Chapter 1

Essentials of C Language

1.1 INTRODUCTION

It is important to note that for the learning of data structures and their implementation effectively and efficiently; in-depth knowledge of C language is essential. In particular, the deep understanding of the pointers and their usage is essential.

This chapter gives an overview of the features of C language that are essential for the study of data structures.

1.2 ENUMERATED DATA TYPE

Variables of enumerated data type enhance the readability of the program. For example, a payroll program deals with various categories of employees, such as clerks, stenos, assistants, supervisor, deputy managers and managers. If we decide that code 0 is used for clerks, 1 for stenos, and so on. Program will not be readable, and later on if the program is to be modified, the task may be difficult if the program is not properly documented. However, if we use names of these categories, the program will be more readable.

In C language, you can define your own data type and specify what values a variable of this type can take. The data type defined this way is known as *enumerated (enum) data type*.

For example, since C language does not support Boolean type data (with false and true), you can create your own as

```
enum boolean { false, true };
```

Here word *boolean* is called *tag name* for the user-defined data type. Then we can declare variables of this data type as follows

```
enum boolean flag;
```

Then we can assign value *false* or *true* to variable *flag*, and also we can compare the value of *flag* with these values.

Internally, these values are represented as 2-byte integer numbers. By default, these numbers start from 0. Thus, the first value in the list is represented by number 0, the second value by number 1, and so on. However, we can override these default numbers by specifying the starting number along with the first value of the list.

Variables of enumerated type can also be used in arithmetic and relational expressions. However, the input and output of the variables of enumerated data type is not possible.

1.3 THE *void* DATA TYPE

The *void* data type, also known as *empty* data type, is useful in many situations. These are enumerated below:

1. When used as a function return type, *void* means that the function does not return a value. For example, we may write as

    ```
    void functionName( int x, int y )
    {
        /*  body of function   */
    }
    ```

    to indicate that the *functionName()* does not return a value.

2. When used in the function heading in place of argument list, *void* means that the function does not take any argument. For example, we may write as

    ```
    int functionName( void )
    {
        . . . .
    }
    ```

    to indicate that the *functionName()* does not take any argument.

3. Used to declare generic pointers. The *void* pointer cannot be directly de-referenced without typecasting. This is because the compiler cannot determine the size of the value the pointer points to.

The following segment illustrates this.

```
void main()
{
  void *ptr;
  int x = 5;
  float y = 20.5;
  ptr = &x;
  printf( "\nValue pointed to generic pointer now is %d",
          *(int*)ptr );
  ptr = &y;
  printf( "\nValue pointed to generic pointer now is %f",
          *(float*)ptr);
}
```

1.4 TYPE DEFINITION

A type definition, *typedef*, gives a name to a data type by creating a new type that can be used anywhere a type is permitted.

The syntax for type definition is

```
typedef dataType IDENTIFIER;
```

where *dataType* is either built-in data type or user-defined data type, and *IDENTIFIER*, usually in uppercase, is the new and convenient name for the *dataType*.

The *typedef* keyword tells the compiler to recognize the *IDENTIFIER* as synonymous of *dataType*.

For example, the statement

```
typedef enum { false, true } BOOLEAN;
```

Now to declare variable *flag* type *BOOLEAN*, we write

```
BOOLEAN flag;
```

1.5 CONTROL STATEMENTS

The C Language provides facilities for controlling the order of execution of the statements, which are referred to as *flow control statements*. The various flow control statements are clubbed in the following categories:

1. *Decision Making Statements* — In this category, C language provides the following statements
   - *if* statement
   - *if – else* statement
   - *else if* construct
   - *switch* statement

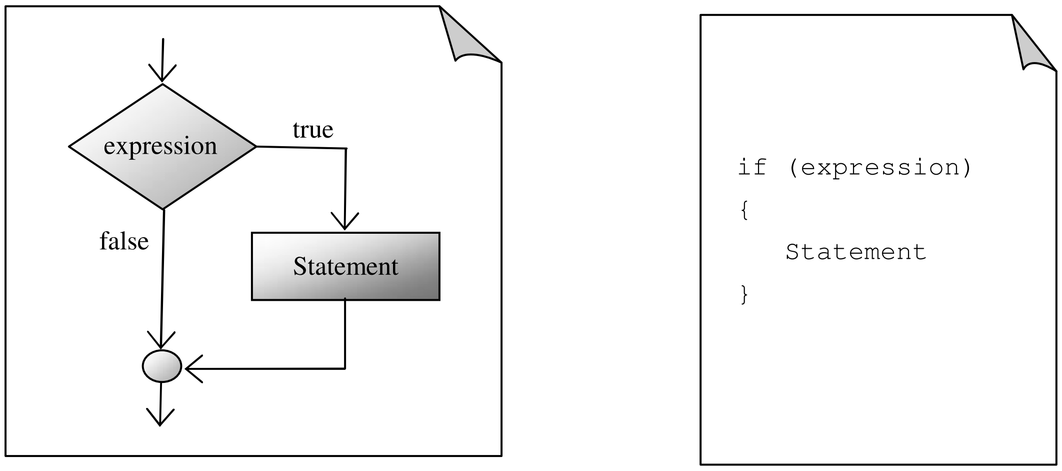
2. *Looping Statements* — In this category, C language provides the following statements
   - *for* statement
   - *while* statement
   - *do – while* statement
3. *Jumping Statements* — In this category, C language provides the following statements
   - *break* statement
   - *continue* statement
   - *goto* statement

1.5.1 Decision Making Statements

These statements allow the execution of selective statements based on certain decision criteria.

1.5.1.1 The *if* Statement

The syntax of *if* statement is



```
if (expression)
{
   Statement
}
```

(*a*) Logic Flow Control of *if* Statement

(*b*) Code of *if* Statement

Figure 1.1: Logic flow control and code of *if* statement

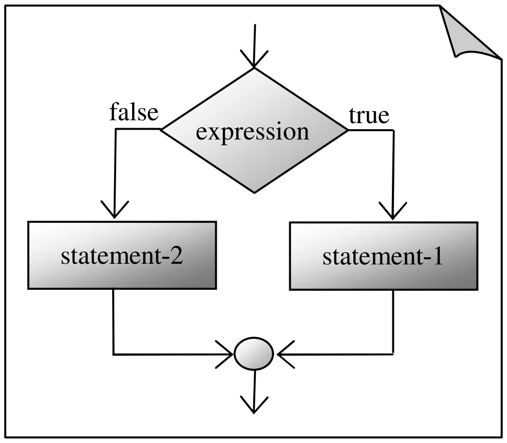
The *expression* may represent a relation expression, a logical expression, a numeric variable or a numeric constant. The specified expression may be a simple expression or compound expression.

The expression in C language evaluates to a *zero* or *non-zero* value. If expression evaluates to a *non-zero* value, then the statements in the statement-block are executed; otherwise they are bypassed.

1.5.1.2 The *if – else* Statement

The *if* statement executes a single statement (simple or compound), when the specified expression evaluates to a *non-zero* value. It does nothing when it evaluates to a *zero* value. *Is there any way whereby a one statement is executed if the expression evaluates to a non-zero value and another statement if the expression evaluates to a zero value*? The answer is yes.

This objective is achieved by using *if – else* statement whose syntax is



(*a*) Logic Flow Control of *if - else* Statement

```
if (expression)
{
   statement-1
}
else
{
   statement-2
}
```

(*b*) Code of *if - else* Statement

Figure 1.2: Logic flow control and code of *if-else* statement

If expression evaluates to a non-zero value, the *statement-1* is executed and the *statement-2* is bypassed. However, if the expression evaluates to a zero value, the *statement-1* is bypassed and the *statement-2* is executed.

