```
8
2
7
6
4
9
10
```

Generally, to read/print array elements, a `for` loop is used starting from 0 to 9. When the value of `i` becomes 10, the loop terminates.

When you accidentally write a statement which involves accessing an array element outside the array bound, C compiler doesn't report neither a compilation error nor a run-time error. It is the responsibility of the user to take care of the array bound.

Program 1.26
Sum of all array elements.

```
#include <stdio.h>
#define  MAX 10
void main()
{
    int a [MAX];     /* declares 10 integer elements */
    int i;
    int  sum = 0;
    /* Read the array elements */
    printf ("Enter 10 integer elements:\n");
    for(i = 0; i < MAX; i++)
         scanf ("%d", &a[i]);
    /* Calculate the sum  */
    for(i = 0; i < MAX; i++)
         sum = sum + a[i];
    /* Print the result  */
    printf("Total = %d\n", sum);
}
```

Sample Run

```
Enter 10 integer elements: 1 2 3 4 5 6 7 8 9 0
Total = 45
```

Initialization of Arrays

Sometimes it is useful to initialize an array at the time of declaration, just like an ordinary scalar variable. The syntax and example is given below:

```
    data_type array_name[size] = {value1, value2, ............};
    int a[5] = {1,2,3,4,5};
    char vowels[] = {'a', 'e', 'i', 'o', 'u'};
```

Note, here the size is missing, as the computer automatically calculates the array size.

```
        int t[5] = {1,2,3,4,5,6};
```

In this example, there is an extra element in the initialization list. This will cause a complication error - `"Too many initializers"`.

```
        int q[5] = {1,2};
```

This example is another possibility where the number of initializers is less than the size. The first two elements of the array will be initialized to 1, 2 and the rest will receive zeros.

1.8.3 Two-Dimensional Arrays

In the single dimensional array (vector), we have used only one index for referencing any element. There are many applications, which require the information to be stored in the form of a table.

Imagine a computer company, ABC Computers Ltd., wishes to store sales figures of individual items, like IBM PCs, software, UPC, etc., for each month. The single dimensional array may not be sufficient for this problem. We can use a tabular structure to store this information.

To declare a two dimensional array in C, you use the following syntax:

```
        int a[ROW_MAX][COL_MAX];
```

Assuming `ROW_MAX = 2` and `COL_MAX = 3`, the compiler allocates 2X3 = 6 locations. That is, 2 rows and 3 columns. To access, for example, 1st row 2nd column, we can write, `a[0][1];`

We shall see an example to understand the working of an array.

Program 1.27
Adding two matrices A and B.

```
#include <stdio.h>
#define  m   3        /* array size 3x3 */
#define  n   3
void main()
{
    int a[m][n];
    int b[m][n];
    int c[m][n];  /* resultant matrix */
    int i, j;

    /* Read MATRIX-A  */
    printf("Enter the array elements for MATRIX-A:\n");
```

```
        for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
             scanf("%d", &a[i][j]);

        /* Read Matrix-B */
        printf("Enter the array elements for MATRIX-B:\n");
        for (i = 0; i < m; i++)
             for(j = 0; j < n; j++)
                  scanf("%d", &b[i][j]);

        /* Add A and B  */
        for (i = 0; i < m; i++)
             for(j = 0; j < n; j++)
                  c[i][j] = a[i][j] + b[i][j];

        /* Print the resultant Matrix-C */
        for (i = 0; i < m; i++)
        {
             for(j = 0; j < n;j++) printf("%d", c[i][j]);
             printf("\n");
        }
}
```

1.8.4 Initializing Two-Dimensional Arrays

An initialization list assigns values to an array in sequential order until the list is exhausted. With a list enclosed by a single set of curly braces, you can omit values from the end of the list like the one shown below:

```
int a[100] = {1,2,3,4,5};
```

Except the first array elements in a, all the rest of the elements would be automatically be initialized to zeros.

Similarly, you can omit values from the end of any dimension of a two-dimensional (or even multi-dimensional) array by using additional set of curly braces. See the following examples:

```
int a[5][3] = {1, 2, 3,
               4, 5, 6,
               7, 8, 9,
               10, 11, 12,
                13, 14, 15};
```

If you want to initialize, only first two columns, then write it as given below:

```
int a[5][3]  = { {1,2},
                 {3,4},
```

Example

```
calling function        called function
main()                  void f1()
{                       {
      :                       :
      f1();                   :
}                       }
```

1.9.4 Return Values and their Types (return statement)

A function can be classified into two types:- one which returns **nothing** (or **void**) and one which returns a value. The syntax of return statement is,

```
(1)return;
(2)return (expression);
```

Or,

```
   return expression;
```

The line named (1) can appear any where in the program and it is normally associated with an `if` statement and it returns *nothing* or *void.* Even without this statement when the closing brace (}) the function is *seen,* the control is transferred back to calling function.

The line named (2) returns the value of the expression (could be a constant value, a variable or an expression). You can either use parentheses around the expression or without parentheses. We shall see two example programs to understand how return statement works.

```
/* Returns the sum of first 10 integers */
int sum()
{
     int total = 0;
     int i;
     for (i = 1; i <= 10;  i++)
           total = total + i;
     return total;
}
```

This piece of code finds the total for numbers from 1 to 10 and returns the sum to the calling program (`total` = 55). The calling function/program can use this function in the following manner:

```
void main()
{
     int t;
     t = sum();
     printf("Addition of first 10 numbers = %d\n", t);
}
```

Major Categorization of Functions

There are two types of functions, which we generally write, namely

- Functions which do not returns any values **void functions**
- Functions, which returns values.

