W1)**Inheritance:**

In C++, inheritance is a process in which one object acquires all the properties and behaviours of its base/parent object automatically. In such way, can reuse, extend or modify the attributes and behaviours which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

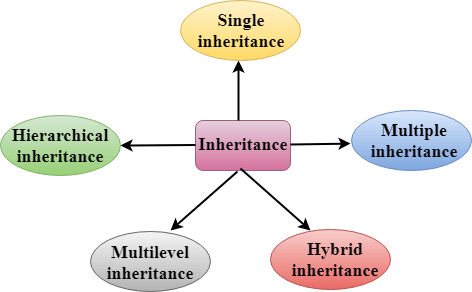
Advantage of C++ Inheritance

**Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

## Types Of Inheritance

**C++ supports five types of inheritance:**

* Single inheritance
* Multiple inheritance
* Hierarchical inheritance
* Multilevel inheritance
* Hybrid inheritance



## Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

1. **class** derived\_class\_name :: visibility-mode base\_class\_name
2. {
3. // body of the derived class.
4. }

**Where,**

**derived\_class\_name:** It is the name of the derived class.

**visibility mode:** The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

**base\_class\_name:** It is the name of the base class.

* When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
* When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

### **Note:**

* In C++, the default mode of visibility is private.
* The private members of the base class are never inherited.

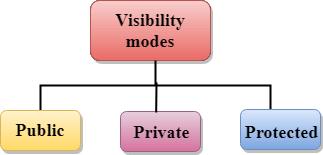
## C++ Single Inheritance

**Single inheritance** is defined as the inheritance in which a derived class is inherited from the only one base class.

C++ Inheritance

Where 'A' is the base class, and 'B' is the derived class.

**Visibility modes can be classified into three categories:**



* **Public**: When the member is declared as public, it is accessible to all the functions of the program.
* **Private**: When the member is declared as private, it is accessible within the class only.
* **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

|  |  |  |  |
| --- | --- | --- | --- |
| **Base class visibility** | **Derived class visibility** | | |
| **Public** | **Private** | **Protected** |
| Private | Not Inherited | Not Inherited | Not Inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |

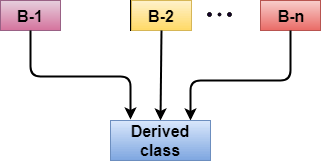
C++ Multilevel Inheritance

**Multilevel inheritance** is a process of deriving a class from another derived class.

C++ Inheritance

## C++ Multiple Inheritance

**Multiple inheritance** is the process of deriving a new class that inherits the attributes from two or more classes.

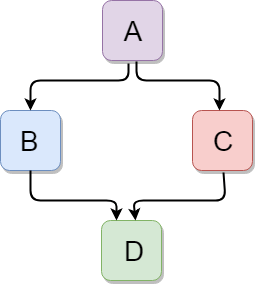


**Syntax of the Derived class:**

1. **class** D : visibility B-1, visibility B-2, ?
2. {
3. // Body of the class;
4. }

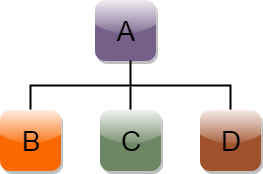
## C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



