C++ Cia

GROUP E

1. Compare and contrast multiple inheritance and multilevel inheritance with the help of classes.

a) **Multiple Inheritance** is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.

```
#include <iostream>
using namespace std;
class Area
{
  public:
    float area_calc(float l,float b)
    {
        return l*b;
    }
};

class Perimeter
{
  public:
    float peri_calc(float l,float b)
    {
        return 2*(l+b);
    }
};

class Rectangle : public Area, public Perimeter
{
    public:
      float length, breadth;
       Rectangle() : length(0.0), breadth(0.0) { }
       void get_data( )
       {
           cout<<"Enter length: ";
           cin>>length;
           cout<<"Enter breadth: ";
           cin>>breadth;
       }
};
```

```
int main()
{
    Rectangle r;
    r.get_data();
    cout<<"Area = "<<r.area_calc(r.length,r.breadth);
    cout<<"\nPerimeter = "<<r.peri_calc(r.length,r.breadth);
    return 0;
}
```

OUTPUT:

```
Enter length: 5.1

Enter breadth: 2.3

Area = 11.73

Perimeter = 14.8
```

- In this program, Rectangle derives from both Area and Parameter.
- When area_calc() or peri_calc() is called the program looks in Area and Perimeter classes and executes the respective call.

b) **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.

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

class A
{
    public:
      void display()
      {
          cout<<"Base class content.";
      }
};

class B : public A
{

};
```

```
class C : public B
{

};

int main()
{
    C c;
    c.display();
    return 0;
}
```

OUTPUT:

```
            Base class content.
```

- In this program, class B is derived from A and C is derived from B.
- When the display() function is called, display() is class A is executed Because there is no display() function in C and B.
- If there was display() function in C too, then it would have override display() in A because of member function overriding.

2. Explain Ambiguity resolution in single and multiple inheritance with the help of classes.

- Inheritance is the process of inheriting properties of objects of one class by objects of another class.
- The class which inherits the properties of another class is called Derived or Child or Sub class and the class whose properties are inherited is called Base or Parent or Super class.
- When a single class is derived from a single parent class, it is called Single inheritance. Ambiguity Resolution for single inheritance:

If parent and child classes have same named method, parent name and scope resolution operator(::) is used. This is done to distinguish the method of child and parent class since both have same name.

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

```
using namespace std;
class staff
{
   private:
      char name[50];
      int code;
   public:
      void getdata();
      void display();
};
class typist: public staff
{
   private:
      int speed;
   public:
      void getdata();
      void display();
};

void staff::getdata()
{
   cout<<"Name:";
   gets(name);
   cout<<"Code:";
   cin>>code;
}

void staff::display()
{
   cout<<"Name:"<<name<<endl;
   cout<<"Code:"<<code<<endl;
}

void typist::getdata()
{
   cout<<"Speed:";
   cin>>speed;
}

void typist::display()
{
   cout<<"Speed:"<<speed<<endl;
}

int main()
{
   typist t;
   cout<<"Enter data"<<endl;
   t.staff::getdata();
   t.getdata();
   cout<<endl<<"Display data"<<endl;
   t.staff::display();
```

```
  t.display();
  getch();
  return 0;
}
```

A derived class with two base classes and these two base classes have one common base class is called
multiple inheritance.
Ambiguity in C++ occur when a derived class have two base classes and these two base classes have one
common base class.

Two WAYS to Ambiguity Resolution for Multiple Inheritance :

a. Using scope resolution operator :
Using scope resolution operator we can manually specify the path from which data member a will be
accessed, as shown in statement 3 and 4, in the above example.