1.5.1.3 The else – *if* Construct

If the nesting of *if* or *if-else* statements gets deep and deep, the program becomes more and more difficult to read. In that case *else-if* construct is handier. But it cannot be used as standalone statement, *i.e.*, it can only be used in conjunction with opening *if* statement.

The syntax of *else-if* construct is given in Figure 1.3.

```
if ( expression-1 )
   statement-1
else if ( expression-2 )
   statement-2
else if ( expression-3 )
   statement-3
    :
   :
else if ( expression-n )
   statement-n
else
   statement-s
```

Figure 1.3: Code of *else if* Construct

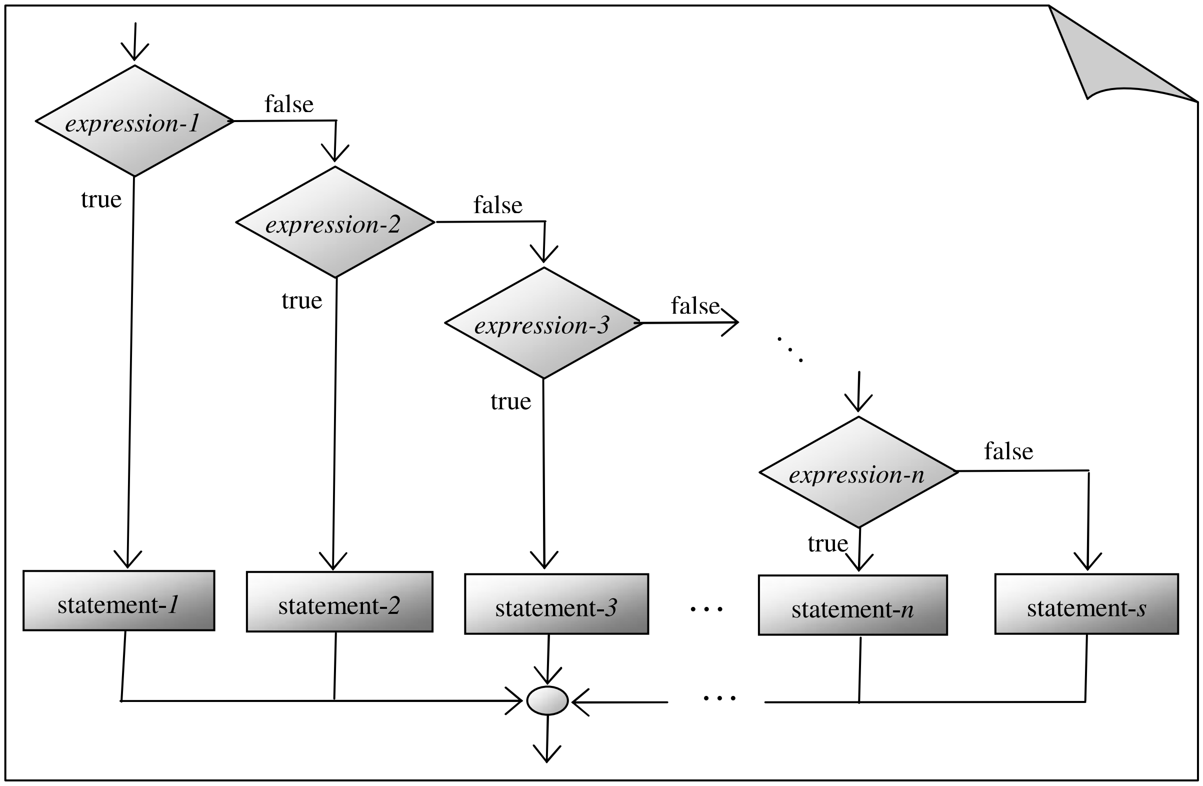


Figure 1.4: Logic Flow Control of *else if* Construct

The expressions are evaluated in order, and if any expression is true then the statement block associated with it is executed, and this terminates the whole chain. The last *else* part handles *none of the above* or default case where none of the specified expressions are satisfied. This sequence of *if* statements is the most general way of writing a multi-way decision.

1.5.1.4 The *switch* Statement

The *switch* statement provides an alternative to *else if* construct. The *switch* statement has more flexibility and a clearer format than *else if* construct. The syntax of *switch* statement is

```
switch ( expression )
{
   case val-1 : statement-1
                break;
   case val-2 : statement-2
                break;
   case val-3 : statement-3
                break;
   :
   case val-n : tatement-n
               break;
   default    : statement-d
}
```

Figure 1.5: Code of *switch* Statement

If *expression* takes any value from *val-1*, *val-2*, *val-3*, ..., *val-n*, the control is transferred to that appropriate case. In each case, the statements are executed and then the *break* statement transfers the control out of *switch* statement. If no *break* statement is used following a case, except the last one in the absence of *default* keyword, the control will fall through to the next case.

If the value of the *expression* does not match any of the case values, control goes to the *default* keyword, which is usually at the end of the *switch* statement. The use of the *default* keyword can be of a great convenience. If there is no *default* keyword, the whole *switch* statement simply terminates when there is no match.

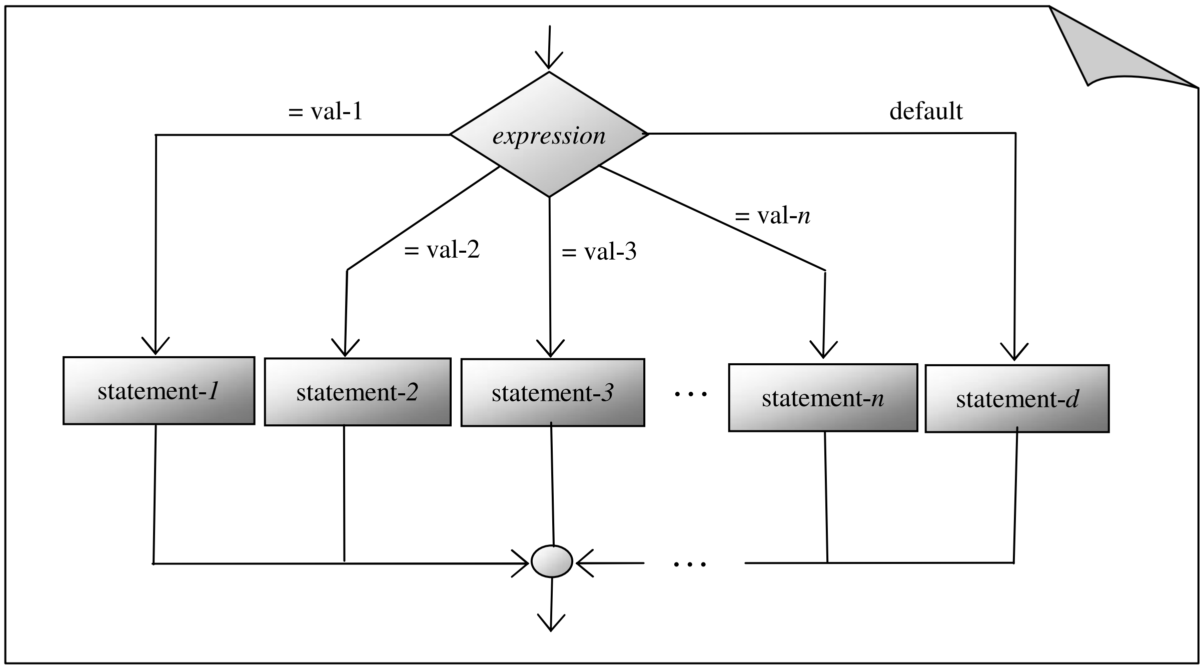


Figure 1.6: Logic Flow Control of *switch* Statement

However, if you want that a same statement is to executed for more than one value of the *expression*, then we have to code these *cases* one after the other, and then the specified statement as shown below:

```
switch ( expression )
{
       :
   case val-4 :
   case val-5 :
   case val-6 : statement-block
                break;
       :
}
```

The *statement-block* will be executed whenever the value of the *expression* is *val-4*, *val-5*, or *val-6*.