C++ Hierarchical Inheritance

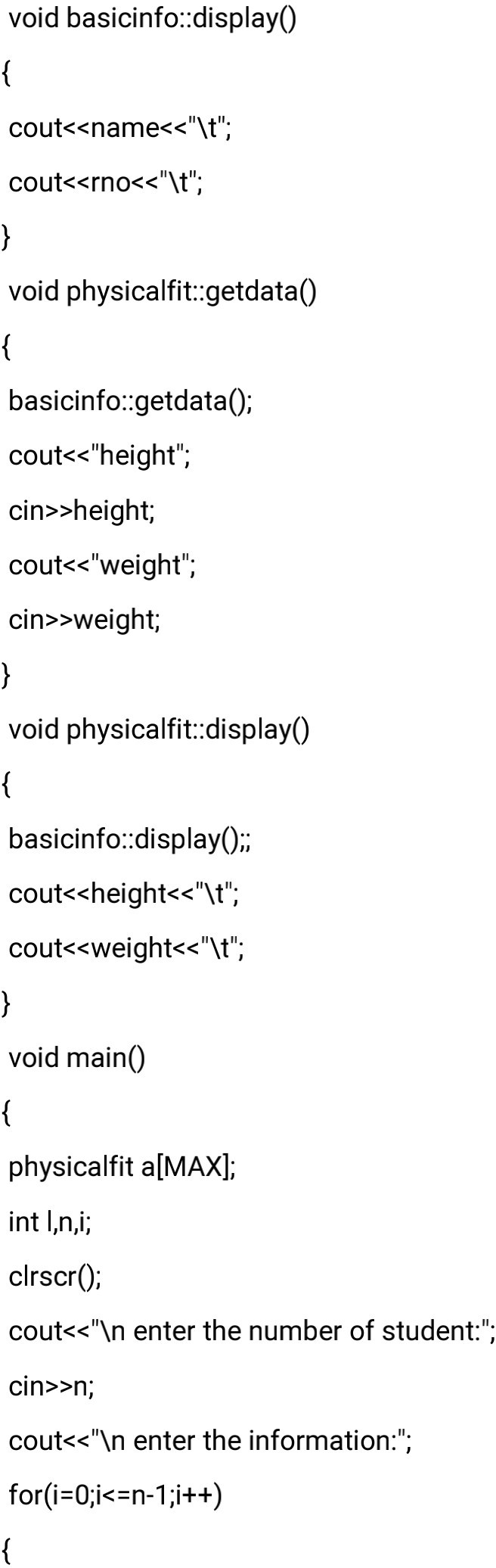
Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

