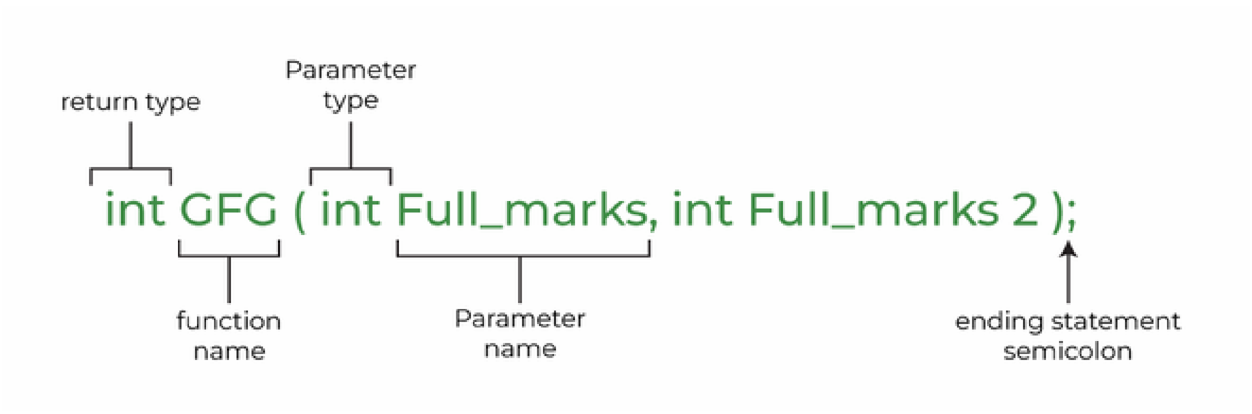
Lab-3

Functions in C++
A function is a set of statements that takes input, does some specific computation, and produces output. The idea is to put some commonly or repeatedly done tasks together to make a **function** so that instead of writing the same code again and again for different inputs, we can call this function.


Parameter type
return type
int GFG (int Full_marks, int Full_marks 2);
function name
Parameter name
ending statement semicolon

```
#include <iostream>
using namespace std;
int max(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
}
int main()
{
   int a = 10, b = 20;

   // Calling above function to find max of 'a' and 'b'
   int m = max(a, b);

   cout << "m is " << m;
   return 0;
}
```

Why Do We Need Functions?

For ***reducing code redundancy***.

Functions make code ***modular***.

Functions provide ***abstraction***.

Function declaration/ prototype

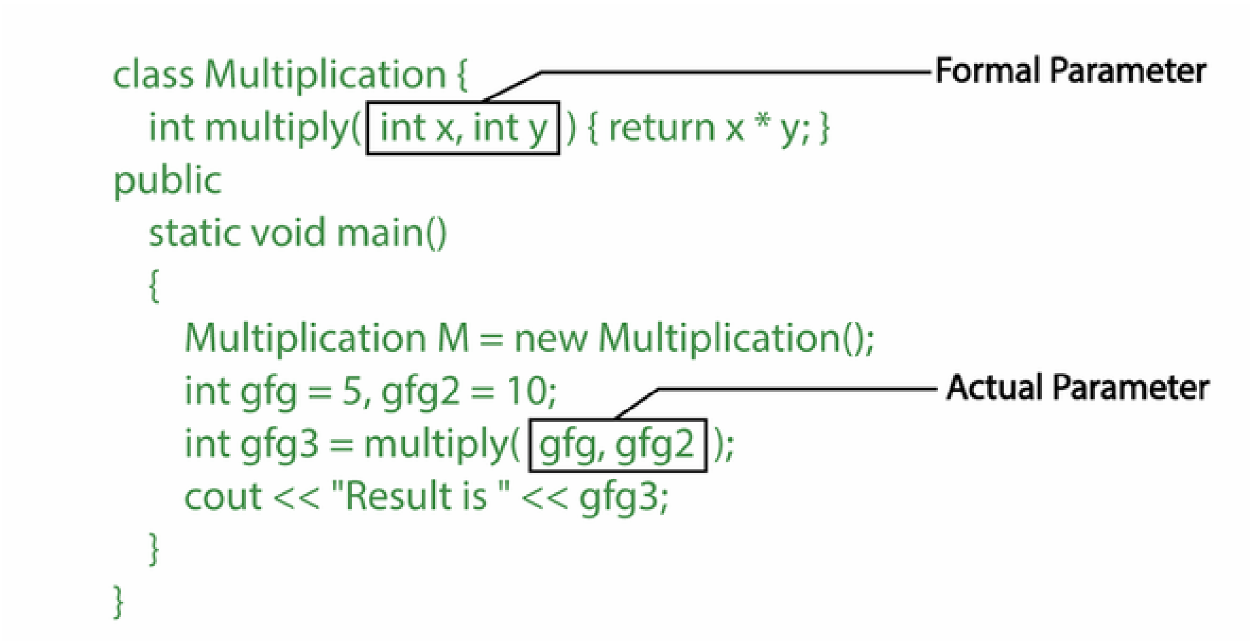
int max(int, int);