The following are some more points about switch statement:

- The *expression* of switch statement must be of type integer or character type.
- The *default* case need not be used as the last case. It can be placed at any place.

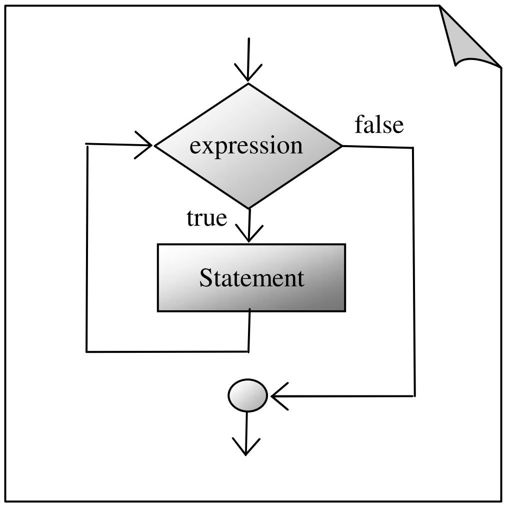
- The case values must be integer constants or character constants. These values can also be constant expressions as these are evaluated at compilation time. For example, if expression 5 + 7 appears as case value, it will be evaluated to value 12 at compilation. Similarly, if expression 5 < 7 appears as the case value, it will evaluated to 1 by compiler at compilation time.
- The case statements can only be used within the scope of switch statement otherwise the compiler will display en error message similar to *"Case outside of switch statement in function …"*.
- The case values need not be in any specific order.

1.5.2 Looping Statements

These statements allow the execution of some set of statements repeatedly till either for a known number of times or till certain conditions are met. In this section, we will see their working and will illustrate their use with well-designed examples.

1.5.2.1 The *while* Statement

The *while* statement is suited for problems where it is not known in advance that how many times a statement or a statement-block will be executed.



```
while (expression)
{
   Statement
}
```

(*a*) Logic Flow Control of *while* Statement

(*b*) Code of *while* Statement

Figure 1.7: Logic flow control and code of *while* statement

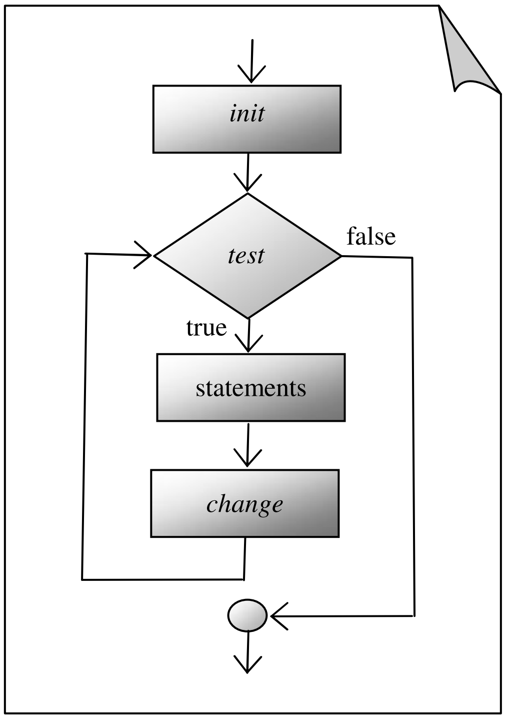
where *expression* is a constant, a variable or an expression. The *statement* is executed repeatedly till the *exp* evaluates to a *non-zero* value. Whenever *exp* evaluates to a *zero* value, the execution of *while* statement will terminate and control will pass to a statement immediately following it.

Though in C language there are certain exceptions, but in general, the following points must be kept in mind while using the *while* statement:

- There must be a statement prior to *while* statement that initializes the *exp*.
- In the statement-block, there must be a statement that modifies the expression.

1.5.2.2 The *for* Statement

The *for* statement is suited for problems where the number of times a statement or statement-block will be executed is known in advance.



(*a*) Logic Flow Control of *for* Statement

```
for ( init; test; change )
{
   statements
}
```

(*b*) Code of *for* Statement

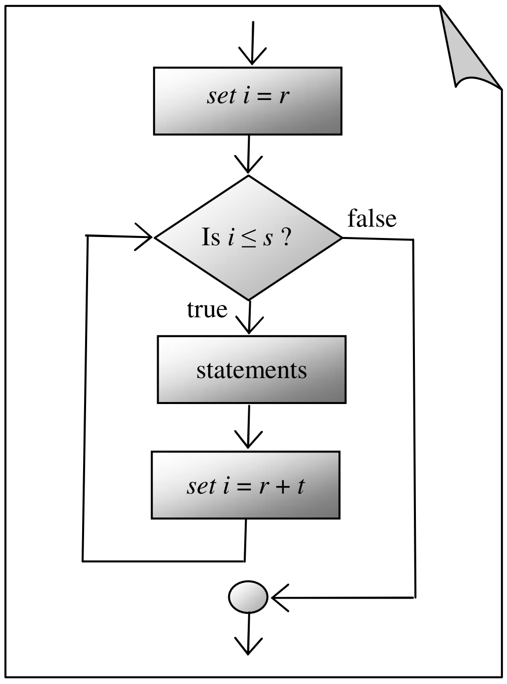
Figure 1.8: Logic flow control and code of *for* statement

where *init* is an expression to initialize the counter, the *test* is an expression to see when to stop iterating, and *change* is an expression to change the counter for each pass of the loop. The *init* and *change* parts can have more than one statement separated by a comma.

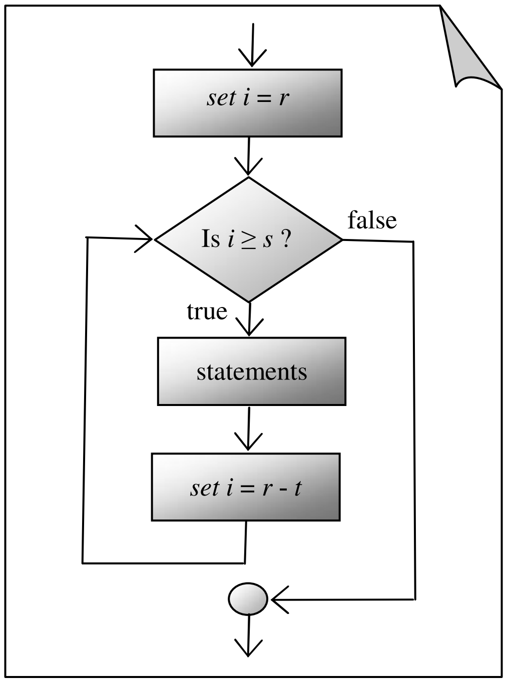
The change for the counter can be positive as well as negative, as illustrated below:

```
          :
for ( k = r; k <= s; k += t )
{
   // statements
}
         :
```

```
          :
for ( k = r; k >= s; k -= t )
{
   // statements
}
          :
```



(*a*) When step size *t* is positive

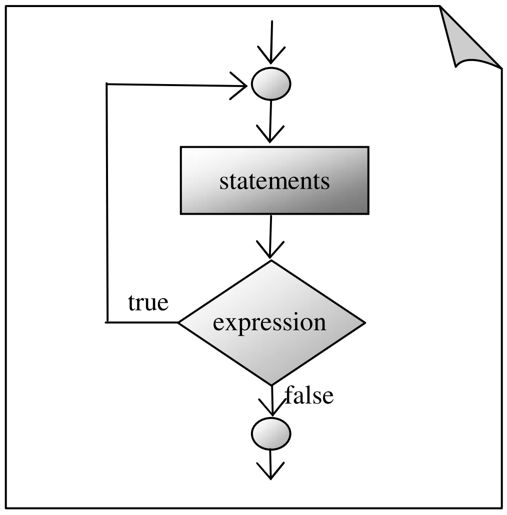


(*b*) When step size *t* is negative

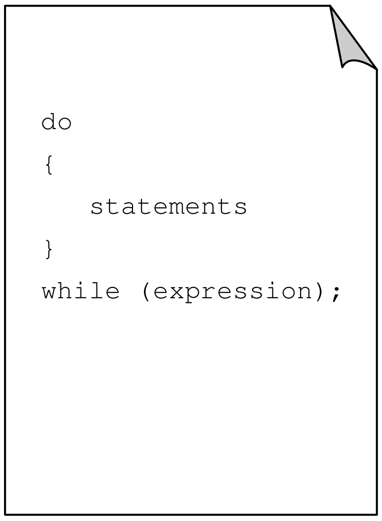
Figure 1.9: Working of *for* statement for positive and negative step size

1.5.2.3 The *do - while* Statement

The *do-while* statement, like *while* statement, is also suited for problems where it is not known in advance that how many times a statement will be executed.



