C.P.PATEL & F.H.SHAH COMMERCE COLLEGE
(MANAGED BY SARDAR PATEL EDUCATION TRUST)
BCA, BBA (ITM) & PGDCA PROGRAMME:
BCA Semester III -Paper Code: US03CBCA23

UNIT 2- Input / Output, Arrays and Working with Classes

Sr. No.	Topics
1	Basic I/O in C++
2	Arrays in C++ : introduction, declaration, initialization of one , two and multi-dimensional arrays, operations on arrays
3	Working with strings : introduction, declaration, string manipulation and arrays of string
4	Classes and objects in C++
5	Constructors : default, parameterized, copy, constructor overloading and destructor
6	Access specifies, implementing and accessing class members
7	Working with objects : constant objects, nameless objects, live objects, arrays of objects

Basic Input / Output in C++

The C++ standard libraries provide an extensive set of input/output capabilities. Very basic and most common I/O operations required for C++ programming. C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device likes a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device likes a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

I/O Library Header Files

There are following header files important to C++ programs −

Sr.No	Header File & Function and Description
1	**<iostream>** This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.

The Standard Output Stream (cout)

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is

CPPHSCC

used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main()
{
   char str[ ] = "Hello C++";
   cout << "Value of str is : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main()
{
   char name[50];
   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

CPPHSCC

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

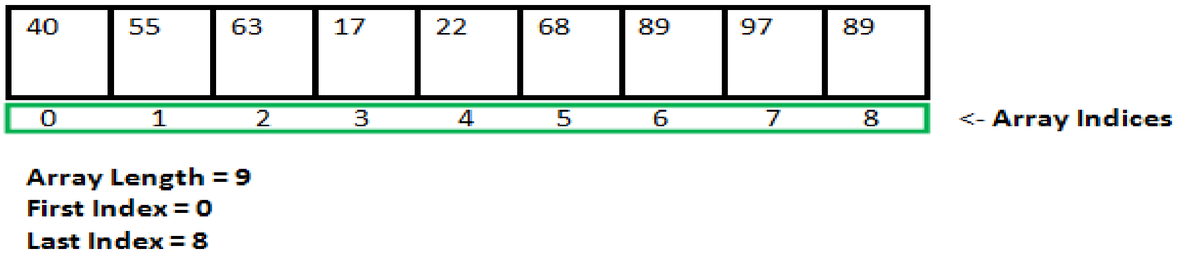
```
cin >> name >> age;
```

This will be equivalent to the following two statements −

```
cin >> name;
cin >> age;
```

Arrays in C++

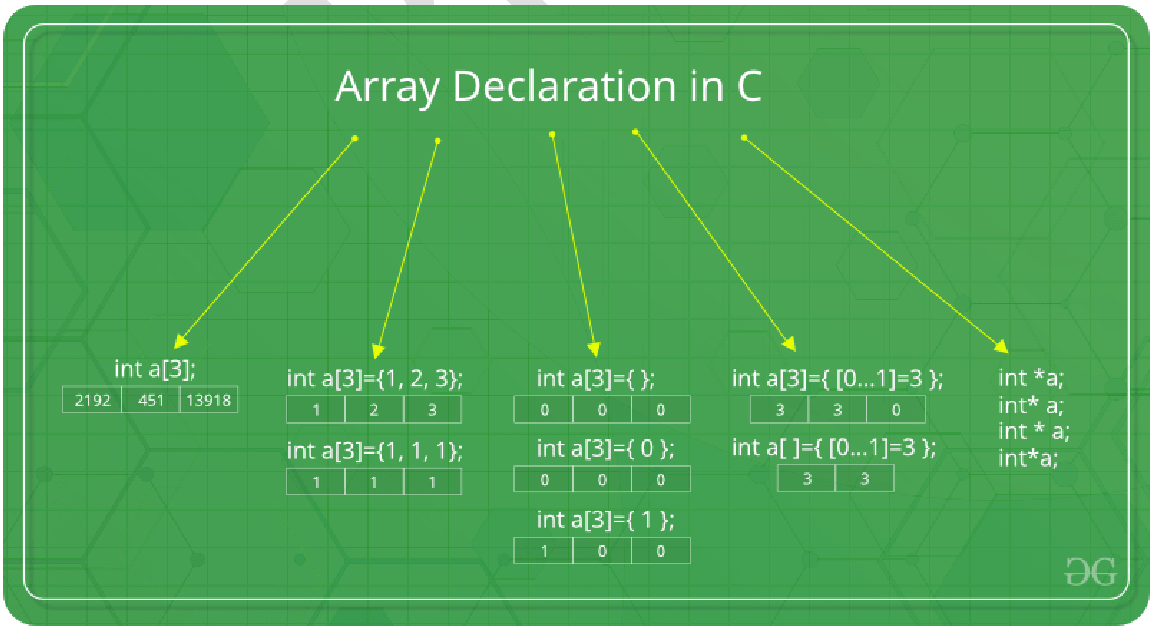
An array is collection of items stored at continuous memory locations.



Why do we need arrays?

We can use normal variables (v1, v2, v3, ..) when we have small number of objects, but if we want to store large number of instances, it becomes difficult to manage them with normal variables. The idea of array is to represent many instances in one variable.

Array declaration in C++:



We can declare an array by specifying its type and size or by initializing it or by both.

1. **Array declaration by specifying size**

```
// Array declaration by specifying size
int arr1[10];

// With recent C/C++ versions, we can also
// declare an array of user specified size
int n = 10;
int arr2[n];
```

2. **Array declaration by initializing elements**

```
// Array declaration by initializing elements
int arr[] = { 10, 20, 30, 40 }
// Compiler creates an array of size 4.
// above is same as  "int arr[4] = {10, 20, 30, 40}"
```

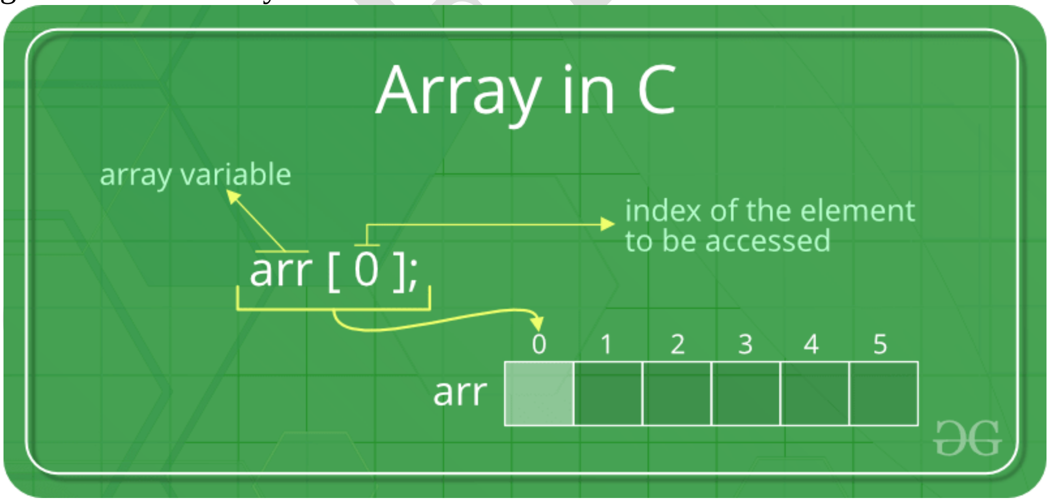
3. **Array declaration by specifying size and initializing elements**

```
// Array declaration by specifying size and initializing
// elements
int arr[6] = { 10, 20, 30, 40 }
// Compiler creates an array of size 6, initializes first
// 4 elements as specified by user and rest two elements as 0.
// above is same as  "int arr[] = {10, 20, 30, 40, 0, 0}"
```

EHSCC

Facts about Array in C/C++:

- **Accessing Array Elements:**
  Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.



