UNIT-II Standard Input/Output

Syllabus

- **Concept of streams**,
  - hierarchy of console stream classes,
  - input/output using overloaded operators >> and <<
  - member functions of i/o stream classes,
- **Formatting output,**
  - formatting using ios class functions and flags,
  - formatting using manipulators.
- **Classes and Objects:**
  - Specifying a class,
  - Creating class objects,
  - Accessing class members,
  - Access specifiers,
  - Scope resolution operator
  - Defining member functions
  - Nesting of member function
  - Private member functions
  - Arrays within the class
  - Memory allocation for objects
  - Static data members,
  - Static member function,
  - Arrays of objects
  - Objects as function arguments
  - Returning an object from the function
  - friends of a class,
  - use of *const* keyword,
  - empty classes,
  - nested classes,
  - local classes,
  - container classes,
  - bit fields and classes**.**

Concept of streams

C++ I/O system are based on streams. Streams are sequence of bytes flowing in and out of the programs. In input operations, data bytes flow from an input source (such as keyboard, file, network or another program) into the program. In output operations, data bytes flow from the program to an output sink (such as console, file, network or another program). Streams acts as an intermediaries between the programs and the actual IO devices, in such the way that frees the programmers from handling the actual devices, so as to archive device independent IO operations.

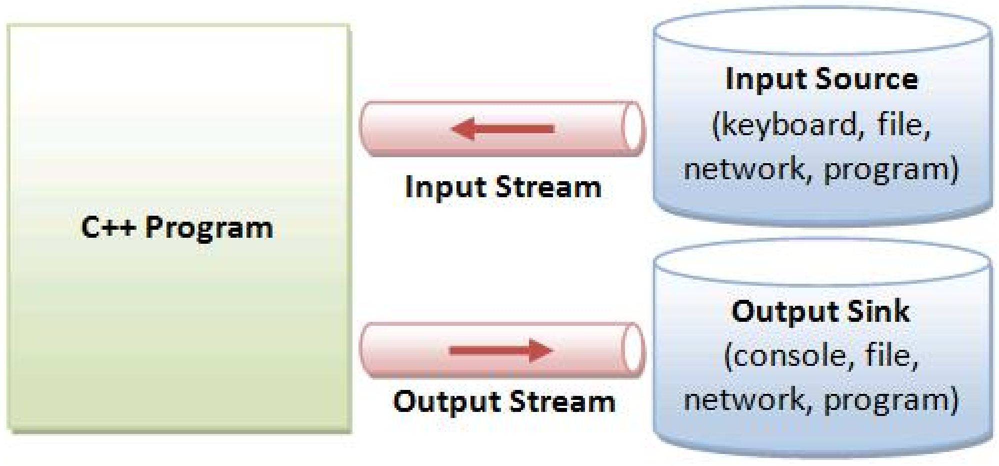


Figure: 2.1: C++ Stream

Hierarchy of console stream classes

Figure 2.2 shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file iostream. This file should be included in all the programs that communicate with the console unit.

As seen in the figure 2.2, ios is the base class for istream (input stream) and ostream (output stream) which are, in turn, base classes for iostream(input/output stream). the class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted an unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input while the ostream provides the facilities for formatted output. The class iostream provides the facilities for handling both input and output streams. Three classes, namely, istream_withassign, ostream_withassign, and iostream_withassign add assignment operators to these classes. Table 2.1 gives the details of these classes.

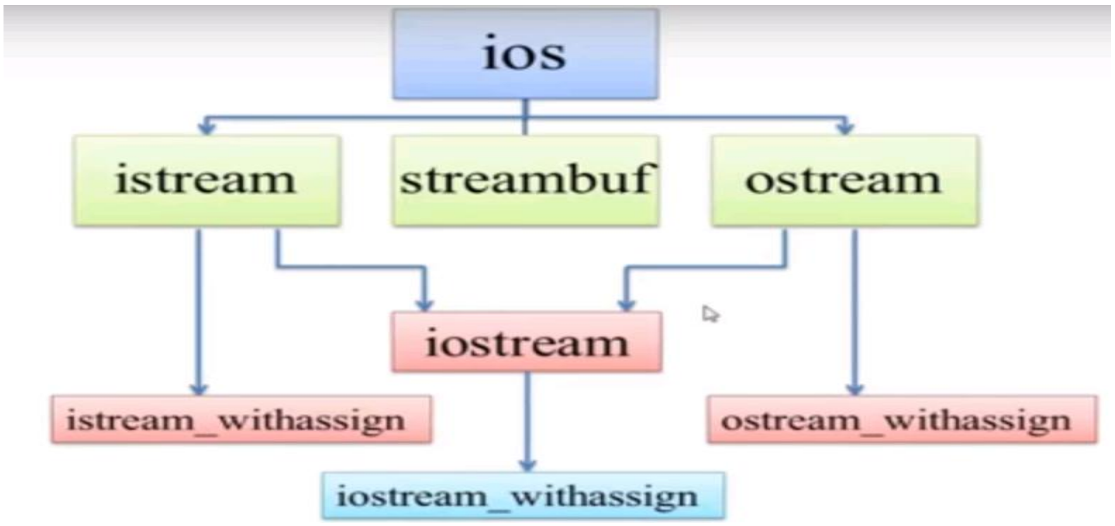
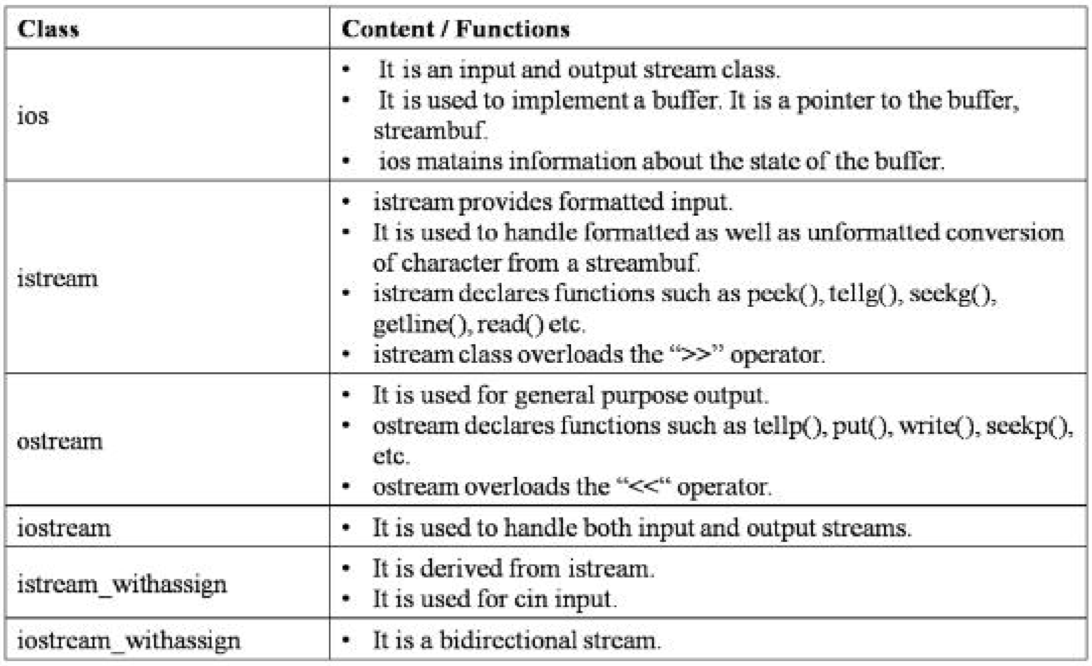


Figure: 2.2: Stream classes for console I/O operations

Table 2.1: Stream classes for console operations



Class	Content / Functions
ios	• It is an input and output stream class. • It is used to implement a buffer. It is a pointer to the buffer, streambuf. • ios matains information about the state of the buffer.
istream	• istream provides formatted input. • It is used to handle formatted as well as unformatted conversion of character from a streambuf. • istream declares functions such as peek(), tellg(), seekg(), getline(), read() etc. • istream class overloads the ">>" operator.
ostream	• It is used for general purpose output. • ostream declares functions such as tellp(), put(), write(), seekp(), etc. • ostream overloads the "<<" operator.
iostream	• It is used to handle both input and output streams.
istream_withassign	• It is derived from istream. • It is used for cin input.
iostream_withassign	• It is a bidirectional stream.

<u>Input/output using overloaded operators >> and <<</u>

Cin and cout are used for the input and output of data of various types. This has been made possible by overloading the operators >>(extraction) and <<(insertion) to recognize all the basic c++ types. The >> operator is overloaded in the istream class and << is overloaded in the ostream class. The following is the general format for reading data from the keyboard:

cin>> variable1>> variable2 >> ……… >> variableN

Variable1, variable2,… are valid C++ variable names that have been declared already. This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be:

data1, data2…….dataN

The input data are separated by white spaces and should match the type of variable in the cin list. Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example, consider the following code:

```
int code;
cin>>code;
```

Suppose the following data is given as input: 4258D

The operator will read the characters upto 8 and the value is assigned to code. The character D remains in the input stream and will be input to the next cin statement. The general form for displaying data on the screen is:

```
cout<<item1<<item2<<……<<itemN
```

The items item1 through itemN may be variables or constants of any basic type.

Member functions of i/o stream classes

get() and put() functions:

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations.

get(): There are two types of get() functions. Both prototypes are use to fetch a character including the blank space, tab and the newline character

get(char *): this assigns the input character to its argument.
Ex:

```
char c;
cin.get(c );
while(c!='\n')
{
cout<<c;
cin.get(c );
}
```

get(void): this returns the input character.

Ex:

```
char c;
c=cin.get( );
cout<<c;
```

The value returned by the function get() is assigned to the variable c.

put(): This function is a member of ostream class, can be used to output a line of text, character by character.

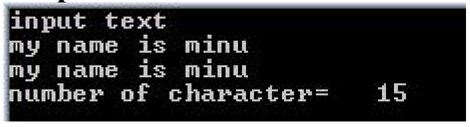
Ex:

```
char c;
cin.get(c);
while(c!='\n')
{
cout.put(c);
cin.get(c);
}
```

Program 1: Write a program to read the text and count the total numbers of characters in it.

```
#include<iostream>
using namespace std;
int main()
{
int count=0;
char c;
cout<<"input text\n";
cin.get( c);
while(c!='\n')
{
cout.put(c);
count++;
cin.get(c);
}
cout<<endl;
cout<<"number of character=      "<< count<<"\n";
return 0;
}
```

Output:



```
input text
my name is minu
my name is minu
number of character=   15
```

getline() and write() functions:

We can read and display a line of text more efficiently using the line-oriented input/output functions getline(0 and write().

getline(): this function reads a whole line of text that ends with a new line character. This function can be invoked by using the object cin as follows:

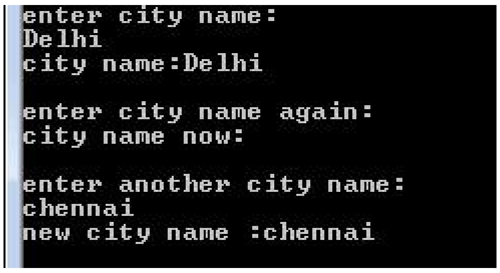
cin.getline(line, size);

This function call invokes the function getline() which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read. The newline character is read but not saved. Instead, it is replaced by the null character. For example, consider the following code:
char name[20];
cin.getline(name, 20);

Program 2: character I/O with get() and put()

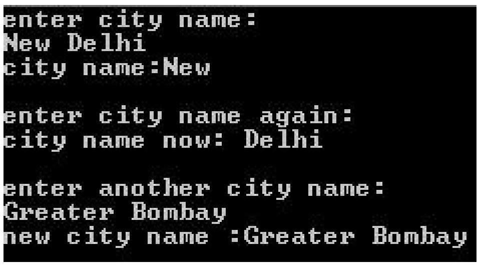
```
#include<iostream>
using namespace std;
int main()
{
int size=20;
char city[20];
cout<<"enter city name:\n";
cin>>city;
cout<<"city name:" <<city <<"\n\n";
cout<<"enter city name again:\n";
cin.getline(city,size);
cout<<"city name now:" <<city <<"\n\n";
cout<<"enter another city name:\n";
cin.getline(city,size);
cout<<"new city name :" <<city <<"\n\n";
return 0;
}
```

Output:
First run:



```
enter city name:
Delhi
city name:Delhi

enter city name again:
city name now:

enter another city name:
chennai
new city name :chennai
```

2nd Run:



```
enter city name:
New Delhi
city name:New

enter city name again:
city name now: Delhi

enter another city name:
Greater Bombay
new city name :Greater Bombay
```

During first run, the newline character '\n' at the end of 'Delhi' which is waiting in the input queue is read by getline() that follows immediately and therefore it does not wait for any response to the prompt 'enter city name again'. The character'\n' is read as an empty line.