(*a*) Logic Flow Control of *do-while* Statement



```
do
{
    statements
}
while (expression);
```

(*b*) Code of *do-while* Statement

Figure 1.10: Logic flow control and code of *do-while* statement

where *exp* is a constant, a variable or an expression. The *statement-block* is executed repeatedly till the *exp* evaluates to a *non-zero* value.

1.5.2.4 Nested *while, for* and *do-while* Statements

Just as *if* statements can be nested, these statements can also be nested. The inner loop is executed from the beginning for each iteration of the outer loop. The following sections of code show the nesting of looping statements within their own types.

```
for(i=0;i<m;i++)         i=0;                   i=0;
{                        while(i<m)             do
   :                     {                      {
   for(j=0;j<n;j++)         :                      :
   {                        j=0;                   j=0;
      :                     while(j<n)             do
                            {                      {
   }                           :                      :
}                             j++;                   j++;
                            }                      } while(j<n);
                            i++;                   i++;
                         }                      } while(i<m);
```

In general, it is possible to nest while and do-while statements inside *for* statement, *for* statement inside the *while* and *do-while* statements, *i.e.*, all sort of nesting combinations are permitted.

1.5.3 Jumping Statements

These statements transfer the control from one part of the program to another part. In this section, we will see their working.

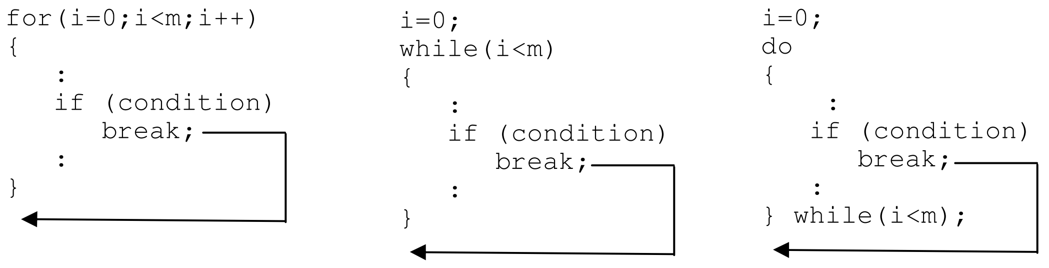
1.5.3.1 The *break* Statement

The *break* statement is always used inside the body of the switch statement, and looping statements.

In *switch* statement, it is used as the last statement of the statement block of every case except the last one. When executed, it transfers the control out of *switch* statement and the execution of the program continues from the statement following *switch* statement.

```
switch ( expression )
{
    case val-1 : statement-block-1
                 break;
    case val-2 : statement-block-2
                 break;
    case val-3 : statement-block-3
                 break;
       :
       :
    case val-n : statement-block-n
                 break;
    default    :
                statement-block-default
}
```

In *for*, *while* and *do-while* statements, it is always used in conjunction with *if* statement. Note that it is never used with *if* statement if it is not part of the body of the looping statement. When executed, it transfers the control out of looping statement and the execution of the program continues from the statement following looping statement.



```
for(i=0;i<m;i++)
{
   :
   if (condition)
      break;
   :
}
```

```
i=0;
while(i<m)
{
   :
   if (condition)
      break;
   :
}
```

```
i=0;
do
{
   :
   if (condition)
      break;
   :
} while(i<m);
```

In simple terms, we can say that the *break* when executed terminates the execution of the loop.

Consider the following code segment

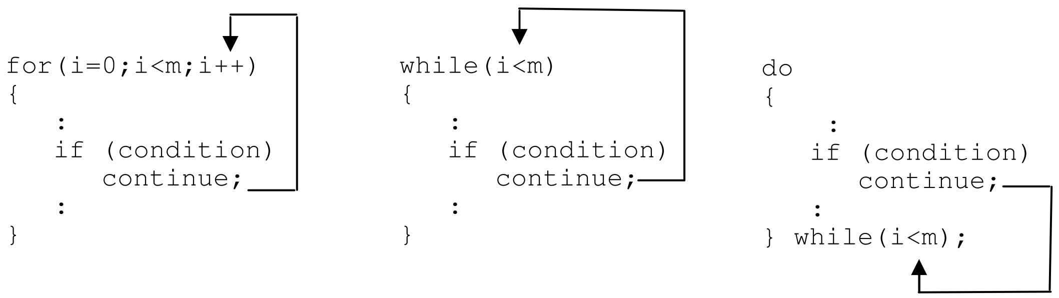
```
i = 0;
while ( i <= 100 ) {
   . . . . . .
   scanf("%d", &k);
   if ( k == 0 )
      break;
   m = i / k;
   . . . . . .
}
```

The execution of the *while* loop will be terminated as soon as the user enters value 0 for variable *k* as input or the value of variable *i* exceeds 100, which ever happens earlier.

1.5.3.2 The *continue* Statement

The *continue* statement is always used inside the body of the looping statements. In looping statements, sometimes a situation may arise where we want that from a given statement onward, the rest of the statement up to the last statement of the loop are to be bypassed. This task is accomplished by using *continue* statement.

The *continue* statement transfers the control to the beginning of the next iteration of the loop thus bypassing the statements which are not yet executed. But note that the *continue* statement is always used in conjunction with the *if* statement.



```
for(i=0;i<m;i++)
{
   :
   if (condition)
      continue;
   :
}
```

```
while(i<m)
{
   :
   if (condition)
      continue;
   :
}
```

```
do
{
   :
   if (condition)
      continue;
   :
} while(i<m);
```

In simple terms, we can say that the continue statement when executed terminates the current iteration of the loop.

Consider the following segment

```
i = 0;
while ( i <= 100 )
{
   . . . . . .
   if ( i  % 10 == 0 )
      continue;
   . . . . . .
}
```

The use of *continue* statement in conjunction with *if* statement skips the rest of the statements of the *while* loop for values of *i* divisible by 10.

1.5.3.3 The *goto* Statement

Though in C language, every program can be written without the use of *goto* statement, still a situation may arise where we are forced to use *goto* statement. For example, to transfer a control from deeply nested statements, such as jumping out of two or more loops at once. Please note a *break* statement can not be used directly since it exits only from the innermost loop.

The *goto* statement can transfer the control to any part of the program. The target destination of the *goto* statement is marked by a label. The syntax for using *goto* statement is

```
   // some statements
   goto label;

   // some more statements

label:
   // still more statements
```

```
   // some statements
label:

   // some more statements

   goto label;

   // still more statements
```

Though in C, every program can be written without the use of *goto* statement, still a situation may arise where we are forced to use *goto* statement. For example, to transfer a control from deeply nested statements, such as jumping out of two or more loops at once. Please note a *break* statement cannot be used directly since it exits only from the innermost loop.