Types of Functions

1. User defined function
2. Library function ex: **For Example:** sqrt(), setw(), strcat(), etc.

Parameter Passing to Functions

The parameters passed to the function are called ***actual parameters***. For example, in the program below, 5 and 10 are actual parameters.
The parameters received by the function are called ***formal parameters***. For example, in the above program x and y are formal parameters.



```
class Multiplication {
   int multiply( int x, int y ) { return x * y; }
public
   static void main()
   {
      Multiplication M = new Multiplication();
      int gfg = 5, gfg2 = 10;
      int gfg3 = multiply( gfg, gfg2 );
      cout << "Result is " << gfg3;
   }
}
```

There are two most popular ways to pass parameters:

Pass by Value:

Pass by Reference:

From theory.

Code for pass by value:

```
// C++ Program to demonstrate function definition
#include <iostream>
```

```
using namespace std;

void fun(int x)
{
	// definition of
	// function
	x = 30;
}

int main()
{
	int x = 20;
	fun(x);
	cout << "x = " << x;
	return 0;
}
```

Functions Using Pointers

```
// C++ Program to demonstrate working of
// function using pointers
#include <iostream>
using namespace std;

void fun(int* ptr) { *ptr = 30; }

int main()
{
	int x = 20;
	fun(&x);
	cout << "x = " << x;

	return 0;
}
```

```
// C++ Program to demonstrate working of
// function using pointers
#include <iostream>
using namespace std;

void fun(int* ptr) { *ptr = 30; }

int main()
{
	int x = 20;
	fun(&x);
	cout << "x = " << x;
```

```
        return 0;
}
```

Types of Main Functions

1. Without parameters:

- CPP

```
// Without Parameters

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

2. With parameters:

- CPP

```
// With Parameters

int main(int argc, char* const argv[]) { ... return 0; }
```

C++ Recursion

```
recursionfunction()
{
   recursionfunction(); // calling self function
}
```

C++ Passing Array to Function

```
function_name(array_name[]); //passing array to function
#include <iostream>
using namespace std;
void printMin(int arr[5]);
int main()
{
        int ar[5] = { 30, 10, 20, 40, 50 };
        printMin(ar); // passing array to function
}
void printMin(int arr[5])
{
        int min = arr[0];
        for (int i = 0; i < 5; i++) {
                if (min > arr[i]) {
                        min = arr[i];
                }
```

```
	}
	cout << "Minimum element is: " << min << "\n";
}
```

C++ Overloading (Function)
If we create two or more members having the same name but different in number or type of parameters, it is known as C++ overloading. In C++, we can overload:

- *methods,*
- *constructors and*
- *indexed properties*

Types of overloading in C++ are:

- *Function overloading*
- *Operator overloading*

Example: changing number of arguments of add() method

```
// program of function overloading when number of arguments
// vary
#include <iostream>
using namespace std;
class Cal {
public:
	static int add(int a, int b) { return a + b; }
	static int add(int a, int b, int c)
	{
		return a + b + c;
	}
};
int main(void)
{
	Cal C; // class object declaration.
	cout << C.add(10, 20) << endl;
	cout << C.add(12, 20, 23);
	return 0;
}
```

Example: when the type of the arguments vary.