During the second run, the word 'Delhi' is read by the function getline() and, therefore, here again it does not wait for any input to the prompt 'enter city name again:'. Note that the line of text "Greater Bombay" is correctly read by the second cin.getline(city,size); statement.

write(): Function displays an entire line and has the following form:

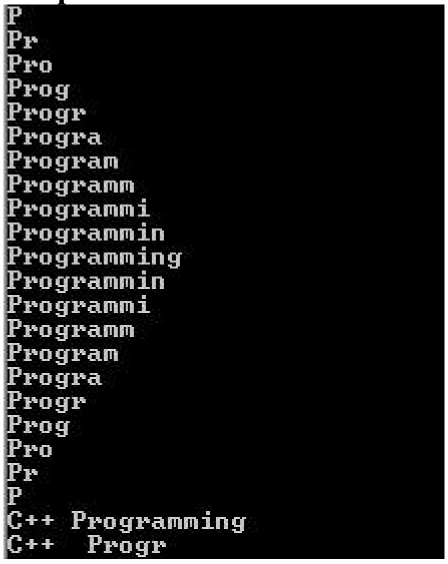
cout.write(line, size)

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display. Note that it does not stop displaying the characters automatically when the null character is encountered. If the size is greater than the length of line, then it displays beyond the bounds of line.

Program 3: Displaying Strings with write().

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

using namespace std;
int main()
{
char * string1="C++ ";
char * string2="Programming";
int m=strlen(string1);
int n=strlen(string2);
for(int i=1; i<n; i++)
{
cout.write(string2, i);
cout<<"\n";
}
for(int j=n; j>0; j--)
{
cout.write(string2, j);
cout<<"\n";
}
cout.write(string1,m).write(string2,n);
cout<<"\n";
cout.write(string1, 10);
return 0;
}
```

Output:



```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
C++ Programming
C++  Progr
```

It is possible to concatenate two strings using the write() function. The statement
cout.write(string1,m).write(string2,n);

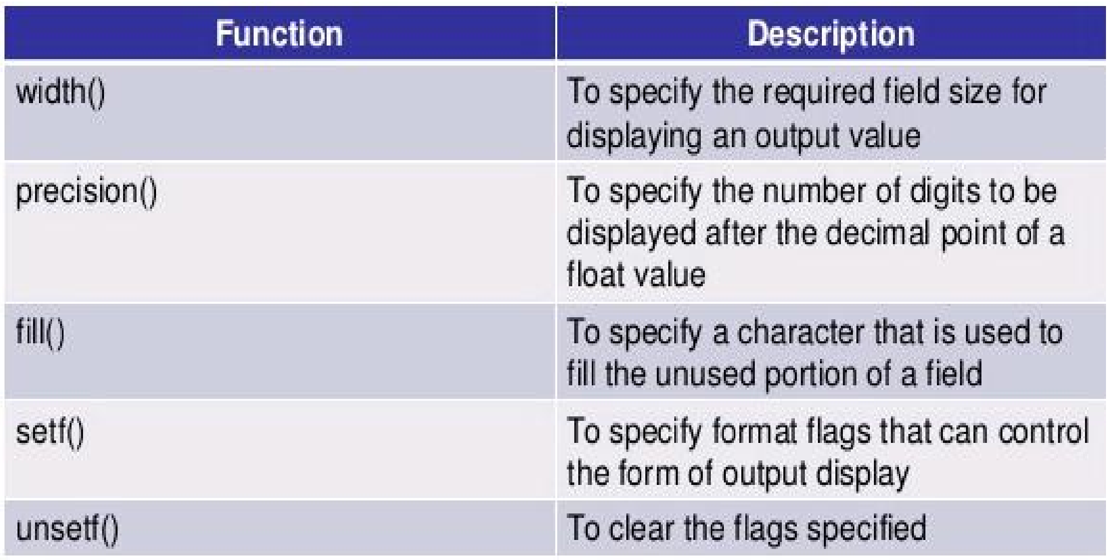
is equivalent to the following two statements:

```
cout.write(string1,m);
cout.write(string2,n);
```

<u>Formatting using ios class functions and flags</u>

The ios class contains a large number of member functions that would help us to format the output in a number of ways. The most important ones among them are listed in Table 2.2.

Table 2.2: ios format functions



Function	Description
width()	To specify the required field size for displaying an output value
precision()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display
unsetf()	To clear the flags specified

(i) <u>Defining Field Width: width()</u>

It is used to define the width of a field necessary for the output of an item. Since it is member function, we have to use an object to invoke it, as shown below:

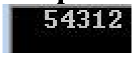
```
cout.width(w);
```

w> is the field width (number of columns).
The output will be printed in a field of w characters wide at the right end of the field. The width() function can specify the field width for only one item. After printing one item it will revert back to the default.

Example:

```
cout.width(5);
cout<<543<<12<<"\n";
```

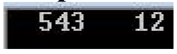
Output:



```
  54312
```

The value 543 is printed right justified in the first five columns. The specification width(5) does not retain the setting for printing the number 12. this can be improved as follows:

Output:



```
  543   12
```

Remember that the field width should be specified for each item separately. C++ never truncates the values and therefore, if the specified field width is smaller than the size of the value to be printed, c++ expands the field to fit the value.

(ii) Setting Precision: precision()

By default, the floating numbers are printed with six digits after the decimal point. However we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done using the precision() member function as follows:

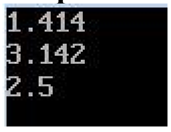
cout.precision(d);

Where d--> is the number of digits to the right of the decimal point.

Example:

```
cout.precision(3);
cout<<sqrt(2)<<"\n";
cout<<3.14159<<"\n";
cout<<2.50032<<"\n";
```

Output:



```
1.414
3.142
2.5
```

1.414 (truncated)
3.142 (rounded to the nearest cent)
2.5 (no trailing zeros)

Unlike the function width(), precision() retains the setting in effect until it is reset. That is why we have declared only one statement for the precision setting which is used by all the three outputs.

We can set different values to different precision as follows:

```
cout.precision(3);
cout<<sqrt(2)<<"\n";
cout.precision(5);              //reset the precision
cout<<3.14159<<"\n";
```

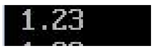
We can also combine the field specification with the precision setting.

Example:

```
cout.precision(2);
cout.width(5);
cout<<1.2345<<"\n";
```

The first two statements instruct: "print two digits after the decimal point in a field of five character width".

Output:



```
1.23
```

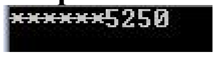
(iii) Filling and Padding: fill()

We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default. However, we can use the fill() function to fill the unused positions by any desired character. It is used in the following form:

cout.fill(ch); Where ch represents the character which is used for filling the unused positions.

Example:
cout.fill('*');
cout.width(10);
cout<<5250<<"\n";

Output:



```
******5250
```

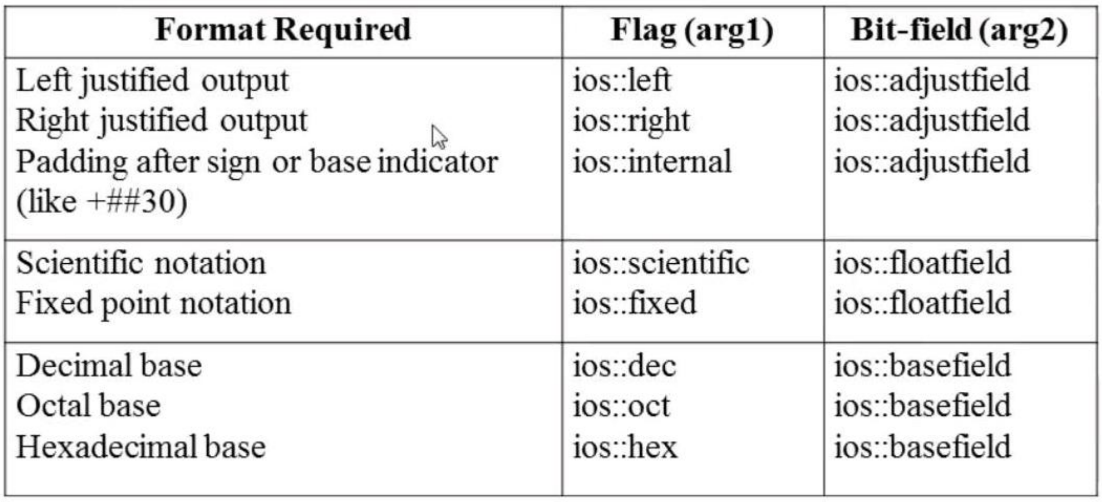
(iv) Formatting flags, Bit-field and setf()

We have seen that when the function width() is used, the value is printed right-justified in the field width created. But it is usual practice to print the text left-justified.

setf() (setf stands for set flags) function is used to print the value left-justified. It is used in the following form:
cout.setf(arg1, arg2)

arg1--> it is one of the formatting flags defined in the class ios.
Arg2--> known as bit field specifies the group to which the formatting flag belongs.

Table 2.3: Flags and bit fields for setf() function



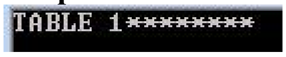
Format Required	Flag (arg1)	Bit-field (arg2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	ios::adjustfield
Padding after sign or base indicator (like +##30)	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

Table 2.3 shows the bit field, flags and their format actions. There are three bit fields and each has a group of format flags which are mutually exclusive.

Example:

```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(15);
cout<<"TABLE 1"<<"\n";
```

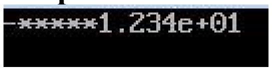
Output:



```
TABLE 1********
```

```
cout.fill('*');
cout.precision(3);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout.width(15);
cout<<-12.34567<<"\n";
```

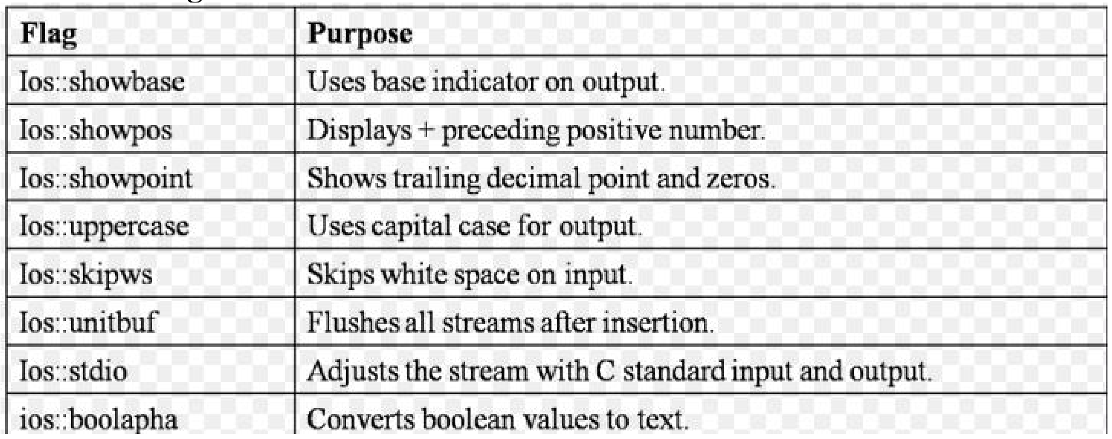
Output:



```
-*****1.234e+01
```

Some flags do not have any bit fields and therefore are used as single arguments in setf(). table 10.4 lists the flags that do not possess a named bit field. These flags are not mutually exclusive and therefore can be set or cleared independently.

Table 2.4: Flags that do not have bit fields

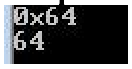


Flag	Purpose
Ios::showbase	Uses base indicator on output.
Ios::showpos	Displays + preceding positive number.
Ios::showpoint	Shows trailing decimal point and zeros.
Ios::uppercase	Uses capital case for output.
Ios::skipws	Skips white space on input.
Ios::unitbuf	Flushes all streams after insertion.
Ios::stdio	Adjusts the stream with C standard input and output.
ios::boolapha	Converts boolean values to text.

Example:

```
    cout.setf ( :ios::hex, std::ios::basefield );
    cout.setf ( :ios::showbase );
    cout << 100 << '\n';
    cout.unsetf ( ios::showbase );
    cout << 100 << '\n';
```

Output:



```
0x64
64
```

Formatting using manipulators

Manipulators are special functions that can be included in the I/O statements to manipulate the output format. Table 2.5 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

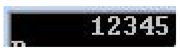
Table 2.5: Manipulators and their meanings

Manipulators	**Meaning**	**Equivalent ios function**
setw(int w)	Set the field width to w.	width()
setprecision(ind d)	Set the floating point precision to d.	precision()
setfill(int c)	Set the fill character to c.	fill()
setiosflags(long f)	Set the format flag f.	setf()
resetiosflags(long f)	Clear the flag specified by f.	unsetf()
endl	Insert new line and flush stream.	

Examples:

```
#include <iostream>
#include<iomanip>
using namespace std;
int main () {
cout<<setw(10)<<12345;
return 0;
}
```

Output:

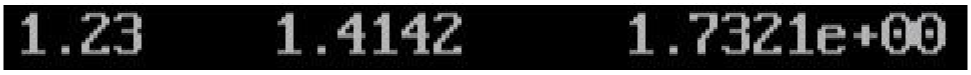


```
     12345
```

One statement can be used to format output for two or more values. For example:

```
#include <iostream>
#include<iomanip>
#include<math.h>
using namespace std;
int main () {
cout<<setw(5)<<setprecision(2)<<1.2345;
cout<<setw(10)<<setprecision(4)<<sqrt(2);
cout<<setw(15)<<setiosflags(ios::scientific)<<sqrt(3);
return 0;
}
```

Output:



```
 1.23    1.4142     1.7321e+00
```

We can jointly use the manipulators and the ios functions in a program.

There is a major difference in the way the manipulators are implemented as compared to the ios member functions. The ios member function return the previous format state which can be used later, is necessary. But the manipulator does not return the previous format state. In case, we need to save the old format states, we must use the ios member functions rather than the manipulators.

Specifying a class

Generally, a class specification has two parts:

- Class declaration
- Class function definition

The declaration specifies the type and scope of both data and member functions of class. Where as definition specifies the executable code of the function.

The following are the characteristics of a class

- The keyword class specifies abstract data type of type class name.
- The body of a class is enclosed with in braces and terminated by a semicolon.
- The functions and variables with in the class are collectively called as members
- The members that have been declared as private can be accessed only from with in the class.
- Class definition is only a template and does not create any memory space for a class.
- By default all the members are of type private .

Example:

```
class item
{
      int number;      //variable declaration
      float cost;       //private by default
  public:
      void getdata(int a, float b); //functions declaration
       void putdata(void);
};
```

Creating class objects

Once a class has been declared, we can create variables of that type by using the class name. For example:
item x;
creates a variable x of type item. In C++, the class variables are known as objects. Therefore, x is called an object of item. We may also declare more than one object in one statement. Example:

item x,y,z;

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. For example:

```
class item
{
………
………
} x,y,z;
```

Would create the objects x,y and z of type item. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

Accessing class members

The private data of a class can be accessed only through the member functions of that class. The following is the format for calling a member function:

object_name.member_function(actual_parameters);

The following is an example of creating an object of type 'item' and invoking its member function.

```
    void main ( )
        {    item x; // creating an object of type item
             x. getdata (20,20.5);
             x.put data ( );
        }
```

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member function. For example:
x.putdata();

Sends a message to the object x requesting it to display its contents. A variable declared as public can be accessed by the objects directly. Examples:

```
class xyz
{
int x;
int y;
public:
int z;
};
….
….
xyz p;
p.x=0;     //error, x is private
p.z=10;    //ok, z is public
…...
…...
```

Access specifiers

There are 3 access specifiers for a class in C++. These access specifiers define how the members of the class can be accessed. Of course, any member of a class is accessible within that class. There are three types of access specifiers, they are:

i. Public
ii. Private
iii. Protected

Public: Any member declared under this specifier is accessible from outside the class, by using the object of the class. The public members of a class get inherited completely to the child classes.

Syntax:

```
class definition
{
  public :
declarations or definitions
} ;
```

Example:

```
#include<iostream>
using namespace std;
class A
{
public:
      int a;
};
int main()
{
      A ob1;
      ob1.a=10;
      cout<<ob1.a<<"\n";
}
```

Private: Any member declared under this specifier is Not Accessible from Outside the Class. The Private members are accessible to only the Member Functions and Friend Functions in the class. The Private members of a Class Never get Inherited to the Child Classes.

Syntax:

```
class definition
{
     private :
declarations or definitions
} ;
```

Example:

```
#include<iostream>
using namespace std;
class A
{         private :
     int a;
     public :
     void fill ( )
     {     cin >> a ;
             cout<<a;
     }
} ;
main()
{     int c;
     A    oba ;

//    oba . a = 10 ; //    invalid as    ‘ a ‘ is private member.
     oba . fill ( ) ; //    valid as    fill ( )    is a public member.
}
```

Protected: Any member declared under this specifier is not accessible from outside the class, by using the object of the class. The protected members of a class get inherited completely to the child classes and they maintain their protected visibility in the child class. The protected members are accessible to only the member functions and friend functions in the class.

Syntax:

```
Class definition
{
  protected :
declarations or definitions
  } ;
```

Example:

```
#include<iostream>
using namespace std;
class A
{
  protected :
     int a;
public :
     void fill ( )
     {     cin >> a ;
             cout<<a;
        }
     } ;
int main()
{
      A    oba ;
//    oba . a = 10 ; //    invalid as    ‘ a ‘ is protected member.
```

```
    oba . fill ( ) ; //    valid as    fill ( )    is a public member.
}
```

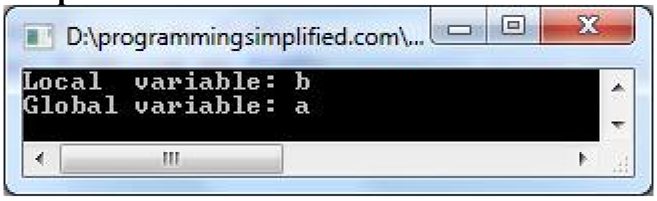
Scope resolution operator

Scope resolution operator (::) is used to define a function outside a class or when we want to use a global variable but also has a local variable with the same name.

Example:

```
#include <iostream>
using namespace std;
char c = 'a';      // global variable
int main() {
  char c = 'b';   //local variable
  cout << "Local   variable: " << c << "\n";
  cout << "Global variable: " << ::c << "\n";  //using scope resolution operator
  return 0;
}
```

Output:



```
Local  variable: b
Global variable: a
```

Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined.

Outside the Class Definition

To declare the member function, outside the class definition, the function prototype declared within the body of a class and defined them out side the body of a class. Membership function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which class the function belongs to.

The general form of a member function definition is:

```
 class class-name
 { access specifier:
   return data type function-name(arguments); //declaration
 };
 return data type class-name :: function-name(argument) // function definition
 {              function body;
 }
```

Example:

```
class a
{ public;
        void show(); // prototype
};

void a:: show()
{
 cout << "this is outside the class " ;
 }
```

Here the membership label class_name:: tells the compiler that the function is the member of the specified class. The scope of the function is restricted to the class-name specified in the header line.

.

Properties of a member function

- Several different classes can use the same function name; the membership label will resolve their scope.
- Member functions can access the private data of the class, but a non-member function cannot.
- A member function can call another member function directly, without using a dot operator.

<u>Inside the Class Definition</u>

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.

Example:

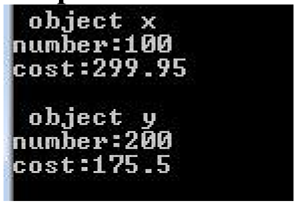
```
class a
{ public;
void show()
{
 cout << "this is inside the class " ;
 }
};
```

<u>Program 4</u>: Class implementation

```
#include<iostream>
using namespace std;
class item
{
```

```
int number;
float cost;
public:
void getdata(int a, float b);
void putdata(void)
{
cout<<"number:"<<number<<"\n";
cout<<"cost:"<<cost<<"\n";
}
};

void item :: getdata(int a, float b)
{
number=a;
cost=b;
}

int main()
{
item x;
cout<<"\n object x"<<"\n";
x.getdata(100, 299.95);
x.putdata();
item y;
cout<<"\n object y"<<"\n";
y.getdata(200, 175.50);
y.putdata();
return 0;
}
```

Output:



```
 object x
number:100
cost:299.95

 object y
number:200
cost:175.5
```

Nesting of member functions

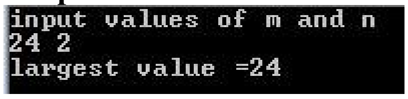
A member function can be called by using its name inside another member function of the same class.

Program 5: Program to read two numbers from keyboard and display the largest on the screen.

:

```
#include<iostream>
using namespace std;
class set
{
        int m, n;
public:
        void input(void);
        void display(void);
        int largest(void);
};

int set :: largest(void)
{
        if(m>=n)
                return(m);
        else
                return(n);
}

    void set :: input(void)
{
        cout<<"input values of m and n"<<"\n";
        cin>>m>>n;
        }
          void set :: display(void)
        {
        cout<<"largest value ="<<largest()<<"\n";
        cin>>m>>n;
        }

        int main()
        {
                set A;
                A.input();
                A.display();
                return 0;
        }
```

Output:



```
input values of m and n
24 2
largest value =24
```

Private member functions

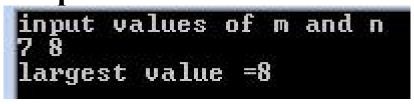
Although it is normal practice to place all the items in a private section and all the functions in public, some situations may require certain functions to be hidden from the outside the calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and there fore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator.

Program 6: Program to read two numbers from keyboard and display the largest on the screen.

