UNIT 2 DATA TYPES IN C

Structure **Page No.**

2.1 INTRODUCTION

Computer programs work with data. For this data has to be stored in the computer and manipulated through programs. To facilitate this, data is divided into certain specific types. 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. 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 as, for example, Pascal is. Thus, it provides the programmer with a far greater degree of flexibility. At the same time you should not forget that with this greater power inherent in language, the C programmer shoulders all the more responsibility for writing code that shall be robust and reliable in all circumstances.

Interestingly enough, **B** and **BCPL**, the ancestors of **C**, were type less languages.

We start this unit with discussion of the data type int in Sec. 2.2. As you can easily guess, this is used for storing data which are integers like number of students in a class etc. In Sec. 2.3, we discuss the data type char which are used to hold characters. In Sec. 2.4, we will discuss float which are used to hold decimal numbers like decimal approximations to π, $\sqrt{2}$ etc. In Sec. 2.5, we will discuss double. They are also used to store decimal numbers, but they can store larger numbers with greater accuracy. The enum and typedef, which we discuss in sections 2.6 and 2.7, respectively, helps us to create user defined data types.

Objectives

After studying this unit, you should be able to

- state what the basic data types are, how they are declared and how they are used;
- explain the use of modifiers **unsigned**, **double** and **long**;
- create simple MACRO definitions and correctly name the identifiers according to the rules;
- use `scanf()` and `printf()` to read and print variables of different data types; and
- use the typedef statement and explain its purpose.

2.2 VARIABLES OF TYPE int

We have already seen that C uses different data types for storing integers and numbers with a decimal digit. Actually, C program variables and constants are of four main types: char, int, float and double. We start our discussion of data types with the discussion of the data type **int**.

Before an identifier can be used in a **C** program its type must be explicitly **declared**. Here's a **declarative 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 old DOS compilers for the IBM PC for which the word size is 16 bits `int`s are stored in two consecutive bytes, and are restricted to the range $[-32768, 32767]$. In most of the modern compilers which are 32 bit **ints** are 4-bytes 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 C it is possible and in fact usual to both declare a variable's type and, where needed, **initialise** 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 initialised, i.e. assigned a value, automatically gets the value 0.

Let's look at Listing 1, and its output. The program adds two ints x and `y`, and prints their sum, `z`. Recall 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 1; file name:unit2-prog1.c */
#include<stdio.h>
int 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);
    return (0);
}
```

Listing 1: A simple program that uses int.

The output of the program in Listing 1 is appended below.

```
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 program in Listing 2 on the facing page and determine its output on your computer:

```
/* Program 2; file name:unit2-prog2.c */
#include<stdio.h>
int main()
{
    int x, y, z;                /* x, y and z are undefined. */
    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);
    return (0);
}
```

Listing 2: When variables are not defined.

On our 32-bit linux machine, the output was:

The value of x is: 134513628

The value of y is: -1208279840

Their sum, z is: -1073766212

You may like to compile the program, run it and check the output on your computer.

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.**

Try the following exercises now.

E1) Execute the program below, with the indicated arithmetic operations, to determine their output:

```
/* Program 3; file name:unit2-prog3.c */
#include<stdio.h>
int main()
{
    int x = 70, y = 15, z;
    printf("x=%d,\n", x);
    printf("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);
    return (0);
}
```

Can you explain your results? What do you think is happening here? In particular, determine the values you obtain for the quotient x/y when:

$x = 60, \quad y = 15;$
$x = 70, \quad y = 15;$
$x = 75, \quad y = 15;$
$x = 80, \quad y = 15;$

E2) Execute Program 4 below for the stated values of x and y. What mathematical operation could the % symbol in the statement:

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

signify?

```
/* Program 4; file name: unit2-prog4.c */
#include<stdio.h>
int 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);
    x = 75;
    y = 15;
    z = x % y;
    printf("z is: %d\n", z);
    x = 15;
    y = 80;
    z = x % y;
    printf("z is: %d\n", z);
    return (0);
}
```

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 `int`s. The following types of **range modifying** declarations are possible:

unsigned

Usage

unsigned int *variable-name*;
Example: 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 arithmetic, in which the sign bit is "weighted", i.e. it has an associated place value which is negative. Thus on a 16-bit machine its value is -2^{15}, or $-32,768$. So a 16-bit signed number such as 10000000 00111111 would have the value $2^0+2^1+2^2+2^3+2^4+2^5-2^{15}=-32,705$. As an unsigned integer this string of bits would have the value 32831.