```
// Program of function overloading with different types of
// arguments.
#include <iostream>
using namespace std;
int mul(int, int);
float mul(float, int);
```

```
int mul(int a, int b) { return a * b; }
float mul(double x, int y) { return x * y; }
int main()
{
        int r1 = mul(6, 7);
        float r2 = mul(0.2, 3);
        cout << "r1 is : " << r1 << endl;
        cout << "r2 is : " << r2 << endl;
        return 0;
}
```

Function Overloading and Ambiguity
When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as *function overloading ambiguity.*
When the compiler shows the ambiguity error, the compiler does not run the program.
Causes of Ambiguity:

- *Type Conversion.*
- *Function with default arguments.*
- *Function with pass-by-reference.*

Type Conversion:-

```
#include <iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i) { cout << "Value of i is : " << i << endl; }
void fun(float j)
{
        cout << "Value of j is : " << j << endl;
}
int main()
{
        fun(12);
        fun(1.2);
        return 0;
}
```

Function with Default Arguments:-

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int, int);
void fun(int i) { cout << "Value of i is : " << i << endl; }
void fun(int a, int b = 9)
{
        cout << "Value of a is : " << a << endl;
        cout << "Value of b is : " << b << endl;
}
int main()
{
        fun(12);

        return 0;
}

// Code Submitted By Susobhan Akhuli
```

Function with Pass By Reference:-

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int&);
int main()
{
        int a = 10;
        fun(a); // error, which fun()?
        return 0;
}
void fun(int x) { cout << "Value of x is : " << x << endl; }
void fun(int& b)
{
        cout << "Value of b is : " << b << endl;
}
```

Friend Function

- A friend function is a special function in C++ which in spite of not being a member function of a class has the privilege to access private and protected data of a class.

Declaration of friend function in C++

Syntax:

```
class <class_name> {

        friend  <return_type>  <function_name>(argument/s);

};
```

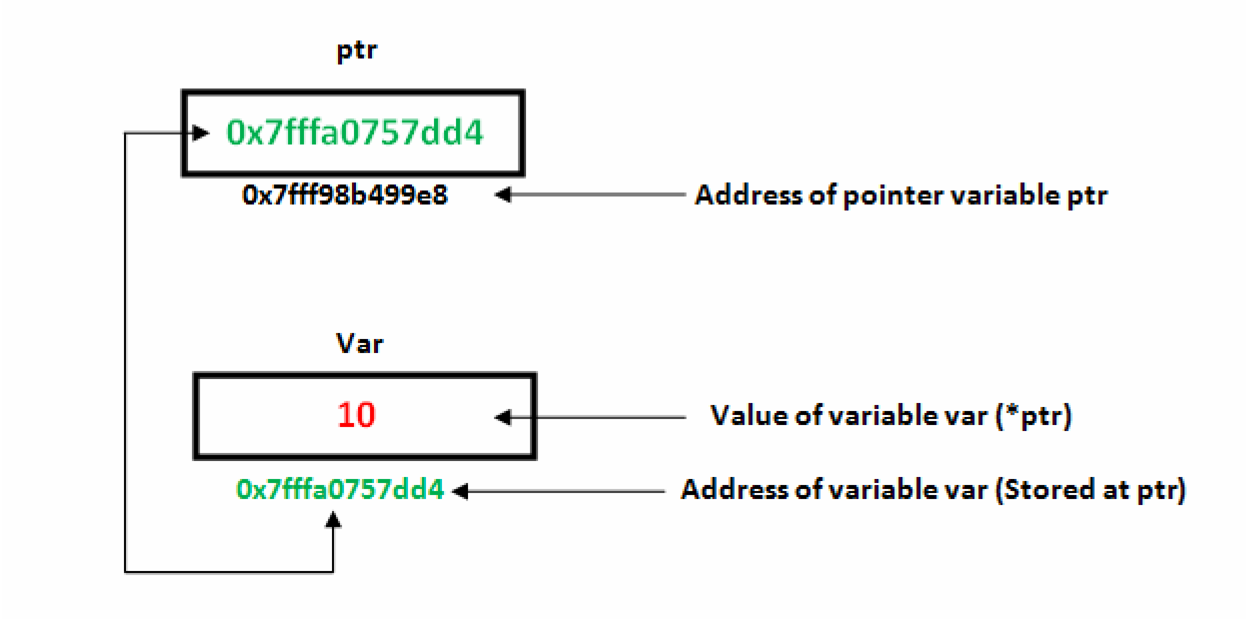
Example_1: Find the largest of two numbers using Friend Function