Following are few examples.

```
#include <stdio.h>

int main()
{
   int arr[5];
   arr[0] = 5;
   arr[2] = -10;
   arr[3 / 2] = 2; // this is same as arr[1] = 2
```

```
    arr[3] = arr[0];

    printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);

    return 0;
}
```

Output:

```
5 2 -10 5
```

One Dimensional Array Program in C++

To print one dimensional array in C++ programming, you have to ask to the user to enter array size and array elements to store all the array elements in one dimensional and then print the array in one dimension using one for loop as shown here in the following program.

C++ Programming Code for One Dimensional (1D) Array

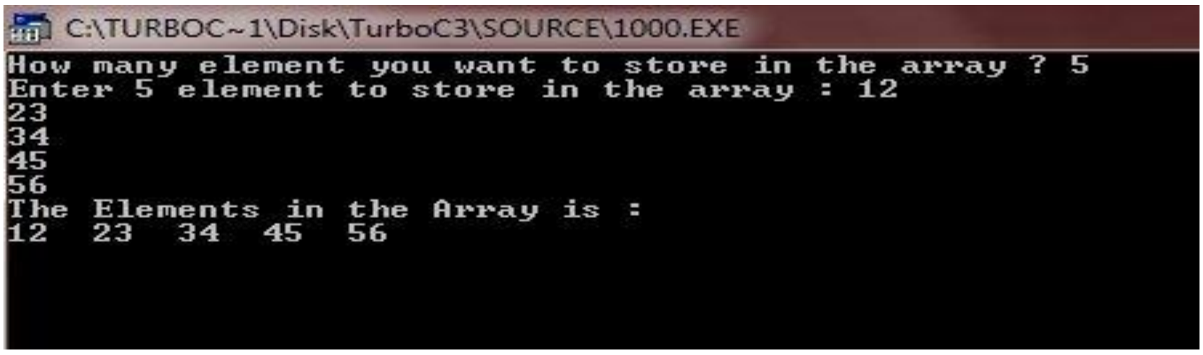
Following C++ program ask to the user to enter the array size, then ask to enter the element of the array to store the elements in the array, then finally display the array element on the screen:

```
/* C++ Program - One Dimensional Array Program */

#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int arr[50], n;
        cout<<"How many element you want to store in the array ? ";
        cin>>n;
        cout<<"Enter "<<n<<" element to store in the array : ";
        for(int i=0; i<n; i++)
        {
                cin>>arr[i];
        }
        cout<<"The Elements in the Array is : \n";
        for(i=0; i<n; i++)
        {
                cout<<arr[i]<<" ";
        }
        getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:

CPPFHSCC



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
How many element you want to store in the array ? 5
Enter 5 element to store in the array : 12
23
34
45
56
The Elements in the Array is :
12  23  34  45  56
```

Two Dimensional Array Program in C++

Two dimensional (2D) array can be made in C++ programming language by using two for loops, first is outer **for** loop and the second one is inner **for** loop. The outer **for** loop is responsible for rows and the inner **for** loop is responsible for columns as shown here in the following program.

C++ Programming Code for Two Dimensional (2D) Array

Following C++ program ask to the user to enter row and column size of the array then ask to the user to enter array elements, and the program will display the array elements in two dimensional (i.e., display array elements in row and column):

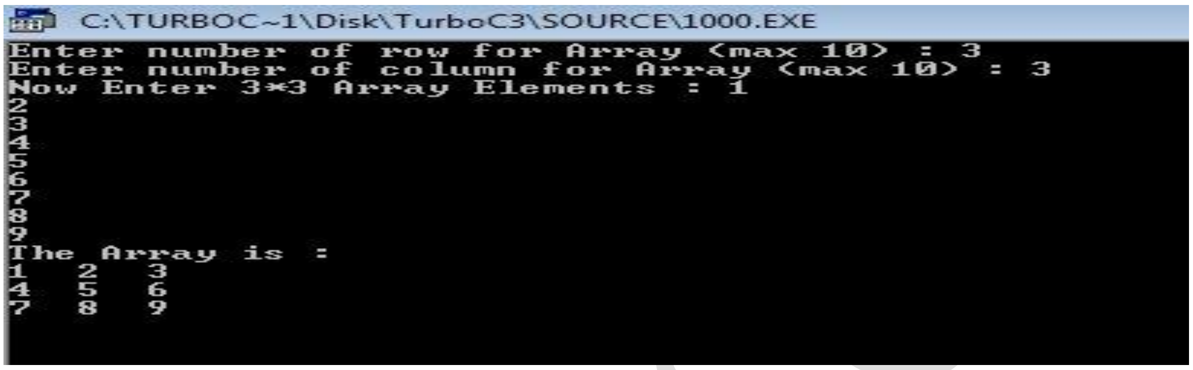
```
/* C++ Program - Two Dimensional Array Program */

#include<iostream.h>
#include<conio.h>
void main()
{
	clrscr();
	int arr[10][10], row, col, i, j;
	cout<<"Enter number of row for Array (max 10) : ";
	cin>>row;
	cout<<"Enter number of column for Array (max 10) : ";
	cin>>col;
	cout<<"Now Enter "<<row<<"*"<<col<<" Array Elements : ";
	for(i=0; i<row; i++)
	{
		for(j=0; j<col; j++)
		{
			cin>>arr[i][j];
		}
	}
	cout<<"The Array is :\n";
	for(i=0; i<row; i++)
	{
		for(j=0; j<col; j++)
```

CPPFHSCC

```
                    {
                              cout<<arr[i][j]<<" ";
                    }
                    cout<<"\n";
          }
          getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
Enter number of row for Array (max 10) : 3
Enter number of column for Array (max 10) : 3
Now Enter 3*3 Array Elements : 1
2
3
4
5
6
7
8
9
The Array is :
1   2   3
4   5   6
7   8   9
```

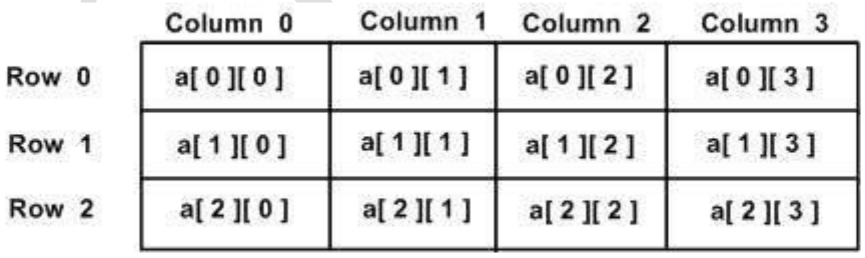
Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below −



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram.

```
#include <iostream>
using namespace std;
int main () {
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
    // output each array element's value
   for ( int i = 0; i < 5; i++ )
      for ( int j = 0; j < 2; j++ ) {
               cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
a[0][0]: 0
```

CPPFHSCC

```
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Example 3: MultiDimensional Array

```
#include <iostream>
using namespace std;
int main()
{
   // This array can store upto 12 elements (2x3x2)
   int test[2][3][2];
   cout << "Enter 12 values: \n";
   // Inserting the values into the test array
   // using 3 nested for loops.
   for(int i = 0; i < 2; ++i)
   {
      for (int j = 0; j < 3; ++j)
      {
         for(int k = 0; k < 2; ++k )
         {
            cin >> test[i][j][k];
         }
      }
   }
   cout<<"\nDisplaying Value stored:"<<endl;
   // Displaying the values with proper index.
   for(int i = 0; i < 2; ++i)
```

CPPFHSCC

```
    {
        for (int j = 0; j < 3; ++j)
        {
            for(int k = 0; k < 2; ++k)
            {
                cout << "test[" << i << "][" << j << "][" << k << "] = " << test[i][j][k] << endl;
            }
        }
    }
    return 0;
}
```

Output

