keyboard, and print, it:

```
/* Program 2.15 */
#include <stclio.h
main ( )
            {
             float val_1, val_2, val_3, val_4, val_5, total 0.0, avg;
             printf ("\nEnter first number...");
             scanf ("%f", &val_l);
             printf ("\nEnter second number...");
             scanf ("%f', &,val_2);
             printf ("\nEnter third number...");
             scanf ("%f', &val_3);
             printf ("\nEnter fourth number...")
             scanf ("%f", &val_4);
             printf ("\nEnter fifth number...");
             scanf ("%f", &val_5);
```

```
    total val_1 + val_2 + val_3 + val_4 + val_5;
    avg totil / 5;
    printf ("\nThe average of the numbers you entered is: %f\n", avg);
   }
     Here's a sample conversation with the program:
     Enter first number .. 32.4
     Enter second number .. 56.7
     Enter third number .. 78.3
     Enter fourth number .. 67.8
     Enter fifth number... -93.9

The average of the numbers you entered is: 28.260000
```

2.5 VARIABLES OF TYPE double

Because the words of memory can store values which are precise only to a fixed number of figures, any calculation involving floating point numbers almost invariably introduces round-oft errors. At the same time scientific computations often demand a far greater accuracy than that provided by single precision arithmetic, i.e. arithmetic with the four-byte float variables. Thus, where large scale scientific or engineering computations are involved, the double declaration becomes the natural choice for program variables. The double specification allows the storage of double precision floating point numbers (in eight consecutive bytes) which are held correct to 15 figures, and have a much greater range of definition than floats [E-308, E307]. Older compilers may also allow the long float specification instead of double, but its use is not recommended.

double lightspeed = 2.997925E10, pi = 3.1415928;

Note: ANSI C has another floating point type called long double which has at least as large a number of significant figures, and as large a range of allowable exponents as double. The system file float . h contains constants pertaining to floating point arithmetic. Some of these (from float.h for TURBO C) are:

DBL_DIG	decimal digits of precision	15
FLT_DIG	"	6
LDBL_DIG	"	19
DBL_MANT_DIG	bils to hold the mantissa	53
FLT_MANT_DIG	"	24
LDBL_MANT_DIG	"	64
DBL_MAX_10_EXP	maximum exponent values	308
FLT_MAX_10_EXP	"	38
LDBL_MAX_10_EXP	"	4932
DBL_MIN_10_EXP	minimum exponent values	-307
FLT_MIN_10_EXP	"	-37
LDBL_MIN_10_EXP	"	-4931

End of Note]
[Important Note:]

C compilers convert ,all single precision floating-point constants to double precision wherever they occur in a computation, so it's as well to declare all floating point constants in a program as double. Note: In ANSI C the suffix f or F after a floating point value forces it to be treated as a single precision constant; the suffix I or L treats as a long double constant. Either the integer part or the fraction part may be absent from the definition of a floating constant value: not both of the decimal point and the e and its exponent may be missing.

End of Note]

The, %if (or % le or %lg) specification in a scanf (is required for the input of double variables, which are however output via %e (or %E), %f or %g like their single precision counterparts. Program 2.16 which computes the volume of a cone, whose base radius and height are read off from the keyboard, illustrates this:

```
/* Program 2.16 */
#define PI 3.1415928
#incIude <stdio.h
main ( )
            {
             double base_radius, height, cone_volume;
             printf' ("This program computes the volume of a cone\n");
             printf'("of which the radius and height are entered\n");
             printf'("from the keyboard.\,n\.nEnter radius of cone base: ");
             scanf'("%if",&base_ radius);
             printf'("\nEnter height of cone: ");
             scanf ("%If ", &height );
             printf'("\n Volume of cone of base radius R and height H\
             is (1 / 3) * PI*R*R* H\n");
             cone_volume = PI * base_radius * base -radius * height / 3;
             printi'("\n Volume of cone is: %f\n", cone-volume);
            }
```