1.6 MEMORY USE IN C

Let us understand what is going on in computer’s main memory when running a C program. When a C program runs, the allocated memory to the program by the operating system is divided into several different sections/areas.

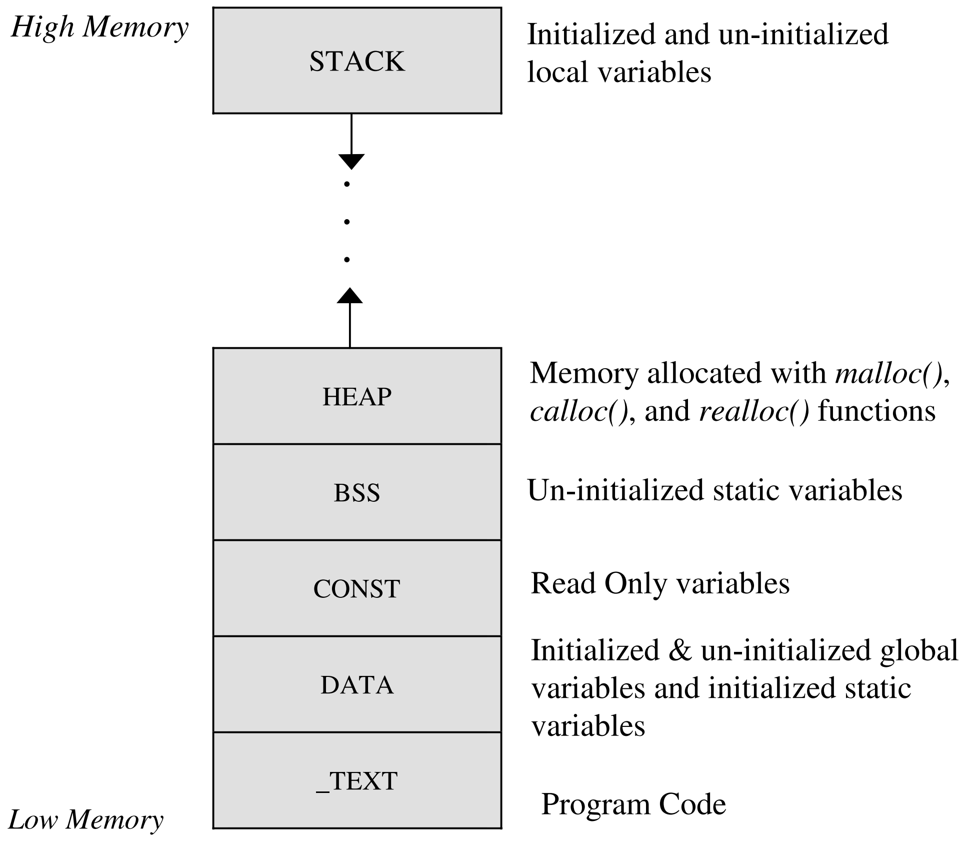
These sections/areas are — *code area*, *data area*, *stack area*, and *heap*.

Code Area — This is the first area, named *_TEXT*, where the program resides. This area, usually, is never changed when a program runs.

Data Area — This area is the second one and is further divided into three sections —

- First is named *DATA* that contains un-initialized global data, initialized global data, and initialized static data.
- Second is named *CONST* that contains read only data, *i.e.*, constants stored in variables.
- Third is named Block Started Segment (*BSS*) that contains un-initialized static data.

Stack Area — This is the area that is set up at the top of available memory and grows towards lower memory as it is filled.



High Memory
STACK
Initialized and un-initialized local variables
HEAP
Memory allocated with *malloc()*, *calloc()*, and *realloc()* functions
BSS
Un-initialized static variables
CONST
Read Only variables
DATA
Initialized & un-initialized global variables and initialized static variables
_TEXT
Program Code
Low Memory

Figure 1.11: Use of Memory in a C Program

Heap — This area is located on the top of code and data area. It grows just like stack except it grows towards high memory rather than towards lower memory.

The C language runtime system automatically maintains stack for us, but the allocation of memory from heap and its use is programmer's responsibility. To allocate memory from heap, library functions *malloc()*, *calloc()*, and *realloc()* functions are used. Once the memory allocated to the program from heap is not required further in the program, it is returned to the heap using library function *free()*.

1.7 POINTERS

Pointers are one of the most important features of C language. Pointers, by most people, are regarded as one of the most difficult topics in C. Let me assure you that it is more a myth than a fact. I would prefer to say that pointers are one of the most delicate aspects of C language, and you know delicate things need very careful handling. Therefore, pointers need very careful handling.

There are many reasons for using pointers. Some of them are enumerated below:

- A pointer allows a function or a program to access a variable (better we say memory location) outside the preview of function or a program. In fact, using a pointer, your program can access any memory location in the computer's memory.
- Since using *return* statement, a function can only pass back a single value to the calling function, pointers allows a function to pass back more than one value by writing them into memory locations that are accessible to the calling function.
- Use of pointer increases makes the program execution faster.
- Using pointers, arrays and structures can be handled in more efficient way.
- Without pointers, it will be impossible to create complex data structures, such as *linked lists*, *trees* and *graphs*.
- To communicate with operating system about memory. For example, the function *malloc()* returns the location of free memory block by using a pointer and the function *free()* returns the memory block pointed to by a pointer to the operating system.

In totality, we can say that the real power of C language lies in the judicious use of pointers. Therefore, every reader of C language must learn and master the art of using pointers.

1.7.1 Understanding Pointers

When you give a command to run a program, the operating system first finds a free block of requisite size from computer (main) memory, and then loads the program into that block. Even in that block, usually, the program's code is loaded in one part and the program's data in another part of the block. Each of the data operand is stored in a cell and the system associates each of the variables with these addresses. This is illustrated in Figure 1.12.

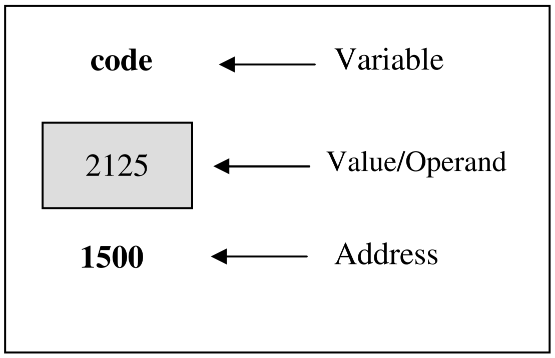


Figure 1.12: Representation of a Variable in Memory

In order to access the data operand either we use either variable name or we can use address of the memory cell. Since these addresses are simply positive integer numbers, they can also be assigned to some variables and stored in memory, like any other variable. Such variables that hold addresses are known as *pointers*. A pointer is, therefore, nothing but a variable that contains address of another variable in memory.

Since pointer is a variable, its value is also stored in memory in another address. Suppose we assign the address of variable *code* to a variable *ptr*. The link between the variables *ptr* and *code* can be visualized as shown in Figure 1.13.

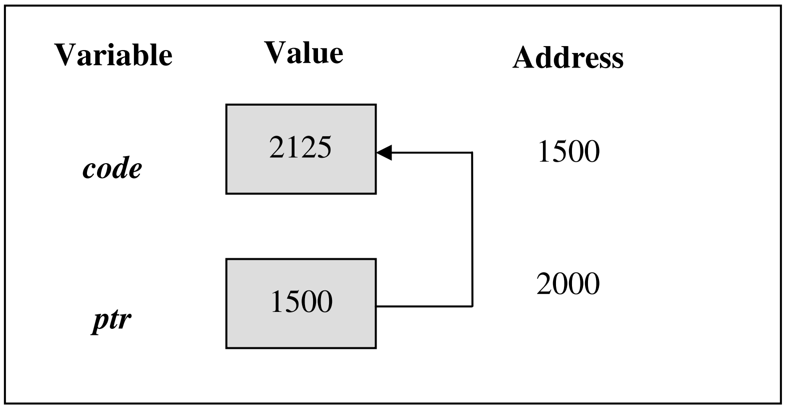


Figure 1.13: Pointer as a Variable

Since the value of the variable *ptr* is the address of the variable *code*, we can access the value of variable *code* by using the value of variable *ptr* and therefore, we say that the variable *ptr* points to the variable *code*, and hence *ptr* gets the name pointer.