```
Enter 12 values:
1
2
3
4
5
6
7
8
9
10
11
12

Displaying Value stored:
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12
```

CPPFHSCC

As the number of dimension increases, the complexity also increases tremendously although the concept is quite similar.

Operations on Array

Insert Element in Array in C++

To insert an element in an array in C++ programming, you have to ask to the user to enter the array size and array elements and ask to the user to enter the element (with their position) to insert the element at desired position in the array.

After inserting the element at the desired position in the array, display the new array on the screen as shown here in the following program.

C++ Programming Code to Insert Element in Array

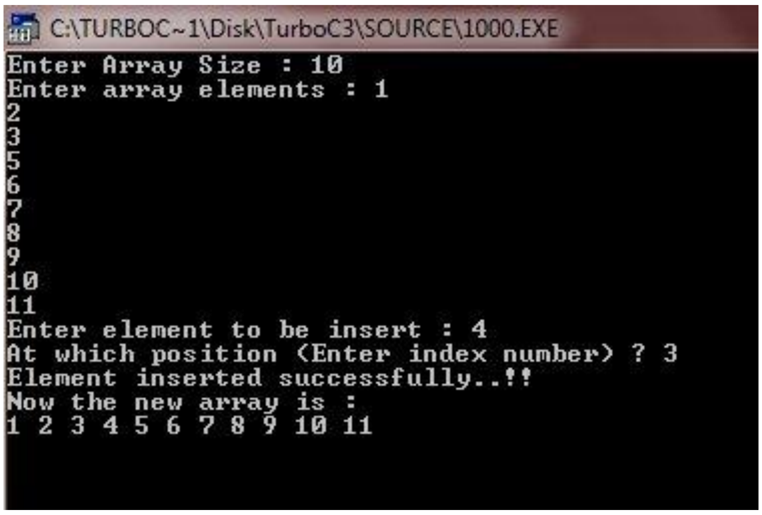
Following C++ program ask to the user to enter array size, then ask to the user to enter array element, then ask to the user to enter element or number to be insert, then at last it will ask to the user to enter the position (index number) where he or she want to insert the desired element in the array, so this program insert the desired element and display the new array on the screen after inserting the element:

```
/* C++ Program - Insert Element in Array */

#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int arr[50], size, insert, i, pos;
        cout<<"Enter Array Size : ";
        cin>>size;
        cout<<"Enter array elements : ";
        for(i=0; i<size; i++)
        {
                cin>>arr[i];
        }
        cout<<"Enter element to be insert : ";
        cin>>insert;
        cout<<"At which position (Enter index number) ? ";
        cin>>pos;
        // now create a space at the required position
        for(i=size; i>pos; i--)
        {
                arr[i]=arr[i-1];
        }
```

```
            arr[pos]=insert;
            cout<<"Element inserted successfully..!!\n";
            cout<<"Now the new array is : \n";
            for(i=0; i<size+1; i++)
            {
                        cout<<arr[i]<<" ";
            }
            getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
Enter Array Size : 10
Enter array elements : 1
2
3
5
6
7
8
9
10
11
Enter element to be insert : 4
At which position (Enter index number) ? 3
Element inserted successfully..!!
Now the new array is :
1 2 3 4 5 6 7 8 9 10 11
```

Delete Element from Array in C++

To delete element from an array in C++ programming, you have to first ask to the user to enter the array size then ask to enter the array elements, now ask to enter the element which is to be deleted. Search that number if found then place the next element after the founded element to the back until the last as shown here in the following program.

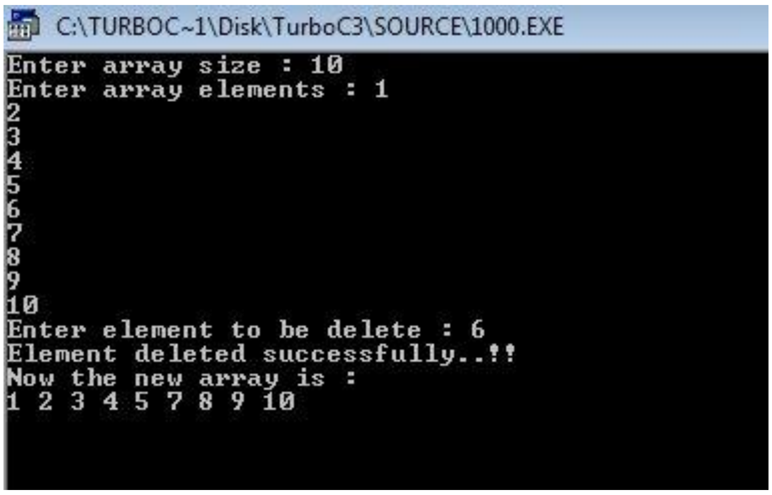
C++ Programming Code to Delete Element from Array

Following C++ program ask to the user to enter array size, then enter array elements then it will ask to enter element to be delete, to delete the desired element from the array, then display the new array on the screen:

```
/* C++ Program - Delete Element from Array */

#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int arr[50], size, i, del, count=0;
        cout<<"Enter array size : ";
        cin>>size;
        cout<<"Enter array elements : ";
        for(i=0; i<size; i++)
        {
                cin>>arr[i];
        }
        cout<<"Enter element to be delete : ";
        cin>>del;
        for(i=0; i<size; i++)
        {
                if(arr[i]==del)
                {
                        for(int j=i; j<(size-1); j++)
                        {
                                arr[j]=arr[j+1];
                        }
                        count++;
                        break;
                }
        }
        if(count==0)
        {
                cout<<"Element not found..!!";
        }
        else
        {
                cout<<"Element deleted successfully..!!\n";
                cout<<"Now the new array is :\n";
                for(i=0; i<(size-1); i++)
                {
                        cout<<arr[i]<<" ";
                }
        }
        getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
Enter array size : 10
Enter array elements : 1
2
3
4
5
6
7
8
9
10
Enter element to be delete : 6
Element deleted successfully..!!
Now the new array is :
1 2 3 4 5 7 8 9 10
```

Code for Program to perform array operations like append, insert, delete, edit, display in C++ Programming

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

main()
{
   int array[15];
   int no_el;
   clrscr();
   cout<<"Enter the no of element :";
   cin>>no_el;
   for(int i=0;i<no_el;i++)
   {
      cout<<"Enter the element : ";
      cin>>array[i];
   }
   while(1)
   {
      clrscr();
      cout<<endl<<"1. Append";
      cout<<endl<<"2. Insert";
      cout<<endl<<"3. Delete by value";
      cout<<endl<<"4. edit";
      cout<<endl<<"5. display";
      cout<<endl<<"6. search";
      cout<<endl<<"7. exit";
      cout<<endl<<"Enter your choice : ";
      int choice;
      cin>>choice;
      switch(choice)
      {
         case 1:
```

```
        cout<<"Enter the new element : ";
        int new_el;
        cin>>new_el;
        array[no_el]=new_el;
        no_el++;
    break;
    case 2:
        cout<<"Enter the position at which you want to insert : ";
        int pos;
        cin>>pos;
        cout<<"Enter the new element : ";
        cin>>new_el;
        pos--;
        for(i=no_el-1;i>=pos;i--)
            array[i+1]=array[i];
        array[pos]=new_el;
        no_el++;
    break;
    case 3:
        cout<<"Enter the value to be search : ";
        int key;
        cin>>key;
        for(pos=0;pos<no_el;pos++)
        {
            if(array[pos]==key)
                break;
        }
        if(pos==no_el)
        {
            cout<<"Search key not found";
            break;
        }
        for(i=pos;i<no_el;i++)
            array[i]=array[i+1];
        no_el--;
    break;
    case 4:
        cout<<"Enter the position to be edit : ";
        cin>>pos;
        cout<<"Enter the new value for old position : ";
        cin>>array[pos-1];
    break;
    case 5:
        cout<<endl;
        for(i=0;i<no_el;i++)
            cout<<endl<<"The element is : "<<array[i];
    break;
    case 6:
```

