Unit

6

ARRAYS AND OPERATOR OVERLOADING

Lesson Structure

6.0 Objective

After going through this unit you will understand:

- What is an Array?
- How to access the elements of an Array?
- Concept related to Array as array as class member, arrays of objects etc.
- Overloading of Operators in C++
- Data Conversion

- Pitfalls associated with Conversion and Overloading

6.1 Introduction

The C++ language offers some broadly used derived or compound data types together with mechanisms which allow the programmer to define variables of these types and access the data stored within them. One of them is array. This unit discusses the fundamentals of array. An array is a collection of like variables that share a single name. Here it is discussed that how they are declared and data is accessed in it. Of particular interest are one-dimensional and two-dimensional arrays and C strings. In addition to array the concept of overloading the operators is introduced. It also talks about the conversion of data from one data type to another.

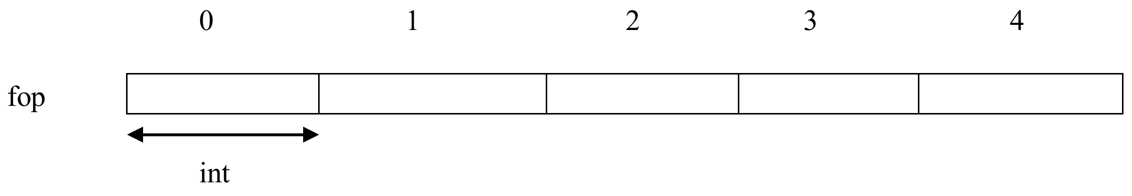
The rest of the chapter is organized as follows. Section 6.2 explains the array fundamentals. Section 6.3 discusses the concept of array as class member. Section 6.4 explains the arrays of objects concept. Section 6.5 discusses the string and functions associated with it. Section 6.6 talks about the overloading of Unary and Binary operators. Section 6.7 discusses data conversion in C++. Section 6.8 describes the pitfalls associated with the overloading and conversion. A brief summary on the unit is given in Section 6.9. Section 6.10 contains some questions for the students to workout. Section 6.11 shows some suggested readings.

6.2 Array Fundamentals

An array is defined as a collection of similar types of elements placed in adjacent memory locations that can be referenced individually by adding an key to a unique identifier.

Suppose there are five values of int type that can be stated as an array without declaring 5 separate variables (each having its own identifier). In its place, using an array, the five values of int type are kept in adjoining memory locations, and all five are accessed by means of the same identifier, with the proper key.

For instance, an array having 5 data values of integer (int) type called fop could be denoted as:



where each blank piece signifies an array element. In this situation, these are values of int type. These data's are sequenced from 0 to 4, being 0 the first element and 4 being last; In C++, the initial element of an array is always pointed with a zero (not a one).

Like a common variable, an array must also be declared before using it. A general type of declaration for an array in C++ is:

datatype name [elements];

where datatype is a legal type (such as char, float, int etc.), name is a valid identifier and the elements field (which is every time enclosed in [] square brackets), identifies the array length in terms of the number of data an array can hold.

Therefore, the fop array, with 5 elements of int type, can be declared as:

int fop [5];

NOTE: The elements field of an array within [] square brackets represent the number of elements. It must be a constant value, since arrays are chunks of static memory whose dimension must be determined at compile time, i.e. before the code executes.

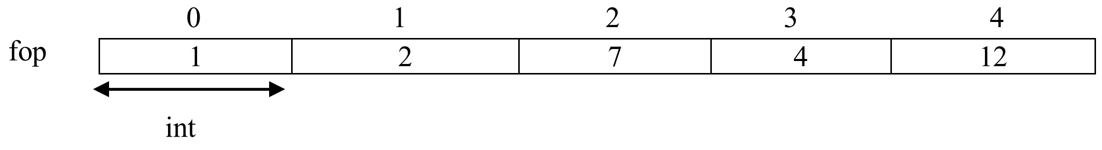
Initializing arrays

By default, normal arrays of scope having local to a function are generally left uninitialized. This means that elements of the array are not set to any specific value; their values are undetermined at the time when the array is declared.

But the array elements can be initialized explicitly to particular values when it is declared, by enfolding those preliminary values in {} braces. For example:

int fop [5] = { 1, 2, 7, 4, 12 };

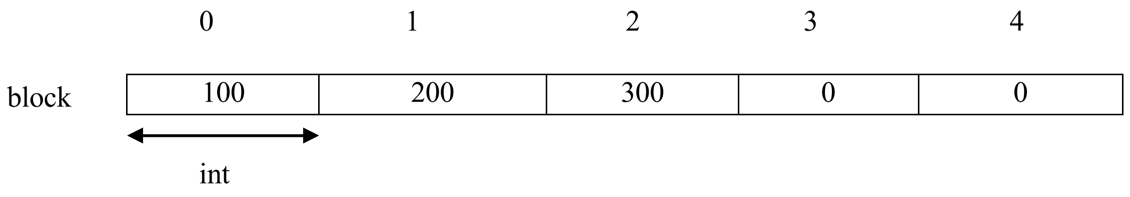
This declaration declares an array that can be characterized like this:



The number of values between {} braces shall not be more than the number of elements in the array. For example, above declaration, fop was declared to hold 5 elements (as stated by the number enclosed in []), and the braces {} contains exactly 5 values, one for each index. If declared with fewer values, the remaining indexes are fixed to their default values (which is generally means that filled with 0's). For example:

int block [5] = { 100, 200, 300 };

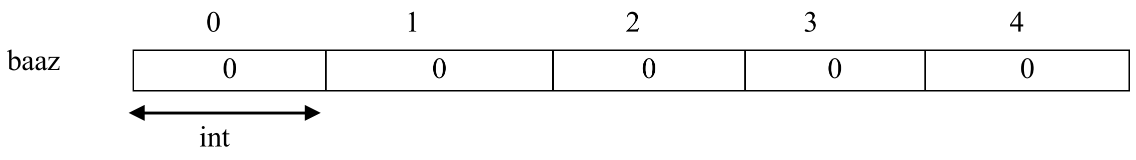
Will create an array like this:



The initializer can even have no values, just the braces:

```
int baaz [5] = { };
```

This makes an array of five values of int type, each index assigned with a value of 0:



When value initialization is provided for an array, C++ lets the option of keeping the [] unfilled. In this situation, the compiler will accept automatically a dimension for the array that matches the number of values inside the braces {}:

```
int fop [] = { 1, 2, 7, 4, 12};
```

After this statement, array fop would be of dimension 5 of int type, as we have given 5 values in initialization.