```
obj.ClassB::a = 10;       //Statement 3
obj.ClassC::a = 100;      //Statement 4
```

Note : still, there are two copies of ClassA in ClassD.

b. Avoid ambiguity using virtual base class :
To remove multiple copies of ClassA from ClassD, we must inherit ClassA in ClassB and ClassC as virtual class.

Example to avoid ambiguity by making base class as a virtual base class

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

class ClassA
{
    public:
    int a;
};

class ClassB : virtual public ClassA
{
    public:
    int b;
};
class ClassC : virtual public ClassA
{
    public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
    public:
        int d;
    };

    void main()
    {
                        ClassD obj;

                        obj.a = 10;     //Statement 3
                        obj.a = 100;    //Statement 4

                        obj.b = 20;
                        obj.c = 30;
                        obj.d = 40;

                        cout<< "\n A : "<< obj.a;
                        cout<< "\n B : "<< obj.b;
                        cout<< "\n C : "<< obj.c;
                        cout<< "\n D : "<< obj.d;

         }
```

Output :

```
    A : 100
    B : 20
    C : 30
    D : 40
```

Thus, this is how ambiguity is resolved in single and multiple inheritance in c++ using classes.

3. Develop a hybrid inheritance program with the help of classes

```
#include<iostream>
using namespace std;
class performance
{
        public:
        int engine;//In cc
        int power;//In bhp
};
class visual
{
        public:
        char color[10];//Any color
};
```

```
class features
{
	public:
	char abs;//Y or N
	char ac;//Y or N

};
class prototype:public performance,public visual,public features//Multiple
Inheritence
{
	public:
	long estimate;//Estimated price
	void estimatep()
	{
		if(engine<1500)
		{
			estimate+=200000;
		}
		else
		{
			estimate+=500000;
		}
		if(abs=='Y','y')
			estimate+=45000;
		if(ac=='Y','y')
			estimate+=50000;
	}
};
class automobile:private prototype//Single/Multilevel Inheritence
{
	char name[30];
	public:
	void input()
	{
		estimate=0;
		cout<<"Name of the car : ";
		fflush(stdin);
		gets(name);
		cout<<"Performance\n";
		cout<<"Engine Displacement(in cc) : ";
		cin>>engine;
		cout<<"Power(in BHP) : ";
		cin>>power;
		cout<<"Visual\n";
		cout<<"Color : ";
		cin>>color;
		cout<<"Features\n";
		cout<<"ABS(Y or N) : ";
```

```
                        cin>>abs;
                        cout<<"Air Conditioner(Y or N) : ";
                        cin>>ac;
                        estimatep();
                }
                void display()
                {
                        system("cls");
                        double tax;
                        tax=estimate*0.18;
                        cout<<"Name : ";puts(name);
                        cout<<"-----------------------------------\n";
                        cout<<"Performance\n";
                        cout<<"Engine Displacement(in cc) : "<<engine<<endl;
                        cout<<"Power(in BHP) : "<<power<<endl;
                        cout<<"Visual\n";
                        cout<<"Color : "<<color<<endl;
                        cout<<"Features\n";
                        cout<<"ABS : "<<abs<<endl;
                        cout<<"Air Conditioner : "<<ac<<endl;
                        cout<<"-Estimated Price : "<<estimate<<endl;
                        cout<<"-Final Price : "<<double(estimate+tax)<<endl;
                        cout<<"\n-----------------------------------\n";
                }
        };
        int main()
        {
                automobile car;
                car.input();
                car.display();
        }
```

4. When do we make a virtual function "pure"? What are the implications of making a function as pure virtual function?

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation hence making it an abstract class. In such cases a pure virtual function (or abstract function) in C++ is a virtual function which is created for which we don’t have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

Example -

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

class Base
{
   int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class ingerits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

Output:

```
fun() called
```

The implications are as follows –

- Only class instance methods can be rendered as pure-virtual functions. Non-member functions and static member methods cannot be declared pure-virtual.
- The implication of declaring a pure-virtual function is that the class to which it is a member becomes an abstract data type (or abstract base class). This means you cannot instantiate an object of that class, you can only derive classes from it. Moreover, the derived classes must also implement the pure-virtual functions or they, too, become abstract data types. Only concrete classes that contain a complete implementation can be instantiated, although they can inherit implementations from their base classes
- Pure-virtual functions ensure that you do not instantiate base class objects that are not intended to be instantiated and that derived objects provide a specific implementation.

NOTE - We cannot create objects of abstract classes.