```
#include<iostream>
using namespace std;
class set
{
        int m, n;
        void input(void);
public:

        void display(void);
        int largest(void);
};

int set :: largest(void)
{
        if(m>=n)
                return(m);
        else
                return(n);
}

  void set :: input(void)
        {
        cout<<"input values of m and n"<<"\n";
        cin>>m>>n;
        }

    void set :: display(void)
      {
        input();
        cout<<"largest value ="<<largest()<<"\n";
        }

        int main()
        {
                set A;
                A.display();
                return 0;
        }
```

Output:



```
input values of m and n
7 8
largest value =8
```

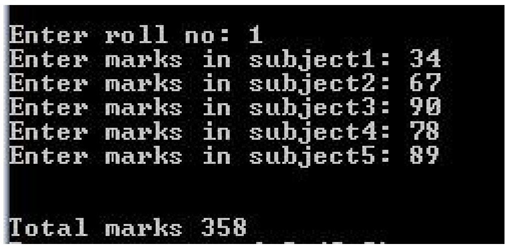
Arrays within the class

Arrays can be used as member variables in a class. The arrays can be declared as private, public or protected members of the class. To understand the concept of arrays as members of a class, consider this example.

Program 7: Program to read a student five subject marks and display the total marks obtained.

```
#include<iostream>
using namespace std;
const int size=5;
class student
{
int roll_no;
int marks[size];
public:
void getdata ();
void tot_marks ();
} ;
void student :: getdata ()
{
cout<<"\nEnter roll no: ";
cin>>roll_no;
for(int i=0; i<size; i++)
{
cout<<"Enter marks in subject"<<(i+1)<<": ";
cin>>marks[i] ;
}
}
void student :: tot_marks() //calculating total marks
{
int total=0;
for(int i=0; i<size; i++)
total = total + marks[i];
cout<<"\n\nTotal marks "<<total;
}
int main()
{
student stu;
stu.getdata() ;
stu.tot_marks() ;
}
```

Output:



```
Enter roll no: 1
Enter marks in subject1: 34
Enter marks in subject2: 67
Enter marks in subject3: 90
Enter marks in subject4: 78
Enter marks in subject5: 89

Total marks 358
```

Memory Allocation for Objects

We have stated that the memory space for subjects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects.

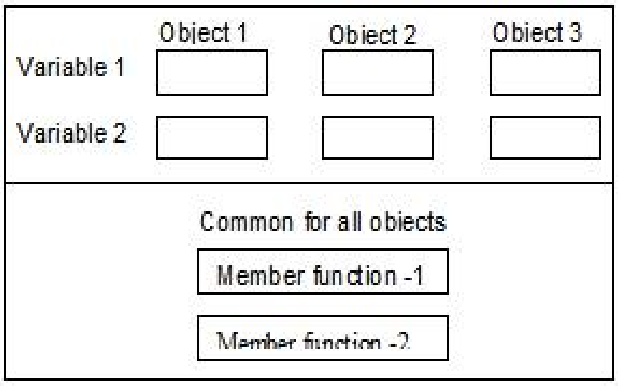


Figure: 2.3:Memory Allocation of Objects

Static Data Members

A data member of a class can be qualified a static. A static member variable has certain special characteristics. These are:

i. It is initialized to zero when the first object of its class is created.no other initialization is permitted.
ii. Only a single copy of the Static data members are shared between Objects of the Class.
iii. It is visible only within the class, but its lifetime is the entire program.

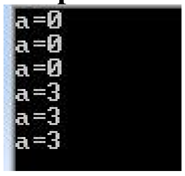
Static variables are normally used to maintain values common to the entire class. For example, a static data member cab be used as a counter that records the occurrences of all the objects.

Syntax:
static data-type variable-name ;

Program 8: Program to illustrate the use of a static data member.

```
#include <iostream>
using namespace std;
class sample
{
    static int a;              // declaration
    public:
            void incr( )
            {
              a=a+1;
            }
            void putdata ( )
            {
              cout <<"a="<<a<<endl;
            }
};
int sample::a;                  //definition of static data member
int main ( )
{
sample x,y,z;                  // a is initialized to zero
x.putdata( );                  //display a
y.putdata( );
z.putdata( );
x. incr ();              //getting data into object x
y. incr ();              //getting data into object y
z. incr ();              //getting data into object z
x.putdata( );              //display a
y.putdata( );
z.putdata( );
return 0;
}
```

Output:



```
a=0
a=0
a=0
a=3
a=3
a=3
```

Note: type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object.

Static Member Function

Like static member variables we can also have static member functions. A member function that is declared as static has the following properties.

- A static member function can have access to only other static members(functions or variable) declared in the same class.
- A static member function can be called using the class name a follows.

class_name : : function_name ( )

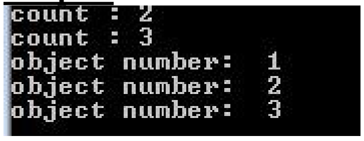
The differences between a static member function and non-static member functions are as follows.

i. A static member function can access only static member(data and member functions) data and functions outside the class. A non-static member function can access all of the above including the static data member.
ii. A static member function can be called, even when a class is not instantiated, a non-static member function can be called only after instantiating the class as an object.
iii. A static member function cannot be declared virtual, whereas a non-static member functions can be declared as virtual
iv. A static member function cannot have access to the 'this' pointer of the class.
v. A static or non static member function cannot have the same name.

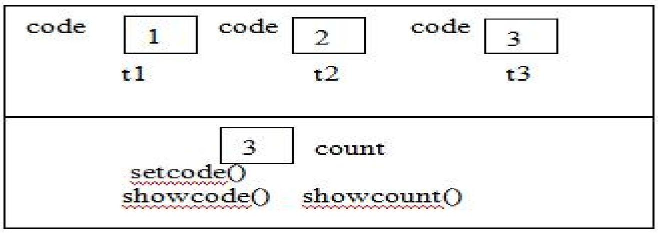
Program 9: Program to illustrate the use of a static member function.

```
#include<iostream>
using namespace std;
class test
{
   int code;
   static int count;
   public:
void setcode(void )
{
code = ++count;
}
void showcode ( void)
{
cout<<"object number:    "<< code<<endl ;
  }
static void showcount ( )         // static member function.
{
cout <<"count : " <<count<<" \n";
// cout<<"code ="<<code ;         //error , code is a non static member function.
}
};
int test :: count;
int main ( )
{
test t1,t2,t3;
t1. setcode ();
t2. setcode ();
test :: showcount ( );    //accessing static function
t3. setcode ( );
test:: showcount ( );
t1. showcode ( );
t2. showcode ( );
t3. showcode ( );
return 0;
}
```

Output:



```
count : 2
count : 3
object number:  1
object number:  2
object number:  3
```



Arrays of Objects

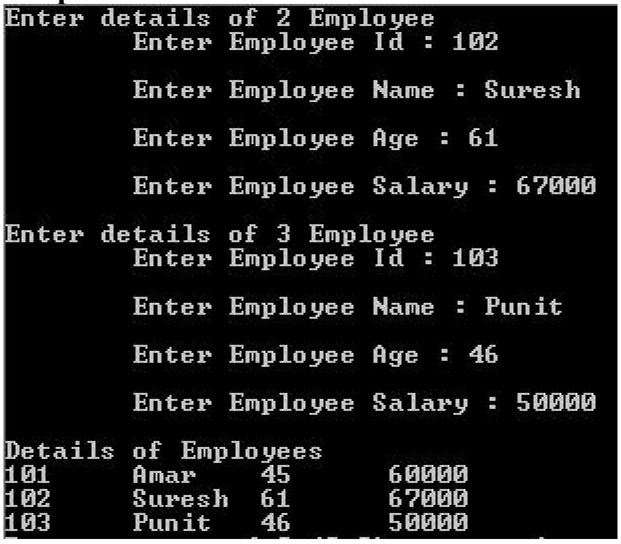
We can also have arrays of variables that are of the type class. Such variables are called arrays of objects.

Program 10: Program to read employee personal details and display.

```
#include<iostream>
  using namespace std;
        class Employee
        {
                int Id;
                char Name[25];
                int Age;
                long Salary;
                public:
                void GetData()                  //Defining GetData()
                {
                        cout<<"\n\tEnter Employee Id : ";
                        cin>>Id;
                        cout<<"\n\tEnter Employee Name : ";
                        cin>>Name;
                        cout<<"\n\tEnter Employee Age : ";
                        cin>>Age;
                        cout<<"\n\tEnter Employee Salary : ";
                        cin>>Salary;
                }
                void PutData()                  //Defining PutData()
                {
                        cout<<"\n"<<Id<<"\t"<<Name<<"\t"<<Age<<"\t"<<Salary;
                }
        };
int main()
```

```
{
        int i;
        Employee E[3];      //Creating Array of 3 Employees
        for(i=0;i<3;i++)
        {
                cout<<"\nEnter details of "<<i+1<<" Employee";
                E[i].GetData();
        }
        cout<<"\nDetails of Employees";
        for(i=0;i<3;i++)
        E[i].PutData();
}
```

Output :



```
Enter details of 2 Employee
        Enter Employee Id : 102

        Enter Employee Name : Suresh

        Enter Employee Age : 61

        Enter Employee Salary : 67000

Enter details of 3 Employee
        Enter Employee Id : 103

        Enter Employee Name : Punit

        Enter Employee Age : 46

        Enter Employee Salary : 50000

Details of Employees
101     Amar    45      60000
102     Suresh  61      67000
103     Punit   46      50000
```

Objects as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- Pass by value
- Pass by reference.

When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object.

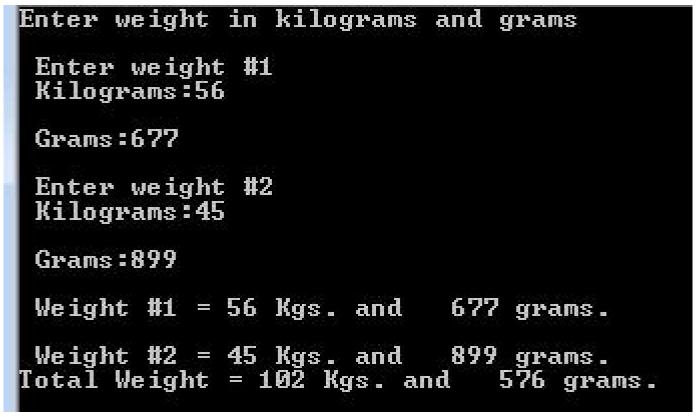
On the other hand, in pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

Program 11: A program to demonstrate passing objects by value to a member function of the same class

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

```
class weight
{
int kilogram;
int gram;
public:
void getdata ();
void putdata ();
void sum_weight (weight,weight) ;
} ;

void weight :: getdata()
{
cout<<"\n Kilograms:";
cin>>kilogram;
cout<<"\n Grams:";
cin>>gram;
}

void weight :: putdata ()
{
cout<<kilogram<<" Kgs. and      "<<gram<<" grams.\n";
}

void weight :: sum_weight(weight wl,weight w2)
{
gram = wl.gram + w2.gram;
kilogram=gram/1000;
gram=gram%1000;
kilogram+=wl.kilogram+w2.kilogram;
}

int main ()
{
weight wl,w2 ,w3;
cout<<"Enter weight in kilograms and grams"<<"\n";
cout<<"\n Enter weight #1" ;
wl.getdata();
cout<<" \n Enter weight #2" ;
w2.getdata();
w3.sum_weight(wl,w2);
cout<<"\n Weight #1 = ";
wl.putdata();
cout<<"\n Weight #2 = ";
w2.putdata();
cout<<"Total Weight = ";
w3.putdata();
return 0;
}
```

Output:



```
Enter weight in kilograms and grams

 Enter weight #1
 Kilograms:56

 Grams:677

 Enter weight #2
 Kilograms:45

 Grams:899

 Weight #1 = 56 Kgs. and   677 grams.

 Weight #2 = 45 Kgs. and   899 grams.
Total Weight = 102 Kgs. and   576 grams.
```

Returning an object from the function

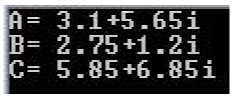
A function cannot only receive objects as arguments but also can return them.

Program 12: Program to add two complex number.