In either case, you can have arguments or need not have arguments. In other words, arguments are optional.

(1) void functions (with no arguments)

When you write a function with *void* return type, the function cannot return any value to the calling function. Some times it is needed to just write a function whose main job is to process some steps and return back to the calling program. For instance, assume that you have to read a two-dimensional matrix, for which we shall write a function:

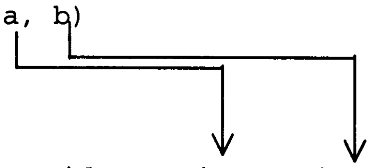
```
void ReadMatrix()
{
      int i, j;
       for (i = 0; j < M; i++)
             for (j = 0; j < N;j++)
                    scanf("%d", &a[i][j]);
}
```

The matrix `a(M×N)` is assumed to be declared globally. We shall see in later sections about passing an array/scalar variable to a function. A very important syntax to be followed in function writing is that the return type should match with the type of value being returned.

(2) Parameter Passing Techniques

The major drawback in the function `ReadMatrix()` is that it can only read the matrix with name `a`. In case, if the same function is to be used for reading any other matrix, the method is to pass the array as an argument. We shall understand the jargons used in a function call.

Function call: `swap(a, b)`



Function Definition: `void swap(int x, int y)`

In a calling sequence, you can have any number of arguments (any type) and the corresponding variable in the function definition is called as parameters. In the example shown above, `a` and `b` are arguments and `x` and `y` are called as **parameters** (or **formal arguments**). The data type of the arguments and the parameters must match, because C is very strict in parameter type checking. Here, `x` and `y` are some times called as placeholders, which behave as good as local variables. There are two types of parameter passing techniques namely.

- Value parameter (One-way communication)
- Reference parameter (Two-way communication)

Value parameter

In the example just explained, the values of a and b are passed to the function swap(). The values of a and b are copied on to x and y before the function starts executing.

We shall write the program with some actual values. The modified program is given below:

```
void swap(int, int);
void main()
{
    int a, b;
    a = 10;
    b = 20;
    printf("%d\t%d\n", a, b);/* before calling swap */
    swap(a,b);
    printf ("%d\t%d\n", a, b); /* after calling swap */
}

void swap(int x, int y)
{
    int temp;  /* local variable */
    temp = x;
    x = y;
    y = temp;
}
```

The value of a and b (i.e. 10 and 20) are copied to x and y. The values of x and y are swapped in the function. However, the swapped values of x and y are not affected to a and b. Because, the parameters are passed by value, i.e., they carry the values to the function and the changed values are not sent back to the calling program (i.e. main()). This is called **one-way communication**. The output of the above program is:

```
before calling swap: 10     20
after calling swap:  10     20
```

There are certain rules to be followed when you write functions. We have already presented one such rule i.e., about the return type and matching of types between arguments and parameters. Now we shall present another example; which prints a line, with a given character:

```
calling function            called function
:
Print('*');                 void Print (char c)
:                           {
:                                 int i;
Print('-');                       for (i = 0; i < 80; i++)
:                                       printf ("%c", c);
Print('-');                       printf("\n");
Print('*');                 }
```

The function `Print()` can now be used for printing a line with a character being sent as a parameter.

Reference parameter

The examples shown earlier cannot return value(s) either through explicit return statement or through parameters or arguments. C does not allow **reference parameter** treatment directly, but it is possible to achieve the same using pointers (see Section 1.11.8).

Functions Returning Values (return statements)

When a function is written to evaluate a given expression or formula, it is always advisable to have the return statement at the end of the function definition. Similarly, a function can be written to return `int`, `long`, `float`, `double`, `char`, pointer, etc. except that **you cannot return an array**. Generally, in any return statement, you can return only a single value to the calling sequence. For example,

```
    :                       int add(int x, int y)
n = add(data1, data2);      {
    :                             return(x + y);
                            }
```

Whenever you declare formal arguments in the function definition, each and every parameter should have appropriate type declared. The following definition is **illegal.**

```
          int add (int x, y)
          {
               .........;
          }
```

The below example program demonstrates the working of a function with a return value.

Program 1.28
To print 10 random numbers

```
#include <stdio.h>
int RandGen(void);    /* prototype  declaration */
void main()
```

```
{
     int i;
     for (i = 0; i < 10; i++)
          printf("Random numbers: %d\n", RandGen());
}
int RandGen()
{
     return (rand());
}
```

The function `rand()` is a library routine which return a random number. When a function definition appears after it is called, then the prototype `return_type` name `(type-1, type-2, ..........., type-n);` should be declared in the beginning of the program itself.

1.10 STRUCTURES

1.10.1 Introduction

When a group of related elements are to be put under a single name, the **structure** definition is useful. We can have different or same data types for the members of the structure. For example, a point in a two-dimensional plane can be represented as a combination of x-coordinate and y-coordinate, or (x, y). Instead of declaring x and y as integer variables, we can group them under a single name called point and define it using a structure.