In conclusion, the development of C++ has directed to the approval of universal initialization for arrays also. Consequently, there is no longer need for the '=' sign between the initializer and the declaration. Hence, both these expressions are equivalent:

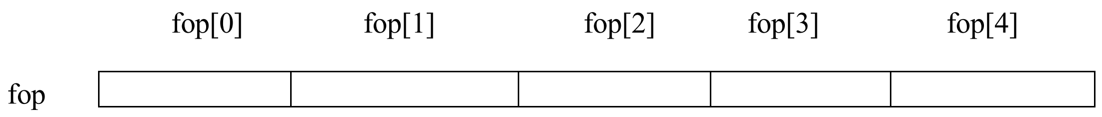
```
int fop[] = { 100, 200, 300 };

int fop[] { 100, 200, 300 };
```

Accessing the values of an array

The elements value in an array can be accessed just alike the value of a regular variable of the similar kind. The syntax is:

```
name[index]
```

Resulting from the preceding examples in which fop had 5 elements and each of these are of int type, the term which can be used to denote each element is the following:



For example, the following statement stores the value 75 in the third element of foo:

```
fop [2] = 7;
```

And, suppose, the following duplicates the value of the third element of fop to a variable called y:

```
y = fop[2];
```

Therefore, the statement fop[2] is itself a variable of int type.

Note that the third element of fop is indicated by fop[2], as the first element is fop[0], the second is fop[1], and consequently, the third is represented fop[2]. By this same goal, its last element is fop[4]. Thus, if fop[5] is written, it is accessing the sixth element of fop, and consequently really surpassing the array size.

In C++, it is syntactically right to surpass the define range of array indices. This can generate difficulties, as accessing elements beyond range do not create compilation errors, but can create runtime errors.

At this instance, it is significant to be capable to noticeably differentiate between the two uses that brackets [] have associated to arrays. They accomplish two different jobs: one is used at declaration time to identify the size of arrays; and the second one is used at time of accessing the array elements to specify indices for concrete array elements. Do not complicate these two potential uses of brackets [] with arrays.

```
int fop[5];        // new array declaration
fop[2] = 7;        // array element accessing
```

The key difference is that the declaration is headed by the datatype of the elements, while the access expression is not.

Some other legal actions with arrays:

```
fop[0] = b;
fop[b] = 75;
a = fop [b+2];
fop[fop[b]] = fop[2] + 5;
```

For example: // arrays example

```
#include <iostream>
using namespace std;
int fop [] = {1, 2, 7, 4, 12};
int n, output=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
        output += fop[n];
  }
  cout << output;
  return 0;
}
```

OUTPUT: 26

Two-Dimensional Arrays

Two-dimensional arrays can be considered a rectangular display of elements with rows and columns, and this is also known as a matrix. Consider the following example int x[3][3]. The two-dimensional array can be declared as shown in Figure 6.1: Two-dimensional array.

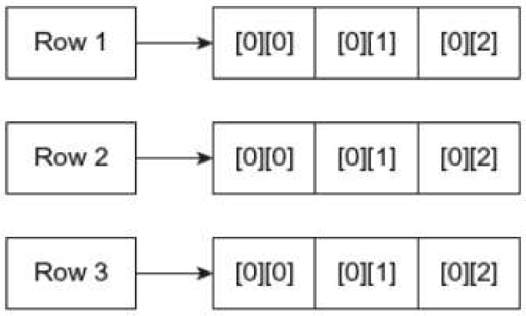


Figure 6.1: Two-dimensional array

Table 6.1: Arrangement of two-dimensional array elements

	Column 0	**Column 1**	**Column 2**
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

The arrangement of array elements shown in Table 6.1 is only for the sake of understanding. Actually, the elements are stored in continuous memory locations. The two-dimensional array is a collection of two one-dimensional arrays. The meaning of the first argument is in x[3][3] and means the number of rows; that is, the number of one-dimensional arrays, and the second argument indicates the number of elements. The x[0][0] means the first element of the first row and column. In one row, the row number remains the same but the column number changes. The number of rows and columns is called the range of the array. A two-dimensional array clearly shows the difference between logical assumptions and the physical representation of data. The computer memory is linear and any type of array may one, two- or multi-dimensional array it is stored in continuous memory location as shown in Figure 6.2: Storage of two dimensional array.

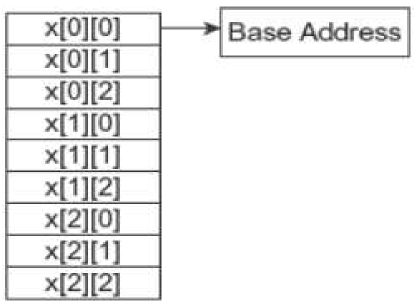


Figure 6.2: Storage of two dimensional array

Program to demonstrate 2-D array.

```
#include<iostream.h>
#include<conio.h>
int main()
{
int m,n;
int a[3][3]={1,2,3,7,8,9,4,5,6};
clrscr();
cout<<"\n Array elements and address ";
cout<<"\n \t Col-0 Col-1 Col-2";
cout<<"\n \t ====== ====== ======";
cout<<"\nRow0";
for (m=0;m<3;m++)
{
for (n=0;n<3;n++)
cout<<"\t "<< a[m][n];
if(m==2)
break;
cout<<"\nRow"<<m+1;
}
return 0;
}
```

OUTPUT

```
Array elements and address

                 Col-0          Col-1          Col-2

                 ======         ======         ======

Row0               1              2               3

Row1               7              8               9
```

```
Row2            4            5            6
```

Explanation: From the above program's output, we can conclude that the one-dimensional array can be accessed using a single loop. However, for the two-dimensional array, two loops are required for rows and columns. The inner loop helps access the row-wise elements, and the outer loop changes the column number.