```
#include <iostream>
using namespace std;
class complex                    //x+iy form
{
        private:
              float x;        //real part
              float y;      //imaginary part
        public:
                void input(float real, float imag)
                {
                      x=real;
                      y=imag;
                }

        complex sum(complex c1, complex c2)
                {
                complex c3 ;
                c3.x = c1.x+c2.x;
                c3.y = c1.y+c2.y;
                return c3;
                }

          void show(complex c)
                {
                        cout    << c.x << "+" << c.y << "i"<<"\n";
                }
};
int main()
{
        complex A, B,C,D;
        A.input(3.1,5.65);
```

```
    B.input(2.75,1.2);
    C=D.sum(A,B);
    cout<<"A= ";C.show(A);
    cout<<"B= ";C.show(B);
    cout<<"C= ";C.show(C);
    return 0;
}
```

Output:



```
A= 3.1+5.65i
B= 2.75+1.2i
C= 5.85+6.85i
```

FRIEND FUNCTION

We have learned that private members cannot accessed from outside the class. That is, a non member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to be use an object name and dot membership operator with each member name.
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

- A friend function can be:
  a) A global function
  b) A member function of another class

A global function

To make an outside function “friendly” to a class, we have to simply declare this function as a friend of the class as shown below:

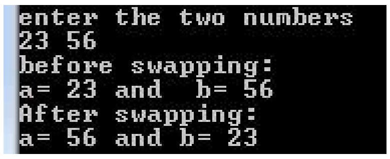
Syntax:
```
 class ABC
{       ………
        ………..
    public:
    …………
    ….……….
    friend void xyz(void);
};
```

The function declaration should be preceded by the keyword friend. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword friend or the scope operator. The function that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

Program 13: Program for swapping of two numbers using call by value.

```
#include<iostream>
using namespace std;
class sample
{
int a,b;
public:
        void get()
        {
        cout<<"enter the two numbers"<<endl;
          cin>>a>>b;
        }

void show()
        {
        cout<<"a= "<< a <<" and    " <<"b= "<<    b<<endl;
        }
        friend void swap(sample s);
  };

void swap( sample    s1)
{
int temp=s1.a;
s1.a =s1.b;
s1.b = temp;
cout<<"a= "<<s1.a    << " and " <<"b= "<<    s1.b<<endl;
}

int main()
{
      sample s2;
      s2.get();
      cout<<"before swapping:"<<endl;
      s2.show();
      cout<<"After swapping:"<<endl;
      swap(s2);       //function call swap(s2) passes the object s2 by value to the friend function.
    return 0;
}
```

Output:



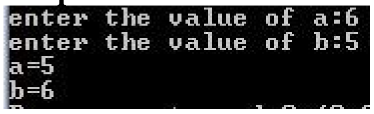
```
enter the two numbers
23 56
before swapping:
a= 23 and  b= 56
After swapping:
a= 56 and b= 23
```

Note: Actual swapping can be done by reference or address, we can make friend void swap(sample &); or swap(sample *);

Program 14: Program for swapping the private data members of two classes using friend function.

```
#include<iostream>
using namespace std;
class second;    // Forward declaration to satisfy the compiler about class B
class first
{       int a ;

        void get()
        {
           cout<<"enter the value of a:";
           cin>>a;
        }
friend   void   swap ( first &,second &) ;
};

class second
{
       int b ;
       void get()
       {
          cout<<"enter the value of b:";
          cin>>b;
       }
friend   void   swap ( first &,second &);
};

void swap ( first &ob1,second &ob2 )
{   ob1.get();
    ob2.get();
    int temp;
    temp = ob1.a ;
    ob1.a = ob2.b ;
    ob2.b = temp ;
    cout<<"a="<<ob1.a<<"\nb="<<ob2.b;
}
int main()
{
  first ob1;
  second ob2;
  swap(ob1,ob2) ;
  return 0;
}
```

Output:



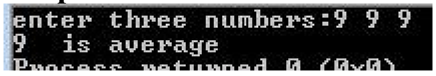
```
enter the value of a:6
enter the value of b:5
a=5
b=6
```

Program 15: Create a class X with integer data member a, class Y with integer data member b and class Z with integer data member c. Write a c++ program to find average of a, b and c using friend function.

```
#include<iostream>
using namespace std;
class Y;
class Z;
class X
{
int a;

public: void setvalve(int x)
{
a=x;
}

friend void avg(X n, Y m, Z p);
};

class Y
{
int b;

public:
void setvalve(int x)
{
b=x;
}
friend void avg(X n, Y m, Z p);
};

class Z
{
int c;

public:
void setvalve(int x)
{
c=x;
}
friend void avg(X n, Y m, Z p);
};
```

```
void avg(X n, Y m, Z p)
{
float avg;
avg= (n.a+m.b+p.c)/3;
cout<<avg<< "    is average";
}

int main()
{

X a1;
Y a2;
Z    a3;
int a,b,c;
cout<<"enter three numbers:";
cin>>a>>b>>c;

a1.setvalve(a);
a2.setvalve(b);
a3.setvalve(c);
avg(a1,a2,a3);
return 0;
}
```

Output:



```
enter three numbers:9 9 9
9   is average
Process returned 0 (0x0)
```

<u>A member function of another class</u>

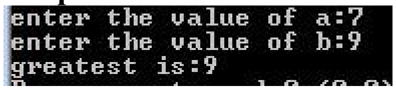
Member function of one class can be friend function of another class. In such cases, they are defined using the scope resolution operator.

```
class first
{
……….
int fun1();
……..
};

class second
{
……….
friend int first :: fun1();
……..
};
```

<u>Program 16:</u> Program to find greatest between data member of one class and another class by making the member function of one class as a friend function of another class.

```
#include<iostream>
using namespace std;
class second;    // Forward declaration to satisfy the compiler about class B
class first
{       int a ;
        void get()
        { cout<<"enter the value of a:";
           cin>>a;
        }
        public: void greatest(second);
};
class second
{
      int b ;
      void get()
      { cout<<"enter the value of b:";
         cin>>b;
      }
      friend    void first:: greatest(second);
};
void first::greatest (second ob2 )
{    get();
     ob2.get();
     if(a>ob2.b)
        cout<<"greatest is"<<a ;
     else
     cout<<"greatest is:"<<ob2.b;
}
int main()
{

  first ob1;
  second ob2;
  ob1.greatest(ob2) ;
  return 0;
}
```

Output:



```
enter the value of a:7
enter the value of b:9
greatest is:9
```

<u>Friend classes</u>

Like a friend function, a class can also be made a friend of another class using keyword friend. For example:

```
class first
{
…………
………….
```

```
        friend class second;    //all member functions of second are friends to first.
};

class second
{
    ... .. ...
};
```

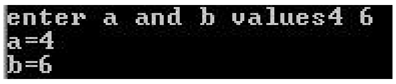
When a class is made a friend class, all the member functions of that class becomes friend functions.

In this program, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A. But, member functions of class A cannot access the data of class B.

Program 17: Example friend class

```
#include <iostream>
using namespace std;
class Test1
{
int a,b;
public:
friend class Test2;
void getab()
{
cout<<"enter a and b values";
cin>>a>>b;
}
};

class Test2
{
public:
void putab(Test1 t1)
{
cout<<"a="<<t1.a<<endl;
cout<<"b="<<t1.b;
}
};

int main()
{
Test1 T1;
Test2 T2;
T1.getab();
T2.putab(T1);
}
```

Output:



```
enter a and b values4 6
a=4
b=6
```

Use of const keyword

Constant is something that doesn't change. In C++ we use the keyword const to make program elements constant. const keyword can be used in many contexts in a C++ program. It can be used with:

i. Variables
ii. Function Parameters and Return type
iii. Pointers
iv. Class Data Members
v. Class Member Functions
vi. Objects

i. Variables

If you make any variable as constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while they are declared.

Once initialized, if we try to change its value, then we will get compilation error.

Example 1:

```
int main()
{
const int a=5;    //declaring and initializing a const variable
a=10;             //this will give compilation error
return 0;
}
```

Example 2:

```
int main()
{
const int a=5;    //declaring and initializing a const variable
a++;              //this will give compilation error
return 0;
}
```

ii. Function Parameters and Return type

We can also make the type of argument and return type of a function const. Once constant declared, we will get compilation error if we try to change it.

Example 1:

```
void func(const int a)      //const argument
{
```

```
a=5;    //compilation error
cout<<c;
}

int main()
{
func(6);
}
```

This will result in compilation error since we cannot change the value of any const variable.

Example 2:

```
const int func()      //const return type
{
return 4;
}

int main()
{
int a;
a=func();
cout<<a;
return 0;
}
```

In the above code, the return type of the function func is const and so we returned a const integer value.

iii. Pointer

Const Pointer: If we make a pointer const, we cannot change the pointer. This means that the pointer will always point to the same address but we can change the value of that address.

We declare a const pointer as follows

```
int a=4;
int * const p=&a; // const pointer p pointing to the variable a.
```

Note that we cannot change the pointer p but can change the value of a.

Example 1:

```
#include<iostream>
using namespace std;
int main()
{
int x=1;
int *const p=&x;
cout<<*p<<endl;
int y=2;
//p=&y;    //error
cout<<*p<<endl;
return 0;
}
```

We can see that a constant pointer variable ‘p’ is declared at line 6 that points to variable x. As the pointer variable ‘p’ is declared as const so it will always point to variable x. At line 9, we tried to change the value of pointer variable ‘p’ which will lead to a compilation error. If the pointer variable was not declared as const then the same code will execute without any error as one pointer variable can point to more than variables at different times.

Pointer to Const Variable: A pointer to const means that we can access the value of the variable through pointer variable but cannot change the value of the pointed variable through the pointer variable as explained in the following example.

const int *p; //const pointer p pointing to a const int variable.

Here, p is a pointer which is pointing to a const int variable. This means that we cannot change the value of that int variable.

Example 2:

```
#include<iostream>
using namespace std;
int main()
{
int x=1;
int const *p=&x;
cout<<*p<<endl;
x=x+10;
*p=*p+10;   //error
cout<<*p<<endl;
return 0;
}
```

iv. Defining Class Data members as const

These are data variables in class which are defined using const keyword. They are not initialized during declaration. Their initialization is done in the constructor.

Example:

```
#include <iostream>
using namespace std;
class SI
{
float p;
int t;
const float r;
public:
```

```
SI() : r(2.5)
{
}

void read(float pa, int ti)
{
p=pa;
t=ti;
}

float show()
{
return(p*t*r)/100;
}
};
int main()
{
SI s;
s.read(1000,10);
cout<<"SI="<<s.show();
return 0;
}
```

v. Defining Class Object as const

When an object is declared or created using the const keyword, its data members can never be changed, during the object's lifetime.

Syntax:

const class_name object;

If in the class defined above, we want to define a constant object, we can do it like:
const A a(5);

Example:

```
class A
{
public:
int x;
A()
{
x=0;
}
};

int main()
{
const A a; //declaring const object ‘a’ of class ‘A’
a.x=10;   //compilation error
return 0;
}
```

The above program will give compilation error. Since we declared ‘a’ as a const object of class A, we cannot change its data members. When we created ‘a’, its constructor got called assigning a value 0 to its data members ‘x’. Since ‘x’ is a data member of a const object, we cannot change its value further in the program. So when we tried to change its value, we got compilation error.

vi. Defining Member function of Class as const

A const member function cannot change the value of any data member of the class and cannot call any member function which is not constant.

To make any member function const, we add the const keyword after the list of the parameters after the function name.