One new feature of Program 2.16 is to be found in its second line

#define PI 3.1415928

The #defines_also called MACRO definitions_are a convenient way of declaring constants in a C program. Like the #includes, the #defines are preprocessor control lines. As before, the # symbol must occur in column 1, and there should be no space between # and define. #defines are processed at an early stage of the compilation. They cause the substitution of the named identifier by the associated token string throughout the text of the part of the program which follows the #defines. Exception: not if the identifier is embedded inside a comment, a quoted string, or a char constant. Typically, a #define is of the form:

#define IDENTIFIER identifier_will_be_replaced_by_this_stuff

In Program 2.16 above wherever PI occurs in the program, it is replaced by 3.1415928. (There is a longstanding tradition in C for writing names for MACRO constants in uppercase characters, and we will follow this practice.)

Like the #includes, #defines are also not terminated by a semicolon; if they were, the semi- colon would become part of the replacement string, and this could cause syntactical errors. For example consider the following program:

```
/* Program 2.17 */
#define PI 3.1415928-, /* Error! */
main ( ) /* finds volume of cone, height H, base radius R */
            {
                   double H = 1.23, R = 2.34, cone-volume;
                   cone-volume = PI * R * R * H / 3;
                }
```

When the replacement for PI is made, the assignment for cone_ volume takes the form:

```
cone_volume = 3.1415928; * 2.34 * 2.34 * 1.23/3;
```

which cannot be compiled. For precisely the same reason the assignment operator = cannot occur in a MACRO definition.

A #defined quantity is not a variable and its value cannot be modified by an assignment. Though the MACRO definition for PI has has been placed before main () in Program 2.17, this is not a requirement: it may occur anywhere in the program before it is referenced. One great convenience afforded by MACRO definitions is that if the token string representing the value of a #defined quantity has to be changed, it need be changed only once, in the MACRO itself. When the program is compiled, the changed value will be recorded in every occurrence of the quantity following the #definition. So if you wish to make a computation in which the value of PI must be accurate to 16 places of decimals, just change it where it's #defined.

Note: ANSI C provides the const declaration for items whose values should not change in a program:

```
const int dozen = 12,
```

The keyword const lets the programmer specify the type explicitly in contrast to the #define, where the type is deduced from the definition. In addition, ANSI C has a modifier volatile, to explicitly indicate variables whose values may change, and which must be accessed for a possibly changed value whenever they are referenced.

Check Your Progress 5

Question 1: Write a program to compute sirnple interest: if a principal of P Rupees is deposited for a period of T years at a rate of R per cent, the simple interest I is:

I = P * R * T / 100.

Your program should prompt for floats P, R and T from the keyboard and output the interest 1.

Question2: Write a C program to compute the volume of a sphere of radius 10.The volume is given by the formula:

V = (4/3) * PI * R * R * R

where R is the radius of the sphere.

2.6 ENUMERATED TYPES

In addition to these four types of program variables, C allows additional user-defined variable types, called enum (from enumerated) types. Consider the following declaration, which bears comparison with the Pascal type statement:

```
enum grades
        {
                F, D, C_MINUS, C, C_PLUS, B_MINUS, B, B_PLUS,
                A_MINUS, A, A_PLUS
        } result;
```

This declaration makes the variable result an enumerated type, namely grades, which is called the tag of the type. The tag is a reference to the enumerated type, and may be used to declare other variables of type enum grades:

```
enum grades final_result;
```

The variables result and Final-result can only be assigned one of the values F, D,..., A_PLUS, and no others. These eleven enumerators have the consecutive integer values 0 (for F) through 10 (for A_PLUS); so that if you assign the value A_PLUS to result and later examine it, it will be found to be 10 (and not the string "A,-PLUS").Though the C compiler stores enumerated values as integer constants, enum variables are a distinct type,, and they should not be thought of as ints. [In ANSI C the enumerators are int constants.]