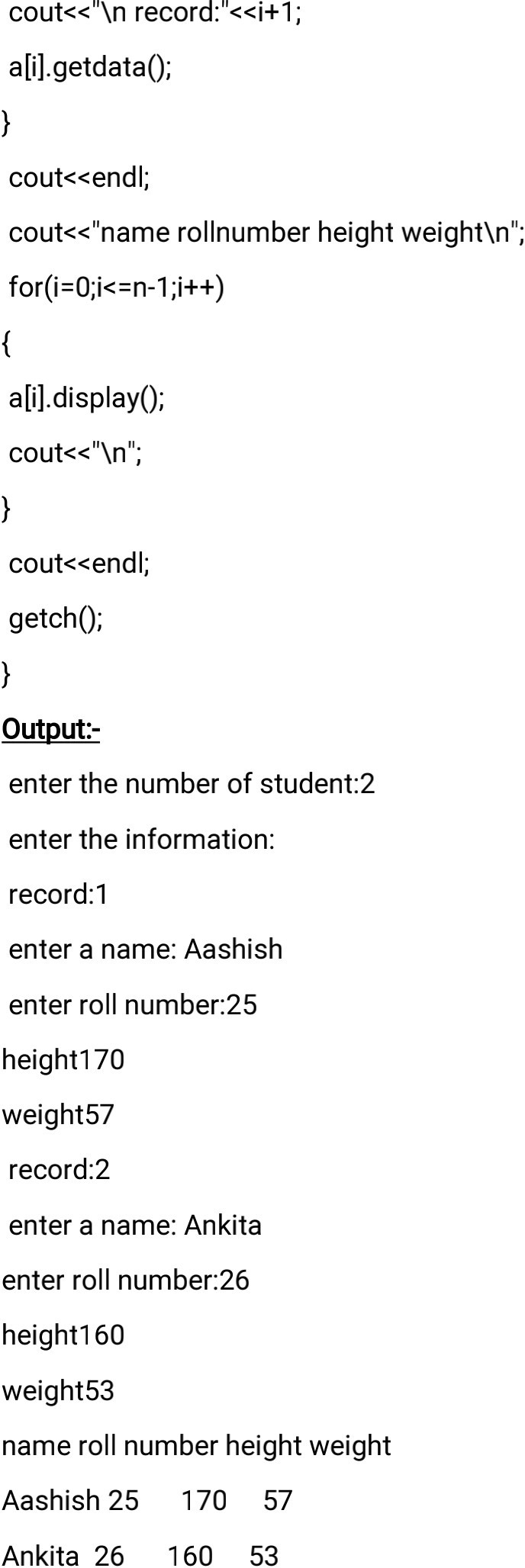


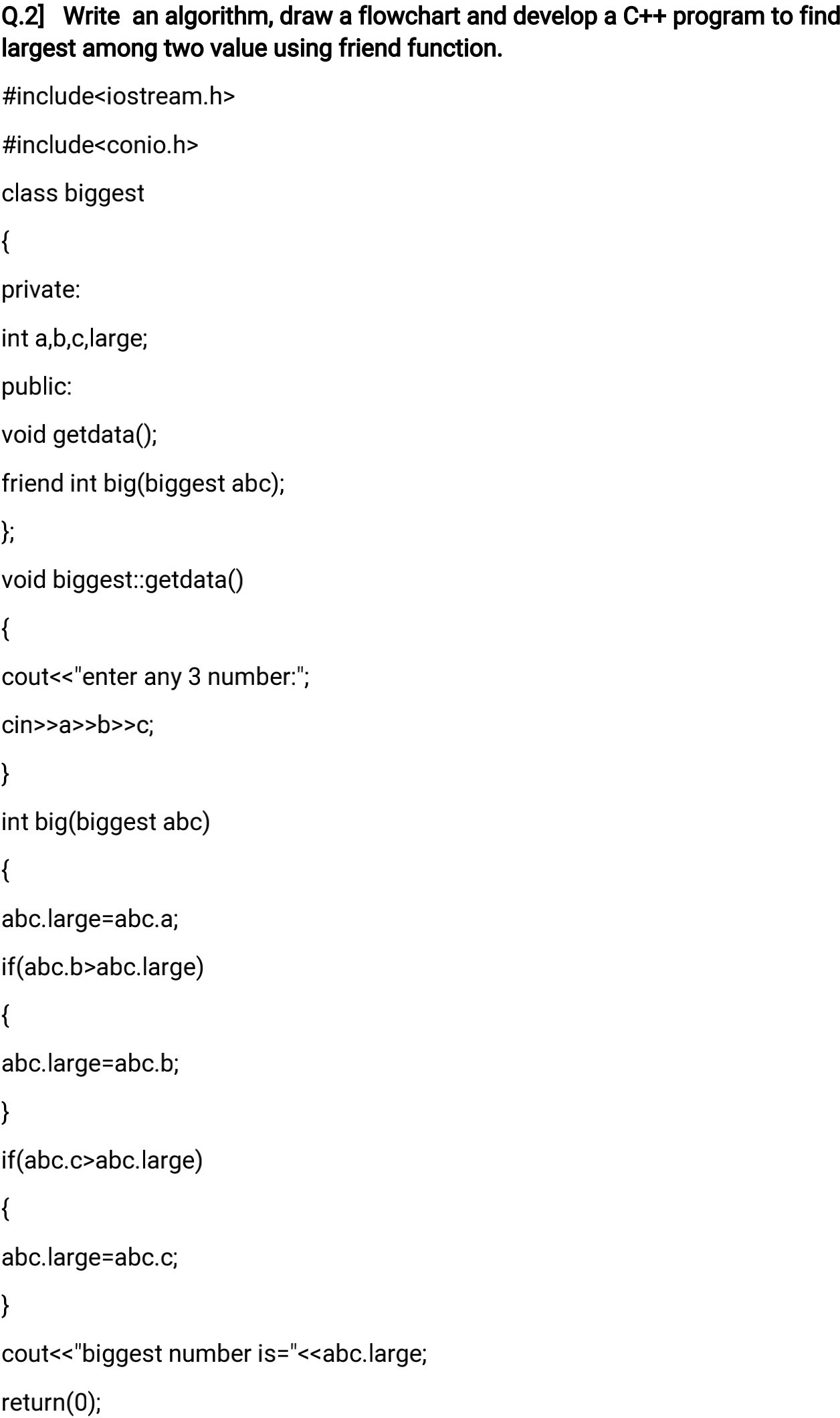
**Syntax of Hierarchical inheritance:**

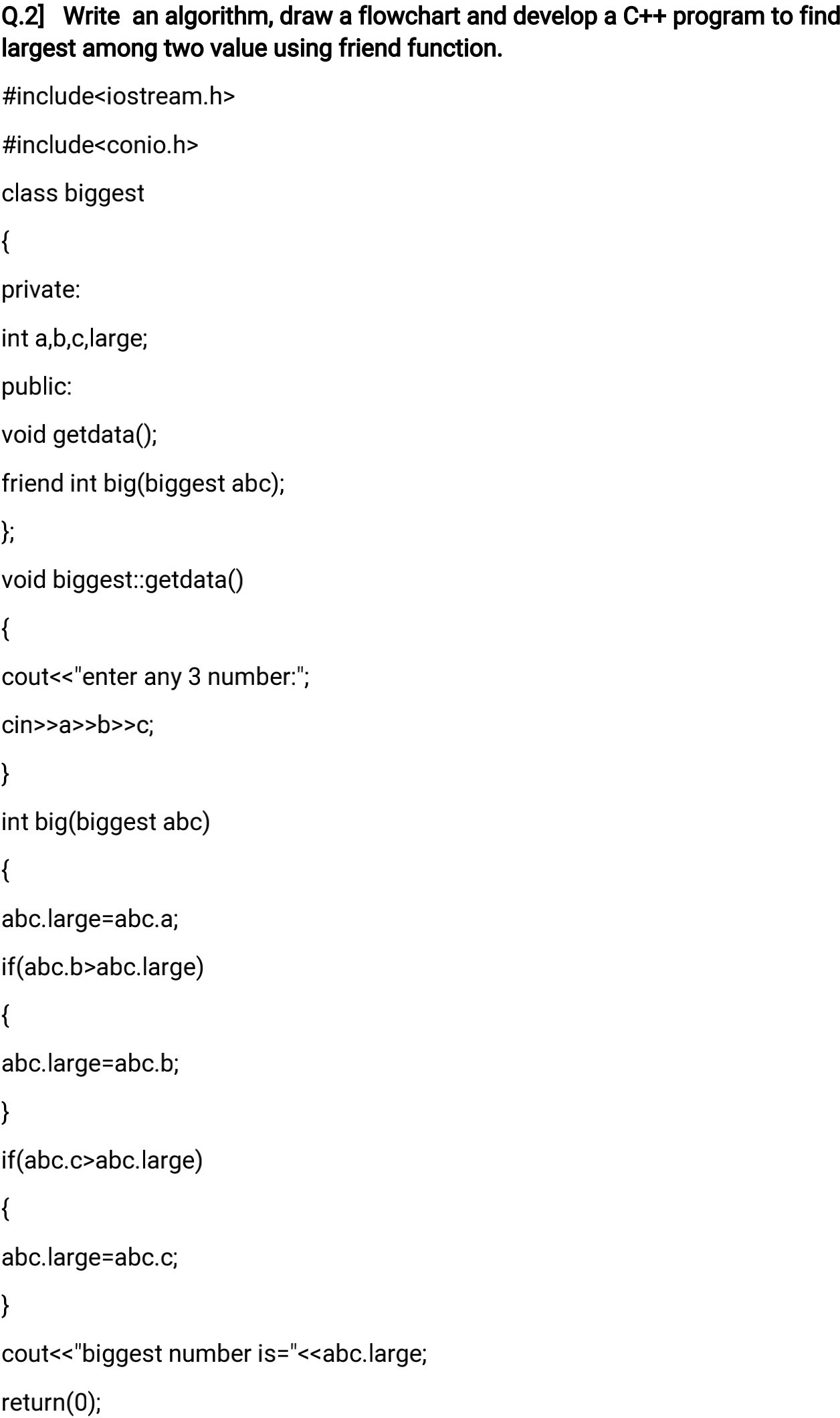
1. **class** A
2. {
3. // body of the class A.
4. }
5. **class** B : **public** A
6. {
7. // body of class B.
8. }
9. **class** C : **public** A
10. {
11. // body of class C.
12. }
13. **class** D : **public** A
14. {
15. // body of class D.
16. }