```
#include <iostream>
using namespace std;
class test
{
int a,b;
public:
void read()
{
a=10;
b=10;
}

void show() const
{
a=30;   //error
b=40;   //error
cout<<"a="<<a<<endl;
cout<<"b="<<b<<endl;
}
};
int main()
{
test t;
t.read();
t.show();
return 0;
}
```

The above code will give us compilation error because show() is a const member function of class test and we are trying to assign a value to its data member ‘a and b’.

EMPTY CLASSES

C++ allows creating an Empty class. We can declare an empty class and its object. The declaration of Empty class and its object are same as normal class and object declaration.

An Empty class's object will take only one byte in the memory; since class doesn't have any data member it will take minimum memory storage. One byte is the minimum memory amount that could be occupied.

Example:

```
#include <iostream>
using namespace std;
class Empty {};
int main()
{
  Empty e;
  cout<< sizeof(e) << endl;
  return 0;
}
```

Output:

```
1
```

LOCAL CLASS

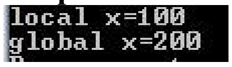
A class declared inside a function becomes local to that function and is called Local Class in C++.

Example:

```
#include <iostream>
using namespace std;
int x=200;
void fun()
{
   class Test
      {
         int x;
           public:
                 void show()
                 {
                       x=100;
                       cout<<"local x=" <<x<<endl;
                       cout<<"global x="<<::x;
                 }
      };
      Test T1;
      T1.show();
      }
```

```
int main()
{
        fun();
        return 0;
}
```

Output:



```
local x=100
global x=200
```

Following are some interesting facts about local classes.

i. A local class name can only be used in the enclosing function.
ii. All the methods of Local classes must be defined inside the class only.
iii. A Local class cannot contain static data members.
iv. Local classes can access global types, variables and functions. Also, local classes can access other local classes of same function.

NESTED CLASS

When a class is defined in another class, it is known as nesting of classes. In nested class the scope of inner class is restricted by outer class.

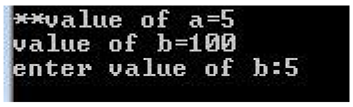
I. Member function of a nested class have no special access to members of an enclosing class.
II. Member function of an outer class have no special access to members of a nested class.
III. Object of inner class can be created by using : : operator.
outer_class: : inner_class obj1;
IV. The member function of the inner class can be defined outside its scope.
return_type of fun outer_class: : inner_class: : fun()   {   }

Example 1: we declare the inner class under public access specifier.

```
#include<iostream>
using namespace std;
class sample
{
        public:
        int a;
        class inner
          {
                public:
                int b;
              void get_in()
              {
              cout<<"enter value of b:";
              cin>>b;
              }
        };
            void show_out()
```

```
                {
                        inner i2;
                i2.b=100;
                cout<<"\n**value of a=";
                cin>>a;
                cout<<"value of b="<<i2.b<<endl;
                }
};

int main()
{
sample a1;
sample::inner i1;
a1.show_out();
i1.get_in();
return 0;
}
```

Output:



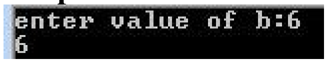
```
**value of a=5
value of b=100
enter value of b:5
```

Example 2: we declare the inner class under private access specifier.

```
#include<iostream>
using namespace std;
class sample
{

        int a;
          class inner
          {
                  public:
                  int b;
                void get_in()
                {
                cout<<"enter value of b:";
                cin>>b;
                }
        };
        public:
              void show_out()
                {
                inner i1;
                i1.get_in();
                cout<<i1.b;
                }
};
```

```
int main()
{
sample a1;
a1.show_out();
return 0;
}
```

Output:



```
enter value of b:6
6
```

CONTAINER CLASSES

A container stores many entities and provide sequential or direct access to them. List, vector and strings are such containers in standard template library. The string class is a container that holds chars. All container classes access the contained elements safely and efficiently by using iterators.

Container class is a class that hold group of same or mixed objects in memory.

Types of container class

i. **Heterogeneous:** Heterogeneous container class can hold mixed objects in memory
ii. **Homogeneous:** when it is holding same objects, it is called as homogeneous container class.

The following are the standardized container classes :

i. **std::map :** Used for handle sparse array or a sparse matrix.
ii. **std::vector :** Like an array, this standard container class offers additional features such as bunds checking through the at () member function, inserting or removing elements, automatic memory management and throwing exceptions.
iii. **std::string :** A better supplement for arrays of chars.

Advantages of Containers

i. Container classes make programmers more productive.
ii. Container classes let programmers write more robust code.

Example:

```
void f(vector<int>& v)
{
// do something with v
for (int i = 0; i<v.size(); ++i) v[i] = i;
}

vector<int> v1(20);
vector<int> v2(10);
```

```
void g()
{
f(v1);
f(v2);
}
```

BIT FIELDS AND CLASSES

A field or member inside a class which allows programmers to save memory are known as bit fields. As we know a integer variable occupies 32 bits(on a 32 bits machine). if we only want to store two values like 0 and 1, only one bit is sufficient. Remaining 31-bits are wasted. In order to reduce such wastage of memory, we can use bit fields.

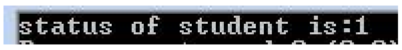
Syntax:
type-specifier   bit-field name: width;

Eg:
Unsigned int status:1;

Example:

```
#include <iostream>
using namespace std;
class student
{
      string name;
      string regdno;
      int age;
      string branch;
      unsigned int status: 1;// size of status field is 1 bit
public:
void set_status(int x)
{
      status=x;
}
int get_status()
{
      return status;
}
};
int main()
  {
    student s1;
    s1.set_status(1);
    cout<<"status of student is:"<<s1.get_status();
    return 0;
  }
```

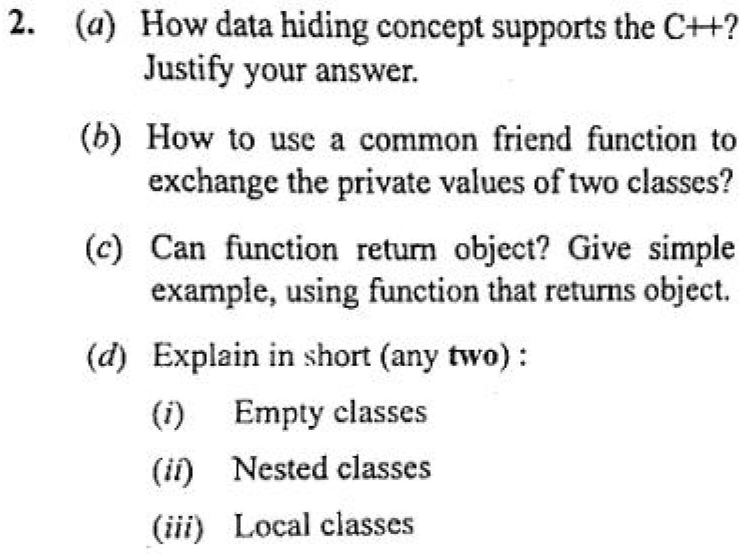
Output:



```
status of student is:1
```

Question Bank

Apr-may 2018



2. (*a*) How data hiding concept supports the C++? Justify your answer.

(*b*) How to use a common friend function to exchange the private values of two classes?

(*c*) Can function return object? Give simple example, using function that returns object.

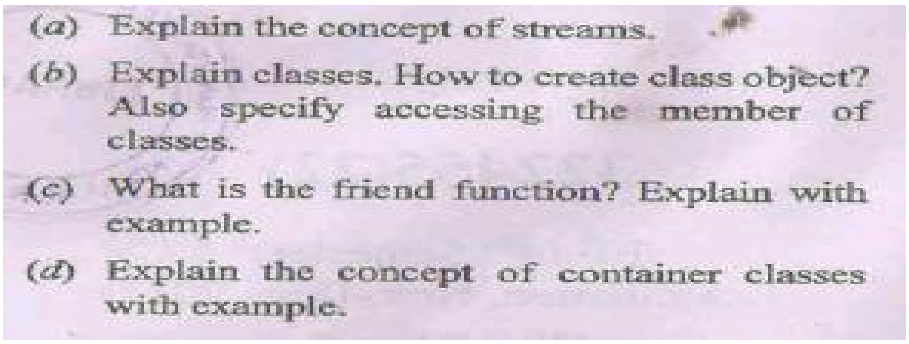
(*d*) Explain in short (any **two**) :

(*i*) Empty classes

(*ii*) Nested classes

(*iii*) Local classes

Nov-Dec 2017



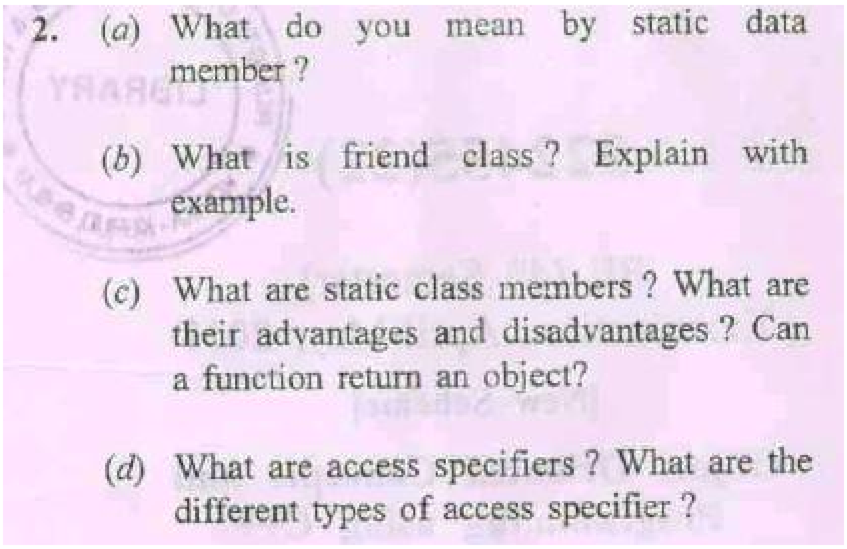
(*a*) Explain the concept of streams.

(*b*) Explain classes. How to create class object? Also specify accessing the member of classes.

(*c*) What is the friend function? Explain with example.

(*d*) Explain the concept of container classes with example.

Apr-may 2017



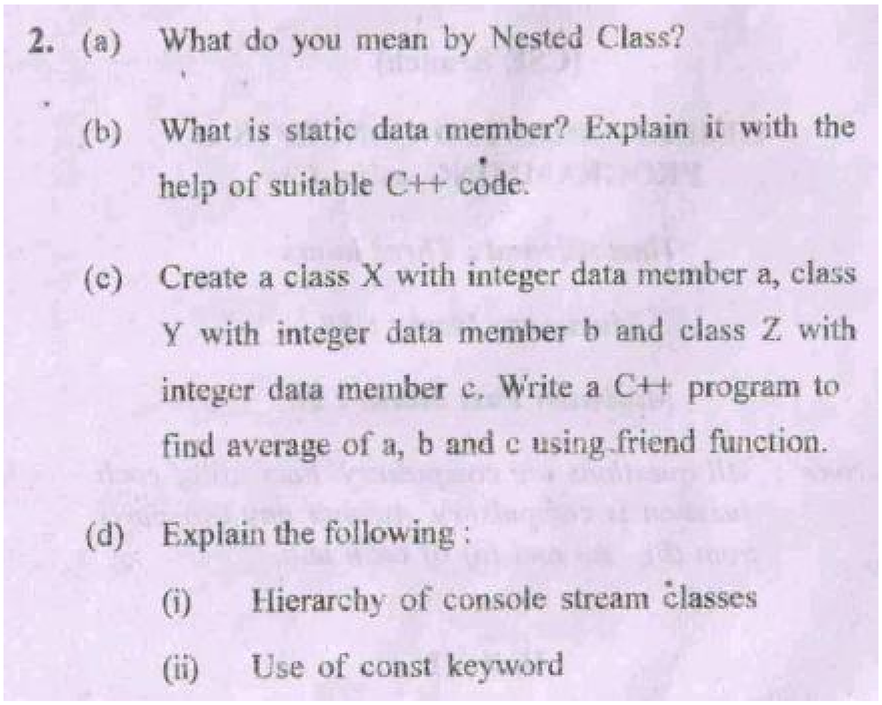
2. (a) What do you mean by static data member ?

(b) What is friend class ? Explain with example.

(c) What are static class members ? What are their advantages and disadvantages ? Can a function return an object?

(d) What are access specifiers ? What are the different types of access specifier ?

Nov-Dec 2016



2. (a) What do you mean by Nested Class?

(b) What is static data member? Explain it with the help of suitable C++ code.

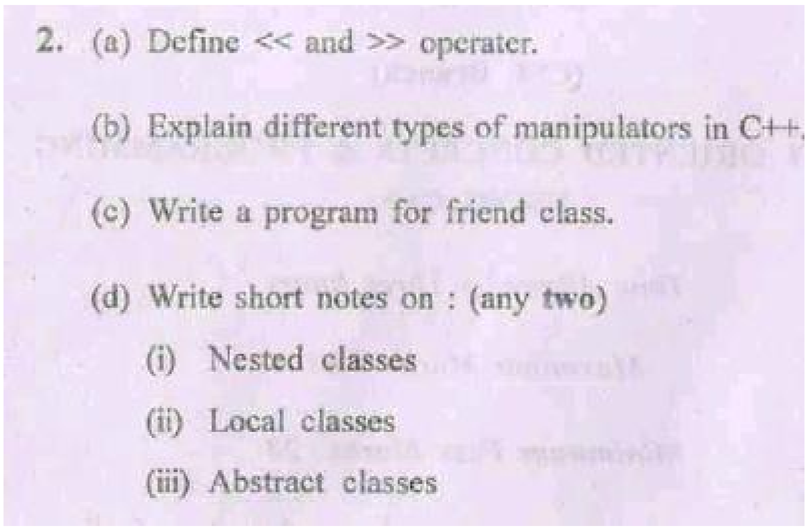
(c) Create a class X with integer data member a, class Y with integer data member b and class Z with integer data member c. Write a C++ program to find average of a, b and c using friend function.

(d) Explain the following :

(i) Hierarchy of console stream classes

(ii) Use of const keyword

Apr-may 2016



2. (a) Define << and >> operater.

(b) Explain different types of manipulators in C++.

(c) Write a program for friend class.

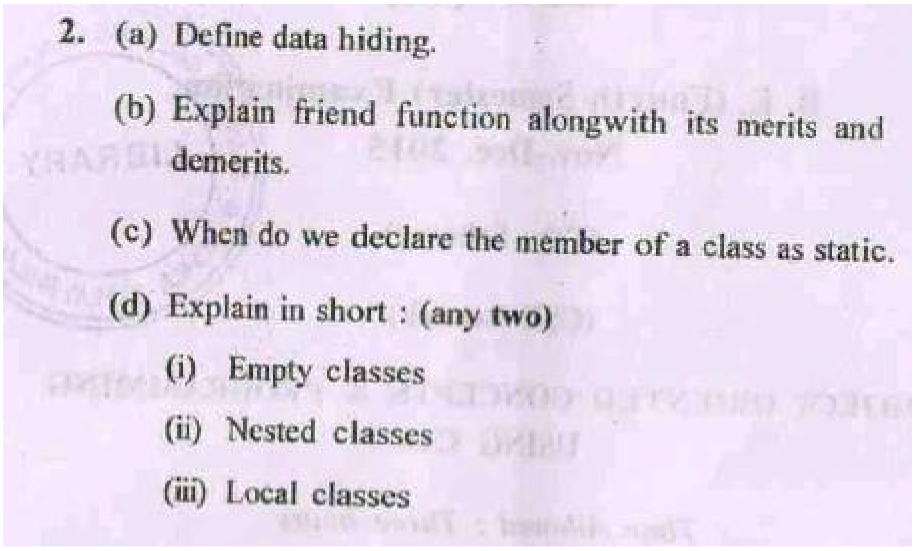
(d) Write short notes on : (any two)

(i) Nested classes

(ii) Local classes

(iii) Abstract classes

Nov-Dec 2015



2. (a) Define data hiding.

(b) Explain friend function alongwith its merits and demerits.

(c) When do we declare the member of a class as static.

(d) Explain in short : (any two)

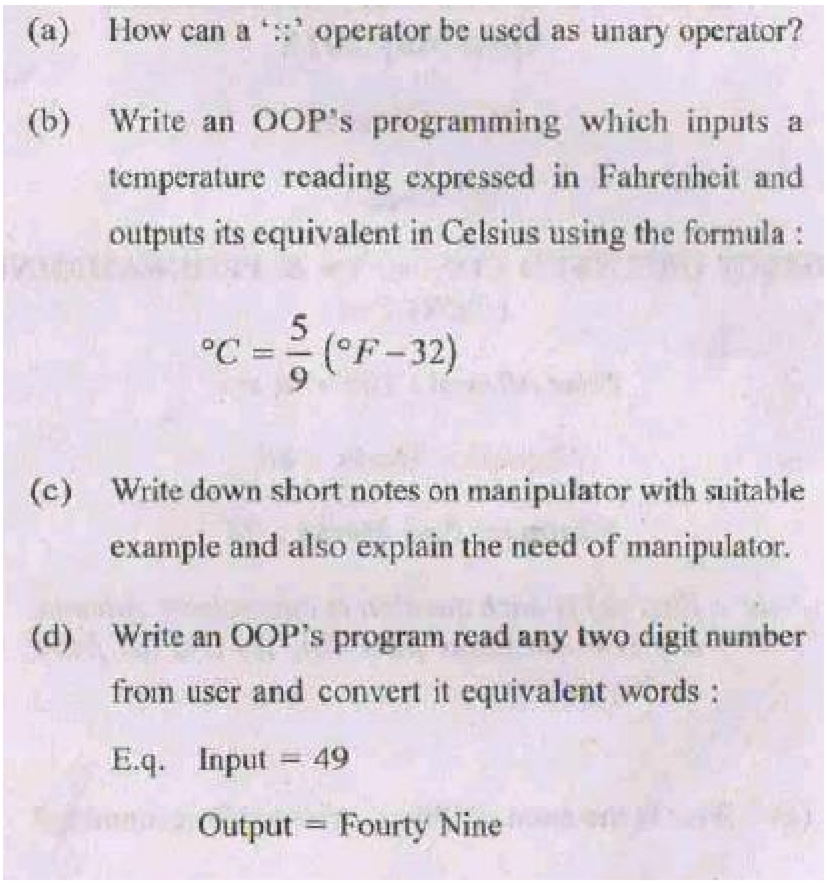
(i) Empty classes

(ii) Nested classes

(iii) Local classes

Explain friend function along with its merit and demerits

Apr-may 2015



(a) How can a '::' operator be used as unary operator?

(b) Write an OOP's programming which inputs a temperature reading expressed in Fahrenheit and outputs its equivalent in Celsius using the formula :

$$°C = \frac{5}{9}(°F - 32)$$

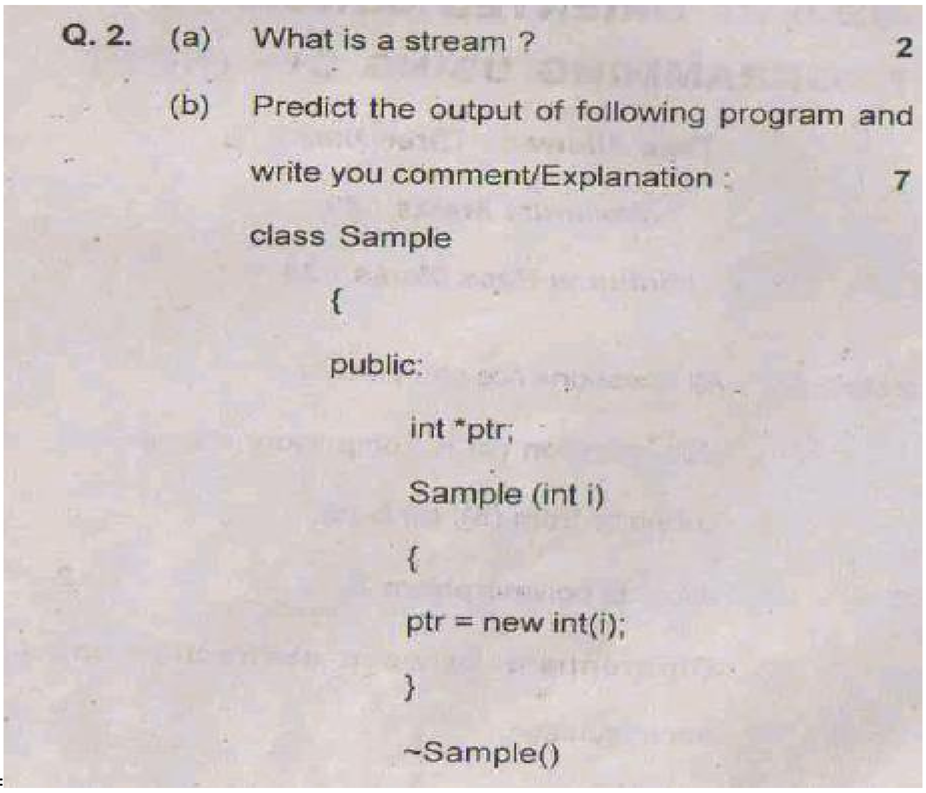
(c) Write down short notes on manipulator with suitable example and also explain the need of manipulator.

(d) Write an OOP's program read any two digit number from user and convert it equivalent words :

E.q. Input = 49

Output = Fourty Nine

Nov-Dec 2014

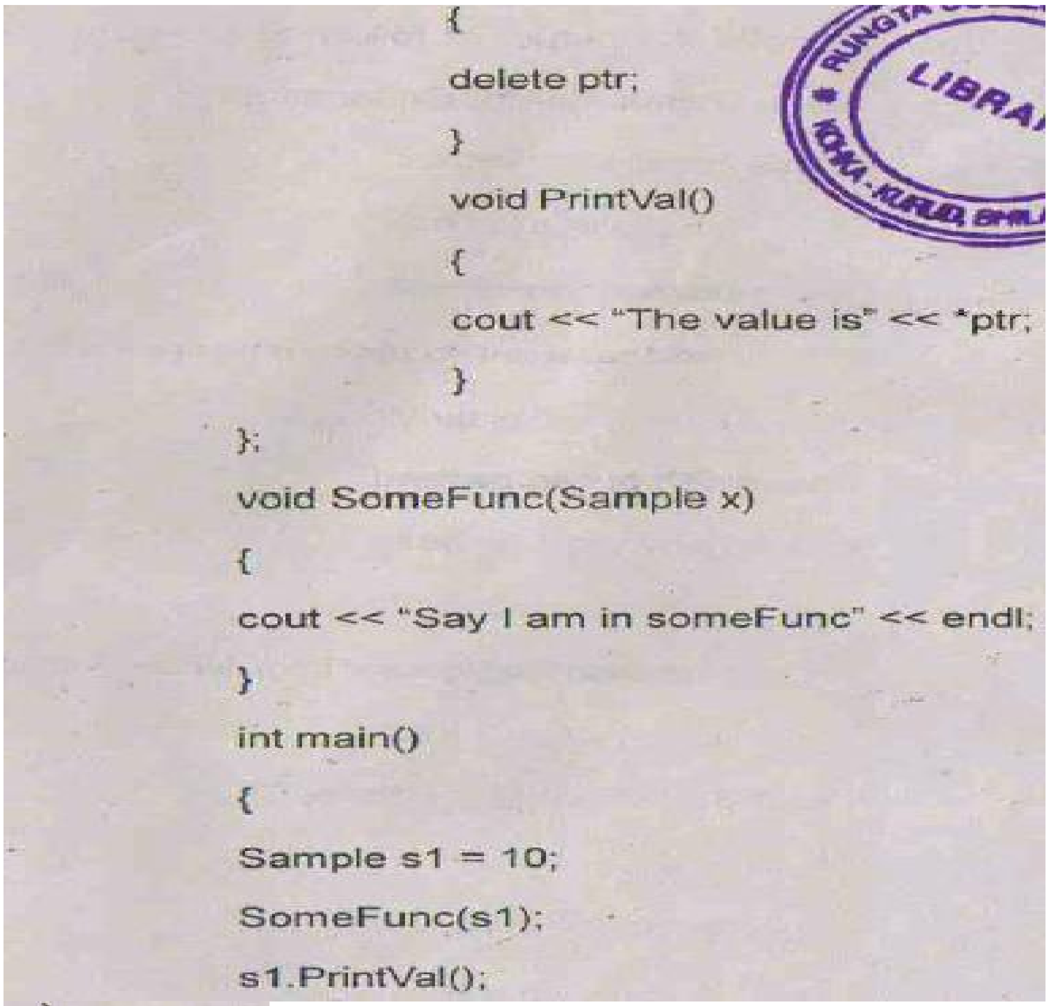


Q. 2. (a) What is a stream ? 2

(b) Predict the output of following program and write you comment/Explanation : 7

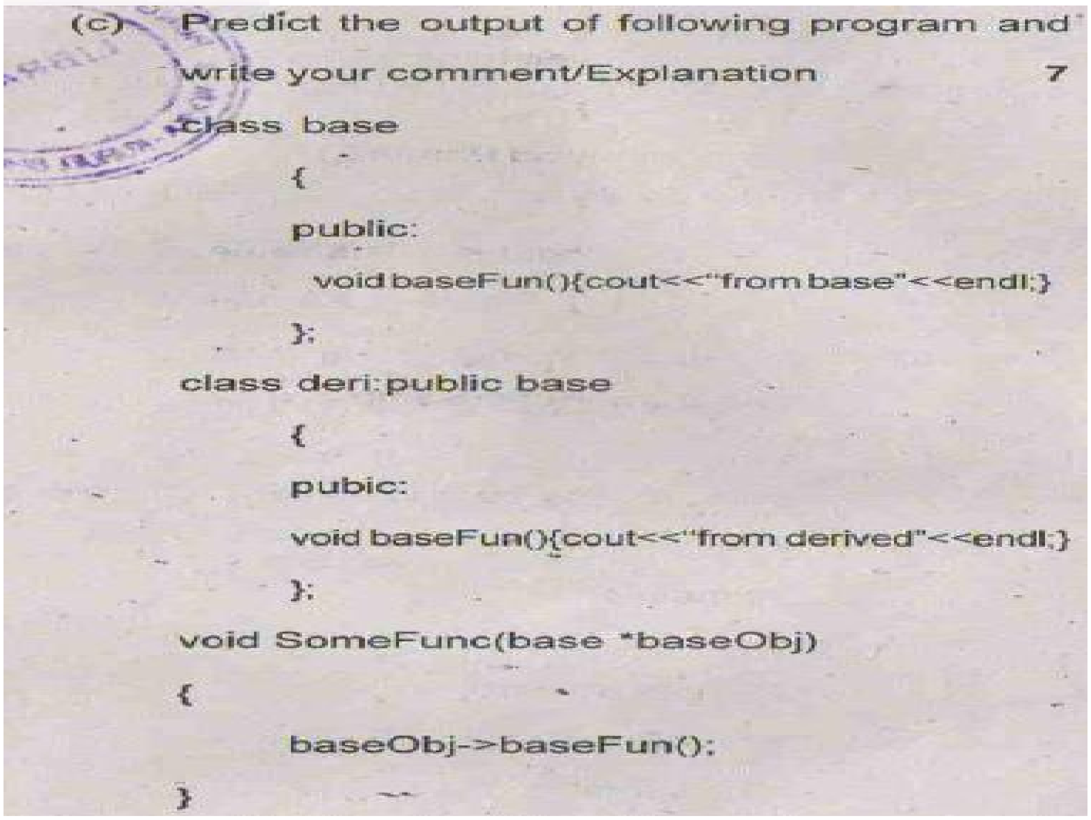
```
class Sample
    {
    public:
        int *ptr;
        Sample (int i)
        {
        ptr = new int(i);
        }
        ~Sample()
```

=



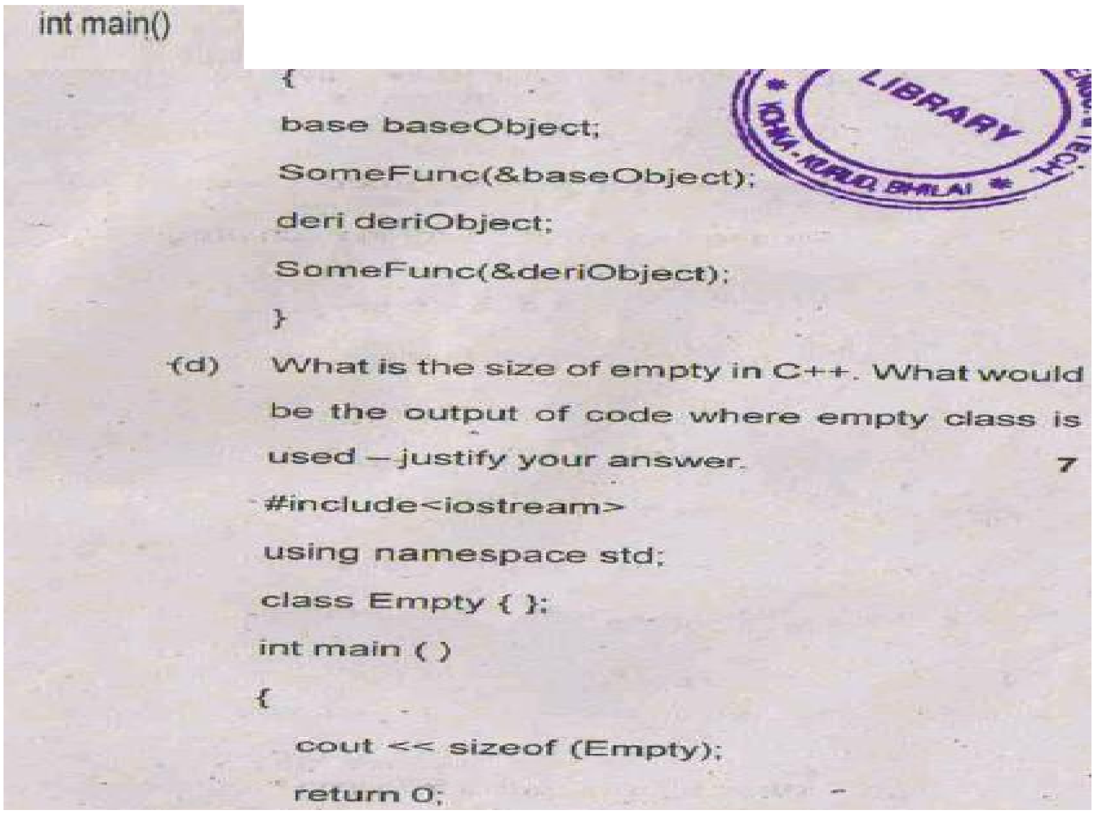
RUNGTA LIBRA KOHKA-KURUD, BHIL