Program to read marks and percentage of students using two dimensional array.

```
#include<iostream.h>
#include<conio.h>
int main()
{
int student[5][2],i,j;
clrscr();
for(i=0;i<5;i++)
{
cout<<"\n Enter the Roll no and percentage of the student:";
cin>>student[i][0]>>student[i][1];
}
cout<<"\n Roll_no \t percentage ";
for(i=0;i<5;i++)
{
cout<<"\n";
cout<<student[i][0]<<"\t"<<(student[i][1]);
}
return 0;
}
```

OUTPUT

```
Enter the Roll no and percentage of the student: 1 89
Enter the Roll no and percentage of the student: 2 78
Enter the Roll no and percentage of the student: 3 76
Enter the Roll no and percentage of the student: 4 56
Enter the Roll no and percentage of the student: 5 90

Roll_no             percentage
1                   89
2                   78
3                   76
4                   56
5                   90
```

Explanation: In the above program, student [5][2] is declared as an array. The array student [5][2] contains 5 rows and 2 columns. Roll number and percentage scored by the students are read through the keyboard and displayed.

6.3 Array as a Class Member Data

Within a class, arrays can be used as member variables. The syntax for the same is as follows.

```
const int size=15;        // provides value for array size
class array
{
        int a[size];      // a is array of type int
    public:
        void setvalue(void);
        void output(void);
};
```

In the class **array** we have declared an array variable **a[]** which is a private data member. We can access it using the public member functions and perform all types of operations on it. In the above example, the member function **setvalue()** is used to set the values of items in the array **a[]**. The member function **output()** is used to display those values.

We will now consider a list of shopping items the order of which is placed with a merchant. The list includes all the necessary details about the shopping items, like the item codes and price per item. We want to perform operations like adding new items to the list, deleting existing items from the list and printing the full cost of the order.

Program below exhibits how these operations are performed using a class which has arrays as data members.

```
#include<iostream>
using namespace std;
const int a=30;
class product
{
        int code[a];
        float price[a];
        int number;
   public:
        void NUM(void){number = 0;}
        void getproducts(void);
        void showtotal(void);
        void del(void);
        void showproducts(void);
};
```

```
void product :: getproducts(void)            // give values to data members of product
{
                    cout << " Enter code of the product: ";
                    cin >> code[number];
                    cout << "Enter cost of the product: ";
                    cin >> price[number];
                    number++;
          }
          void product :: showtotal(void)  //show the total cost of all products
          {
                    float total = 0;
                    for(int k=0; k<number; k++)
                    total = total + price[k];
                    cout << "\n Sum of the values: " << total << "\n";
          }
          void product :: del(void)            // remove a selected product
          {
                    int z;
                    cout << "Enter code of the product: ";
                    cin >> z;
                    for(int k=0; k<number; k++)
                    if (code[k] ==z)
                    price[k] = 0;
          }
          void product :: showproducts(void)        // list all the products
          {
                    cout << "\ncode price\n";
                    for(int k=0; k<number; k++)
                    {
                              cout << "\n" << code[k];
                              cout << "         " << price[k];
                    }
                    cout << "\n";
          }
          int main()
          {
                    product order;
                    order.NUM();
                    int y;
                    do                   // do……..while loop
                    {
cout << "\nYou are allowed to perform the following;" << "Please enter a correct number \n";
cout << "\n1 : Insert a product";
cout << "\n2 : Show the total cost";
cout << "\n3 : Remove a product";
cout << "\n4 : List all products";
cout << "\n5 : Leave";
cout << "\n\nWhat do you choose?";
cin >> y;
switch(y)
{
                                        case 1 : order.getproducts(); break;
                                        case 2 : order.showtotal(); break;
                                        case 3 : order.del(); break;
                                        case 4 : order.showproducts(); break;
                                        case 5 : break;
default : cout << "You have selected a wrong option !!! Please try again !!!\n";
}
                    }
                    while(y != 5);          // do……..while ends
                    return 0;
          }
```

The **output** of program 2.3 is:

```
You are allowed to perform the following; Please enter a correct number

1 : Insert a product
2 : Show the total cost
3 : Remove a product
4 : List all products
5 : Leave

What do you choose?1
          Enter code of the product: 12
Enter cost of the product: 50

You are allowed to perform the following; Please enter a correct number

1 : Insert a product
2 : Show the total cost
3 : Remove a product
4 : List all products
5 : Leave

What do you choose?1
          Enter code of the product: 23
Enter cost of the product: 100
```

```
You are allowed to perform the following;Please enter a correct number

1 : Insert a product
2 : Show the total cost
3 : Remove a product
4 : List all products
5 : Leave

What do you choose?1
        Enter code of the product: 9
Enter cost of the product: 150

You are allowed to perform the following; Please enter a correct number

1 : Insert a product
2 : Show the total cost
3 : Remove a product
4 : List all products
5 : Leave

What do you choose?2

        Sum of the values: 300

You are allowed to perform the following;Please enter a correct number

1 : Insert a product
2 : Show the total cost
3 : Remove a product
4 : List all products
5 : Leave

What do you choose?4

code price

12      50
23      100
9      150

You are allowed to perform the following;Please enter a correct number
1 : Insert a product
2 : Show the total cost
```

```
3 : Remove a product
4 : List all products
5 : Leave

What do you choose?3
Enter code of the product: 12

You are allowed to perform the following;Please enter a correct number
1 : Insert a product
2 : Show the total cost
3 : Remove a product
4 : List all products
5 : Leave

What do you choose?4

code price
12      0
23      100
9      150
```

Two arrays are being used by the program: **code[]** that holds the code number of products and **price[]** that holds the prices of products. A data member **number** is used to maintain of the record of products in the list. The statement

```
const int a = 30;
```

defines the size of the array members.