```
#include <iostream>
```

```
using namespace std;
class Largest {
        int a, b, m;

public:
        void set_data();
        friend void find_max(Largest);
};

void Largest::set_data()
{
        cout << "Enter the first number : ";
        cin >> a;
        cout << "\nEnter the second number : ";
        cin >> b;
}

void find_max(Largest t)
{
        if (t.a > t.b)
                t.m = t.a;
        else
                t.m = t.b;

        cout << "\nLargest number is " << t.m;
}

int main()
{
        Largest l;
        l.set_data();

        find_max(l);

        return 0;

}
```

C++ Pointers
Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures.
Syntax:
datatype *var_name;

int *ptr;   // ptr can point to an address which holds int data



ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

```
// C++ program to illustrate Pointers

#include <bits/stdc++.h>
using namespace std;
void geeks()
{
        int var = 20;

        // declare pointer variable
        int* ptr;

        // note that data type of ptr and var must be same
        ptr = &var;
        // assign the address of a variable to a pointer
        cout << "Value at ptr = " << ptr << "\n";
        cout << "Value at var = " << var << "\n";
        cout << "Value at *ptr = " << *ptr << "\n";
}
// Driver program
int main()
{
geeks();
return 0;
}
```

OOP
There are some basic concepts that act as the building blocks of OOPs i.e.

1. Class
2. Objects
3. Encapsulation
4. Abstraction
5. Polymorphism
6. Inheritance
7. Dynamic Binding
8. Message Passing

```
// C++ Program to show the syntax/working of Objects as a
// part of Object Oriented PProgramming
#include <iostream>
using namespace std;

class person {
        char name[20];
        int id;

public:
        void getdetails() {}
};

int main()
{

        person p1; // p1 is a object
        return 0;
}
```

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

```
#include <bits/stdc++.h>
using namespace std;
class Geeks {
	// Access specifier
public:
	// Data Members
	string geekname;
	// Member Functions()
	void printname() { cout << "Geekname is:" << geekname; }
};
int main()
{
	// Declare an object of class geeks
	Geeks obj1;
	// accessing data member
	obj1.geekname = "Abhi";
	// accessing member function
	obj1.printname();
	return 0;
}
```

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

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

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

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

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

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

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

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

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

Constructors

Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. There are 3 types of constructors:

- Default Constructors
- Parameterized Constructors
- Copy Constructors

```
// C++ program to demonstrate constructors
#include <bits/stdc++.h>
using namespace std;
class Geeks
{
	public:
	int id;

	//Default Constructor
	Geeks()
	{
		cout << "Default Constructor called" << endl;
		id=-1;
	}

	//Parameterized Constructor
```

```
        Geeks(int x)
        {
                cout <<"Parameterized Constructor called "<< endl;
                id=x;
        }
};
int main() {

        // obj1 will call Default Constructor
        Geeks obj1;
        cout <<"Geek id is: "<<obj1.id << endl;

        // obj2 will call Parameterized Constructor
        Geeks obj2(21);
        cout <<"Geek id is: " <<obj2.id << endl;
        return 0;
}
```

A **Copy Constructor** creates a new object, which is an exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes.
Syntax:
class-name (class-name &){}

Destructors
Destructor is another special member function that is called by the compiler when the scope of the object ends.

```
// C++ program to explain destructors
#include <bits/stdc++.h>
using namespace std;
class Geeks
{
        public:
        int id;