1.7.2 Accessing Address of a Variable

The actual address of a variable is a system dependent. During compilation and linking, addresses are assumed to be relative to some base address, usually 0. When the operating system loads the program in a free block, all the address are transformed with relative to address of the first memory location of the free block. Hence, we do not know the address of a variable. Therefore, a question arises — *how can we determine the address of a variable*? This is done with the help of *address of* operator (&).

Consider the following statement

```
ptr = &code;
```

It will assign the number 1500 (address of variable *code*) to the variable *ptr*.

1.7.3 Declaring and Initializing Pointers

Like other variables, the pointer variables are to be declared first in the declaration block to tell the compiler that to which kind of values these variables will be pointing.

The syntax for declaring pointer variable is

```
type *ptrName;
```

where *type* is a *pre-defined* or *user-defined* data types and indicates that the pointer will point to the variables of that specific data type, and character '*' indicates that variable is a pointer variable.

For example, the statement

```
int *intPtr;
```

declares a pointer variable *intPtr* that can point to an integer type variable.

The pointer variable(s) declaration can also be combined with the declaration of other variables provided they are of same type. For example, the statement

```
int x, *ptrX;
```

declares an integer variable *x* and a pointer variable *ptrX*.

Once a pointer has been declared, it must be initialized prior to its use. It is important to note here that like other variables, a pointer variable will take a garbage value, which can be an address of any storage location. Therefore, if not properly initialized, a pointer may point to any location in memory including those locations where operating system is running, and your system may hang-up, *i.e.*, stop responding. Such un-initialized pointers are sometimes referred to as *dangling pointers*.

1.7.4 Accessing a Variable through a Pointer

Once a pointer has been assigned the address of a variable, the question remains as how to access the value of the variable using the pointer. This is done by using operator * (asterisk), usually known as the *indirection or dereferencing* operator.

Consider the following statements

```
int *ptrX, x = 10, y;
     :
ptr_x = &x;
     :
y = *ptrX;
printf( "Value of variable x = %d\n", x );
printf( "Value of variable pointed to by ptrX = %d\n", *ptrX );
printf( "Address of variable x = %u\n", ptrX );
printf( "Value of variable y  = %d\n",  y );
    :
```

The declaration statement tells the compiler that *ptrX* is pointer variable that will point to an integer value, *x* is an integer variable and it also initializes the variable *x* with value 10, and *y* is another integer variable.

And suppose that during the execution of the above segment, a memory location with address 1098 is set aside for variable *x*. The output of the above segment will look like:

```
Value of variable x = 10
Value of variable pointed to by ptrX = 10
Address of variable x = 1098
Value of variable y = 10
```

Note that writing the statements

```
ptrX = &x;
y = *ptrX
```

is equivalent to writing

```
y = *&x;
```

or for more clarity as

```
y = *(&x);
```

which in turn is equivalent to writing

```
y = x;
```

It is important to note that assignment of pointers and addresses is always done symbolically, by means of symbolic names. You cannot access the value stored at address 1098 by writing *1098. It will not work.

1.7.5 Pointer to a Pointer

As mentioned, a pointer is variable that holds the address of another variable. This concept can be further extended. We can have a variable that hold an address of a variable that in turn holds an address of another variable. This type of variable is known as *pointer to a pointer*. The underlying concept is known as *double indirection*. This concept can be further extended to triple level, fourth level, and so on. The use of pointer to a pointer usually occurs where calling function passes pointer argument by reference and the called function can modify the contents of a pointer variable.

A pointer to a pointer will be declared as

```
    int **ptr;
```

The value pointed to by a pointer to another pointer will be accessed as

```
    **ptr;
```

1.7.6 Pointer to a Function

A pointer to a function is declared as

```
    ReturnType (*fname)(list of arguments);
```

However, if the declaration is

```
    ReturnType *fname(list of arguments);
```

The compiler will consider it as a prototype of a function that returns a pointer.

1.8 MEMORY ALLOCATION

It is important to note that anything that needs to be executed (programs) or processed (data) must be loaded into memory before it can be executed or processed. Therefore, for every program to be executed or data to be processed, some memory must be allocated for them. The memory allocation can be done in two ways – *statically* and *dynamically*. Let us look at both the ways.

1.8.1 Static Memory Allocation

When you give a command to run your program, the operating system allocates a block of memory for your program, loads it from the secondary storage (Floppy disk, hard disk, CD or DVD, etc.) into that block and then initiates its execution. When the program finishes its execution, the memory occupied by the program is automatically released. It means that the memory requirements are determined prior to execution of your program, and thereafter your program neither can acquire more memory nor can release the free memory, if not required.

The above situations depicts a situation where the amount of memory required is known before hand, and the compiler then can determine the memory requirements for the data as well as the instructions. In such a situation, memory allocation is known as *static memory allocation*.

1.8.2 Dynamic Memory Allocation

Further, it is important to note that the memory requirements for the instructions are always fixed. However, there may be situation when the exact memory requirements for the data may not be known in advance or the data requirements may vary from one program execution to another program execution, *i.e.*, memory requirements for the data are dynamic.

Therefore, when the amount of memory is not known before hand for a particular data item(s), then it is allocated at the execution time, *i.e.*, when the program is running. In such a situation, memory allocation is known as *dynamic memory allocation*.

1.8.3 Heap/Free Store

As mentioned earlier, every program is provided with a pool of unallocated memory that it can utilize during execution. This pool of unallocated memory is known as *heap* or *free store*. Therefore, whenever the memory of requisite size (amount) is required by the program, it is taken from the free store, and when the previously allocated memory is not required further, it is returned back to the free store.

The memory requirements for the instructions (program code) are always fixed.

1.9 DYNAMIC MEMORY MANAGEMENT

As mentioned earlier, when you give a command to run your program, the operating system allocates a block of memory for your program, loads it from the disk into that block and then initiates its execution. When the program finishes its execution, the memory occupied by the program is automatically released.

It means that the memory requirements are determined prior to execution of your program. Thereafter, your program can neither acquire more memory nor can release the free memory, if not required. However, C language provides a set of library functions (listed in Table 1.1), called *dynamic memory management functions*, to allocate and de-allocate memory at execution time, *i.e.*, dynamically.

Table 1.1: Memory Management Functions

Function Name	Description
malloc	Allocates memory from heap.
calloc	Allocates memory from heap and initializes the allocated memory to zeros.
realloc	Readjusts the existing block and copies the contents to the new location.
free	Deallocates a block allocated by *malloc*, *calloc* and *realloc* functions.
coreleft	Returns a measure of unused memory.

By allocation, we mean that your program can obtain as much memory as required by your program even during execution of your program. By de-allocation, we mean that the memory acquired dynamically can be released at any time during your program execution. It is important to note that memory allocated dynamically, must be de-allocated before your program finishes its execution. Otherwise, even if your program terminates, memory allocated dynamically is never released automatically, and from operating system point of view that memory is still in use. Therefore, if you run the same program many times, many users are using your program concurrently, the operating system may run out of memory. Each of these memory management functions is described below.

1.9.1 Dynamic Memory Allocation with *malloc()* Function

The *malloc()* function dynamically allocates memory from heap. The prototype for *malloc()* function is

```
void *malloc(size_t size);
```

It takes one argument that specifies the size of the block in bytes. The function returns a pointer to the allocated memory on success or a null pointer (0) in case of failure.