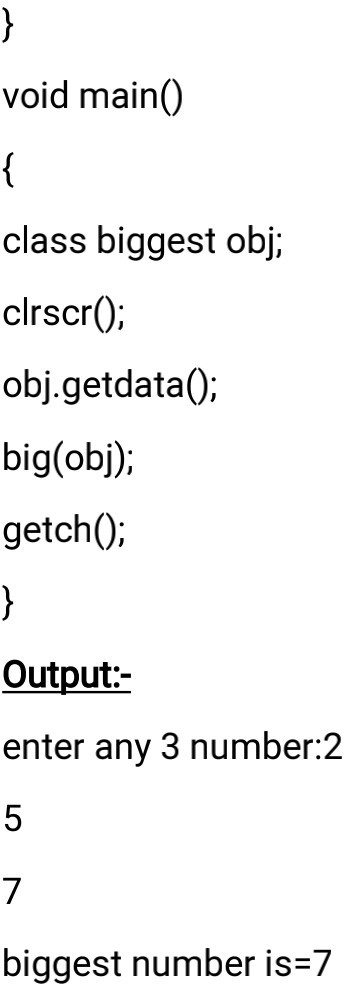


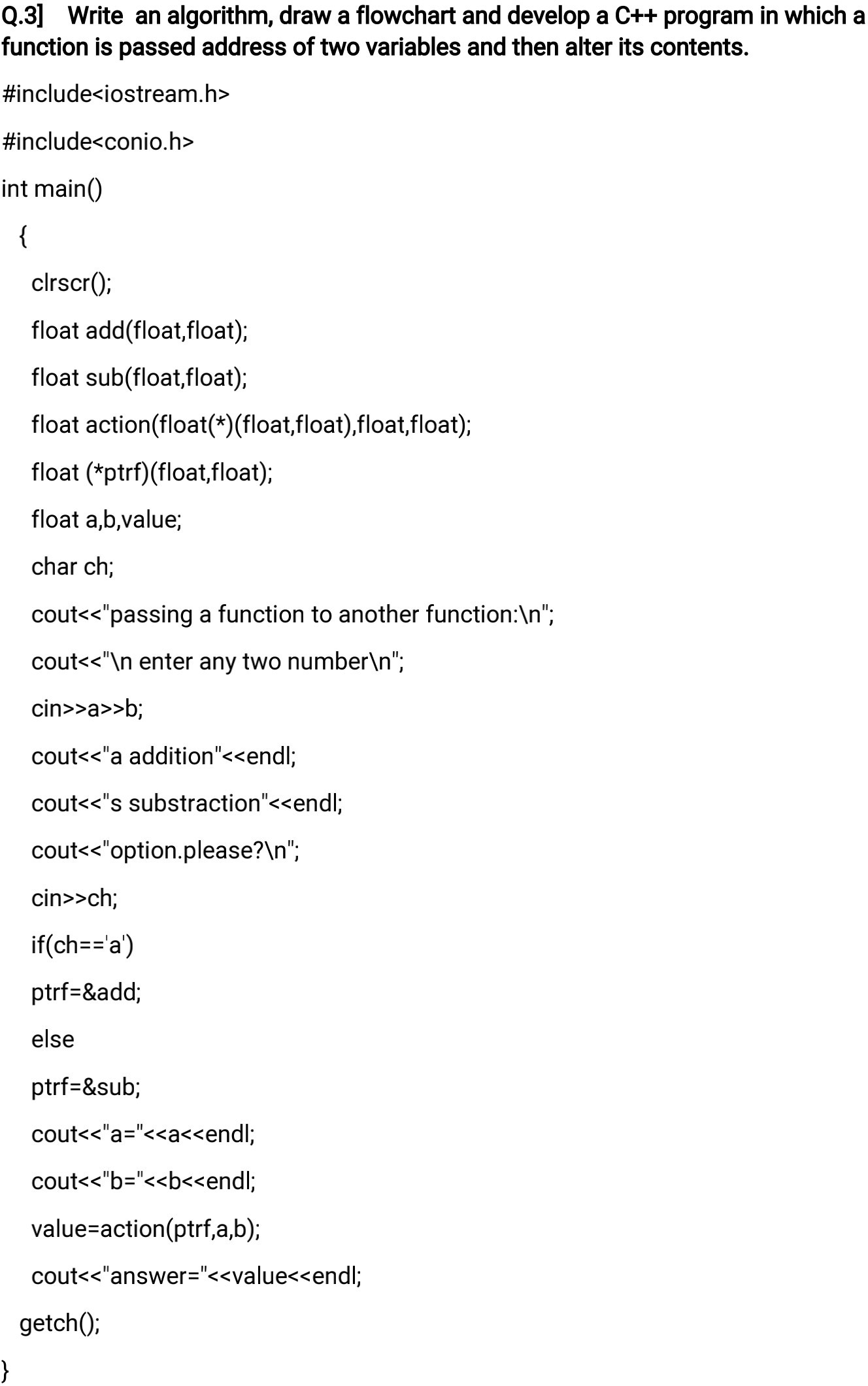


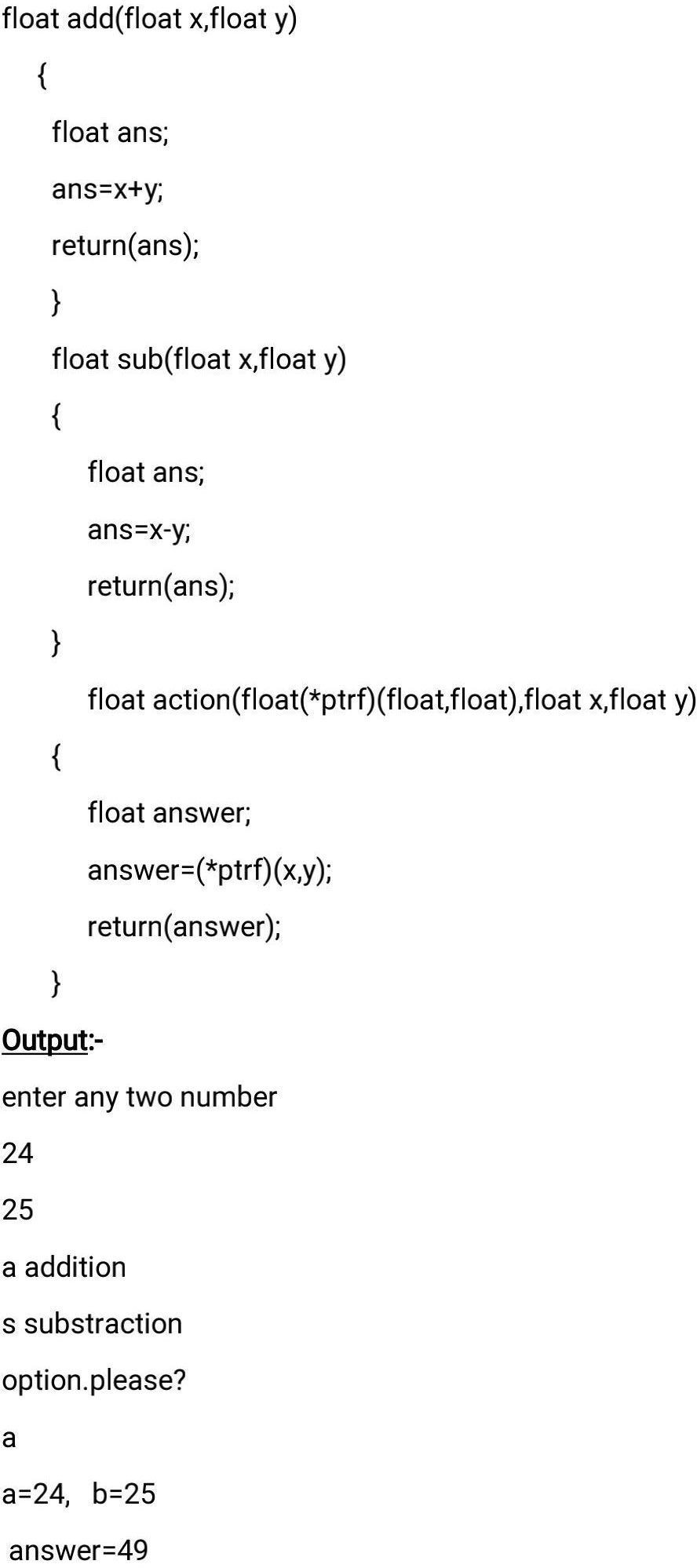


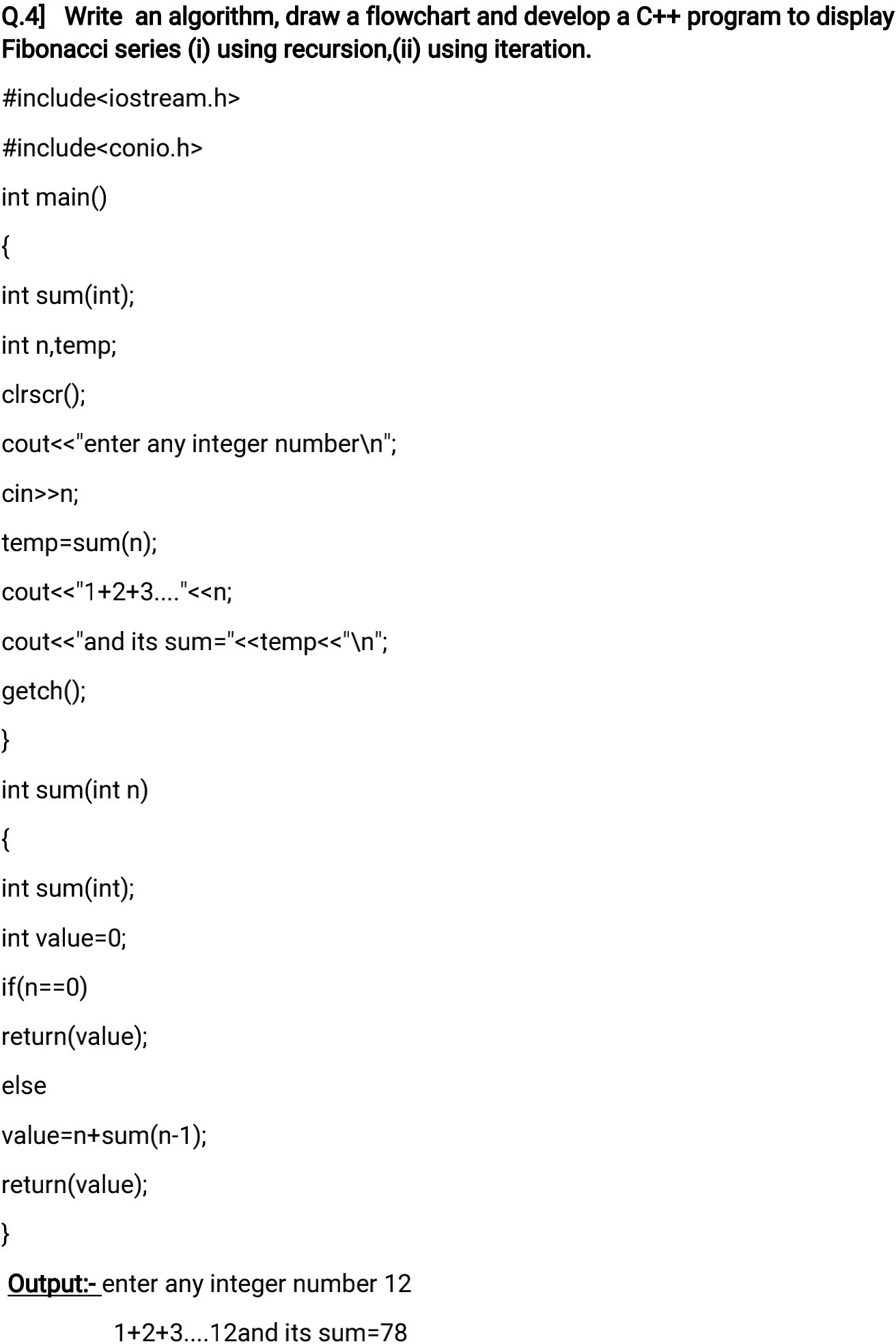


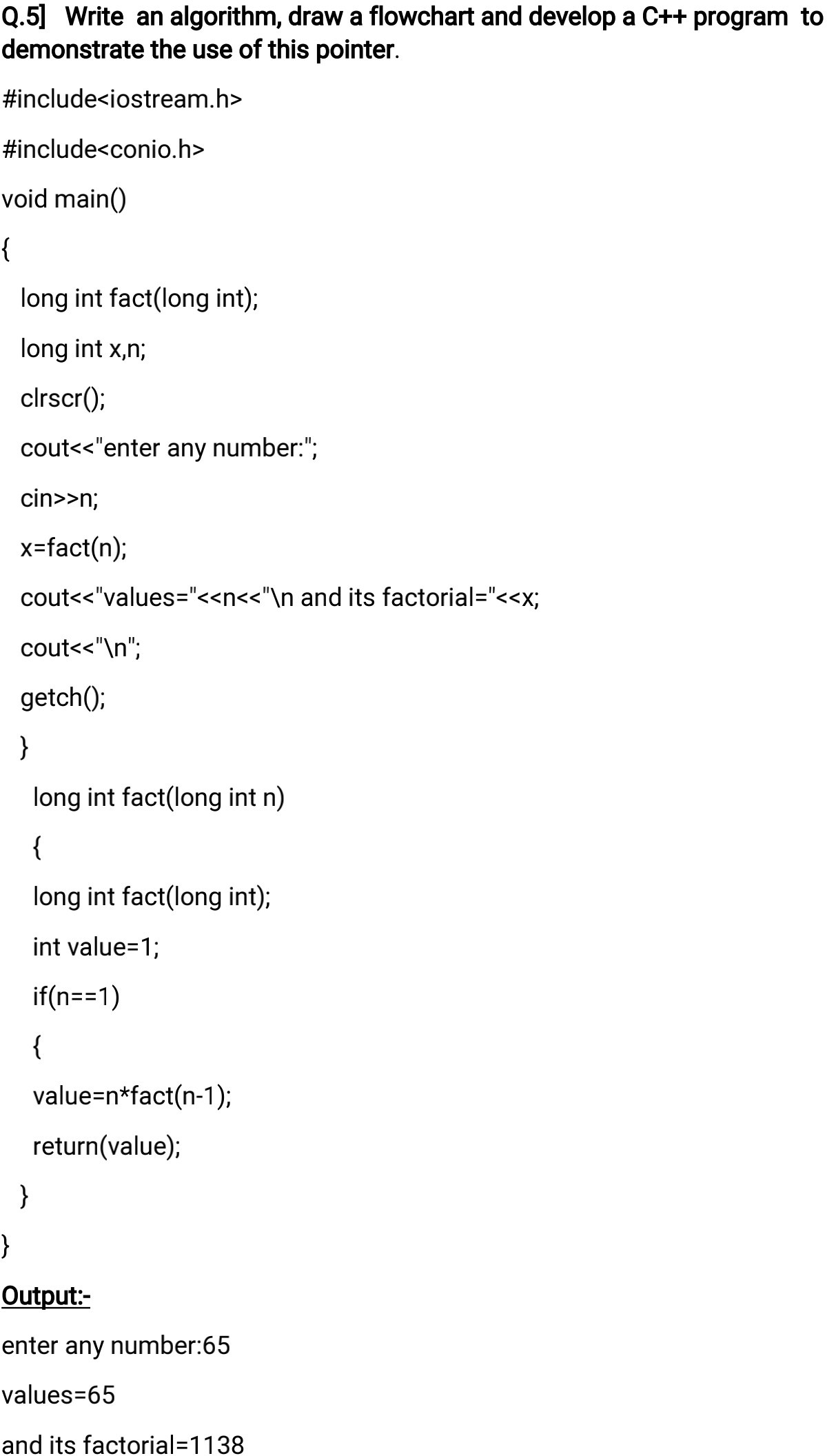












Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

class ClassName {

private:

int dataMember;

public:

method(int a) {

// this pointer stores the address of object obj and access dataMember

this->dataMember = a;

... .. ...

}

}

int main() {

ClassName obj;

obj.method(5);

... .. ...

}

## Applications of this pointer

### **1. Return Object**

One of the important applications of using thispointer is to return the object it points. **For example**, the statement

return \*this;

inside a member function will return the object that calls the function.