In an enum list any enumerator may be specified an integer value different from the constant associated with it by default. The enumerator to its right gets a value 1 greater, and further down the list, each enumerator becomes 1 more than the preceding.

```
enum flavours
                    {
                            sweet, sour, salty = 6, pungent, hot, bitter
                    }    pickles;
```

In the declaration above sweet and sour have the values 0 and 1 respectively, while pungent, hot and bitter are 7, 8 and 9 in that order.

2.7 THE typedef STATEMENT

The typedef statement is used when one wants to refer to a variable type by an alternative name, or alias. This often makes a great convenience for the programmer: suppose you are writing a program to keep track of all the cutlery__spoons, forks, knives'and serving ladles_in a restaurant Then, the statement

typedef int cutlery;

will enable you to declare:

cutlery spoons, forks, knives, serving-ladles;

which may be more meaningful than:

int spoons, forks, knives, serving_ladles;

cutlery becomes an alias for int. typedef does no more than rename an existing type.

Exercise : Name the types of C variables you would choose to represent the following Quantities _ char, int, float, double or enum:

the velocity of sound in air, approximately 750 miles/hour
the velocity of light in vacuum, 2.997925E10 cm/sec
he number of seconds in 400 years
the punctuation symbols of the English alphabet
the months of the year
the population of the city of Delhi, approximately 9 million
the population of planet Earth, approximately 5.4 billion
the value of Avogadro's number, 6.0248E23
the value of Planck's constant, 6.6IE-27
6,594,126,820,000,000,000,000, which is the approximate value of the mass of the earth, in tons
the colours of the rainbow
the II class train fare between any two cities in India
the seat capacity of a Boeing 747
the vowels of the English alphabet
the days of the week

a savings bank balance of Rs. 12,345.67
a savings bank balance of Rs. 12,34,56,789.10
the square-root of 6 correct to 6 significant figures
the cube-root of 15 correct to 1 5 significant figures

2.8 IDENTIFIERS

We have already seen several examples of C identifiers, or names for program variables (more precisely, the names of storage locations): newline, octal_num, apples, volts, etc. Identifiers for variables and constants, as well as for functions, are sequences of characters chosen from the set {A - Z, a - z, 0 - 9, _}, of which the first character must not be a digit. C is a case sensitive language, so that ALFA and ALFA are different identifiers, as are main () and Main () The underscore character (_) should not be used as the first character of a variable name because several compiler defined identifiers in the standard C library have the underscore for the beginning character, and inadvertently duplicated names can cause "definition conflicts". Identifiers may be any reasonable length; generally 8-10 characters should suffice, though certain compilers may allow for very much longer names (of upto 63 characters). [Note on ANSI C: Two identifiers will be considered different if they differ upto their first 31 characters.] C has a list of keywords e.g. int, continue, etc. which cannot be used in any context other than that predefined in the language. (This implies, for example, that you can't have a program variable named int.) A list of keywords is appended below; take care that you do not choose identifiers from this list. Your programs will not compile. In- deed one should consistently follow the practice of choosing names for variables which indicate the roles of those variables in the program. For example, the identifier savings-balance in a program that processes savings-bank balances is clearly a better choice for representing the variable than is asdf.

C Keywords

auto	Break	case	charc	onst
Continue	Default	do	double	else
Enum	extem	Float	for	goto
If	int	Long	register	return
Short	signed	sizeof	static	struct
Switch	typedef	union	unsigned	void
Volatile	while			

Check Your Progress 6

Question 1: The output of the program below on an ASCII machine was the alphabetical character e. What is the ASCII decimal equivalent of d?

```
/* Program 2.18 */
#include < stdio.h
main ( )
            {
             char a, b, c = 'd';
             b = c / 10;
             a = b * b + 1;
             putchar (a);
            }
```