1.9.2 Dynamic Memory Allocation with *calloc()* Function

The *calloc()* function dynamically allocates memory and automatically initializes the memory to zeroes. The prototype for *calloc()* function is

```
void *calloc(size_t nitems, size_t size);
```

It takes two arguments. The first argument is the number of elements and the second argument is the size of each element. The function returns a pointer to the allocated memory on success or a null pointer (0) in case of failure.

1.9.3 Changing Size of Dynamically Allocated Memory with *realloc()* Function

The *realloc()* function changes the size of previously dynamically allocated memory with *malloc()*, *calloc()* or *realloc()* functions. The original contents held in the previously allocated memory are not changed provided the memory allocated is larger than the amount of memory previously allocated otherwise the contents are unchanged upto the size of new object.

The prototype for *realloc()* function is

```
void *realloc(void *block, size_t size);
```

It takes two arguments. The first argument is the pointer to the original object and the second argument is the new size of the object.

Depending on the values of these arguments, following things can happen

- If the pointer to the original object is 0 (NULL), the *realloc()* function works identical to *malloc()* function.
- If size of the new object is 0 and pointer is not 0, the memory of the object is freed.
- If the pointer is not 0 and the size of the new object is greater than zero, *realloc()* function tries to allocate new block of memory. If new space cannot be allocated, the object pointed to by the pointer is unchanged.

The *realloc()* function returns either a pointer to the reallocated block or a null pointer.

1.9.4 Deallocating Memory with *free()* Function

The *free()* function deallocates a memory block previously allocated with *malloc()*, *calloc()* or *realloc()* functions. The prototype for *free()* function is

```
void free(void *block);
```

It takes one argument that specifies the pointer to the allocated block.

But it is important to note that only the allocated block is deallocated, the pointer variable is not deleted.

Once the block is freed, an attempt to dereference a pointer variable yields undefined results. Such a pointer variable is called a *dangling pointer*. Therefore, it is recommended that after freeing the allocated block, set the pointer variable to NULL explicitly as shown in the next program. Dereferencing a null pointer is always safe.

1.10 DEBUGGING POINTERS

Pointers can be the source of mysterious and catastrophic program bugs. Common bugs related to pointers and memory management is *dangling pointers*, *memory leaks*, and *allocation failures*. Each of these problems is discussed below. Every student must understand these carefully.

1.10.1 Problem of Dangling Pointers

The most common problem with pointers is that the programmer fails to initialize a pointer with a valid address. Such an initialized pointer, referred to as *dangling pointer*, can end up pointing anywhere in memory that may include the program code itself or to the code of the operating system. If the programmer assign a value to such pointer, the value will overwrite the program or operating system instructions, and the computer program may show undesirable behaviour or may even crash, *i.e.*, stop responding. When the system stop responding, we say the system has hanged up and the only option left is to reboot the system. Therefore, care must be taken to initialize pointers with valid address.

1.10.2 Problem of Null Pointer Assignment

One particular situation that happens is when the pointer points to address 0, which is called NULL. For example, this may happen that if the pointer variable is declared as global since global variables are initialized to 0. Likewise, this can also happen for a local un-initialized pointer variable, particularly for local static variables, they are also initialized to 0. When it happens, and no catastrophic situations had occurred, then the system will display a message "*Null pointer assignment*" on termination of the program.

1.10.3 Problem of Memory Leaks

Another common problem with pointers is that of *memory leak*. Memory leak is a situation where the programmer fails to release the memory allocated at run time in a module. Note that when memory is allocated, a pointer variable is used to hold the address of the allocated block, however, when the module completes its execution, the pointer variable goes out of scope and there will be no way to reach that memory block. Therefore, care must be taken to release the allocated memory block in a module where memory was allocated.

1.10.4 Problem of Allocation Failures

Still another problem with pointers is that of *allocation failures*. An *allocation failure* is a situation when the program through *malloc()*, *calloc()*, or *realloc()* function request for a block of memory, and the operating system could not fulfill the request of the program because the sufficient memory may not be available in the free pool. In that case, these functions return a NULL pointer indicating the allocation failure. Therefore, the programmer should take care of allocation failures and provide a safe exit in their programs on such failures.

1.11 ARRAYS AND STRINGS

An *array* is a collection of homogeneous data elements, *i.e.,* of same data type, described by a single name, and each individual element of an array is referenced by a *subscripted* variable, formed by affixing to the array name a *subscript* or *index* enclosed in brackets. The term subscript has the same meaning as in the mathematical notation. If single subscript is required to refer to an element of an array, the array is known as *one-dimensional* or *linear array*, and if two subscripts are required to refer to an element of an array, the array is known as *two-dimensional* array, and so on. Analogously, the arrays whose elements are referred by two or more subscripts are called *multi-dimensional* arrays.

Using array of characters, we can store string data. But do remember that as such strings are not directly supported, though sometimes loosely we refer to array of characters as strings. Array of characters or the programming languages to manipulate text such as words and sentences use strings. As with other data types, strings can also be used as constants or variables.

1.12 STRUCTURES

A *structure* is a collection of data elements, called *fields*, which may be of different type. Individual elements of a structure variable are accessed using dot operator ('.'). If a pointer is used to point to a structure variable, then arrow operator ('->') is used. But from the data structures point of view, we have interest in *self-referential structures*. A *self-referential structure* is a structure that include an at least one element that is a pointer to itself. Self-referential structures are used in building complex data structure such as *linked lists*, *trees* and *graphs*, etc.

Following are some examples of complex data structures and the corresponding self-referential structures to represent these data structures:

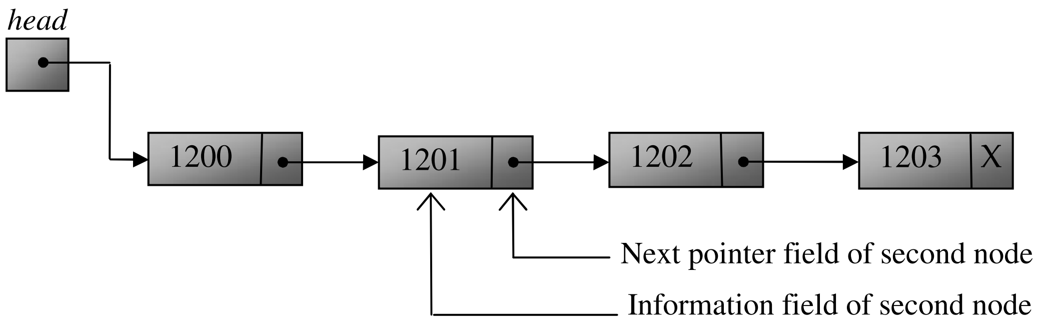

head
1200
1201
1202
1203
X
Next pointer field of second node
Information field of second node

Figure 1.14: Linear linked list of integer values with nodes 4

To represent the above linear linked list in memory, we need following declarations

```
typedef struct nodeType
{
    int info;
    struct nodeType *next;
} node;

node *head;
```

To represent the above linear linked list in memory, we need following declarations

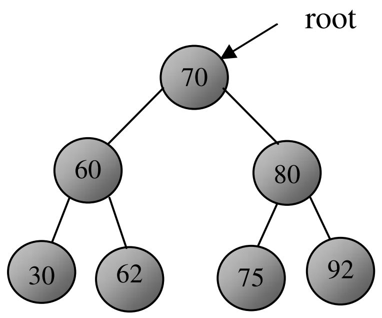
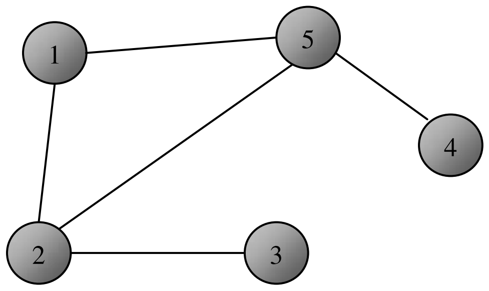