```
                {
                delete ptr;
                }
                void PrintVal()
                {
                cout << "The value is" << *ptr;
                }
        };
        void SomeFunc(Sample x)
        {
        cout << "Say I am in someFunc" << endl;
        }
        int main()
        {
        Sample s1 = 10;
        SomeFunc(s1);
        s1.PrintVal();
}
```



(c) Predict the output of following program and write your comment/Explanation 7

```
class base
        {
        public:
          void baseFun(){cout<<"from base"<<endl;}
        };
class deri:public base
        {
        pubic:
        void baseFun(){cout<<"from derived"<<endl;}
        };
void SomeFunc(base *baseObj)
{
        baseObj->baseFun();
}
```



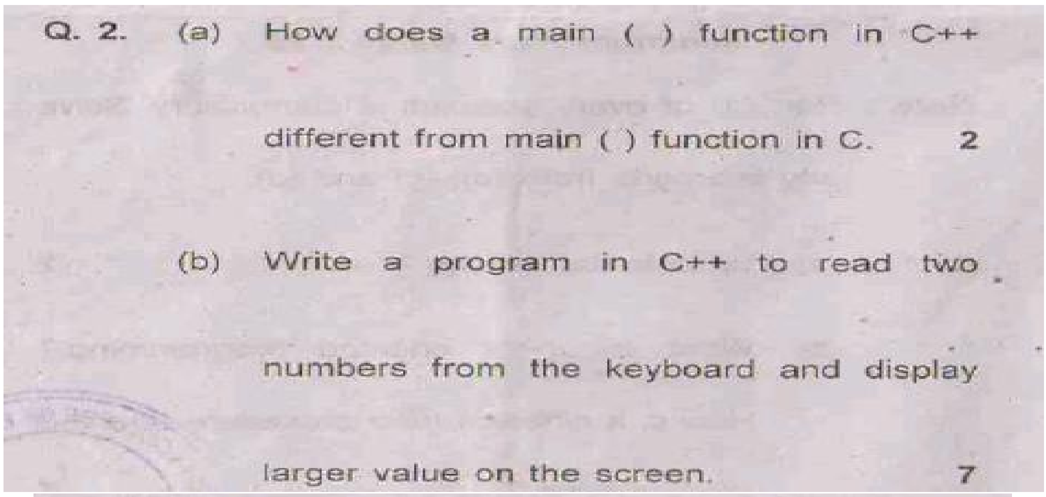
```
int main()
{
base baseObject;
SomeFunc(&baseObject);
deri deriObject;
SomeFunc(&deriObject);
}
```

LIBRARY
BHILAI

(d) What is the size of empty in C++. What would be the output of code where empty class is used – justify your answer. 7

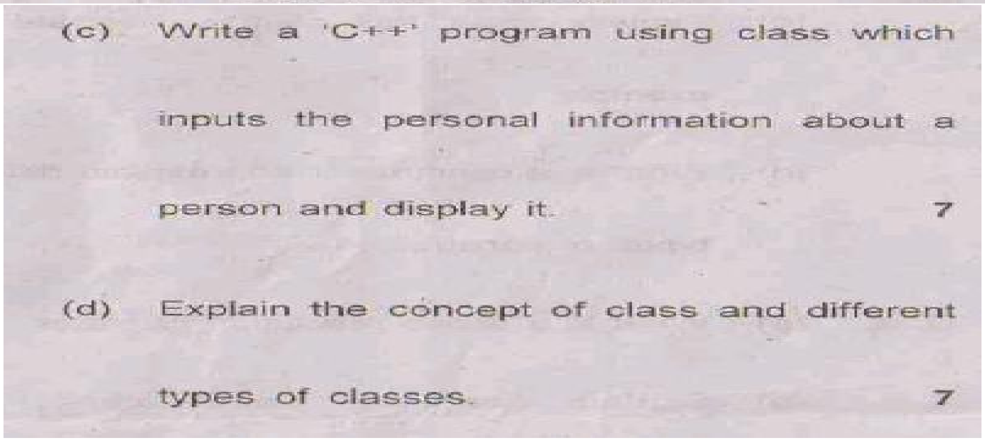
```
#include<iostream>
using namespace std;
class Empty { };
int main ( )
{
  cout << sizeof (Empty);
  return 0;
}
```

Apr-may 2014



Q. 2. (a) How does a main ( ) function in C++ different from main ( ) function in C. 2

(b) Write a program in C++ to read two numbers from the keyboard and display larger value on the screen. 7



(c) Write a 'C++' program using class which inputs the personal information about a person and display it. 7

(d) Explain the concept of class and different types of classes. 7