Question 2: State the output of programs 2.19 and 2.20, and verify your results on a computer

```
/* Program 2.19 */
#include < stdio.h
main ( )
```

```
                {
                 int alpha = 077, beta = Oxabc, gamma = 123, q;
                 q = alpha + beta - gamma;
                 printf ("%d\n", q);
                 q = beta / alpha;
                 printf ("%d\,n", q);
                 q = beta % gamma;
                 printf ("%d\n", q);
                 q = beta / (alpha + gamma);
                 printf ("%d\n", q);
                }
/* Program 2.20 */
#include <stdio.h
main ( )
        {

        char c = 72;                    putchar (c);
        c = c + 29;                     putchar (c);
        c = c + 7;                      putchar (c); putchar (c);
        c = c + 3;                      putchar (c);
        c = c - 67;                     putchar (c);
        c = c - 12;                     putchar (c);
        c = c + 87;                     putchar (c);
        c = c - 8;                      putchar (c);
        c = c + 3;                      putchar (c);
        c = c - 6;                      putchar (c);
        c = c - 8;                      putchar (c);
        c = c - 67;                     putchar (c);
                                        putchar (\n');
        }
```

Question 3: Write C programs to verify whether:

36* 36+37 * 37+38* 38+ 39* 39+40*40=
41 * 41 +42* 42+43 *43 +44 * 44
23 * 23 * 23 +24 * 24* 24 +25 * 25 * 25 = 204 * 204
$5^8 + 12^8 + 13^8 = 59^4 + 120^4 + 179^4$

Question 4: The Fibonacci numbers FI,F2,F3,..., Fn are defined by the relations:

F1 = 1
F2 = 1
F(n) =F(n-1) + F(n-2), n 2

F3 and successive numbers of the series are obtained by adding the preceding two numbers. For large n the ratio of two consecutive Fibonacci numbers is approximately 0.618033989. Given that F(100) is

354,224,848,179,261,915,075 (21 digits)

write a C program to find approximations to F(99) and F(101).

Question 5: The Lucas numbers are defined by the same recurrence relation as the Fibonaccis, where L1 is 1 but L2 is 3. Write a C program to print the first 10 Lucas numbers.

Question 6: the first large number to be factorised by a mechanical machine was one of l9 digits:

1,537,228,672,093,301,419.

The two prime factors found (which were known beforehand) were 29,510,939 and 2,903,110,321. Writing about that event later in Scripta Mathmatica1 (1933) [quoted by Malcolm E Lines in 'Mink of a Number, Pub. Adam Hilger, 19901 Professor D. N. Lehmer (who with his son had invented the machine) remarked, "It would have surprised you to see the excitement in the group of professors and their wives, as they gathered around a table in the laboratory to discuss, over coffee, the mysterious and uncanny powers of this curious machine. It was agreed that it would be unsportsmanlike to use it on small numbers such as could be handled by factor tables and the like, but to reserve it for numbers which lurk as it were, in other galaxies than ours, outside the range of ordinary telescopes."

Write a C program to determine, to the best extent you can, whether the two factors found are indeed correct.

2.9 SUMMARY

The fundamental data types of C are declared through the seven keywords: int, long, short, unsigned, char, float and double. (The keywords signed and long double are ANSI C extensions.)

1. Integers: ranges are in general machine dependent, but in any implementation a short will be at most as long as an int, and an int will be no longer than long. Unsigned integers have zero or positive values only.
2. Characters: char variables are used to represent byte-sized integers and textual symbols.
3. Floating point numbers: may be single or double precision numbers with a decimal point. Unless specified to the contrary (in ANSI C), single precision noats are automatically converted to double in a computation double variables allow a larger number of significant figures, and a larger range of exponents than floats. getchar ( ) gets a keystroke from the keyboard; putchar ( ) deposits its character argument on the monitor.

printf () uses the following format conversion characters to print variables:

d	Decimal integers
u	Unsigned integers
o	Octal integers
x	Hex integers, lowercase
X	Hex integers, uppercase
f	Floating point numbers
e	Floating point numbers in exponential format, lowercase e
E	Floating point numbers in exponential format, uppercase E
g	Floating point numbers in the shorter of f or e format
G	Floating point numbers in the shorter of for E format c single characters

The modifier # with **x** or **X** causes a leading Ox or OX to appear in the output of hexadecimal values; scanf () uses %If to read **doubles.**