Similarly, a student record consisting of register number, name, address, marks, etc. may be stored as members of this record structure. Structure is suited, because they are of different data types. The structure definition is again useful for this example.
A typical C structure for Point can be declared as,

```
struct Point
{
     int x;
     int y;
};
```

A structure is basically a **template**, this means that no memory is allocated for Point. You can access the structure members `x` and `y`. However, once we define a variable `p`, we can access its members, as the memory will be allocated for the entire structure (see below).

```
struct Point p;
```

Let us declare another structure for an employee record and is as follows,

```
struct employee
{
      int emp_id;
      char name[20];
      float salary;
};
struct employee e;
```

1.10.2 Accessing structure members

The structure member(s) can be accessed using the following syntax:

```
struct_name.struct_member;
```

To access the member we may use, `e.emp_id;` `e.name;` `e.salary;`. Similarly, the members of `Point` can be accessed as `p.x` and `p.y`.

Now that you have understood the syntax of structure definition, we shall present a small example program, which uses structures.

Program 1.29
Employee record

```
#include <stdio.h>
struct employee
{
     int emp_id;
     char name[20];
     float salary;
};
void main()
{
     struct employee emp;
     float temp;
     /* Read employee information  */
     printf("Enter id:");
     scanf("%d", &emp.emp_id);
     printf("Enter employee name:");
     scanf("%s", emp.name);
     printf("Enter employee salary:");
     scanf("%f", &temp);   /* you can't read a float */
     emp.salary = temp;
     /* print */
     printf("emp_id:%d \t emp_name:%s\t temp_salary:%f \n",
                emp.emp_id, emp.name, emp.salary);
}
```

1.10.3 Initializing a Structure

You have studied the way by which a variable-scalar or vector or array - can be initialized at the time of declaration. Similarly, a structure can also be initialized as,

```
            struct Point p = {20, 30};
```

With this, `p.x` and `p.y` are initialized to 20 and 30 respectively.

1.10.4 Arrays of Structures

In section 10.2, we have used a structure definition for `employee`. Subsequently a scalar `emp` has been defined, but supposing if you wish to maintain the information of, say, `n` employees, you need to declare an array of structures. Similar to array of integers, it is also possible to have an array of structures.

```
            #define n 100
            struct employee e[n];
```

Accessing of structure members is same as earlier, except that each array location now consists of structure members. For example, to access the 6th employee's salary, we use `e[5].salary`.

1.10.5 Structures and Functions

One of the very important advantages of structures is to minimize the number of arguments being sent from calling function to called function. Imagine that you wish to pass the employee record to a function `f()`. Without a structure definition, you have to pass three parameters (`emp_id, name` and `salary`) and with a structure, you can just send the name of the structure. Assuming `e` is the structure variable name, you can pass `e` as, `f(e)` and the called function will have the header like,

```
            void f (struct employee x) { }.
```

Similarly, a structure can be returned from a called function to calling function. The return type of the function should be declared appropriately with a **tag-name**.

This way of passing a structure to a function is called as **value parameter treatment**. When you pass `e` to function `f()`, a copy of entire structure is made to `x` in `f()` and the structure of values are available to the whole of function scope. Of course, an important requirement is that the structure definition should be global, as it could be used by functions, may need it.

In the previous case, the functions `f()` is `void` which means that it does not return any value to calling routine. The piece of code below shows how a structure can be returned to a calling function.

```
            struct employee f (struct employee x)
            {
```

```
        struct  employee temp;
        :
        :
        return (temp);
    }
```

The return type `temp` and the function return type should be same as normally imposed by C language.

1.10.6 Size of Structures

The size of a structure variable can be found using C compile-time unary operator called `sizeof`.

```
sizeof object
```

Or,

```
sizeof(type_name)
```

For example, to print the size of the employee structure, we may use,

```
printf("size = %d\n", sizeof(struct employee));
```

The output printed by this statement is 26 bytes. Therefore, the `sizeof` operator returns the total number of space occupied by the structure.

1.10.7 Union

A **union** is same as structure in terms of syntax. However, there are few differences as given below:

- The keyword `union` is used instead of `struct`.
- The size of structure is the sum of all members. But for union the memory allocated is the highest of all its members.

In other words, all the union members share a common memory area. Unions provide a way to declare memory space that can hold values for different data type at different times. You can initialize a union with the data type of the first member declared. For example,

```
union
{
     int int_data;
     float float_data;
     char char_data;
} x;
```

Now, you can store values only in one member at a given time. The memory allocated for this union is 4 bytes (since `float_data` occupies the highest memory). If you assign a value to integer member, say, `int_data = 42;` then displaying its value is legal (see next page).

```
            printf("%d\h", x.int_data);
```

However, you can't display the *float* member;

```
            printf("%f\n", x.float_data);
```

When you assign a value to `float_data`, you can only use that member and the value of the other *int* member is no longer available.

1.10.9 Structure Assignment Comparison

Borland C allows structure assignment, which means that structure variables can be used is an assignment statement. For example, assuming the employee structure as defined earlier with the following two variables `e1` and `e2`, we can write,

```
             e1 = e2;
```

This statement assigns value of `e2` to `e1` member-wise.
Similarly when you have an array member, still the assignment holds good which is illegal otherwise. Look at the piece of code below,

```
            char name1[20] = "RAMA";
            char name2[20] = "KRISHNA";
            name1 = name2;  /* Illegal */
```

However, when you declare the array in a structure, then it is possible to use assignment and the compiler does not report an error.