The value of **number** variable is set to zero by the first function NUM(). The next function **getproducts()** gets the code and price of the products and allots them to array members **code[number]** and **price[number]**. The function **showtotal()** computes the total cost of the order and then prints the value. The function **del()** removes a product from the list. With the help of the product code it locates the product in the list and then sets the price to zero. The last function **showproducts()** shows the list containing all the products.

Program: Write C++ Programs to find out Prime Number using Class.

```
#include<iostream>
using namespace std;
// Declaration of Class
class prime_no
{
   // Declaration of Member Variable
 int b, j, m;
 public:
```

```
 prime_no(int y)
 {
 b=y;
 }

 // Object Creation for Class
 void calc()
 {
            j=1;
    {
     for(m=2;m<=b/2;m++)
     if(b%m==0)
       {
             j=0;
             break;
       }
     else
      {
             j=1;
      }
    }
 }
void display()
 {
 if(j==1)
 cout<<"\n"<<b<<" is Prime Number.";
 else
 cout<<"\n"<<b<<" is Not Prime Numbers.";
 }
};
//Main Function
int main()
{
 int b;
 cout<<"Enter the Number:";
 cin>>b;
 // Object Creation for Class
 prime_no obj(b);
 // Call Member Functions
 obj.calc();
 obj.display();
 getch();
 return 0;
}
```

Output:

```
Enter the Number: 12
12 is Not Prime Numbers.

Enter the Number:11
11 is Prime Number.
```

6.4 Array of Objects

We can have an array of any data type, counting **struct**. C++ also allows us to have an array of variables that is of type **class**. We call these variables *arrays of objects*. Take a look at the class definition below.

```
class worker
{
        char Fname[35];
        float age;
   public:
        void get_data(void);
        void put_data(void);
};
```

The identifier worker is a user-defined data type that can be used to create objects belonging to different categories of workers. For example

```
worker engineer[3];       //array of engineer
worker salesperson[85];  //array of salesperson
worker supervisor[20];   // array of supervisor
```

There are 3 objects, i.e., engineer [0], engineer [1] and engineer [2] of type **worker** class in the array **engineer**. Similarly array **salesperson** consists of 85 objects and array **supervisor** consists of 20 objects.

An array of objects can be treated as any other array. So, individual elements can be accessed using the basic array accessing methods and we can also use the dot operator to access member functions. The statement below,

```
engineer[k].putdata();
```

shows the kth element of array **engineer**. In other words, member function **putdata()** is called by the object **engineer[k]**.

An array of objects is stored in memory just like a multi-dimensional array. Figure 6.3 below shows how data items of an object array are stored. Here we should remember that memory space is created only for data items of the objects. Member functions which are used by all the objects are stored separately.

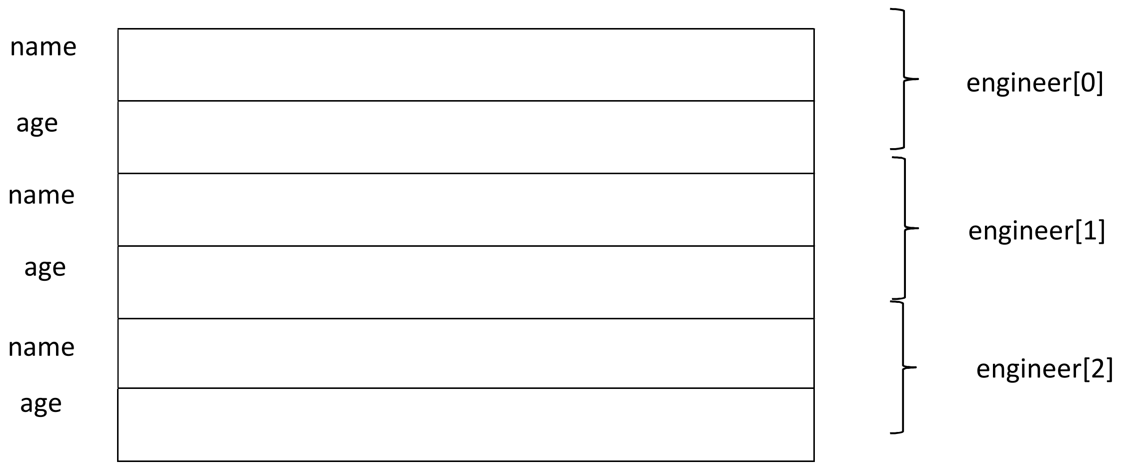


Figure 6.3: Storing of data items of an object array

Program below shows how object arrays are used.

```
#include<iostream>
using namespace std;
class worker
{
        char name[35];  //string as class member
        float age;
   public:
        void getdata(void);
        void putdata(void);
};
void worker :: getdata(void)
{
        cout << "Enter the name of the worker: ";
        cin >> name;
        cout << "Enter the age of the worker: ";
        cin >> age;
};

void worker :: putdata(void)
{
        cout << "Name of the worker: " << name << "\n";
        cout << "Age of the worker: " << age << "\n";
}

const int size = 3;
int main()
{
        worker engineer[size];
        for(int k=0; k<size; k++)
        {
                cout << "\nDetails of the Engineer" << k+1 << "\n";
                engineer[k].getdata();
        }

        cout << "\n";
        for(int k=0; k<size; k++)
        {
                cout << "\nEngineer" << k+1 << "\n";
                engineer[k].putdata();
        }
        return 0;
}
```