SPFHSCC

```
          cout<<"Enter the value to be search : ";
          cin>>key;
          for(pos=0;pos<no_el;pos++)
          {
             if(array[pos]==key)
                break;
          }
          if(pos==no_el)
          {
             cout<<"Search key not found";
             break;
          }
          cout<<"Search key found at : "<<pos+1;
       break;
       case 7:
          return(0);
       break;
    }
    getch();
 }
}
```

C++ Strings

C++ provides following two types of string representations −

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

The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

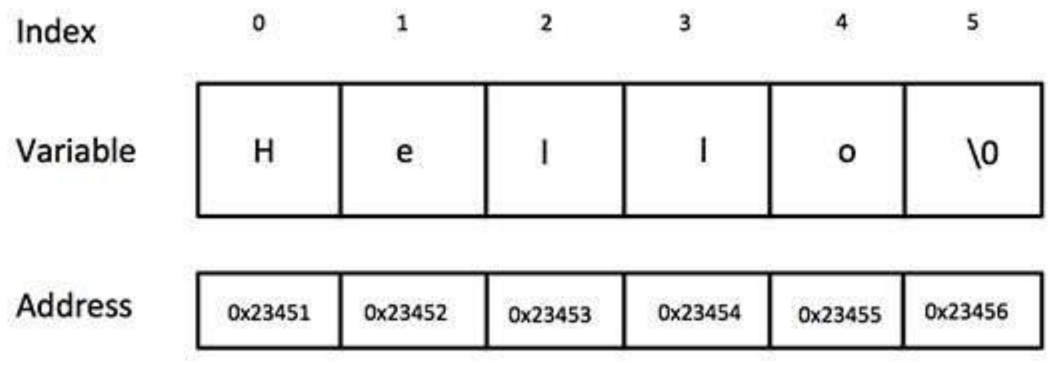
```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

CPPFHSCC

Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

```
#include <iostream>
using namespace std;
int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings −

Sr. No	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.

CPPFHSCC

6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions −

```
#include <iostream>
#include <cstring>

using namespace std;

int main () {

   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example

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

using namespace std;

int main () {

   string str1 = "Hello";
```

CPPFHSCC

```
  string str2 = "World";
  string str3;
  int  len ;

  // copy str1 into str3
  str3 = str1;
  cout << "str3 : " << str3 << endl;

  // concatenates str1 and str2
  str3 = str1 + str2;
  cout << "str1 + str2 : " << str3 << endl;

  // total length of str3 after concatenation
  len = str3.size();
  cout << "str3.size() :  " << len << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

Some more detailing about String:

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class)
- C-strings (C-style Strings)

In C programming, the collection of characters is stored in the form of arrays, this is also supported in C++ programming. Hence it's called C-strings.

C-strings are arrays of type char terminated with null character, that is, \0 (ASCII value of null character is 0).

How to define a C-string?

```
char str[] = "C++";
```

In the above code, str is a string and it holds 4 characters.

Although, "C++" has 3 character, the null character \0 is added to the end of the string automatically.

Alternative ways of defining a string

```
char str[4] = "C++";

char str[] = {'C','+','+','\0'};

char str[4] = {'C','+','+','\0'};
```

CPPFHSCC

Like arrays, it is not necessary to use all the space allocated for the string. For example:

```
char str[100] = "C++";
```

Example 1: C++ String to read a word

C++ program to display a string entered by user.

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

int main()
{
   char str[100];

   cout << "Enter a string: ";
   cin >> str;
   cout << "You entered: " << str << endl;

   cout << "\nEnter another string: ";
   cin >> str;
   cout << "You entered: "<<str<<endl;

   return 0;
}
```

Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Notice that, in the second example only "Programming" is displayed instead of "Programming is fun".
This is because the extraction operator >> works as scanf() in C and considers a space " " has a terminating character.

Example 2: C++ String to read a line of text

C++ program to read and display an entire line entered by user.

```
#include <iostream>
using namespace std;
int main()
{
   char str[100];
   cout << "Enter a string: ";
   cin.get(str, 100);
   cout << "You entered: " << str << endl;
   return 0;
}
```

CPPFHSCC

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

To read the text containing blank space, cin.get function can be used. This function takes two arguments.
First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.
In the above program, str is the name of the string and 100 is the maximum size of the array.

string Object

In C++, you can also create a string object for holding strings.
Unlike using char arrays, string objects has no fixed length, and can be extended as per your requirement.

Example 3: C++ string using string data type

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

int main()
{
   // Declaring a string object
   string str;
   cout << "Enter a string: ";
   getline(cin, str);

   cout << "You entered: " << str << endl;
   return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

In this program, a string str is declared. Then the string is asked from the user.
Instead of using cin>> or cin.get() function, you can get the entered line of text using getline().
getline() function takes the input stream as the first parameter which is cin and str as the location of the line to be stored.

Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

```
#include <iostream>

using namespace std;

void display(char *);
void display(string);

int main()
```

CPPFHSCC

```
{
   string str1;
   char str[100];

   cout << "Enter a string: ";
   getline(cin, str1);

   cout << "Enter another string: ";
   cin.get(str, 100, '\n');

   display(str1);
   display(str);
   return 0;
}

void display(char s[ ])
{
   cout << "Entered char array is: " << s << endl;
}

void display(string s)
{
   cout << "Entered string is: " << s << endl;
}
```

Output

```
Enter a string:  Programming is fun.
Enter another string:  Really?
Entered string is: Programming is fun.
Entered char array is: Really?
```

In the above program, two strings are asked to enter. These are stored in str and str1respectively, where str is a char array and str1 is a string object.
Then, we have two functions display() that outputs the string onto the string.
The only difference between the two functions is the parameter. The first display() function takes char array as a parameter, while the second takes string as a parameter.
This process is known as function overloading.

How to define a class in C++?

A class is defined in C++ using keyword class followed by the name of class.
The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className
  {
  // some data
  // some functions
```

CPPFHSCC

```
  };
```

Example: Class in C++

```
class Test
{
   private:
       int data1;
       float data2;

   public:
       void function1()
       {   data1 = 2;  }

       float function2()
       {
           data2 = 3.5;
           return data2;
       }
  };
```

Here, we defined a class named Test.

This class has two data members: data1 and data2 and two member functions: function1()and function2().

Keywords: private and public

You may have noticed two keywords: private and public in the above example.

The private keyword makes data and functions private. Private data and functions can be accessed only from inside the same class.

The public keyword makes data and functions public. Public data and functions can be accessed out of the class.

Here, data1 and data2 are private members where as function1() and function2() are public members.

If you try to access private data from outside of the class, compiler throws error. This feature in OOP is known as data hiding.

C++ Objects

When class is defined, only the specification for the object is defined; no memory or storage is allocated.

CPPHSCC

To use the data and access functions defined in the class, you need to create objects.

Syntax to Define Object in C++

```
className objectVariableName;
```

You can create objects of Test class (defined in above example) as follows:

```
class Test
{
   private:
       int data1;
       float data2;

   public:
       void function1()
       {   data1 = 2;  }

       float function2()
       {
          data2 = 3.5;
          return data2;
       }
  };

int main()
{
   Test o1, o2;
}
```

Here, two objects o1 and o2 of Test class are created.

In the above class Test, data1 and data2 are data members and function1() and function2() are member functions.

Example: Object and Class in C++ Programming

```
// Program to illustrate the working of objects and class in C++ Programming
#include <iostream>
using namespace std;
class Test
{
   private:
       int data1;
       float data2;
   public:
        void insertIntegerData(int d)
      {
         data1 = d;
         cout << "Number: " << data1;
```

CPPFHSCC

```
        }
       float insertFloatData()
       {
          cout << "\nEnter data: ";
          cin >> data2;
          return data2;
        }
};
 int main()
 {
      Test o1, o2;
      float secondDataOfObject2;

      o1.insertIntegerData(12);
      secondDataOfObject2 = o2.insertFloatData();

      cout << "You entered " << secondDataOfObject2;
      return 0;
 }
```

Output

```
Number: 12
Enter data: 23.3
You entered 23.3
```

C++ Constructors

A constructor is a special type of member function that initializes an object automatically when it is created.
Compiler identifies a given member function is a constructor by its name and the return type.
Constructor has the same name as that of the class and it does not have any return type. Also, the constructor is always public.