1.11 POINTERS

1.11.1 Introduction

The concept of pointers in C is the prime power of the language. **Pointers** are data, similar to other kinds of integer data that you have used already. However, pointer data is special because you can use them to access other data. The programming examples shown in the previous chapters can all be written suing pointers, which would make the programs efficient and flexible.
There are several reasons why we use pointers and few are listed below:

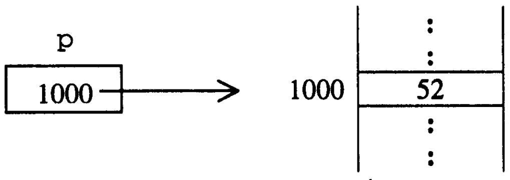
- A pointer variable can be used to access the array with out explicit indexing, especially for character strings.
- Arrays of pointers to refer to other blocks of data.
- Dynamic memory allocation is possible only through pointers.
- A function can be called dynamically using pointers (pointer to a function).

The disadvantages of using pointers are:

- Unless defined and initialized properly use of pointers may lead to disastrous situations.
- Using pointers may some times be confusing than ordinary methods.

1.11.2 Understanding Pointers

When a pointer variable is declared, it is ready to hold a valid memory address. This memory address in turn will be having some data value. This illustrated in Figure 1.5.



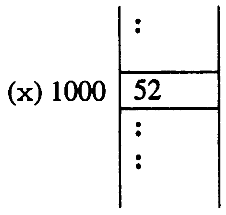
main memory

Fig. 1.5 pointer pointing to a memory address

Here, the pointer variable points to memory address 1000, which contains data 52. You should not confuse a pointer variable with an ordinary variable. For example,

```
int x;
x = 52;
```

declares an integer variable x and the assignment statement puts 52 into x. What is the address of variable x? Generally, the C compiler allocates memory addresses to each of the variable declared in the program at the time of compilation. For instance, x might have been allocated 1000 as the address.



memory

However, you can't alter the address of x by any means. But pointer variables can be assigned any valid address during run-time.

The size of a pointer variable is always 2 bytes (some machines may use 4 byte address) irrespective of what data type it points to. Because pointer variables hold only addresses and therefore point to `int` or `float` or `char`, etc.

You can read an address for a pointer variable using `scanf ("%p",...)` and similarly you can print the address using the same syntax or with %x.

1.11.3 Declaring and initializing pointer variable

The syntax for declaring a pointer variable is:

```
data_type *var_name;
```

Where data_type is any primitive or user defined data type in C including void. The character * is the indirection operator and this informs the compiler that it is a pointer variable. For example,

```
int *int_ptr;          /* integer pointer */
float *float_ptr;      /* float pointer   */
struct point
{
     int x;
     int y;
};
struct point *str_ptr;
```

After declaring a pointer variable of particular type, **it is the responsibility of the programmer to assign a valid memory address to that variable**. Now, how to assign a valid address? There are two ways by which a valid address can be associated:

(1) Declare an ordinary variable and put that address to the pointer variable. That is,

```
int a;
int *p = &a;
```

The ampersand sign '&' is called the **address operator** which returns the address of the variable. The address of a is valid, since the compiler has offered the address.

(2) Getting the address dynamically using malloc() function.

```
p = (int *) malloc (sizeof(int));
```

The malloc() allocates memory of size int (i.e., 2 bytes) and the returned memory is *typecasted* to the required type matching with the type of left hand side. Now if you want to allocate memory for the *float* variable, then you could write:

```
float *float_var;
float_var = (float *) malloc (sizeof(float));
```

Similarly, for The structure Point, (defined earlier)

```
stp_ptr = (struct point *) malloc (sizeof(struct Point));
```

We shall see in the following section, how to access the memory pointed by a pointer variable.

1.11.4 Accessing a Variable through its Pointer Variable

Once you assign a valid address to a pointer variable, then accessing via the pointer variable (i.e., **indirection**) is easier. For example,

```
int *p1, *p2;
int a, b;
p1 = &a;  /* p1 gets a valid address     */
p2 = &b;  /* p2 gets a valid address too */
```

```
5    *p1 = 10;
6    *p2 = 20;
7    p1 = p2;   /* p1 points to what p2 points to */
8    *p1 = 30;
```

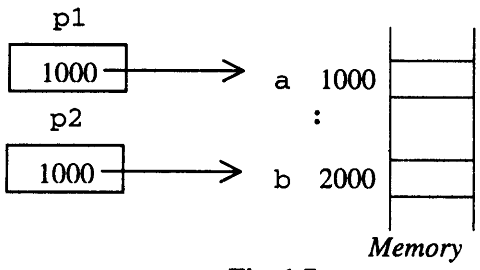
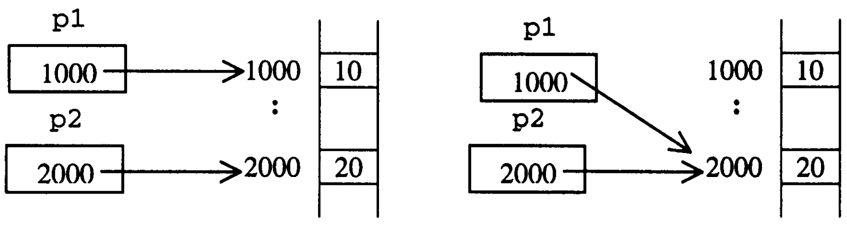


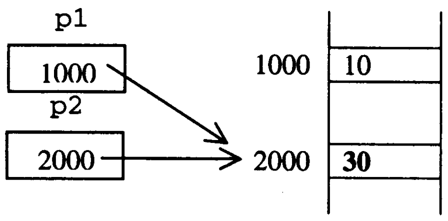
Fig. 1.7

Assuming that the addresses of and B are 1000 and 2000 respectively, the Figure 1.7 shows the situation up to line-4. When you want to access the address of a pointer variable, you use just the name of the pointer variable like in line-3 and line-4. To access the value pointed by the pointer variable, you must use, `*ptr_var`. For example, `*p1 = 10;`, puts 10 to the address pointed by `p1`. The Line –7 assigns the address of `p2` to `p1`.