### **2. Method Chaining**

After returning the object from a function, a very useful application would be to chain the methods for ease and a cleaner code.

**For example,**

positionObj->setX(15)->setY(15)->setZ(15);

Here, the methods setX, setY, setZ are chained to the object, positionObj. This is possible because each method return \*this pointer.  
This is equivalent to

positionObj->setX(15);

positionObj->setY(15);

positionObj->setZ(15);

### **3. Distinguish Data Members**

Another application of this pointer is distinguishing data members from local variables of member functions if they have same name. **For example**,

#include<iostream>

#include<conio.h>

using namespace std;

class student

{

char name[100];

int age,roll;

float percent;

public:

void getdata()

{

cout<<"Enter data"<<endl;

cout<<"Name:";

cin>>name;

cout<<"Age:";

cin>>age;

cout<<"Roll:";

cin>>roll;

cout<<"Percent:";

cin>>percent;

cout<<endl;

}

student & max(student &s1,student &s2)

{

if(percent>s1.percent && percent>s2.percent)

return \*this;

else if(s1.percent>percent && s1.percent>s2.percent)

return s1;

else if(s2.percent>percent && s2.percent>s1.percent)

return s2;

}

void display()

{

cout<<"Name:"<<name<<endl;

cout<<"Age:"<<age<<endl;

cout<<"Roll:"<<roll<<endl;

cout<<"Percent:"<<percent;

}

};

int main()

{

student s,s1,s2,s3;

s1.getdata();

s2.getdata();

s3.getdata();

s=s3.max(s1,s2);

cout<<"Student with highest percentage"<<endl;

s.display();

getch();

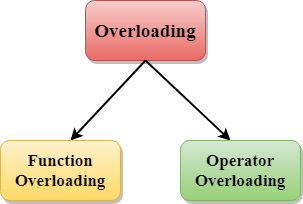
return 0;

}

Operator Overloading:

Types of overloading in C++ are:

* Function overloading
* Operator overloading



C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

**Operator that cannot be overloaded are as follows:**

* Scope operator (::)
* Sizeof
* member selector(.)
* member pointer selector(\*)
* ternary operator(?:)

Syntax of Operator Overloading

1. return\_type class\_name  : : operator op(argument\_list)
2. {
3. // body of the function.
4. }

Where the **return type** is the type of value returned by the function.

**class\_name** is the name of the class.

**operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

* Existing operators can only be overloaded, but the new operators cannot be overloaded.
* The overloaded operator contains atleast one operand of the user-defined data type.
* We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
* When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
* When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