The program on execution gives the following **output**:

```
Details of the Engineer1
Enter the name of the worker: raghav
Enter the age of the worker: 35

Details of the Engineer2
Enter the name of the worker: raman
Enter the age of the worker: 32

Details of the Engineer3
Enter the name of the worker: rajat
Enter the age of the worker: 28
Engineer1
Name of the worker: raghav
Age of the worker: 35

Engineer2
Name of the worker: raman
Age of the worker: 32

Engineer3
Name of the worker: rajat
Age of the worker: 28
```

6.5 Strings

C++ offers two types of string representations which is as following −

- The C-style character string.

- The string class type introduced with Standard C++.

The C-Style Character String

The C-style character string invented in the C language and endures to be held within C++. This string is in fact a one-dimensional array that holds elements of character type and is terminated by a character '\0'(null). Therefore a null-terminated string comprises the characters that contains the string trailed by a null.

The following statement construct a string containing the word "Howdy". To have the null character at the array end, the array size having the string is one additional than the number of characters in the word "Howdy."

```
char wishes[6] = {'H', 'o', 'w', 'd', 'y', '\0'};
```

If the rule of array initialization is followed, then the above statement can be written as follows −

```
char wishes[] = "Howdy";
```

Subsequent is the memory arrangement of overhead defined string in C/C++ −

index	0	1	2	3	4	5
Variable	H	o	w	d	y	\0
Address	0x234	0x235	0x236	0x237	0x238	0x239

In fact, the user doesn't places the null character at the end of a string constant. The C++ compiler by default puts the '\0' at the end of the string when it initializes the array. Following program prints above-mentioned string−

```
#include <iostream>
using namespace std;
int main () {
  char wishes[6] = {'H', 'o', 'w', 'd', 'y', '\0'};
  cout << "My wishes are: ";
  cout << wishes << endl;
  return 0;
}
```

When the above program is compiled and executed, it yields the following outcome −

```
My wishes are: Howdy
```

C++ offers a wide variety of functions that operate on null-terminated strings as shown in Table 6.2 –

Table 6.2: Functions to manipulate String.

Function	Purpose
strcpy(st1, st2);	Duplicates string st2 into string st1.
strcat(st1, st2);	Joins string st2 onto the end of string st1.
strlen(st1);	Yields the string st1 length.
strcmp(st1, st2);	Yields 0 if st1 and st2 are the same; less than 0 if st1<st2; greater than 0 if st1>st2.
strchr(st1, ch);	A pointer is returned to the first existence of character 'ch' in string st1.
strstr(st1, st2);	A pointer is returned to the first existence of string st2 in string st1.

Following programs uses few of the above-mentioned string functions.

```
#include <cstring>
#include <iostream>
using namespace std;
int main () {
  char st1[10] = "Howdy";
  char st2[10] = "Earth";
  char st3[10];
  int length ;
  // copy st1 into st3
  strcpy( st3, st1);
  cout << "strcpy( st3, st1) : " << st3 << endl;
  // concatenates st1 and st2
  strcat( st1, st2);
  cout << "strcat( st1, st2): " << st1 << endl;
  // total length of st1 after appending
  length = strlen(st1);
  cout << "strlen(st1) : " << length << endl;

  return 0;
}
```

When the above program is compiled and executed, it yields outcome something as following −

```
strcpy( st3, st1) : Howdy
strcat( st1, st2): HowdyEarth
strlen(st1) : 10
```

The String Class in C++

The standard C++ library offers a string class type that supports all the functions stated above, moreover much extra functionality. Following example will illustrate them−

```
#include <string>
#include <iostream>
```

```
using namespace std;
int main () {

   string st1 = "Howdy";
   string st2 = "Earth";
   string st3;
   int  length ;

   // copy st1 into st3
   st3 = st1;
   cout << "st3 : " << st3 << endl;

   // concatenates st1 and st2
   st3 = st1 + st2;
   cout << "st1 + st2 : " << st3 << endl;

   // total length of st3 after concatenation
   length = st3.size();
   cout << "st3.size() :  " << length << endl;
   return 0;
}
```

When the above program is compiled and executed, it yields output as follows −

```
st3 : Howdy
st1 + st2 : HowdyEarth
st3.size() :  10
```

Program: Write a C++ program to find String length without Using Library function.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

 void main()
  {
   int j, tally=0;
   char chr[20];
   clrscr();
   cout<<"Enter any string: ";
   gets(chr);
   for(j=0;chr[j]!='\0';j++)
   {
```

```
  tally++;
  }
  cout<<"String Length: "<<tally;
  getch();
 }
```

Output:

```
Enter any String: computer
String Length: 8
```

Explanation:

```
for(j=0;chr[j]!='\0';j++)
  {
  tally++;
  }
```

Here we check the condition chr[j]!='\0' its means loop run till string is not null, when it reaches to null character loop terminates.

Program: Write a C++ program to find String Length Using Library Function.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>

 int main()
  {
  char st[20];
  int len;
  cout<<"Enter any string: ";
  gets(st);
  len = strlen(st);
  cout<<"String Length: "<<len;
  return 0;
}
```