5. Draw a table to explain the visibility of the inherited members during the private, protected and public inheritance.

Accessibility in Public Inheritance

Accessibility	private variables	protected variables	public variables
Accessible from own class?	yes	Yes	yes
Accessible from derived class?	no	Yes	yes
Accessible from 2nd derived class?	no	Yes	yes

Accessibility in Protected Inheritance

Accessibility	private variables	protected variables	public variables
Accessible from own class?	yes	Yes	yes
Accessible from derived class?	no	Yes	yes (inherited as protected variables)
Accessible from 2nd derived class?	no	Yes	yes

Accessibility in Private Inheritance

Accessibility	private variables	protected variables	public variables
Accessible from own class?	Yes	Yes	yes
Accessible from derived class?	No	yes (inherited as private variables)	yes (inherited as private variables)
Accessible from 2nd derived class?	No	No	no

6. What is the advantage of using inheritance in C++?

DEFINITION:

Inheritance is the process of creating new classes, called derived classes, from existing classes or base classes. The derived class inherits all the capabilities of the base class, but can add embellishments and refinements of its own.

ADVANTAGES:

- One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.
- Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type of a method is superclass
- **Reusability** - facility to use public methods of base class without rewriting the same.
- **Extensibility** - extending the base class logic as per business logic of the derived class.
- **Data hiding** - base class can decide to keep some data private so that it cannot be altered by the derived class
- **Overriding** -With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

Since inception, C++ has been intended as an OOP language extending from its C predecessor. A major concept behind OOP is polymorphism. Polymorphism allows different objects of different types to conform to a common interface while uniquely defining behavior. One form of polymorphism which is critical in OOP is inheritance.

Inheritance allows objects of different types to share behavior and implementation, while also allowing these objects to specialize components of their behavior. Consider a simple example of an object like “Organism.” All organisms may allow the following interface:

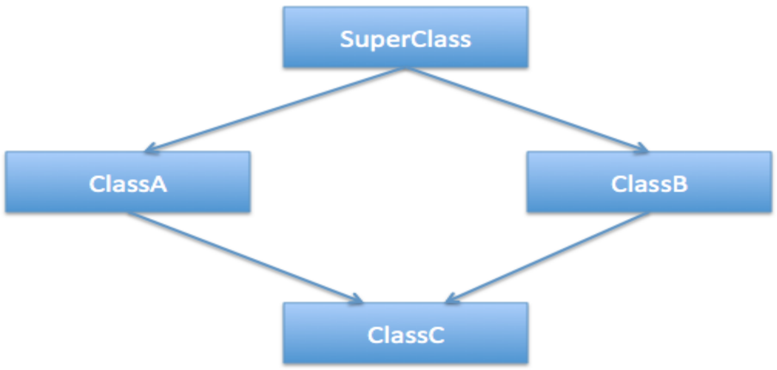
EXAMPLE:

Here we have two classes Teacher and MathTeacher the MathTeacher class inherits the Teacher class which means Teacher is a parent class and MathTeacher is a child class. The child class can use the property collegename of parent class.

Another important point to note is that when we create the object of child class it calls the constructor of child class and child class constructor automatically calls the constructor of base class.

```
#include <iostream>
using namespace std;
class Teacher {
public:
  Teacher(){
    cout<<"Hey Guys, I am a teacher"<<endl;
  }
  string collegeName = "Beginnersbook";
};
//This class inherits Teacher class
class MathTeacher: public Teacher {
public:
  MathTeacher(){
    cout<<"I am a Math Teacher"<<endl;
  }
  string mainSub = "Math";
  string name = "Negan";
};
int main() {
  MathTeacher obj;
  cout<<"Name: "<<obj.name<<endl;
  cout<<"College Name: "<<obj.collegeName<<endl;
  cout<<"Main Subject: "<<obj.mainSub<<endl;
  return 0;
}
```

7. **Describe the syntax of multiple inheritance and explain when we use such an inheritance?**



Syntax :

Class child_class_name : Access specifier 1st parent
, Access specifier 2nd parent

{

Data memebers

Data Functions

}

We use Multiple inheritance when we want a child class to inherit from more than one parent class.

Example : If we have a base class named as **LivingThing**. And this class has function as breathe(). The **Animal** and **Reptile** classes inherit from it. Only the **Animal** class overrides the method breathe(). The **Snake** class inherits from the **Animal** and **Reptile** classes. It overrides their methods. And hence it can be useful in such cases.