On PCs the unsigned declaration allows for int variables the range at least [0, 65535] and is useful when one deals with quantities 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 in Listing 3 on the next page illustrates. Execute the program and determine its output; also examine the effect of changing the `%u` format conversion specifiers to `%d` in the `printf()`s. Can you explain your results?

```
/* Program 5; file name:unit2-prog5.c */
#include<stdio.h>
int 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("Seats available: %u\n", stadium_seats);
    printf("There could be a stampede because\n");
    printf("there may be nearly %u standees \
at the match.\n", standing_viewers);/*Continued*/
    return (0);
}
```

Listing 3: Example to show the use of unsigned modifier.

Usage

short int *variable-name*
Example: 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 exceed 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. But for present-day PCs, with memory cheap and plentiful, most compiler writers make no distinction between ints and short ints.)

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

unsigned short

Usage

unsigned short int *variable-name*;
Example: unsigned short int books;

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

long

Usage

long int *variable-name*;
Example: 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 `l`) must be written immediately after the rightmost digit:

```
long int big_num = 1234567890L;
```

`%ld` in the `printf()` outputs long decimal **int**s, as you may verify by executing the program in Listing 4:

```
/* Program 6; file name:unit1-prog6.c */
#include<stdio.h>
int main()
```

```
{
    long int population_2020 = 424967295L;
    printf("The population of this country in 2020 AD\n");
    printf("will exceed %ld if we do\n", population_2020);
    printf("not take remedial steps now.\n");
    return (0);
}
```

Listing 4: Format modifier for printing long integers.

unsigned long

Usage

unsigned long int *variable-name*;
Example: usage: **unsigned long int** population_2020;

The **unsigned long** declaration transforms the range of **long int**s to the set of 4-byte non-negative integers. Here population_2020 is allowed to range over [0, 4294967295], (Let's hope that larger-sized words will not be required to store this value!) **unsigned long**s are output by the %lu format conversion specifier in the printf().

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

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

etc.

E3) State the output from the execution of the following C program:

```
/* Program 7; file name:unit1-prog7.c */
#include<stdio.h>
int main()
{
    unsigned cheque = 54321;
    long time = 1234567890L;    /*seconds */
    printf("I\'ve waited a long\
time (%ld seconds)\n", time);/*Continued*/
    printf("for my cheque (for Rs.%u/-), \
and now\n", cheque);/*Continued*/
    printf("I find it\'s unsigned!\n");
    return (0);
}
```

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
                and 30 seconds.
```

(**Hint:** If a and b are int variables, then a/b is the quotient and a % b the remainder after division of a by b if b is less than a.)

As remarked above, on most current compilers for the PC the range of shorts is not different from the 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 `int`s. `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 `0x` or `0X`, for example

```
int p = 012345, q = 0x1234; /*p is in octal notation, q is in hex */
long octal_num = 012345670L, hex_num = 0X7BCDEF89L;
```

The `printf()` can be used to output octal and hexadecimal integers just as easily as it prints decimal integers: `%0` prints octal `int`s, while `%x` (or `%X` prints hexadecimal `int`s; `long` octal or hex `int`s are printed using `%lo` and `%lx`, respectively, `%#o` and `%#x` (or `%#X`) cause octal and hexadecimal values in the output to be preceded by 0 and by 0x (or 0X) 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.

What are the decimal values of `p, q, octal_num` and `hex_num`?

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 = %l0\n", x);
printf ("As a decimal number, octal_num = %ld\n", x);
printf ("As a hexadecimal number, octal\num = %lx\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. The program in Listing 5 on the next page of exercise. 5 demonstrates this. There is also an exercise to test your understanding of format modifiers for printing hexadecimal and octal numbers. Please try them.

E4) Print the values of `p, q, octal_num` and `hex_num` using each of the %#0, %#l0, %#lx and %#lX format conversion characters respectively.

E5) Execute the program in Listing 5 on the next page(in which a variable x is multiplied by itself several times), and determine its output on your machine:

```
/* Program 8; file name: unit2-prog8.c */
#include <stdio.h>
int main()
{
    int x = 5;
    printf("x = %d\n", 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 Old 16 bit compilers. */
    printf("x = %d\n", x);
    x = x * x;                    /*now x exceeds
                 the limit of int in 32 bit machines also */
    printf("x = %d\n", x);
    return (0);
}
```

Listing 5: Example demonstrating the effects of overflow.

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.!" 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 8, exercise. 5 on the preceding page, 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.

In 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. `0xffffffful`.

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 your `#include <stdio.h>`:

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

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

```
long largest = LONG_MAX;
```

Table 1: Limits for `int` types.

CHAR_BIT	bits in a char	8
CHAR_MAX	maximum value of char	UCHAR_MAX or SHCAR_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

will initialise `largest` to 2147483647. The values stated below are accepted minimum magnitudes. Larger values are permitted: Let us now write a program that uses `limits.h`. The program in Listing 6 prints the maximum and minimum values of `int` and the maximum value of `unsigned int`.(What is the minimum value of `unsigned int`?)

```
/*Program to print limits; file name: printlimits.c*/
#include<stdio.h>
#include<limits.h>
int main()
{
    printf("Maximum value of int is %d: \n", INT_MAX);
    printf("Minimum value of int is %d: \n", INT_MIN);
    printf("Maximum value of unsigned is %u: \n", UINT_MAX);
    return (0);
}
```

Listing 6: Program that prints limits.

Here is an exercise for you to check for yourself whether you can use the values in limits.h file.

E6) Modify the program so that it prints the limits for the other types in Table. 1. Compile and run the program on your computer and see what are the values you get.

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 designed variable. This is the value used for the variable in any subsequent computation. Program 9 below illustrates how this is done. It accepts two numbers that you type in, and adds them. It uses the `scanf()` function, the counterpart of the `printf`() for the input of data. We have already seen an example in Unit 1, that uses `scanf()` function. Here we will give some more details about this function. We will discuss both this function and `printf()` in greater detail later. 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 that variable can be found. Variables which can store pointers are called **pointer variables**. Where `printf()` uses variable names, the `scanf()` function uses pointers.

Experiment with the program in Listing 7 to see what happens if any of `x` or `y` or `z` exceed the limit of `int`s for your computer.

```
/* Program 9; file name:unit2-prog9.c */
#include<stdio.h>
int main()
{
    int x, y, z;
    printf("Type in a value for int x, and press <CR>:");
    scanf("%d", &x);
    printf("\nx is: %d\n\n", x);
    printf("Type in a value for int y, and press <CR>:");
    scanf("%d", &y);
    printf("\ny is: %d\n\n", y);
    z = x + y;
    printf("The sum of x and y is %d\n", z);
    return (0);
}
```

Listing 7: Experimenting with limits of ints.

E7) Write a program that reads two non-negative integers of type **unsigned long** and checks if their sum entered exceeds the limit for **unsigned long**.

We end this section here. In the next section we will discuss another type of variable that can be used to store characters.

2.3 VARIABLES OF TYPE char

Character variables are used to store single characters for 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 named see. The `char` variable `ccc` is undefined.

Character constants are single characters. They must be enclosed in right single quotes. 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 `nextline`.

> Note
>
> 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 subrange 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 be 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 `char`s 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 char`s.

One consequence of the fact that C does not distinguish between the internal representation of byte-sized `int`s and `char`s is that arithmetic operations which are allowed on `int`s are also allowed on `char`s! Thus in C, if you wish to, you may multiply one character value by another. I list the ASCII decimal equivalents of the character set of C. Here are some variables declared as `char`s, and defined as escape sequences:

```
char newline = '\n', 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 is ANSI C:
>
> `char backspace ='\xA';`
>
> 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 singe characters which are the values of char variables. Let's look at Programs 10 and 11 below: c

```
/* Program 10; file name: unit2-prog10.c */
#include <stdio.h>
int main()
{
    char a = 'H', b = 'e', c = 'l', d = 'o', newline = '\n';
    printf("%c", a);
    printf("%c", b);
    printf("%c", c);
    printf("%c", c);
    printf("%c", d);
    printf("%c", newline);
    return (0);
}
```

The output of Program 10 is easily predictable (what is it?).

```
/* Program 11; file name:unit2-prog11.c */
#include <stdio.h>
int main()
{
    char val_1 = 'a', val_2 = 'z';
    int val_3;
    printf("The first character is %c \
and its decimal equivalent is %d.\n", val_1, val_1);/*Continued*/
    printf("The second character is %c \
and its decimal equivalent is %d.\n", val_2, val_2);/*Continued*/
    val_3 = val_1 * val_2;
    printf("Their product is %d\n", val_3);
    return (0);
}
```

Execute the program above to verify that the 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 10. Execute Program 12 if you want a little music:

```
/* Program 2.12;File name:unit2-prog12.c */
#include <stdio.h>
int main()
{
    char bell = '\007'; /* octal code of the terminal bell */
    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);/*Continued*/
    return (0);
}
```

Program 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 13; file name: unit2-prog13.c*/
#include <stdio.h>
```

```
int 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);
   return (0);
}
```

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-Of-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 `char`s, any program that's written to sense it must declare an `int` variable to store character values. As we see in Program 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 `putchar`() function (pronounced "put character"), which takes a character variable or constant as its sole argument, is often a more convenient alternative for screen output than 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 `#include`ed 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 14 below illustrates the usage of these functions. Your compiler may require the inclusion of `stdio.h` to invoke `getchar`() and `putchar`().

```
/* Program 14; file name:unit2-prog14.c */
#include<stdio.h>
int main()
{
    char key_pressed;
    printf("Type n a lowercase letter (a-z), press <CR>:");
    key_pressed = getchar();    /* get char from keyboard */
    printf("You pressed ");
    putchar(key_pressed - 32);
    /* put uppercase char on Terminal */
    putchar('\n');
    /* converts to uppercase beacause
    ASCII decimal equivalent is 32
    less than for the corresponding
    lower case character. */
    return (0);
}
```

Note

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	Substitutesfor
??=	#
??(	[
??)	]
??<	{
??>	}
??/	\
??'	^
??!	|
??-	~

Here are some exercises to give you some practice in working with `char`.

E8) In Program 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.

E9) Write a program which gets a character via `getchar`(), and prints its ASCII decimal equivalent.

We close this section here. In the next section, we will take up another data type called float that is used for storing fractions and irrational numbers.

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 "floating point" numbers, i.e. numbers with a decimal (internally a binary) point. Such numbers are called `floats` because the binary point can only notionally 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 0.0023×10^3, 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 eighth, 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 finite 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. The number π, which is the ratio of the circumference of any circle to its diameter, is another number that you cannot represent as a finite sequence of digits. As you know 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 computation, that has a long string of digits 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 `scanf()` 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 15 below finds the average of five numbers input from the keyboard, and prints it:

```
/* Program 15; file name:unit2-prog15.c */
#include<stdio.h>
int main()
{
    float val_1, val_2, val_3, val_4,
    val_5, total = 0.0, avg;
    printf("\nEnter first number...");
    scanf("%f", &val_1);
    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 = total / 5;
    printf("\nThe average of the numbers \
you entered is: %f\n", avg);/*Continued*/
    return (0);
}
```

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
```

You can check your understanding of our discussion of floats by doing the following exercise.