Before executing line-7. **After executing line-7.**



Now both `*p1` and `*p2` will access the data element 20. Line-8 puts a new value–30 pointed by `p1`. Pointer `p1` holds the address 2000 (because of line-7) and the value 20. Now, `*p1` and `*p2` both will have values 30 only (see below).



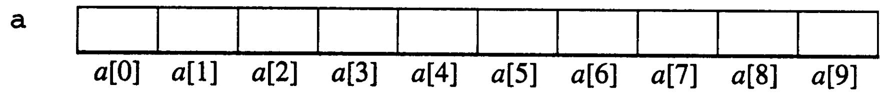
```
            vp = &x;
            printf ("%d\n", *(int *)vp);
            :
Output: 27
```

The statement `(int *)vp` makes `vp` to become an integer pointer and `*(int *) vp` access the address pointed by `vp`. The `void` pointer can be *typecasted* to any type during the execution of the program.

1.11.6 Pointers and Arrays

The relationship between pointers and arrays is strong, as the name of the array is nothing but the address of the array itself. The accessing concept of an array by p using subscripts can also be done with pointers. In fact, pointers offer convenient, flexible and efficient way of accessing the arrays.

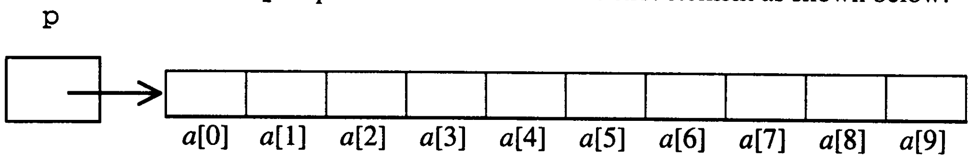
We shall bring out the correspondence between arrays and pointers with a simple declaration `int a[10];`It is a static declaration where the C compiler allocates 10 memory locations. That is,



Each location in a can be accessed using the subscript notation as shown above. For example, `a[i]` refers `i`th element of the array. Now, we shall declare a pointer, p, to point to the array.

```
            int *p;
            p = &a[0];
```

This statement makes p to point to the address of the first element as shown below:



Now the assignment, `x = *p;` will assign the contents of `a[0]` to x. If p points to a particular element of an array, then by definition `p + 1` points to the next element, irrespective of the data type. Or, `(p + i)` points to the *i*th element from p and `(p – i)` points to `i` elements before. The statement `*(p + i)` refers to the contents of `a[i]`.

You can also assign the address of the array by writing

```
            p = a;
```

which is same as

```
p = &a[0];
```

Also, to reference an element of the array, you can write either `a[i]` or `*(a + i)`. This is because `&a [i]` and `a + i` are identical.

Arithmetic is allowed on pointers, like p++ or p–but however, it is not true for array names. The following statements are illegal.

```
a = p;
a ++;
```

This is because, the address of the array is specified by the compiler. Hence, this cannot be altered during program execution.

1.11.7 Character Pointers

As it is pointed out in earlier sections, pointers are very useful in accessing character arrays. The character strings can be assigned with a character pointer like the one shown below.

```
char arr_var[] = "Good day";     /* array version   */
char *ptr_var = "Good day";      /* pointer version */
```

Both these variables can allocate memory just big enough to hold the **string constant** padded with the null character (`\0`) at the end. Even in the pointer version, you need not worry about the storage, as it is taken care by the compiler and `ptr_var` will point to the first character in the string.

1.11.8 Pointers and Functions

You have seen how the primitive data types can be declared arguments in a function. In the same way pointer also can be declared as arguments. Pointer arguments are useful in functions, because they allow you to access the original data in the calling program. When you pass a pointer to a function, you pass an address, For example:

```
Calling sequence        Called sequence
:                       void f (int *x, int *y)
int a, b;               {
:                       :
f(&a, &b);              }
:
```

The calling sequence passes the address of the variables `a` and `b` (you can not pass with out `&` - address operator) and the receiving parameter in the function `f()` are two pointer variables `x` and `y`. Now in the function, you can access the contents with the usual syntax `*x` and `*y`.

When you want to pass an array, you simply write the name of the array.

```
struct Point
{
      int x;
      int y;
};
struct Point p;
struct Point *ptr;
ptr = &p;  /* address of struct to a pointer */
```

You can now access the structure members using

(i)	`(*ptr).x`	access the value of first structure member pointed by `ptr`.
	`(*ptr).y`	accesses the value of the second structure member.
(ii)	`ptr->x`	Another operator called structure.
	`ptr->y`	pointer operator (->) is used here, which consists of two characters on the keyboard i.e., - and >. Compared to the first method, this is much easier.