But, It is not prefered using multiple inheritance and use *virtual* inheritance instead.

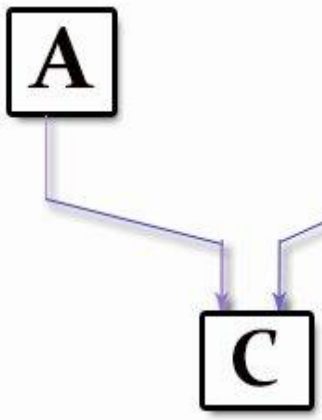
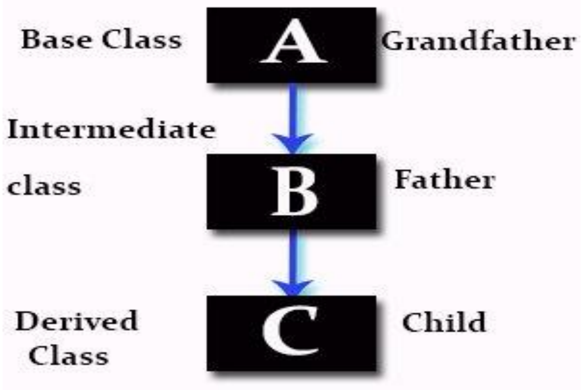
8. Develop a constructor using 'this' pointer in C++?

```
#include<iostream>
#include<string.h>
using namespace std;
class q8
{
   private:
   int rno;
          char name[10];
   public:
   q8(int rno,char *name)
   {
          this->rno=rno;
```

```
                strcpy(this->name,name);
                cout<<endl<<"Parameterized Constructor Called ('this' pointer used)";
        }
        void display()
        {
                cout<<endl<<"Registration number : "<<rno;
                cout<<endl<<"Name : "<<name<<endl;
        }
    };
    int main()
    {
                char a[]="S1",b[]="S2";
        q8 stud1= q8(1741000,a);
        cout<<endl<<"Student 1";
        stud1.display();
        q8 stud2= q8(1741000,b);
        cout<<endl<<"Student 2";
        stud2.display();
        return 0;
    }
```

9. Compare multiple inheritance and multilevel inheritance.

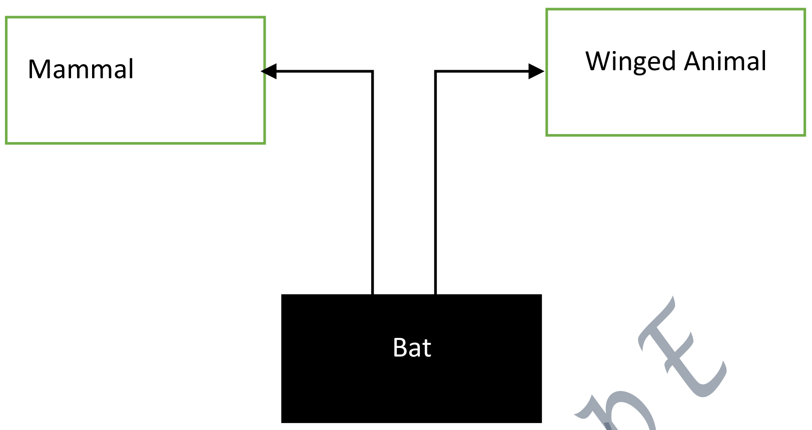
MULTIPLE	MULTILEVEL
Multiple Inheritance is an Inheritance type where a class inherits from more than one base class.	Multilevel Inheritance is an Inheritance type that inherits from a derived class, making that derived class a base class for a new class.
Multiple Inheritance is not widely used because it makes the system more complex.	Multilevel Inheritance is widely used.
Multiple Inheritance has two class levels namely, base class and derived class.	Multilevel Inheritance has three class levels namely, base class, intermediate class and derived class.

 MULTIPLE INHERITANCE	 Base Class Grandfather Intermediate class Father Derived Class Child Fig: Multilevel Inheritance

Syntax –
class base_class1
 	
{ properties; methods;};
 	
class base_class2
 	
{ properties; methods;};
 	
... ...
 	