E10) Write a program to compute simple 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 $I = \frac{PRT}{100}$. Your program should prompt for `float`s P, R and T from the keyboard and output the interest I.

We end this secton here. In the next section, we discuss **double**, a data type for storing larger floating numbers.

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-off** 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;
```

The `%lf` (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 16 which computes the volume of a cone, whose base radius and height are read off from the keyboard, illustrates this:

```
/* Program 16; file name:unit2-prog16.c */
#include <stdio.h>
#define PI 3.1415928
int 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("%lf", &base_radius);
    printf("\nEnter height of cone:");
    scanf("%lf", &height);
    printf("\nVolume of cone of base radius R and height H \
is (1/3)*PI*R*R*H\n");/*Continued*/
    cone_volume = PI * base_radius * base_radius * height / 3;
    printf("\nVolume of cone is: %f\n", cone_volume);
    return (0);
}
```

One new feature of Program 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 **#include**s, the **#define**s are preprocessor control lines. #define**s** 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 #define. 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 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 uppercases characters, and we will follow this practice.)

Like the #include, #define**s** are also not terminated by a semicolon; if they were, the semicolon would become part of the replacement string, and this could cause syntactical errors. For example consider the following program:

```
/* Program 17; file name:unit2-prog17.c */
#define PI 3.1415928;          /* Error! */
int 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;
    return (0);
}
```

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 #define**d** quantity is **not** a variable and its value cannot be modified by an assignment. Though the MACRO definition for PI has been placed before main() in Program 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 #define**d** 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 #defin**i**tion. 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 #define**d**.

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 for gcc on 32 bit Linux are:

DBL_DIG	decimal digits of precision	15
FLT_DIG	decimal digits of precision	6
LDBL_DIG	decimal digits of precision	18
DBL_MANT_DIG	bits to hold the mantissa	53
FLT_MANT_DIG	bits to hold the mantissa	24
LDBL_MANT_DIG	bits to hold the mantissa	64
DBL_MAX_10_EXP	maximum exponent values	308
FLT_MAX_10_EXP	maximum exponent values	38
LDBL_MAX_10_EXP	maximum exponent values	4932
DBL_MIN_10_EXP	minimum exponent values	-307
FLT_MIN_10_EXP	minimum exponent values	-37
LDBL_MIN_10_EXP	minimum exponent values	-4931

You can always print these values for your machine by writing a small C program similar to the one we wrote for integers.

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`. 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 `l` 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.

Here is a version of Simpson rule that uses `#define` to define the integrand.

```
/*Numerical integration on the interval [0,1] using Simpson's
Rule. It uses 6 sub intervals. File name:unit2-simpson.c*/
#include<stdio.h>
#include<math.h>
#define f(x) (1/(1+(x)*(x)))/* You can change this.*/
/* Note that the previous line does not contain a =
 between f(x) and 1/(1+(x)*(x))*/
int main()
{
    float h = 1 / 4.0;
    printf("The value of the integral is \
%f\n", (h / 3.0) * (f(0) + 4 * (f(h) +/*Continued*/
f(3 * h)) + 2 * f(2 * h) + f(4 * h)));/*Continued*/
    return (0);
}
```

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.

E11) Write a C program to compute the volume of a sphere of radius 10. The volume is given by the formula:

$$V = \frac{4}{3}\pi R^3$$

where R is the radius of the sphere.

In the next section, we will discuss some other data types called enumerated types.

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.

```
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 on 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 `int`s. [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.