The structure members sometimes may be pointers and in which case the accessing of the same has to be done with care. For example,

```
struct
{
      int len;
      char *str;
}*p;
```

`++p->len` is same as `++(p->len)` i.e., access the contents `len` pointed by `p` and then increment. Alternatively if you write `(++p)->len` will first increment `p` first then access `len`.

`*p->str`	fetches whatever `str` points to.
`*p->str++`	accesses what `str` points to and then increment.

That is, `(*p->str)++;` Since -> has a higher precedence than ++.

Program 1.34
Pointer and Structure – demo1

```
struct int_ptr
{
    int *p1;
    int *p2;
};
void main()
{
    struct int_ptr p; /* note that p is not a pointer */
    int i1 = 100, i2;
    p.p1 = &i1;
    p.p2 = &i2;
    p.p2 = -92;             /* p->p2  is illegal */
    printf("%d\n",*p.p1); /* p->p1 */
    printf("%d\n",p.p2);  /* p->p2 */
}
```

Sample Run

```
100
-92
```

Program 1.35
Pointer and Structure – demo-2

```
void main()
{
    struct tag
    {
        int *p1, *p2;
    };
    struct tag *p, dummy;
    int dp1, dp2;
    p = &dummy;
    (*p).p1 = &dp1;
    (*p).p2 = &dp2;
    *p->p1 = 10;
    *p->p2 = 20;
    printf("%d\t%d\n",*p->p1,*p->p2);
}
```

1.11.10 Pointers and functions

Passing a structure to a function can be done by sending the address of the structure, rather than by sending its value. Because, sending by value takes time by copying the structure to the function parameter. The following examples explain two ways of sending a structure to a function.

Program 1.36
Structure to a function-ordinary method

```
#include <stdio.h>
struct s {
     int a;
     int b;
};
struct s var1, var2;
void f(struct s);

void main()
{
     var1.a = 10;
     var1.b = 20;
     var2 = var1; /* var2 is a copy of var1 */
     printf("%d\t%d\n", var2.a, var2.b);
     f(var2);          /* send the structure by value */
}

void f(struct s x)
{
     printf("f = %d\t%d\n", x.a, x.b);
}
```

Sample Run

```
    10      20
f = 10   20
```

Program 1.37
Structure to a function-second method

```
#include <stdio.h>
struct s
{
     int a;
     int b;
};
struct s var1, var2;
void f(struct s *);
void main ()
{
     var1.a = 10;
     var1.b = 20;
     var2 = var1; /* var2 is a copy of var1 */
     printf("%d\t%d\n", var2.a, var2.b);
     f(&var2);  /* pass the address of structure */
}
```

```
void f(struct s *x)
{
    printf ("f = %d\t%d\n", x->a, x->b);
}
```

Sample Run

```
     10  20
f = 10  20
```

When you send large structures, it is always efficient to pass the address of the structure (reference parameter) from the calling program to the function rather than sending it by value parameter.

1.12 Dynamic Memory Allocation

Allocating memory to program variables at run time may be required for some applications. For example, to store list of elements whose size is not known in advance (at the time of compilation) need dynamic allocation. In general, requesting for some memory size at run time using C language memory allocation function is called as **dynamic memory allocation.**

When C compiler encounters a request for dynamic allocation of memory through an appropriate function memory is obtained from a heap. This is a separate memory area maintained by the compiler which is logical separation in RAM. Initially, entire heap area is available for dynamic allocation. The amount of memory available for allocation is called free list and the occupied area is called as allocated list. Generally, memory is allocated from the free list and the free list itself is organized as a linked list (refer Chapter 6 for more details).

This section will focus on important functions that are used for dynamic memory allocation as listed below:

- *malloc()*
- *calloc()*
- *realloc()*
- *free()*

The first three functions are used to allocate memory dynamically and the last one is to deallocate the memory. The main advantage of dynamic memory allocation is to save memory from unnecessary wastage, because it is allocated as and when required.

1.12.1 malloc() function

The simplest function to allocate dynamic memory is by using *malloc()* function. This function allocates a specified amount memory (uninitialized) and returns a *void* pointer (why?). The syntax of *malloc()* is given below:

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

Here, `size_t` is the data type used for memory object sizes and `size` is the size of memory (in bytes) required. For example,

```
p = (int *) malloc(10);
```

allocates 10 bytes of integer memory and the address of the allocated block is assigned to the pointer p. Notice the type casting done to the malloc function, because this function returns void pointer.

Return value:

⇒ On success, *malloc* returns a pointer to the newly allocated block of memory.
⇒ On error (if not enough space exists for the new block), *malloc* returns null. The contents of the block are left unchanged.
⇒ If the argument `size` == 0, *malloc* returns *null.*

Before we present a complete C program for *malloc*, following is a piece of code for dynamic allocation and static way of allocating memory.

```
/* Dynamic memory allocation */
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
void main()
{
      int *p;
      p = (int *) malloc (2);
      if (p == NULL)
      {
        printf("Error\n");
        exit(1);
      }
      *p = 10;
      printf("Data = %d\n", *p);
      free(p);
}
```

```
/* Static memory */
#include <stdio.h>
void main()
{
   int p;
   p = 10;
   printf("Data = %d\n", p);
}
```

In the above program (left one) p gets 2 bytes of integer dynamic memory. After obtaining the memory location you can store an integer (10 in our example). To release the memory simply use *free()*. In contrast, for static allocation we do not require *malloc*, because memory allocation is taken care by the C compiler (see the right one).