... ...
 	
class base_classN
 	
{ properties; methods;};
 	
class derived_classname : visibility_mode base_class1, visibility_mode base_class2,... ,visibility_mode base_classN
 	
{ properties; methods;};
	Syntax -
	
	class base_classname
	
	{ properties; methods;};
	
	class intermediate_classname:visibility_mode base_classname
	
	{ properties; methods;};
	
	class child_classname:visibility_mode intermediate_classname
	
	{ properties; methods;};

10. Draw neat diagrams and compare the differences between multiple and multilevel inheritance.

Multiple Inheritance

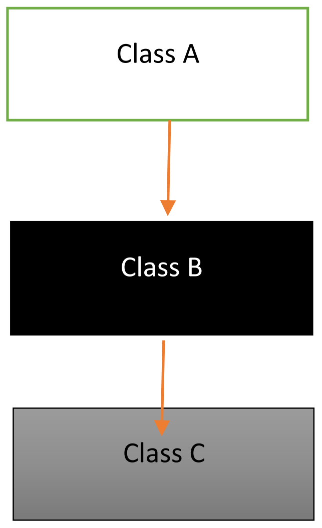


In C++ programming, a class can be derived from more than one parents. For example: A class Bat is derived from base classes Mammal and WingedAnimal It makes sense because bat is a mammal as well as a winged animal.

Program:-

```
#include <iostream>
using namespace std;
class Mammal {
  public:
    Mammal()
    {
      cout << "Mammals can give direct birth." << endl;
    }
};

class WingedAnimal {
  public:
    WingedAnimal()
    {
      cout << "Winged animal can flap." << endl;
    }
};

class Bat: public Mammal, public WingedAnimal {

};

int main()
{
   Bat b1;
   return 0;
}
```

Multilevel Inheritance



In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

Program

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

class A
{
   public:
     void display()
     {
        cout<<"Base class content.";
     }
};

class B : public A
{

};

class C : public B
{

};

int main()
{
   C obj;
   obj.display();
   return 0;
}
```

Output:

Base class content

In this program, class C is derived from class B (which is derived from base class A).The obj object of class C is defined in the main() function.When the display() function is called, display() in class A is executed. It's because there is no display() function in class C and class B.The compiler first looks for the display() function in class C. Since the function doesn't exist there, it looks for the function in class B (as C is derived from B).The function also doesn't exist in class B, so the compiler looks for it in class A (as B is derived from A).If display() function exists in C, the compiler overrides display() of class A (because of member function overriding).

11. Develop a c++ program to make use of constructors in derived classes.

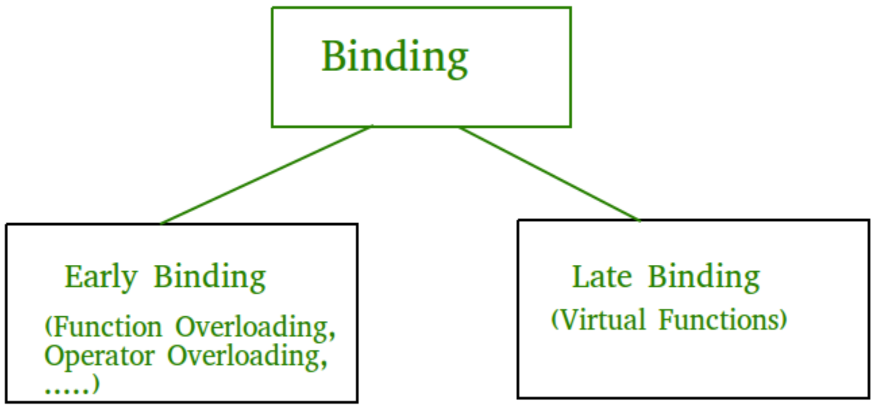
```
#include<iostream>
using namespace std;
class A
{
        protected:
                int a;
        public:
                A(int i)
                {
                        a=i;
                        cout<<"\nClass A  \n";
                }