```
... .. ...
class temporary
{
private:
            int x;
            float y;
public:
            // Constructor
            Temporary (): x(5), y(5.5)
            {
                        // Body of constructor
            }
            ... ..  ...
};
```

CPPFHSCC

```
int main()
{
        Temporary t1;
        ... .. ...
}
```

Above program shows a constructor is defined without a return type and the same name as the class.

How constructor works?

In the above pseudo code, temporary () is a constructor.
When an object of class temporary is created, the constructor is called automatically, and x is initialized to 5 and y is initialized to 5.5.
You can also initialize the data members inside the constructor's body as below. However, this method is not preferred.

```
temporary()
{
  x = 5;
  y = 5.5;
}
// This method is not preferred.
```

Use of Constructor in C++

Suppose you are working on 100's of Person objects and the default value of a data member age is 0. Initializing all objects manually will be a very tedious task.
Instead, you can define a constructor that initializes age to 0. Then, all you have to do is create a Person object and the constructor will automatically initialize the age.
These situations arise frequently while handling array of objects.
Also, if you want to execute some code immediately after an object is created, you can place the code inside the body of the constructor.

Example 1: Constructor in C++

Calculate the area of a rectangle and display it.

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

class Area
{
   private:
      int length;
      int breadth;

   public:
      // Constructor
```

CPPHSCC

```
      Area(): length(5), breadth(2){ }

      void GetLength()
      {
         cout << "Enter length and breadth respectively: ";
         cin >> length >> breadth;
      }

      int AreaCalculation() {  return (length * breadth);  }

      void DisplayArea(int temp)
      {
         cout << "Area: " << temp;
      }
};

int main()
{
   Area A1, A2;
   int temp;

   A1.GetLength();
   temp = A1.AreaCalculation();
   A1.DisplayArea(temp);

   cout << endl << "Default Area when value is not taken from user" << endl;

   temp = A2.AreaCalculation();
   A2.DisplayArea(temp);

   return 0;
}
```

CPPFHSCC

In this program, class Area is created to handle area related functionalities. It has two data members length and breadth.
A constructor is defined which initialises length to 5 and breadth to 2.
We also have three additional member functions GetLength(), AreaCalculation() and DisplayArea() to get length from the user, calculate the area and display the area respectively.
When, objects A1 and A2 are created, the length and breadth of both objects are initialized to 5 and 2 respectively, because of the constructor.
Then, the member function GetLength() is invoked which takes the value of length and breadth from the user for object A1. This changes the length and breadth of the object A1.
Then, the area for the object A1 is calculated and stored in variable temp by calling AreaCalculation() function and finally, it is displayed.
For object A2, no data is asked from the user. So, the length and breadth remains 5 and 2 respectively.
Then, the area for A2 is calculated and displayed which is 10.

Output

```
Enter length and breadth respectively: 6
7
Area: 42
Default Area when value is not taken from user
Area: 10
```

Constructor Overloading

Constructor can be overloaded in a similar way as function overloading.
Overloaded constructors have the same name (name of the class) but different number of arguments.
Depending upon the number and type of arguments passed, specific constructor is called.
Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Example 2: Constructor overloading

```
// Source Code to demonstrate the working of overloaded constructors
#include <iostream>
using namespace std;

class Area
{
    private:
       int length;
       int breadth;

    public:
       // Constructor with no arguments
       Area(): length(5), breadth(2) { }

       // Constructor with two arguments
       Area(int l, int b): length(l), breadth(b){ }

       void GetLength()
       {
           cout << "Enter length and breadth respectively: ";
           cin >> length >> breadth;
       }

       int AreaCalculation() {  return length * breadth;  }

       void DisplayArea(int temp)
       {
           cout << "Area: " << temp << endl;
       }
};
```

CPPFHSCC

```
int main()
{
   Area A1, A2(2, 1);
   int temp;

   cout << "Default Area when no argument is passed." << endl;
   temp = A1.AreaCalculation();
   A1.DisplayArea(temp);

   cout << "Area when (2,1) is passed as argument." << endl;
   temp = A2.AreaCalculation();
   A2.DisplayArea(temp);

   return 0;
}
```

For object A1, no argument is passed while creating the object.
Thus, the constructor with no argument is invoked which initialises length to 5 and breadthto 2. Hence, area of the object A1 will be 10.
For object A2, 2 and 1 are passed as arguments while creating the object.
Thus, the constructor with two arguments is invoked which initialises length to l (2 in this case) and breadth to b (1 in this case). Hence, area of the object A2 will be 2.
Output

```
Default Area when no argument is passed.
Area: 10
Area when (2,1) is passed as argument.
Area: 2
```

Default Copy Constructor

An object can be initialized with another object of same type. This is same as copying the contents of a class to another class.
In the above program, if you want to initialise an object A3 so that it contains same values as A2, this can be performed as:

```
....
int main()
{
  Area A1, A2(2, 1);

  // Copies the content of A2 to A3
  Area A3(A2);
    OR,
  Area A3 = A2;
}
```

You might think, you need to create a new constructor to perform this task. But, no additional constructor is needed. This is because the copy constructor is already built into all classes by default.

CPFHSCC

CPPFHSCC

Types of Constructors

1. **Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}
```

Output:

```
a: 10

b: 20
```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2. **Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;
```

```
class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
 Example e = Example(0, 50); // Explicit call
 Example e(0, 50);           // Implicit call
```

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

Can we have more than one constructors in a class?
Yes, It is called Constructor Overloading.

CPPFHSCC

3. **Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.

C++ Constructor Overloading Example

```
/*.....A program to highlight the concept of constructor overloading.......... */
#include <iostream>
using namespace std;
class ABC
{
    private:
      int x,y;
    public:
      ABC ()      //constructor 1 with no arguments
     {
         x = y = 0;
      }
      ABC(int a)    //constructor 2 with one argument
     {
          x = y = a;
      }
      ABC(int a,int b)    //constructor 3 with two argument
      {
           x = a;
           y = b;
      }
      void display()
      {
           cout << "x = " << x << " and " << "y = " << y << endl;
      }
};

int main()
{
    ABC cc1; //constructor 1
    ABC cc2(10); //constructor 2
    ABC cc3(10,20); //constructor 3
    cc1.display();
    cc2.display();
    cc3.display();
    return 0;
 }  //end of program
```

Output

```
x = 0 and y = 0
x = 10 and y = 10
x = 10 and y = 20
```

Explanation

In the above program, three constructors have been defined. The first one is invoked when no

CPPFHSCC

arguments is passed in ABC cc1. The second one is invoked when we pass one integer value as an argument as the constructor has one integer parameter. Similarly, when we pass two arguments in ABC cc3, the constructor with two arguments is invoked.

Destructors in C++

What is destructor?
Destructor is a member function which destructs or deletes an object.

When is destructor called?
A destructor function is called automatically when the object goes out of scope:
(1) the function ends
(2) the program ends
(3) a block containing local variables ends
(4) a delete operator is called

How destructors are different from a normal member function?
Destructors have same name as the class preceded by a tilde (~)
Destructors don't take any argument and don't return anything

Can there be more than one destructor in a class?
No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?
If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Can a destructor be virtual?
Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function. See virtual destructor for more details.

Destructors don't take any argument and don't return anything

```
class String
{
private:
   char *s;
   int size;
public:
   String(char *); // constructor
   ~String();      // destructor
};

String::String(char *c)
{
   size = strlen(c);
   s = new char[size+1];
   strcpy(s,c);
}

String::~String()
{
   delete []s;
}
```

C++ Access Specifiers

Access specifier can be either private or protected or public. In general access specifiers are the access restriction imposed during the derivation of different subclasses from the base class.