To illustrate working of *malloc()* further, a small but complete C program is given in Program 1.38. The notion is to allocate dynamic memory for 10 integer data, store some data, and display the same.

Program 1.38
Demo of malloc

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>

void main()
{
        int *p, *q;
        int i;
        p = (int *) malloc(10 * sizeof(int));
        if (p == NULL) exit(1);

        q = p; /* save starting address */
        for (i = 1; i <= 10; i++)
        {
          *q = i;
          q++;
        }
        q = p; /* regain starting address */
        for (i = 1; i <= 10; i++)
                printf("%d ", *q++);
        printf("\n");
}
```

Sample Run

```
1 2 3 4 5 6 7 8 9 10
```

The program first allocates dynamic memory for 10 integer data elements. Next, it stores values 1 to 10 into those locations. Finally, all ten data elements are displayed. Notice how the temporary variable q is used to keep track of the starting address of the allocated block.

1.12.2 calloc() function

The *calloc()* function is same *malloc()* except in the parameter list. This function also is used to allocate memory dynamically. Below is the syntax:

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

The *calloc()* function allocates storage space for an array of num elements, each of length size bytes. **Each element is initialized to 0.**

Parameters:
num - Number of elements.
size - Length in bytes of each element.

Return value:

⇒ *calloc* returns a pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than *void*, use a type cast on the return value just like malloc.

Program 1.39 illustrates the working of *calloc()*. You may be wondering that it looks almost similar to *malloc()*. The meaning of *calloc()* in this program is that 10 locations are allocated (first parameter) and each location is meant for integer data (second parameter).

Program 1.39
Demo of calloc

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
void main(void)

{
        int *p;
        int i;
        p = (int *) calloc(10, sizeof(int));
        if (p == NULL) exit(1);
        for (i = 1; i <= 10; i++)
                printf("%d ", *p++);
        printf("\n");
         free(p);

}
```

We haven't stored any data in the memory. In fact, it is done deliberately to show that by default *calloc()* puts 0 in the allocated block. Obviously, the output generated by this program is: 0 0 0 0 0 0 0 0 0 0.

1.12.3 realloc() function

Assume that you have allocated few block of memory using *malloc* or *calloc*. Now you may want to increase or decrease the previously allocated memory. Obviously, you can not go for the earlier two functions, but we have another function in C that can solve this problem – **realloc()**. Similar to the other two functions, this also returns void pointer. The syntax is shown below:

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

realloc adjusts the size of the allocated block to `size`, copying the contents to a new location if necessary. *block* is pointer to previously allocated block of memory.

Parameters:

`block` - Points to a memory block previously obtained by calling *malloc*, *calloc*, or *realloc*. If block is a null pointer, *realloc* works just like *malloc*.

`size` – new size for the allocated block.

Return value:

⇒ On success, realloc returns the address of the reallocated block which might be different than the address of the original block.

⇒ On failure (if the block can't be reallocated or if `size` == 0), it returns null.

Program 1.40 illustrates first allocating memory through *malloc* and then increase the memory block by using *realloc*.

Program 1.39
Demo of realloc

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
void main()
{
    char *p;
    p = (char *) malloc (4);
    if (p == NULL)
    {
        printf("Error in allocating memory\n");
        exit(1);
    }
    p = ("BIT");
    printf("Before realloc, p = %s\n", p);
```

```
        p = (char *) realloc(p, 14);
        p = ("BIT,Bangalore");
        printf("After realloc, p = %s\n", p);

       free(p);
}
```

The pointer p gets 4 bytes of space through *malloc* and the string `'BIT'` is stored (3 bytes for *BIT* and one byte for '*\0*'). Later to store `'BIT, Bangalore'`, we need 14 bytes and this achieved by reallocating the space. Again p points to the starting address pf the newly allocated block of storage.

1.12.4 free() function

Another useful and important function is *free().* This function releases the memory allocated previously by *malloc* or *calloc* or *realloc.* free does not return any thing and its syntax is as follows:

```
void free (pointer-variable);
```

For example, if you have used a statement as: `p = (int *) malloc(10);` you can deallocate this memory as: `free(p);`

1.13 FILE MANAGEMENT

It is very easy to read from the keyboard and write to the console using *scanf* and *printf* functions. The prototypes of these functions are defined in *stdio.h.* However, it is not possible to read/write data only from these standard I/O devices. Suppose a large amount of data is stored in a file and this data is to be processed; the C program should be written to read this data from the file. Similarly if a large block of data to be written on to a disk file, some other method must be followed.

C allows data to be read from a file and also to write into a file called as I/O file management. This kind of high level I/O environment is called as stream I/O. The word **stream** means flow of data from an input file to some buffer and buffer to an output file. For any opened file a buffer is allocated and each file will have a file pointer attached to it. Normally, access of data is done through this file pointer.

We will study, in this section, some of the basics of stream I/O and functions that are supported by C to operate on files. Any opened stream allows read/write in one of the following ways:

- Character I/O
- String I/O
- Formatted I/O
- Binary I/O

A detailed set of stream I/O functions are shown in Table 1.10. We will not discuss all of these functions in this section but only those are required for file operations.

Table 1.10 Stream I/O functions

Control functions	**Description**
fopen()	Creates a file for read/write/append.
fclose()	Closes a file associated with a particular file pointer.
fcloseall()	Closes all opened files.
feof()	End-of-file flag detection.
ferror()	Tests whether an error has occurred in a stream.
fflush()	flushes a stream.
ftell()	Returns the current file pointer.
fseek()	Sets the file pointer anywhere in the file.
Character I/O	**Description**
getc()	Reads one character from the current file pointer (Macro).
putc()	Writes one character into a file (Macro).
fgetc()	Reads one character from the current file pointer.
fputc()	Writes one character into a file.
String I/O	**Description**
fgets()	Reads a string from the file.
fputs()	Writes a string into a file.
gets()	Reads a string from stdin (keyboard).
puts()	Writes a string to a stdout (console).
Formatted I/O	**Description**
fprintf()	Writes formatted data into a file.
fscanf()	Reads formatted data from a file.
fread()	Reads block of structured data written by *fwrite()*.
fwrite()	Writes a block of data

Basically the following topics are covered in this section:

- Opening a file in read, write, and append modes (*fopen*)
- Closing a file (*fclose*)
- I/O file operations (*fgetc*, *fputc*, *fscanf*, *fprintf*, etc.)
- Example programs to show working of certain functions.

1.13.1 fopen() function

Before doing any file related operations it should be opened. For opening a file C has fopen() function. Using this function you can open a file for read/write/append. The syntax to be used for fopen() is given below:

```
FILE *fopen(char *filename, char *mode);
```

There are two arguments that must be supplied to *fopen*. The first argument is the name of the file for a particular type of operation specified in the second argument.

Parameters:

`filename` - File that the functions open

mode - defines the mode of the opened file (r, w, a, r+, w+, a+, etc.)

Return value:

⇒ On success, returns a pointer to the newly opened stream.
⇒ On error, it returns null.

In this function the key issue is the mode string. You must select an appropriate mode for file operations depending upon the application. Table 1.11 gives the complete details of mode string and subsequent examples show how to use them.

The following piece of code demonstrates a basic procedure (will be used in all programs) to open a file.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
     FILE *fp;
     fp = fopen("in.dat", "r");
     if (fp == NULL)
     {
          printf("Error in opening file\n");
          exit(1);
     }

     /* statements corresponding to file operations */
     fclose(fp);
}
```

Table 1.11 Details of mode string (text mode only)

Mode String	Description	Effect
"r"	Read only	(i) if file exists, then it is opened in read-only mode.
		(ii) file pointer points to the beginning of the file.
		(iii) if file doesn't exist, then null is returned.
"w"	Write only	(i) if file exists, then its contents are lost.
		(ii) if file doesn't exist a new file created and opened in write-only mode.
		(iii) file pointer will point to the beginning of the file.
"a"	Append	(i) if file exists it is opened for writing.
		(ii) file pointer is moved to the end of the file.
		(iii) if file doesn't exist, a new file is created for write-only and file pointer points to the beginning of the file.
"r+"	Read & Write	(i) if file exists, then it is opened in read & write mode.
		(ii) file pointer is moved to the beginning of the file.
		(iii) if file doesn't exist, then null is returned.
"w+"	Read & Write	(i) if file exists, then it is opened in read & write mode.
		(ii) file pointer is moved to the beginning of the file.
		(iii) if file doesn't exist, then new file is created for reading and writing.
"a+"	Read, Write & Append	(i) if file exists, then it is opened in read, write & append mode.
		(ii) file pointer is moved to the end of the file.
		(iii) if file doesn't exist, then new file is created for reading, writing and appending.

The first statement declares a file pointer `fp` and the next statement opens a file called "*in.dat*" in *read* mode. If the specified file doesn't exist in the logged on drive, an error message is displayed else you can start performing file operations. We will a complete example C program later in this section. To specify binary mode, append "**b**" to the string (**rb, wb, a+b**, etc.).

1.13.2 fclose() function

It is always suggested that an opened stream must be closed before the end of the program. What does a *fclose()* function does? Technically, *fclose* flushes the buffer contents and disconnects from the opened file. This means that you can't access the file using the file pointer that was used earlier. The syntax for *fclose* is given below:

```
int fclose(FILE * file-pointer);
```

On success, *fclose* returns 0 otherwise -1. The code shown in Section 1.13.2 uses *fclose* as the last statement.

1.13.3 Stream Functions – "r" mode

Functions that are mainly used to read/write characters/strings/block of data will be mainly discussed in this section. We shall start with character based I/O with few example programs.

(1) getc()
The *getc()* is a macro that reads a single character at a time from the opened stream (i.e. a file). The syntax of getc is given below:

```
int getc(FILE *fp);
```

Parameters:
`fp` – file pointer of the opened stream with *fopen()*.
Return value:
⇒ On success it returns the character read.
⇒ On error it returns -1 (i.e. EOF).

(2) putc()
The putc() is also a macro that writes a character into an opened file using fopen(). The syntax is given below:

```
int putc(char c, FILE *fp);
```

Parameters:
`c` – character value to be written.
`fp` – file pointer of the opened stream with *fopen()*.
Return value:
On success it returns the character written.
On error it returns -1 (i.e. EOF).

To illustrate the working of these two functions, see the example Program 1.40.

Program 1.40
getc and putc

```
#include <stdio.h>
#include <stdlib.h>
void main()
```