};
class B:public A
{
        protected:
                int b;
        public:
                B(int x,int y):A(x)
                {
                        b=y;
                        cout<<"\nClass B \n";
                }
};
class C:public B
{
        protected:
                int c;
        public:
                C(int m,int n,int o):B(m,n)
                {
                c=o;
                cout<<"\nClass C\n";
                }
                void disp()
                {
                        cout<<"\n Your Salary is "<<a;
                        cout<<"\n Your Broher's Brother is "<<b;
                        cout<<"\n Your Sister's Salry is "<<c;
                }
                void compare()
                {
                        if((a>b)&&(a>c))
                        {
                                cout<<"\nYour are earning more. Cheer up
hardworking person\n";
                        }
                        else if((c>b)&&(a<c))
                        {
                                cout<<"\nWomen are ahead of men.Dont get
demotivated.You only need to power up by "<<((c-a)/(c+a))*100<<"%";

                        }
                        else if((b>c)&&(a<b))
                        {
                                cout<<"\n:)\n";
                                cout<<"\nDont get demotivated.You only need
to power up by "<<((b-a)/(b+a))*100<<"%";

                        }
                        else
                        {
                                cout<<"\nWork hard ! :)\n";
                        }
                }
};
int main()
{
        int a,b,c;
        cout<<"What's is your salary\t";
```

```
                cin>>a;
                cout<<"What's your brother's salary\t";
                cin>>b;
                cout<<"What's your sister's salary\t ";
                cin>>c;
                C c1(a,b,c);
                c1.disp();
                cout<<"\n COMPARE\n";
                c1.compare();
        }
```

12. Explain early binding and late binding with suitable example.

Early binding and Late binding in C++

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early Binding (compile-time time polymorphism)

As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.
By default early binding happens in C++. Late binding (discussed below) is achieved with the help of virtual keyword)

```
// CPP Program to illustrate early binding.
// Any normal function call (without virtual)
// is binded early. Here we have taken base
// and derived class example so that readers
// can easily compare and see difference in
// outputs.
#include<iostream>
using namespace std;
class Base
{
public:
    void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived;
    // The function call decided at
    // compile time (compiler sees type
    // of pointer and calls base class
    // function.
    bp->show();
    return 0;
}
```

13.Compare the following two statements differ in operation

cin>>c
cin.get(c);

Assuming c begin your single character.cin>>c and cin.get() both takes a character input. The difference being >> is an operator which returns istreamobject, whereas istream::get() returns traits::int_type(If there is input) or traits::eof()(If no input or input in bad format). You cannot take newline('\n') or Horizontal Tab('\t') as input using cin>>c. To do so you need to enable noskipwsflag or disable skipws flag.

14.Compare the input and output facilities in C++ differ from C language?

C++ approach differs in the following main points:

it is type-safe. There is no possibility to mismatch the format specifier and the argument type.
it exposes the complexity of C streams and allows the programmer to access, extend, and modify every part of it, which means
user-defined types can support I/O, and far more standard types come with I/O as well (both C and C++ have complex number types, but only C++ gives them I/O)
C++ streams can be attached to TCP sockets, memory-mapped files, and other custom sources/sinks, they can compress/decompress data on the fly (e.g. streaming a gzipped file), split and join, etc
each stream has its own locale and the facets can be modified as well: you can stream in a CSV file treating commas as whitespace. You can stream in from a UTF-8 file and out into a GB18030 one.
the exposed complexity is overwhelming to many programmers.

15. Both cin and getline() function can be used for reading a string. Comment.

In most program environments, the standard input by default is the keyboard, and the C++ stream object
defined to access it is cin(Standard Input).
For formatted input operations, cin is used together with the extraction operator, which is written as
>> (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted
data is stored. For example:
int age;
cin >> age;

getline() is a standard library function in C++ and is used to read a string or a line from input stream.
It is present in the <string> header.
So basically, what the getline function does is extracts characters from the input stream and appends it
to the string object until the delimiting character is encountered.

Since cin does not read complete string using spaces, stings terminates as you input space. While
cin.getline() – is used to read unformatted string (set of characters) from the standard input device
(keyboard). This function reads complete string until a give delimiter or null match.

In this program will read details name, address, about of a person and print in different lines, name
will contain spaces and dot, address will contain space, commas, and other special characters, same
about will also contains mixed characters. We will read all details using cin.getline() function and
print using cout.

cin.getline() example in C++ :