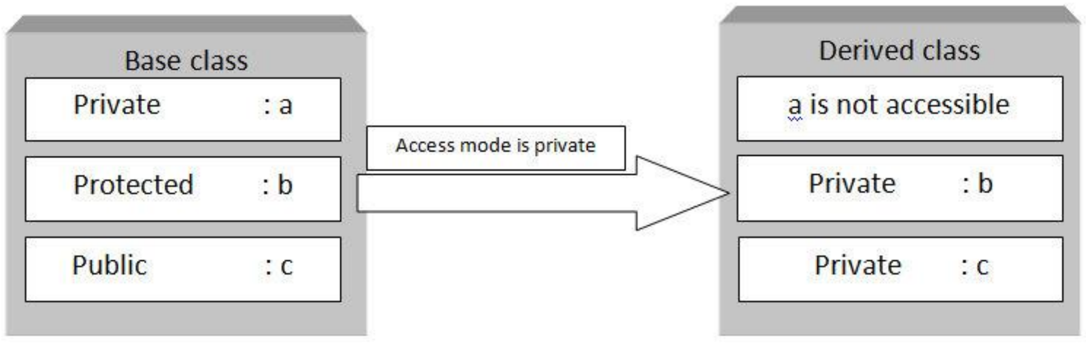
- private access specifier
- protected access specifier
- public access specifier

Private access specifier

If **private access specifier** is used while creating a class, then the public and protected data members of the base class become the private member of the derived class and private member of base class remains private.

In this case, the members of the base class can be used only within the derived class and cannot be accessed through the object of derived class whereas they can be accessed by creating a function in the derived class.

Following block diagram explain how data members of base class are inherited when derived class access mode is private.



Note: Declaring data members with private access specifier is known as data hiding.

Sample program demonstrating private access specifier

```
// private access specifier.cpp
#include <iostream>
using namespace std;

class base
{
            private:
        int x;

            protected:
```

CHSCC

```
                int y;

            public:
                int z;

            base() //constructor to initialize data members
            {
               x = 1;
               y = 2;
               z = 3;
            }
};

class derive: private base
{
            //y and z becomes private members of class derive and x remains private
            public:
                void showdata()
                {
                   cout << "x is not accessible" << endl;
            cout << "value of y is " << y << endl;
            cout << "value of z is " << z << endl;
                }
};
int main()
{
    derive a; //object of derived class
    a.showdata();
    //a.x = 1;   not valid : private member can't be accessed outside of class
    //a.y = 2;   not valid : y is now private member of derived class
    //a.z = 3;   not valid : z is also now a private member of derived class
    return 0;
}           //end of program
```

Output

```
x is not accessible
value of y is 2
value of z is 3
```

Explanation

When a class is derived from the base class with private access specifier the private members of the base class can’t be accessed. So in above program, the derived class cannot access the So in above program, the derived class cannot access the member x which is private in the base class, however, derive class has access to the protected and public members of the base class. So the

Hence the function showdata in derived class can access the public and protected member of the base class.

Common Programming Error

When function which is not the member of class or friend try to access a private member of that class results in an error.

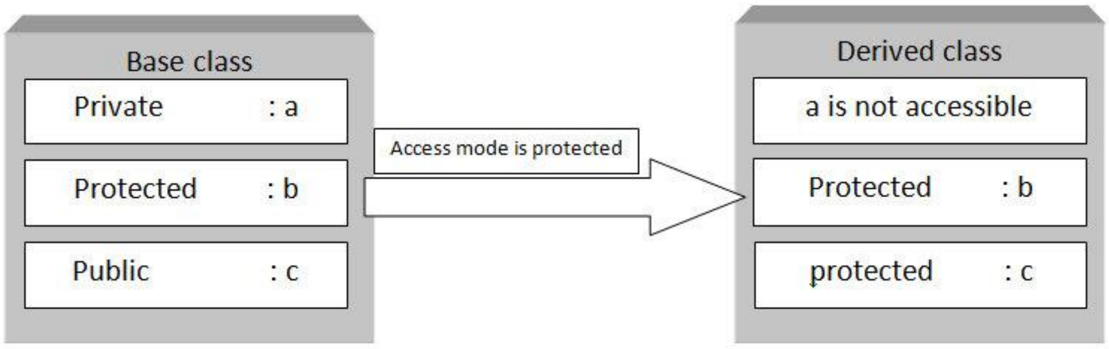
CPPFHSCC

Protected Access Specifier

If **protected access specifier** is used while deriving class then the public and protected data members of the base class becomes the protected member of the derived class and private member of the base class are inaccessible.

In this case, the members of the base class can be used only within the derived class as protected members except for the private members.

Following block diagram explain how data members of base class are inherited when derived class access mode is protected.



Sample program demonstrating protected access specifier

```
// protected access specifier.cpp
#include <iostream>
using namespace std;

class base
{
            private:
      int x;

            protected:
               int y;

            public:
               int z;

            base() //constructor to initialize data members
            {
              x = 1;
              y = 2;
              z = 3;
            }
};

class derive: protected base
{
            //y and z becomes protected members of class derive
            public:
```

CPPFHS

```
                void showdata()
                {
                  cout << "x is not accessible" << endl;
             cout << "value of y is " << y << endl;
             cout << "value of z is " << z << endl;
                }
};
int main()
{
    derive a; //object of derived class
    a.showdata();
    //a.x = 1;   not valid : private member can't be accessed outside of class
    //a.y = 2;   not valid : y is now private member of derived class
    //a.z = 3;   not valid : z is also now a private member of derived class
    return 0;
}            //end of program
```

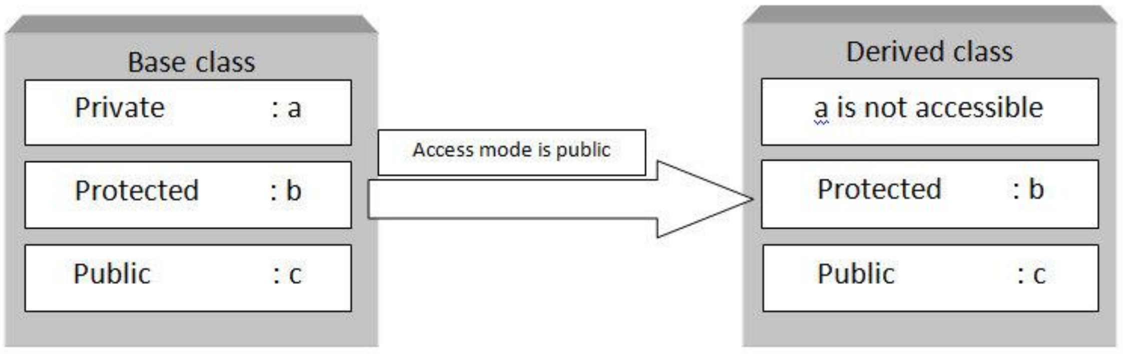
Output

```
x is not accessible
value of y is 2
value of z is 3
```

Public access specifier

If **public access specifie**r is used while deriving class then the public data members of the base class becomes the public member of the derived class and protected members becomes the protected in the derived class but the private members of the base class are inaccessible. Following block diagram explain how data members of base class are inherited when derived class access mode is public

UHSCC



Sample program demonstrating public access specifier

```
// public access specifier.cpp
#include <iostream>
using namespace std;

class base
{
            private:
      int x;
```

```
            protected:
               int y;

            public:
               int z;

            base() //constructor to initialize data members
            {
              x = 1;
              y = 2;
              z = 3;
            }
};

class derive: public base
{
            //y becomes protected and z becomes public members of class derive
            public:
               void showdata()
               {
                 cout << "x is not accessible" << endl;
            cout << "value of y is " << y << endl;
            cout << "value of z is " << z << endl;
               }
};
int main()
{
    derive a; //object of derived class
    a.showdata();
    //a.x = 1;   not valid : private member can't be accessed outside of class
    //a.y = 2;   not valid : y is now private member of derived class
    //a.z = 3;   not valid : z is also now a private member of derived class
    return 0;
}           //end of program
```