E12) Name the types of C variables you would choose to represent the following quantities—`char, int, float, double` or `enum:`

1) The velocity of sound in air, approximately 750 miles/hour.
2) The velocity of light—in vacuum, 2.997925E10 cm/sec.
3) The number of seconds in 400 years.
4) The punctuation symbols of the English alphabet.
5) The months of the year.
6) The population of the city of Delhi, approximately 9 million.
7) The population of planet Earth, approximately 5.4 million.
8) The value of Avogadro's number, 6.0248E23.
9) The value of Planck's constant, 6.61E-27.
10) 6,594,126,820,000,000,000,000, which is the approximate value of the mass. of the earth, in tons.
11) The colours of the rainbow.
12) The II class train fare between any two cities in India.
13) The seat capacity of a Boeing 747.
14) The vowels of the English alphabet.
15) The days of the week.
16) The savings bank balance of Rs. 12,345.67.
17) The savings bank balance of Rs. 12,34,56,89.10.
18) The square-root of 6 correct to 6 significant figures.
19) The cube-root of 15 correct to 15 significant figures.

So far we have merrily given names to variables in C programs without bothering about their validity. However, there are some rules governing variable names. We will discuss these rules in the next section.

2.8 IDENTIFIERS

We have already seen several examples of C identifiers, or names for program variables (more precisely, the name of storage locations): `newline, octal_num, apples, volt`, 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 up to 63 characters).

Note

ANSI C: According to ANSI C standards, **at least** first 31 characters of internal variables are significant.(All the variables that we have seen so far are internal variables only.) In other words, two identifiers must be regarded as different if at least one of the first 31 of their characters are different. We will see what are external variables in the next block. For external variables, **at least** 6 characters are significant.

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. Indeed 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 `saving_balance` in a program that processes savings-bank balances is clearly a better choice for representing the variables than is `asdf`.

C Keywords

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

E13) 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; File name:unit2-prog18.c*/
#include <stdio.h>
int main()
{
    char a, b, c = 'd';
    b = c / 10;
    a = b * b + 1;
    putchar(a);
    return (0);
}
```

E14) State the output of programs 19 and 20, and verify your results on a computer:

```
/* Program 2.19; File name:unit2-prog19.c */
#include <stdio.h>
int main()
{
    int alpha = 077, beta = 0xabc, 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);
    return (0);
}
```

```
/* Program 2.20;File name unit2-prog20.c */
#include <stdio.h>
int 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');
    return (0);
}
```

E15) Write C programs to verify whether:

$$36^2 + 37^2 + 38^2 + 39^2 + 40^2 = 41^2 + 42^2 + 43^2 + 44^2$$

$$23^3 + 24^3 + 25^3 = 204^2$$

$$5^8 + 12^8 + 13^8 = 59^4 + 120^4 + 179^4$$

E16) The Fibonacci numbers F_1, F_2, F_3,..., F_n are defined by the relations:

$F_1 = 1$

F2 = 1

$F_n = F_{n-1} + F_{n-2}, n > 2$

F_3 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).

E17) The Lucas numbers are defined by the same recurrence relation as the Fibonacci's, where L_1 is 1 but L_2 is 3. Write a C program to print the first 10 Lucas numbers.

E18) The first large number to be factorised by a mechanical machine was one of 19 digits:

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

The two prime factors found (which were known beforehand) were 529,510,939 and 2,903,110,321. Writing about that event later in Scripta Mathematica 1 (1933) [quoted by Malcolm E Lines in Think of a Number, Pub. Adam Hilger, 1990] 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.

We close the Unit here. In the next section, we will summarise the Unit.

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 `float`s are automatically converted to `double` in a computation. `double` variables allow a larger number of significant figures, and a larger range of exponents than `float`s. `getchar`() gets a keystroke from the keyboard; `putchar`() deposits its character argument on the monitor.
4. `printf`() uses the following format conversion characters to print variables:

d	decimal integers
u	unsigned integers
o	octal integers
l	long
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 f or E format c single characters

   The modifier # with `x` or `X` causes a leading `0x` or `0X` to appear in the output of hexadecimal values; `scanf`() uses `%lf` to read `doubles`.

2.10 SOLUTIONS/ANSWERS

E1) Here x/y gives the quotient of x when divided by y. When x = 60, y = 15, x/y is 4; when x = 70, y = 15 x/y is 4; when x = 75, y = 15, x/y is 5 etc.

E2) Here x % y gives remainder when x is divided by y if y < y. It gives y if x < y and 0 if x = y.

E3) Here is the program:

```
/* Answer to Exercise 2.3; File name:unit2-ex3ans.c*/
#include<stdio.h>
int main()
{
    unsigned cheque = 54321;
    int seconds, minutes, hours, days;
    long time = 1234567890L;    /*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");
    printf("That's a wait of:\n");
    seconds = time % 60;
    time = time / 60;
    minutes = time % 60;
    time = time / 60;
    hours = time % 24;
    days = time / 24;
    printf("%d days\n", days);
    printf("%d hours\n", hours);
    printf("%d minutes\n", minutes);
    printf("%d seconds\n", seconds);
    return (0);
}
```

E4) Here is the example program:

```
/*Answer to exercise 4; File name: unit2-ex4ans.c*/
#include<stdio.h>
int main()
{/* p is in octal notation, q is in hex. */
    int p = 012345, q = 0x1234;
    long octal_num = 012345670L, hex_num = 0x7BCDEF89L;
    printf("Value of p is %#o\n", p);
    printf("Value of q is %#lx\n", q);
    printf("Value of octal_num is %#lo\n", octal_num);
    printf("Value of hex_num is %#lX\n", hex_num);
    return (0);
}
```

E5) The following is the output on a 32-bit Linux machine when the program is compiled with gcc:

```
x = 5
x = 25
x = 625
x = 390625
x = -2030932031
```

E6)

```
/*Answer to exercise 6; Filename: unit2-printlimitsfull.c*/
#include<stdio.h>
#include<limits.h>
int main()
{
    printf("Number of bits in a char is: %d\n", CHAR_BIT);
    printf("Maximum value of char is %d: \n", CHAR_MAX);
    printf("Minimum value of char is %d: \n", CHAR_MIN);
    printf("Maximum value of int is %d: \n", INT_MAX);
    printf("Minimum value of int is %d: \n", INT_MIN);
    printf("Maximum value of long is %ld: \n", LONG_MAX);
    printf("Minimum value of long is %ld: \n", LONG_MIN);
    printf("Maximum value of signed \
```

```
char is %d: \n", SCHAR_MAX);/*Continued*/
    printf("Minimum value of signed is %d: \n", SCHAR_MIN);
    printf("Maximum value of short is %d: \n", SHRT_MAX);
    printf("Minimum value of short is %d: \n", SHRT_MIN);
    printf("Maximum value of unsigned \
char is %d: \n", UCHAR_MAX);/*Continued*/
    printf("Maximum value of unsigned \
int is %u: \n", UINT_MAX);/*Continued*/
    printf("Maximum value of unsigned \
long is %lu: \n", ULONG_MAX);/*Continued*/
    return (0);
}
```

E7)
```
/*Program that checks for overflow.
File:unit2-ex7ans.c*/
#include<stdio.h>
#include<limits.h>
int main()
{
    unsigned long x, y;
    printf("Enter the value of x...\n");
    scanf("%lu", &x);
    printf("Enter the value of y...\n");
    scanf("%lu", &y);
    if (x > ULONG_MAX - y)
        printf("The sum of x and y exceeds\
the limit of ULONG_MAX\n");/*Continued*/
    else
        printf("\nThe sum of x and y is %lu\n", x + y);
    return 0;
}
```

E8)
```
/* Program ans-ex7; file name: prog13-ex7.c*/
#include <stdio.h>
int main()
{
    int alpha = 'A', beta;
    char gamma = 122;
    unsigned char delta = alpha - gamma;
    beta = gamma - alpha;
    printf("beta seen as an int is: %d\n", beta);
    printf("beta seen as a char is: %c\n", beta);
    printf("delta as an int %d\n", delta);
    printf("delta as an unsigned int %u\n", delta);
    printf("delta as a char is %c\n", delta);
    return (0);
}
```

The output is

beta seen as an int is: 57

beta seen as a char is: 9

delta as an int 199

delta as an unsigned int 199

delta as a char is

E9)
```
/*Answer to exercise 9; File name unit2-ans-ex9.c*/
#include<stdio.h>
int main()
{
    int i;
    printf("Enter the character...\n");
```