Output:

```
Enter any String: computer
String Length: 8
```

Program: Write a C++ program for comparison of two strings.

```
#include<stdio.h>
#include<iostream.h>
```

```
#include<conio.h>

void main()
 {
 char st1[20], st2[20], m, n, check=0;
 clrscr();
 cout<<"Enter first string: ";
 gets(st1);
 cout<<"Enter Second string: ";
 gets(st2);
 m=0;
 n=0;
  while(st1[m]!='\0')
  {
   m++;
  }
  while(st2[n]!='\0')
  {
   n++;
  }
 if(m!=n)
 {
 check=0;
 }
 else
 {
 for(m=0,n=0;st1[m]!='\0',st2[n]!='\0';m++,n++)
 {
 if(st1[m]==st2[n])
 {
 check=1;
 }
 }
 }
 if(check==0)
 {
 cout<<"Strings are not equal";
 }
 else
 {
 cout<<"Strings are equal";
 }
 getch();
}
```

Output:

```
Enter First String : bihar
Enter Second String : bihaar
Strings are not equal
```

Program: Write C++ Program to Reverse a String.

```
#include<conio.h>
#include<iostream.h>
#include<stdio.h>
#include<string.h>

void main()
{
 char st[100],tmp;
 int m,n=0;
 clrscr();
 cout<<"Enter any the string :";
 gets(st);  //  gets function for input string
 m=0;
 n=strlen(st)-1;
  while(m<n)
   {
   tmp=st[m];
   st[m]=st[n];
   st[m]=tmp;
   m++;
   n--;
   }
 cout<<"Reverse string is: "<<st;
 getch();
}
```

Output

```
Enter any the string  : bihar
Reverse string is : rahib
```

Explanation of Program:

Initially the program finds the size of the string using strlen()library function.

Code

```
n = strlen(st)-1;
```

Presume String "bihar" is accepted then

Code

```
n = strlen(st)-1;
```

```
 = strlen("bihar") - 1
 = 5 - 1
 = 4
```
As we know string is character array and character array have character ranging between 0 to string_length-1. Thus we have position of last character in variable 'n'. Current values of 'm' and 'n' are :

Code
```
m = 0;
n = 4;
```
'm' situated on first character and 'n' placed on end character. Now we are exchanging characters at position 'm' and 'n'. After switching characters we are incrementing value of 'm' and decrementing value of 'n'.

Code
```
while(m<n)
    {
    tmp   = st[m];
    str[m] = str[n];
    st[n] = tmp;
    m++;
    n--;
    }
```
If m crosses n then process of swapping character is stopped.

Program: Write a C++ program to Count Occurrence of Characters.

(Occurrence of character in any string means how many times a specific character is existing in any string. For example; suppose we have string manas in this term ‘a’ is repetitive 2 times, it is the occurrence of ‘a’ in string manas.)

```
#include<stdio.h>
#include<iostream.h>
#include<conio.h>

 void main()
 {
  int m, tally=0;
  char chr[20], d;
  clrscr();
  cout<<"Enter Any String: ";
  gets(chr);
  cout<<"Enter any Character form string: ";
  cin>>d;
  for(m=0;chr[m]!='\0';m++)
  {
  if(chr[m]==c)
  tally++;
  }
  if(tally==0)
  {
  cout<<"Given character not found";
  }
  else
  {
  cout<<"Repetition of " <<d<<" "<<tally<<" times";
  }
  getch();
 }
```

Output:

```
Enter any String: Bharat
Enter any Character form string a
Repetition of a 2 times
```

Program: **Write a C++ program to copy one string into another without Using Library Functions.**

```
#include<conio.h>
#include<iostream.h>

int main()
{
  char st1[100], st2[100], m;
  clrscr();
  cout<<"Enter string st1: ";
  cin>>st1;
  for(m=0; st1[m]!='\0'; ++m)
  {
    st2[m]=st1[m];
  }
  st2[m]='\0';
  cout<<"String st2: "<<st2;
getch();
}
```

Output

```
Enter string st1 : Road
String st2: Road
```

Explanation of Code

```
for(m=0; st1[i]!='\0'; ++m)
```

```
 {
  st2[i]=st1[i];
 }
```

In the above program first string is checked till it is not null and after that initialization is done such that one by one elements of first string is copied into second string.

Program: Write a C++ program to find Number of Vowels, Consonants, Digits, Spaces in a String.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

int main()
{
   char stmt[150];
   int i1,v1,c1,ch1,d1,s1,o1;
   o1=v1=c1=ch1=d1=s1=0;
   clrscr();
   cout<<"Enter a statement of string:\n";
   gets(stmt);
   for(i1=0;stmt[i1]!='\0';++i1)
   {
      if(stmt[i1]=='a' || stmt[i]=='e' || stmt[i1]=='i' || stmt[i1]=='o' || stmt[i1]=='u' || stmt[i1]=='A' ||
stmt[i1]=='E' || stmt[i1]=='I' || stmt[i1]=='O' || stmt[i1]=='U')
         ++v1;
      else if((stmt[i1]>='a'&& stmt[i1]<='z') || (stmt[i1]>='A'&& stmt[i1]<='Z'))
         ++c1;
      else if(stmt[i1]>='0'&& stmt[i1]<='9')
         ++d1;
      else if (stmt[i1]==' ')
         ++s1;
   }
   cout<<"Vowels: "<<v1;
   cout<<"\nConsonants: "<<c1;
   cout<<"\nDigits: "<<d1;
   cout<<"\nWhite spaces: "<<s1;
   getch();
}
```

Output

```
Enter a statement of string: This is 10th C program
Vowels: 4
Consonants: 11
```

Digits: 2
White spaces: 4

6.6 Overloading Unary and Binary Operators

Operator Overloading

In C++, Operator Overloading is an important concept. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example, '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.

We can write any C++ program without the knowledge of operator overloading. Nevertheless, programmers tend to use operator overloading to make program intuitive. For example,

We can replace the code like:

```
compute = add(divide(m, n),multiply(m, n));   to   compute = (m/n)+(m*n);
```

To overload an operator, a special operator function is defined inside the class:

```
class class-name


          {                              Keyword          operator to be overloaded
          ... .. ...
          public
            returnType operator Symbol (arguments)
            {
                    ... .. ...
            }
          ... .. ...
 };
```

Here, returnType is the return type of the function. The returnType of the function is followed by the keyword ‘operator’. Symbol is the symbol of the operator that we want to overload. Just like we pass arguments to functions we can also pass arguments to the operator function.

Try to understand program 6.3 which is an example of **binary operator overloading** in C++.