Output

```
x is not accessible
value of y is 2
value of z is 3
```

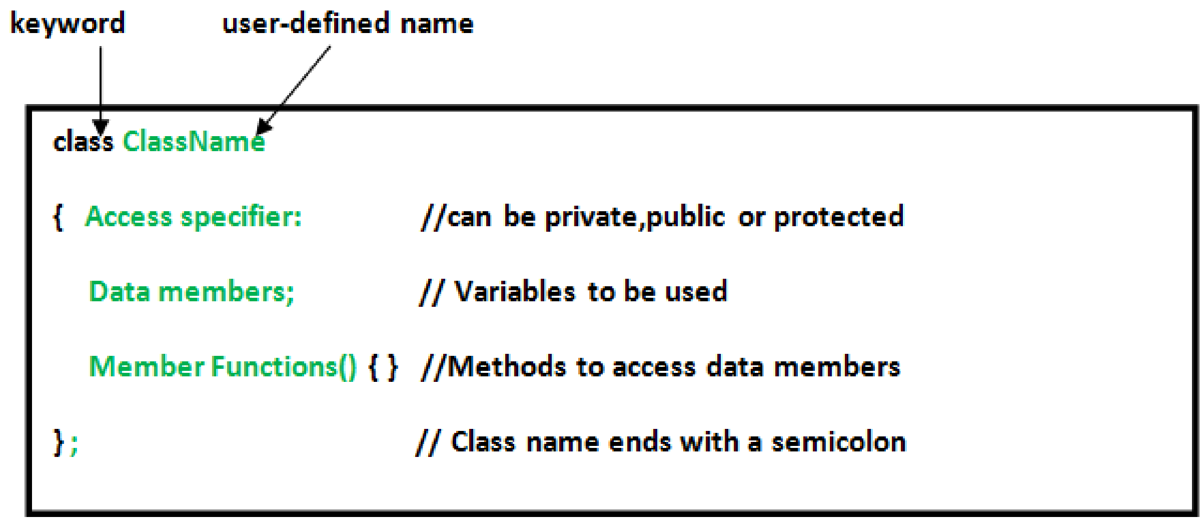
This is all about C++ access Specifiers.

CPPFHSCC

Implementing and accessing class members

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



```
class ClassName

{  Access specifier:          //can be private,public or protected

   Data members;              // Variables to be used

   Member Functions() { }     //Methods to access data members

};                            // Class name ends with a semicolon
```

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

```
// C++ program to demonstrate
// accessing of data members

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
   // Access specifier
   public:

   // Data Members
```

```
    string geekname;

    // Member Functions()
    void printname()
    {
       cout << "Geekname is: " << geekname;
    }
};

int main() {

    // Declare an object of class geeks
    Geeks obj1;

    // accessing data member
    obj1.geekname = "Abhi";

    // accessing member function
    obj1.printname();
    return 0;
}
```

Output:

```
Geekname is: Abhi
```

Member Functions in Classes
There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```
// C++ program to demonstrate function
// declaration outside class

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    string geekname;
    int id;

    // printname is not defined inside class defination
    void printname();

    // printid is defined inside class defination
    void printid()
    {
        cout << "Geek id is: " << id;
```

CPPFHSCC

```
    }
};

// Definition of printname using scope resolution operator ::
void Geeks::printname()
{
    cout << "Geekname is: " << geekname;
}
int main() {

    Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id=15;

    // call printname()
    obj1.printname();
    cout << endl;

    // call printid()
    obj1.printid();
    return 0;
}
```

Output:

```
Geekname is: xyz

Geek id is: 15
```

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.
Note: Declaring a friend function is a way to give private access to a non-member function

The members of a class can be directly accessed inside the class using their names. However, accessing a member outside the class depends on its access specifier. The access specifier not only determines the part of the program where the member is accessible, but also how it is accessible in the program.

Accessing Public Members: The public members of a class can be accessed outside the class directly using the object name and dot operator '. The dot operator associates a member with the specific object of the class.
The syntax for accessing a public data member outside the class is
obj_name.member_name; The syntax for calling a public member function is
object_name.function_name(parameter_list);
To understand the concept of accessing public members of a class, consider this example.
Example : A code segment to demonstrate the concept of accessing public members of a class

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

SDJ-BPFHSCC

```
{
int x;
public:
int y;
int z;
void fn(int a);
} ;
int main ()
{
number p;
p.y = 7;
p.z = 2;
p.x = 3;
p.fn (10) ;
return 0;
}
```

In this example, a class number having three data members x, y and z is defined. The data member x is private by default, whereas, y and z are declared as public. Hence, y and z can be accessed directly outside the class using the object name and the dot operator. However, x being a private data member cannot be accessed directly outside the class.

Accessing Private Members: The private members of a class are not accessible outside the class not even with the object name. However, they can be accessed indirectly through the public member functions of that class.

To understand the concept of accessing private members of a class, consider this example.

Example : A program to demonstrate the concept of accessing private members of a class

```
class book
{
/ / body of class as in Example1
} ;
int main ()
{
book bookl;
book1.price = 350;
bookl.title="Exploring IT";
bookl.getdata ("Exploring IT", 350);
return 0;
}
```

In this example, the object bookl of class book is used to access the public member function getdata (), which provides an indirect access to private data members title and price.

The basics of classes and objects can be summarized in a single program as shown in this example.

Example : A program to demonstrate the concept of classes and objects in C++

```
#include<iostream>
using namespace std;
class book
{
// definition of a class
char title [30];
float price;
public:
void getdata(char[] ,float);
void putdata ();
} ;
void book :: getdata (char a [],float b)
{
// definition of member function
strcpy(title, a);
price = b;
}
void book :: putdata(). II Definition of member function
{
Cout<<"Title: "<<title<<", ";
Cout<<"Price:Rs"<<price;
}
int main ()
{
book book1, book2, book3; // creating objects
book1.getdata("Exploring IT" ,350);
// reading data into book 1
book2. getdata ("JAVA", 300) ;
//reading data into book 2
book3.getdata("Computer Applications",400);
// reading data into book 3
Cout<<"\nTitle and Price of Book l\n";
bookl.putdata () ; II displaying data of book 1
cout<<"\nTitle and Price of Book 2\n";
book2.putdata (); II displaying data of book 2
cout<<"\nTitle and Price of Book 3\n";
book3.putdata();
return 0;
}
```

The output of the program is

```
 Title and Price of Book 1
Title: Exploring IT, Price: Rs 350
Title and Price of Book 2
Title: JAVA, Price: Rs 300
Title and Price of Book 3
Title: Computer Applications, Price: Rs 400
```

Working with objects : constant objects, nameless objects, live objects, arrays of objects

Constant Objects:

Class member functions can be made **const**. What does this mean? To understand, you must first grasp the concept of **const** objects.

A **const** object is defined the same for a user-defined type as a built-in type. For example:

const int i = 1;

```
const blob b(2);
```

Here, **b** is a **const** object of type **blob**. Its constructor is called with an argument of two. For the compiler to enforce **const**ness, it must ensure that no data members of the object are changed during the object's lifetime. It can easily ensure that no public data is modified, but how is it to know which member functions will change the data and which ones are "safe" for a **const** object?

If you declare a member function **const**, you tell the compiler the function can be called for a **const** object. A member function that is not specifically declared **const** is treated as one that will modify data members in an object, and the compiler will not allow you to call it for a **const** object.

It doesn't stop there, however. Just *claiming* a member function is **const** doesn't guarantee it will act that way, so the compiler forces you to reiterate the **const** specification when defining the function. (The **const** becomes part of the function signature, so both the compiler and linker check for **const**ness.) Then it enforces **const**ness during the function definition by issuing an error message if you try to change any members of the object *or* call a non-**const** member function. Thus, any member function you declare**const** is guaranteed to behave that way in the definition.

To understand the syntax for declaring **const** member functions, first notice that preceding the function declaration with **const** means the return value is **const**, so that doesn't produce the desired results. Instead, you must place the **const** specifier *after* the argument list. For example,

```
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

Note that the **const** keyword must be repeated in the definition or the compiler sees it as a different function. Since **f( )** is a **const** member function, if it attempts to change **i**in any way *or* to call another member function that is not **const**, the compiler flags it as an error.