Figure 1.15: Binary tree

```
typedef struct nodeType {
    struct nodeType *left;
    int info;
    struct nodeType *right;
} node;

  node *root;
```



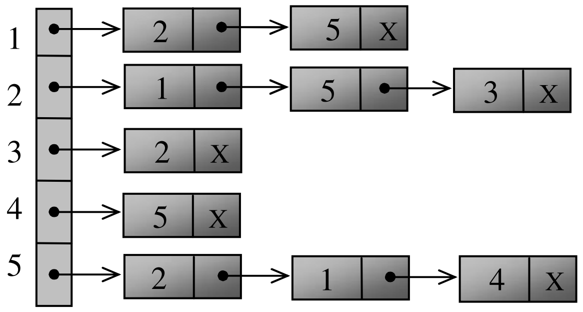


Figure 1.16: Undirected graph **Figure 1.17:** Adjacency list for graph of Figure 1.16

To represent a graph in memory, one of the best representation schemes is adjacency list as shown in Figure 1.17.

To represent a adjacency list, for a graph with maximum of 5o vertices, we need following declarations

```
#define MAX 50
typedef struct nodeType
{
   int vertex;
   struct nodeType *next;
} node;

node *adj[MAX];
```

1.13 FUNCTIONS

The best way to handle complex problems is to split the problem into small problems, called *subproblems* that can be handled easily. This we are used to do in our day-to-day working. Once all these subproblems are solved separately, solutions of these subproblems can be synthesized to generate the solution of the given complex problem.

Like was, if program to be developed is very large and complex, it is always recommended to split into two or more functions (subprograms) that can be developed separately, tested separately and then integrated.

1.13.1 Defining a Function

The general form of function definition is

```
ReturnType functionName(LostOfArguments)
{
   Local declarations
   . . . . . . . . . . .
   Executable body
   . . . . . . . . . . .
}
```

Following are some points to remember while defining functions:

- The first statement of the function must be the function defining statement specifying the return type, name and formal arguments.
- If the type of the function is omitted, then it is assumed of type integer (*int*). But it is recommended that the return type must be explicitly specified.
- Rules for naming a function are same as for variable names.
- The formal arguments should neither be constants nor expressions.

1.13.2 Function Prototype

A function *prototype* is a function declaration that specifies the return type and the data types of the arguments. The main purpose of the function prototyping is to prevent errors caused by data type mismatches between the values passed to a function and the type of values function is expecting.

The general form of function prototype is

```
ReturnType functionName(ListOfArgumentTypes);
```

The following is an example of function prototype for a function that computes factorial of a given positive integer number:

```
int factorial( int k );
```

It is important to note that name of the formal argument is optional. Further, names of formal argument in a function prototype and in the function defining statement can be different.

Therefore, following is equally valid prototype

```
int factorial( int );
```

1.13.3 Accessing a Function

Like library functions, the user-defined functions are accessed from a function simply by its name, including the actual arguments, if any, enclosed within parentheses. However, parentheses must follow the function name even if there is no actual argument to be passed to the function. The actual arguments, if any, must correspond in *number*, *type*, and *order* with formal arguments. The actual arguments can be constants, variable names, subscripted variables, or expressions.

When the name of the function is encountered, the control is transferred to the called function. The formal arguments are replaced by the actual arguments and the execution of the function is carried out. When the *return* statement is executed or last statement has finished its execution, the control is transferred back to the calling function.

Note that the semicolon must follow the function call if it is a solitary statement but not the function definition. However, if the function call occurs as part of another statement then semicolon after the function call may or may not be used. This will be determined the actual place of call.

.

1.13.4 The *return* Statement

The syntax of *return* statement is

```
return [exp];
```

where *exp* can be a constant, variable or expression. Use of parentheses around *exp* is optional. The *return* statement serves two purposes:

- Execution of *return* statement immediately transfers control from the function back to the calling function.
- Whatever is following the *return* statement is returned as a value to the calling function.

The *return* statement need not be at the end of the function. It can be used any where in the function. As soon as it is executed, the control will return to the calling function. A function can contain any number of *return* statements.

There is the key limitation of *return* statement is that it can return only one value. If you want your function to return two or more values to the calling function, you need another mechanism.

1.13.5 Passing Arguments to a Function

The mechanism used to pass data to a function is via argument list, where individual arguments are called *actual arguments*. These arguments are enclosed in parentheses after the function name. The actual arguments must correspond in *number*, *type*, and *order* with formal arguments specified in the function definition. The actual arguments can be *constants*, *variables*, *array names*, or *expressions*.

There are two approaches to passing arguments to a function:

- Call by value
- Call by address, also called call by reference

Let us describe these one by one.

1.13.5.1 Call by Value

In this approach, the names of the actual arguments are used in the function call. In this way the values of the actual arguments are passed to the function. When control is transferred to the called function, the values of the actual arguments are substituted to the corresponding formal arguments and the body of the function is executed. If the called function is supposed to return a value, it is returned via *return* statement.

1.13.5.2 Call by Address

In this approach, the addresses of the actual arguments are used in the function call. In this way the addresses of the actual arguments are passed to the function. When control is transferred to the called function, the addresses of the actual arguments are substituted to corresponding formal arguments and the body of the function is executed. The formal arguments are declared as pointers to *types* that match the data types of the actual arguments. This approach is of practical importance while passing arrays and structures among function, and also for passing back more than one value to the calling functions.

It is important to note that arguments in a function in C language are passed from right to left.

1.13.6 Specifying Argument Data Types

While defining a function, the information to the compiler regarding the *return type* of the function and the data types of the arguments must be provided. As described in the beginning of this chapter that there are two ways to specify the data types of the formal arguments:

1. In the argument list, only the names of the formal arguments are used. The data types of these arguments are specified before the body of the function begins. To illustrate this approach, consider an example of defining a *sampleFunction()* function that takes four arguments. Of these arguments, first is of type integer, second is of type real, third is of type character, and fourth one of type integer.

   The function definition may look like

   ```
   float sampleFunction(arg1,arg2,arg3,arg4 )
   int arg1, arg4;
   float arg2;
   char arg3;
   {
      body of the function
   }
   ```

2. In the argument list itself, the data type of each argument is specified individually. Using this approach the definition of the above *sampleFunction()* function may look like

   ```
   float sampleFunction( int arg1,float arg2, char arg3,int arg4 )
   {
      body of the function
   }
   ```

   Throughout the present text, we will consider the later approach.

I hope with this short recap of C language, you will in a position to take off and enjoy implementing different data structures.

A QUICK REVIEW

In this chapter, we have learnt

- About the enumerated data type.
- About the *void* data type.
- About the *typedef* statement.
- About the working of different control statements.
- About the use of memory by a C program.
- About the different aspects of pointers.
- About the different dynamic memory management functions.
- About the arrays, strings, and structures.
- About the different aspects of functions.

In the next chapter, we will discuss about the various data structures and operations performed on them in brief to give you a feel about what data structures is all about.

REVIEW EXERCISES

1. Explain the usefulness of enumerated data type.
2. Describe the usage of *void* data type.
3. How is *typedef* statement useful?
4. Describe the working of every control statement with suitable examples.
5. With the help of a diagram, demonstrate the use of memory by a C program.

6. What is a pointer? What are the advantages using pointers? Discuss various aspects associated with the use of pointers.
7. Discuss different memory management functions.
8. What is a dangling pointer?
9. What is a memory leak?
10. What is a NULL pointer assignment?
11. How can we declare a pointer to a function?
12. Where we may need to use pointer to a pointer? Illustrate with suitable example.

❑ ❑ ❑