```
/*C++ program to read string using cin.getline().*/
#include  <iostream>
using namespace std;

//definitions for maximum length of variables
#define MAX_NAME_LENGTH     50
```

```
#define MAX_ADDRESS_LENGTH  100
#define MAX_ABOUT_LENGTH    200

using namespace std;

int main()
{
   char name[MAX_NAME_LENGTH],address[MAX_ADDRESS_LENGTH],about[MAX_ABOUT_LENGTH];

   cout << "Enter name: ";
   cin.getline(name,MAX_NAME_LENGTH);

   cout << "Enter address: ";
   cin.getline(address,MAX_ADDRESS_LENGTH);

   cout << "Enter about yourself (press # to complete): ";
   cin.getline(about,MAX_ABOUT_LENGTH,'#');    //# is a delimiter

   cout << "\nEntered details are:";
   cout << "Name: "  << name << endl;
   cout << "Address: " << address << endl;
   cout << "About: " << about << endl;

   return 0;
}
```

Thus, this is how both cin and getline() function can be used for reading a string.

16. What is the difference between get(char *) and get(void)?

Character pointer (char*) has overloaded output operator (operator<<), and prints underlying C-style string instead of adress of a pointer. This overload exists in order to support code like this:

```
std::cout << "some string";
```

“”some string type” type is actually const char*, without the overload it would print an adress of string instead of the string itself.**Void pointer (`void*`)**, or any other pointer type (AFAIK) doesn't provide this kind of overload, so the printed value is pointer address.

17. What are put() and get() functions in C++?

C++ gets()

The gets() function in C++ reads characters from stdin and stores them until a newline character is found or end of file occurs.
gets() prototype : char* gets(char* str);

The `gets()` function reads characters from stdin and stores them in *str* until a newline character or end of file is found.

It is defined in <cstdio> header file.

gets() Parameters

`str`: Pointer to an character array that stores the characters from stdin.

gets() Return value

- On success, the gets() function returns str
- On failure it returns null.
  - If the failure is caused due to end of file condition, it sets the eof indicator on stdin.
  - If the failure is caused due to some other error, it sets the error indicator on stdin.

Example: How gets() function works

```
#include <iostream>

#include <cstdio>

using namespace std;

int main()

{

   char str[100];

   cout << "Enter a string: ";

  gets(str);

  cout << "You entered: " << str;

  return 0;

}
```

When you run the program, a possible output will be:

```
Enter a string: Have a great day!

You entered: Have a great day!
```

C++ puts()

The puts() function in C++ writes a string to stdout.

puts() prototype : int puts(const char *str);

The `puts()` function takes a null terminated string str as its argument and writes it to `stdout`. The terminating null character '\0' is not written but it adds a newline character '\n' after writing the string.

It is defined in <cstdio> header file.

puts() Parameters

`str`: The string to be written.

puts() Return value

On success, the `puts()` function returns a non-negative integer. On failure it returns `EOF` and sets the error indicator on `stdout`.

Example: How puts() function work.

```
#include <cstdio>

int main()

{

  char str1[] = "Happy New Year";

 char str2[] = "Happy Birthday";

   puts(str1);




  *  Printed on new line since '/n' is added *

    puts(str2);

    return 0;

}
```

When you run the program, the output will be

```
Happy New Year

Happy Birthday
```

18. Discuss the syntax of setf() function.

We can use the setf() function to configure **formatting** for the **cout** object. We pass the setf() function arguments made up of ios_base class constants such as **ios_base::boolalpha** to display bool values as true or false instead of 1 or 0, and **ios_base::showpoint** to show a trailing decimal point.

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

int main()
{
  // Turn on showpos and scientific flags.
  cout.setf(ios::showpos);
  cout.setf(ios::scientific);

  cout << 123 << " " << 123.23 << " ";

  return 0;
}
```

```
OUTPUT:+123 +1.232300e+002
```