You can see that a **const** member function is safe to call with both **const** and non-**const** objects. Thus, you could think of it as the most general form of a member function (and because of this, it is unfortunate that member functions do not automatically default to **const**). Any function that doesn't modify member data should be declared as **const**, so it can be used with **const** objects.

Here's an example that contrasts a **const** and non-**const** member function:

```
//: C08:Quoter.cpp
// Random quote selection
```

```
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
  int lastQuote() const;
  const char* quote();
};

Quoter::Quoter(){
  lastquote = -1;
  srand(time(0)); // Seed random number generator
}

int Quoter::lastQuote() const {
  return lastquote;
}

const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
    "Things that make us happy, make us wise",
  };
  const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // OK
//!  cq.quote(); // Not OK; non const function
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

Neither constructors nor destructors can be **const** member functions because they virtually always perform some modification on the object during initialization and cleanup. The **quote( )** member function also cannot be **const** because it modifies the data member **lastquote** (see the **return** statement). However, **lastQuote( )** makes no modifications, and so it can be **const** and can be safely called for the **const** object **cq**.

Nameless Objects:

Sometimes to reduce the code size, we create nameless temporary object of class. When we want to return an object from member function of class without creating an object, for this: we just call the constructor of class and return it to calling function and there is an object to hold the reference returned by constructor. This concept is known as **nameless temporary objects**, using this we are going to implement a C++ program for **pre-increment operator overloading**.

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

class Sample
{
        //private data section
        private:
        int count;

        public:         //default constructor
        Sample()
        { count = 0;}
        //parameterized constructor
        Sample(int c)
        { count = c;}
        //Operator overloading function definition
        Sample operator++()
        {
                ++count;
                //returning count of Sample
                //There is no new object here,
                //Sample(count): is a constructor by passing value of count
                //and returning the value (incremented value)
                return Sample(count);
        }
                //printing the value
        void printValue()
        {
                cout<<"Value of count : "<<count<<endl;
        }
};
//main program
```

CPPFHSCC

```
int main()
{
        int i = 0;
        Sample S1(100), S2;

        for(i=0; i< 5; i++)
        {
                S2 = ++S1;

                cout<<"S1 :"<<endl;
                S1.printValue();

                cout<<"S2 :"<<endl;
                S2.printValue();
        }
        return 0;
}
```

In this program, we used **nameless temporary object in overloaded member function**. Here, we did not create any object inside the member function. We are just calling the constructor and returning incremented value to calling function

rSCC

Live Objects

Objects created dynamically with their data members initialized during creation are known as Live Objects. To create a live object, constructor must be invoked automatically which performs initialization of data members. Similarly the destructor for an object must be invoked automatically before the memory for that object is deallocated.

A class whose live object is to be crated must have atleast one constructor. The syntax for creating a live object is as follows.

Pointer_to_Object = new Class_name(Parameters)

Sample Program

```
#include <iostream.h>
#include <string.h>
class student
{
int rno;
char *name;
public:
studen(void)
```

```
{
char flag, str[50];
cout<<”Do u want to initialize the object y/n”;
cin>>flag;
if(flag == ‘y’)
{
cout<<”Enter the student number”;
cin>>rno;
cout<<”Enter the student name”;
cin>>name;
}


else
{
rno =0;
name = NULL;
}
}
student(int rn)
{
rno = rn;
name = NULL;
}
student(int rn, char *n)
{
rno = rn;
name = n;
}
~student()
{
if(name)
delete name;
}
void show(void)
{
if(rno)
cout<<”Roll number is”<<rno<<endl;
else
cout<<”Number not initialized”<<endl;
if(name)
cout<<”Student name is “<<name<<endl;
else
cout<<"Name not initialized"<<endl;
}
};
void main()
{
student *s1, *s2, *s3;
s1 = new student;
s2 = new student(1);
s3 = new student(1,"Magesh");
cout<<"Live objects contents…………"<<endl;
s1->show();
s2->show();
s3->show();
delete s1;
delete s2;
delete s3;
}
```

Array of Object

An array of objects, all of whose elements are of the same class, can be declared just as an array of any built-in type. Each element of the array is an object of that class. Being able to declare arrays of objects in this way underscores the fact that a class is similar to a type.

Declaring Arrays of Objects
The simplest way to create an array of Frame objects is with the following declaration:

```
  Frame windowList[5];          // an array of 5  Frame objects
```

An important aspect of declaring arrays of objects in this way is that all of the objects in the array must be constructed in the same way. It is not possible with this declaration to give each different object in the array a different set of constructor values. Furthermore, since no constructor arguments are given, the class must contain a constructor that has no arguments. Arrays of this form are useful when all of the objects should be constructed in a uniform way or when the "real" constructor information will not be know until sometime during the computation. In the later case, the array can be declared and the individual objects manipulated when the information is discovered. For example, the user may be asked to supply the name of a file which contains the desired locations and shapes for each of the windowList objects. This information can be read and each array element can then be moved and resized accordingly.

In other cases, it is desired that each of the objects in an array be specifically and individually constructed at the time the array is declared. This can be done as follows:

CPPH

```
 Frame windowList[5] = {Frame("Window 0",    0, 100, 100, 100),
                        Frame("Window 1",   25, 100, 100, 100),
                        Frame("Window 2",   50, 100, 100, 100),
                        Frame("Window 3",   75, 100, 100, 100),
                        Frame("Window 4",  100, 100, 100, 100)
                       };
```

Each object in the array is constructed using explicitly specified values for each constructor argument. This allows the programmer complete control over the initialization of the objects in the array.

It is not necessary to specifiy all of the constructor arguments if there are overloaded constructors, as there are for the Frame class. An object in an array can be constructed using any of the constructors. For example, if it was only desired to specify the name and initial location, but not the shape, for each object in the array then the following declaration would suffice:

```
 Frame windowList[5] = {Frame("Window 0",    0, 100),
                        Frame("Window 1",   25, 100),
                        Frame("Window 2",   50, 100),
                        Frame("Window 3",   75, 100),
                        Frame("Window 4",  100, 100)
                       };
```

In this case the overload constructor will determine the shape of each object. It is also possible to use different constructors for each objects as shown here:

```
 Frame windowList[5] = {Frame("Window 0",    0, 100, 100, 100),

                        Frame("Window 1",   25, 100),
                        Frame("Window 2"),
                        Frame(),
                        Frame("Window 4",  100, 100, 100, 100)
                       };
```

In this version, the first and last objects in the array are constructed by explicitly providing each constructor argument. The constructor for the object named "Window 1" specifies only the location. The constructor for the object named "Window2" specifies only the name. The constructor for the object as subscript position 3 specifies no constructor arguments, allowing all defaults to apply, including the name.

Manipulating Objects in an Array

An object in an array can be manipulated by a combination of the subscripting operator "[]" - to select which object of the array is to be manipulated - and the "." (dot) operator - to apply the operation to the selected object. For example:

```
   windowList[3].MoveTo(100, 50);
```

CPPHSCC

moves the object with subscript 3 to a new position. Remember that the subscripts begin with 0.

One of the advantages of working with arrays of objects is that it is easy to program the same operation over all of the objects. For example, a single loop can shrink all of windows by 10% as follows:

```
  for (int i = 0; i++; i<5)
            windowList[i].resize(0.9);
```

More complex operations involving the elements in the array are also possible. For example the following loop positions the windows along a diagonal from upper left toward lower right, makes them all of the size:

```
   for (int i = 0; i++; i<5) {
      windowList[i].MoveTo(10*i+1, 10*i+1);
      windowList[i].Resize(50, 50);
   }
```

CPPFHSCC