```
    i = getchar();
    printf("The decimal equivalent is %d.", i);
    return 0;
}
```

E10)
```
/*Program to calculate simple interest.
File name: unit2-ansex10.c*/
#include<stdio.h>
int main()
{
    double principal, rate_of_interest, period,interest;
    printf("This program calculates simple interest\n");
    printf("Enter the principal\n");
    scanf("%lf", &principal);
    printf("Enter the rate of interest:\n");
    scanf("%lf", &rate_of_interest);
    printf("Enter the period in months:\n");
    scanf("%lf", &period);
    interest=(principal * rate_of_interest * period) / 100;
    printf("The simple interest is \n");
    printf("%lf", interest);
    return (0);
}
```

E11)
```
/*Program to calculate the volume of a sphere
File name: unit2-ansex11.c*/
#include<stdio.h>
#define PI 3.1415928
int main()
{
    double radius;
    printf("Enter the radius of the sphere:\n");
    scanf("%lf",&radius);
    printf("\n The volume of the sphere is:\n");
    printf("%lf",4*PI*radius*radius*radius/3);
    return (0);
}
```

E12)

1)	float	11)	enum
2)	float	12)	float
3)	It is an integer type, but none of the integer types is big enough. So, we have to use float.	13)	short int
4)	char	14)	char
5)	enum	15)	enum
6)	int	16)	float
7)	float	17)	float
8)	float	18)	float
9)	float	19)	double
10)	double		

E13) 100

E14) The output from program 19 is

2688

43

42

14

The output from program 20 is

E15)
```
/*Answer to exercise 14; File name unit2-ans-ex14.c*/
#include<stdio.h>
long sq(int x);
long cu(int x);
main()
{
    long ans, ans1;
    ans = sq(36) + sq(37) + sq(38) + sq(39) + sq(40);
    ans1 = sq(41) + sq(42) + sq(43)+sq(44);
    printf("%ld", ans - ans1);
    ans = cu(23) + cu(24) + cu(25);
    ans1 = sq(204);
    printf("\n%ld", ans - ans1);
    ans = cu(sq(5)) * sq(5) + cu(sq(12)) * sq(12)
+ cu(sq(13)) * sq(13);/*Continued*/
    ans1 = sq(sq(59)) + sq(sq(120)) + sq(sq(179));
    printf("\n%ld", ans - ans1);
    return 0;
}
long sq(int x)
{
    return x * x;
}
long cu(int x)
{
    return x * x * x;
}
```

E16)
```
/*Answer to exercise 15.*/
/*File name: unit2-ans-ex15.c*/
#include<stdio.h>
int main()
{
    long double f99, f100 = 354224848179261915075.0L, f101;
    long double ratio = 0.618033989L;
    printf("\nf100=%Lf\n", f100);
    f99 = f100 * ratio;
    f101 = f100 / ratio;
    printf("f99 = %Lf\n f100 = %Lf\n f101 = %Lf\n",
f99, f100, f101);/*Continued*/
    printf("f101-f100-f99=%Lf", f101 - f99 - f100);
    return 0;
}
```

Note the %Lf format modifier for long double. %Le is also allowed for printing long double in exponential notation.

E17)
```
/*Answer to exercise 18; File name: unit2-ansex18.c*/
#include<stdio.h>
int main()
{
    long double prime_1 = 529510939.0L;
    long double prime_2 = 2903110321.0L, product;
    product = prime_1 * prime_2;
    printf("The product of the primes is %Lf", product);
    return 0;
}
```

Does the program give the correct answer?

ASCII Table

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	‘	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	’	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

Note that the characters from 0–31 and character 127 are non-printing characters and character 32 is the space.