```
#include <iostream>
class test
 {
          public:
          int m;
          int n;
          test operator+(const test &object1);
          void operator=(const test &object1);
};

void test::operator=(const test &object1)
{
        (*this).m = object1.m;
        (*this).n = object1.n;

        return;
}

test test::operator+(const test &object2)
{
        test temp_object1 = *this;
        temp_object1.m = temp_object1.m + object2.m;
        temp_object1.n = temp_object1.n + object2.n;
        return temp_object1;
}

int main(void)
{
        test object1, object2, object3;
        object1.m = 1;
        object1.n = 1;
        object2.m = 2;
        object2.n = 2;
        object3.m = 0;
        object3.n = 0;
        object3 = object1 + object2;
        std::cout<<object3.m<<" "<<object3.n<<"\n";
        return 0;
}
```

When ‘**object1 + object2**’ is encountered, function corresponding to overloaded operator + is called. We can think of ‘**object1 + object2**’ as something like ‘**object1.add(object2)**’. The function corresponding to overloaded operator + is called in context of **object1** and hence only **object2** is needed to be passed as argument. **object1** can be accessed through **this** pointer in that function. Here in this function, individual integer member is added and the resultant object is returned.

Similarly, everything happens the same way when the resultant object of the sum of **object1** and **object2** is assigned to **object3** through overloaded operator ‘=’. Each integer member of class is assigned to corresponding member of **object3**.

When we compile and execute the above program we get the following output:

```
3 3
```

Now we see **program below in which the unary operator '++' is overloaded.**

```
#include <iostream>
using namespace std;

class sample
{
        private:
        int cou;
        public:
        sample(): cou(7){}
        void operator ++()
        {
                cou = cou+1;
        }
        void show()
        {
                cout<<"Count is "<< cou;
        }
};

int main()
{
        sample s;
        // this calls void operator ++() function
        ++s;
        s.show();
        return 0;
}
```

This function is called when ++ operator operates on the object of **sample** class (object **s** in this case). When we compile and execute the above program we get the following output:

```
Count is 8
```

Program that overloads '<<' and '>>' operators which has been shown below.

```
#include <iostream>
using namespace std;

class complex
{
        private:
        int real;
        int imaginary;

        public:
        complex(int real, int imaginary)
        {
                this->real = real;
                this->imaginary = imaginary;
        }

        void display()
        {
                cout<< this->real<<"+"<<
                this->imaginary<<"i"<< endl;
        }

        friend ostream& operator <<(ostream& out,complex C);
        friend istream& operator >>(istream& in,complex& C);
};
ostream& operator <<(ostream& out,complex C)
{
        out<< C.real<<" + "<< C.imaginary<<"i"<< endl;
        return out;
}
istream& operator >>(istream& in,complex& C)
{
        cout<<"\nEnter a real part"<< endl;
        in>>C.real;
        cout<<"\nEnter an imaginary part"<< endl;
        in>>C.imaginary;
        return in;
}
int main()
{
        complex c1(3,4);
        complex c2(0,0); //Initialized with zero for real and imag,
        //lets take input by overloading using >> operator.
        cout<< c1;
        cin>>c2;
        cout<<"Complex No. is ";
        cout<< c2;
        return 0;
}
```

On compiling and executing the above program we get the following **output:**

```
3 + 4i

Enter a real part
7

Enter an imaginary part
5
Complex No. is 7 + 5i
```

Following paragraph will explain why we need to override << and >> operators using friend function only?

- cout and cin are the objects of ostream and istream classes respectively, in these classes << and >> operators are overloaded for built in data type.
- Obviously it does not have any operator overloaded for user defined data types.
- cout<< c1 ;
- Above statement can be resolved as cout.operator <<(c1) only if ostream class has overloaded operator << for user defined data type in our case complex.
- However we cannot implement this operator << in ostream class because its predefined class in library.
- So in another way cout<< c1; can also resolved as operator<< (cout, c1); In this case operator will be global function accessing the private members of user defined class i.e. complex. Hence it should be declared as friend of user defined class.

Again, why cout and cin objects are necessary to be passed by reference?

- When objects are passed by value in that case copy constructor of that particular class gets called.
- In this case if we pass cout by value copy constructor which takes cout object as an argument will be called, but copy constructor of ostream class is protected hence it is not accessible outside the class which is why we need to pass cout and cin by reference.

Table 6.3 below contains the name of all those operators which can be overloaded.

Table 6.3: Operators that can be overloaded

+	*	%	/	-	^
&	~	,	!	|	=
<	<=	++	>=	>	--
<<	==	&&	!=	>>	||
+=	/=	^=	%=	-=	&=

|=	<<=	[]	>>=	*=	()
->	New	delete	new []	->*	delete []

Table 6.4 below contains the name of all those operators which cannot be overloaded.

Table 6.4: Operators that cannot be overloaded

:: (Scope resolution operator)	.* (pointer to member)	. (dot operator)	?: (ternary conditional operator)	sizeof operator

6.7 Data Conversion

Implicit conversion

Conversions that are performed automatically when a value is copied to a compatible type. For example:

```
short b=200;
int a;
a=b;
```

Here, the value of b is endorsed from short to integer without the requirement of any explicit operatation. This is recognized as a standard conversion. Standard conversions touches fundamental data-types, and permit the changes between numerical types (short to integer, integer to float, double to integer etc.), to or from Boolean and some pointer conversions.

Converting to integer from some smaller data-type, or to double from float is known as promotion, and is definite to yield the exact similar value in the target type. Other transformations between arithmetic types may not always be able to characterize the similar value accurately:

- If a negative int data is transformed to an unsigned type, the subsequent value corresponds to its 2's complement bitwise representation (i.e., -1 come to be the largest value representable by the type, -2 the next largest, ...).
- The changes from/to Boolean reflect false corresponding to 0 (for numeric types) and to '\0' (null pointer) for pointer types; true is corresponding to all other values and is transformed to the equivalent value as 1.
- If the change is from a floating point type to an int type, the value is curtailed (the decimal part is detached). If the outcome is beyond the range of representable values by the type, the change causes indeterminate behavior.
- Otherwise, if the change is between numeric types of the similar type (int-to-int or float-to-float), the conversion is legal, but the value depends on implementation (and may not be movable).

Some of these conversions may infer a loss of accuracy, which the compiler can indicate with a caution.

This caution can be escaped with an explicit change.

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in common permit the following conversions:

- Null pointers can be changed to pointers of any type
- Pointers of any type can be transformed to void pointers.
- Pointer up cast: pointers to a sub-class can be transformed to a pointer of an unambiguous and accessible and super-class, without altering its volatile or const qualification.

Implicit conversions with classes

In the classes' world, implicit conversions can be manage by means of three member functions:

- Single-argument constructors: implicit change is allowed from a specific type to initialize an object.
- Assignment operator: implicit change is allowed from a specific type on assignments.
- Type-cast operator: implicit change is allowed to a specific type.

For example:

```
// implicit conversion of classes:
#include <iostream>
using namespace std;

class X {};

class Y {
public:
  // conversion from X (constructor):
  Y (const X &a) {}
  // conversion from X (assignment):
  Y& operator= (const X& a) {return *this;}
  // conversion to X (type-cast operator)
  operator X() {return X();}
};

int main ()
{
  X fop;
  Y ba = fop;    // calls constructor
  ba = fop;      // calls assignment
  fop = ba;      // calls type-cast operator
  return 0;
}
```

Output

A specific syntax is used by the type-cast operator: it uses the operator keyword trailed by the target type and an empty set of parentheses. Note that the return type is the target type and therefore is not indicated afore the operator keyword.

Type casting

C++ is a strong-typed language. Many conversions, especially those that denote a different explanation of the value, need an explicit conversion is known as type-casting in C++. There is two main syntaxes for generic type-casting: c-like and functional:

```
double a = 100.3;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

The working of these generic forms of type-casting is sufficient for most requirements with fundamental data-types. However, these operators can be universally useful on classes and pointers to classes, which can lead to code that -though being syntactically correct- can originate runtime errors. For example, the following code compiles without errors:

```
// type-casting class
#include <iostream>
using namespace std;

class Replica {
   double m, n;
};

class Add {
   int a, b;
  public:
   Add (int x, int y) { a=x; b=y; }
   int result() { return a+b;}
};

int main () {
  Replica r;
  Add * ptadd;
  ptadd = (Add*) &r;
  cout << ptadd->result();
  return 0;
}
```

Output

The program declares a pointer to Add, but then it allots to it a reference to an object of another unrelated type using explicit type-casting:

```
 ptadd = (Add*) &r;
```

Unrestricted explicit type-casting permits to change any pointer into some other type of pointer, individually of the types they point to. The following call to member consequences in either a runtime error or some other unpredicted outcomes.

In order to regulate these types of changes between classes, four specific casting operators are there: dynamic_cast, static_cast, reinterpret_cast, and const_cast. Their arrangement is to follow the new type surrounded between angle-brackets (<>) and instantaneously afterward, the expression to be converted between parentheses.

```
dynamic_cast <new_type> (expression)
static_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
const_cast <new_type> (expression)
```

The traditional type-casting equivalents to these expressions would be:

```
(new_type) expression
new_type (expression)
```

But each one with its own distinctive features.

6.8 Pitfalls of Overloading and Conversion

There are some restrictions that should be kept in mind when we implement **operator overloading**.

- We cannot change the Precedence and Associativity of an operator.
- Only existing operators can be overloaded. We cannot create any new operators.
- Unary operator works on a single operand, binary operator works on two operands etc. In other words, number of operands can't be changed.
- The meaning of a procedure cannot be redefined, i.e., we cannot change how integers are added.

In explicit Type Conversion following problems may arise.

- Assigning a value of smaller data type to a larger data type, may not pose any problem. But, assigning a value of larger data type to smaller type, may poses problems. The problem is that assigning to a smaller data type may loose information, or result in losing some precision.

Conversion Problems –

Conversion	Potential Problems
Double to float	Loss of precision(significant figures)
Float to int	Loss of fractional part
Long to int/short	Loss of Information as original value may be out of range for target type

6.9 Summary

This unit summarizes the concept of Array. An array is a collection of similar data types that are stored in different memory locations. The array elements are stored in continuous memory locations. The amount of storage required for holding the elements of an array depends on its type and size. The declaration and initialization of one and two dimensional arrays are studied in this chapter with programming examples.

6.10 Questions

1. What is the meaning of base address of the array?
2. Differentiate between one-dimensional and two-dimensional array.
3. Write a program to reverse the string "Hello".
4. What is data conversion? Write shortcomings of Data Conversion.
5. Explain Array of Object in C++.
6. Write a program to overload the unary operator "- -"?
7. Write a program to display the number of vowels in a string.
8. Write a C++ program to find the largest element in an Array.
9. Mention the binary operators that can be overloaded in C++
10. Explain Arrays of Objects.

6.11 Suggested Readings

1. Object oriented Programming with ANSI and Turbo C++ (Pearson): Ashok N Kamthane
2. Object Oriented Programming with C++, 3/e by E. Balagurusamy, McGraw Hill