        //Definition for Destructor
        ~Geeks()
        {
                cout << "Destructor called for id: " << id <<endl;
        }
};

int main()
{
        Geeks obj1;
        obj1.id=7;
        int i = 0;
        while ( i < 5 )
        {
                Geeks obj2;
                obj2.id=i;
                i++;
        } // Scope for obj2 ends here

        return 0;
} // Scope for obj1 ends here
```

Why do we give semicolons at the end of class?

Many people might say that it's a basic syntax and we should give a semicolon at the end of the class as its rule defines in cpp. But the main reason why semi-colons are there at the end of the class is compiler checks if the user is trying to create an instance of the class at the end of it.

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

class Demo{
int a, b;
        public:
        Demo() // default constructor
        {
                cout << "Default Constructor" << endl;
        }
        Demo(int a, int b):a(a),b(b) //parameterised constructor
        {
                cout << "parameterized constructor -values" << a << " "<< b << endl;
        }

}instance;


int main() {

        return 0;
}

#include <iostream>
using namespace std;

class Demo{
        public:
```

```
	int a, b;
	Demo()
	{
		cout << "Default Constructor" << endl;
	}
	Demo(int a, int b):a(a),b(b)
	{
		cout << "parameterized Constructor values-" << a << " "<< b << endl;
	}



}instance(100,200);


int main() {

	return 0;
}
```

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

```
// C++ program to demonstrate public
// access modifier

#include<iostream>
using namespace std;

// class definition
class Circle
{
	public:
		double radius;

		double compute_area()
		{
			return 3.14*radius*radius;
		}

};

// main function
```

```
int main()
{
	Circle obj;

	// accessing public datamember outside class
	obj.radius = 5.5;

	cout << "Radius is: " << obj.radius << "\n";
	cout << "Area is: " << obj.compute_area();
	return 0;
}
```

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
	// private data member
	private:
		double radius;

	// public member function
	public:
		double compute_area()
		{ // Can member function access private
			// data member radius??
			return 3.14*radius*radius;
		}

};

// main function
int main()
{
	// creating object of the class
	Circle obj;

	// trying to access private data member
	// directly outside the class
```

```
        obj.radius = 1.5;

        cout << "Area is:" << obj.compute_area();
        return 0;
}
```

// **Corrected C++ program to demonstrate private**
// access modifier

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

class Circle
{
        // private data member
        private:
                double radius;

        // public member function
        public:
                void compute_area(double r)
                { // member function can access private
                        // data member radius
                        radius = r;

                        double area = 3.14*radius*radius;

                        cout << "Radius is: " << radius << endl;
                        cout << "Area is: " << area;
                }

};

// main function
int main()
{
        // creating object of the class
        Circle obj;

        // trying to access private data member
        // directly outside the class
        obj.compute_area(1.5);


        return 0;
}
```

Or

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

class Circle {
private:
    double radius;

public:
    // Setter function to set radius
    void setRadius(double r) {
        radius = r;
    }

    // Function to compute the area
    double compute_area() {
        return 3.14 * radius * radius;
    }
};

int main() {
    Circle obj;

    // Set radius using the setter function
    obj.setRadius(1.5);

    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;

// base class
class Parent
{
	// protected data members
	protected:
	int id_protected;

};

// sub class or derived class from public base class
class Child : public Parent
{
	public:
	void setId(int id)
	{

		// Child class is able to access the inherited
		// protected data members of base class

		id_protected = id;

	}

	void displayId()
	{
		cout << "id_protected is: " << id_protected << endl;
	}
};

// main function
int main() {

	Child obj1;

	// member function of the derived class can
	// access the protected data members of the base class

	obj1.setId(81);
	obj1.displayId();
	return 0;
}
```

Lab-4

Friend Class and Function in C++
A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.

Syntax:
friend class class_name;   // declared in the base class

```
// C++ Program to demonstrate the
// functioning of a friend class
#include <iostream>
using namespace std;
class GFG {
private:
	int private_variable;
protected:
	int protected_variable;
public:
	GFG()
	{
		private_variable = 10;
		protected_variable = 99;
	}
	// friend class declaration
	friend class F;
};

// Here, class F is declared as a
// friend inside class GFG. Therefore,
// F is a friend of class GFG. Class F
// can access the private members of
// class GFG.
class F {
public:
	void display(GFG& t)
	{
		cout << "The value of Private Variable = "
			<< t.private_variable << endl;
		cout << "The value of Protected Variable = "
			<< t.protected_variable;
	}};

// Driver code
int main()
{
	GFG g;
	F fri;
	fri.display(g);
	return 0;
}
```

Friend Function

Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are the non-member functions that can access and manipulate the private and protected members of the class for they are declared as friends.

A friend function can be:

1. **A global function**
2. **A member function of another class**

Syntax:

```
friend return_type function_name (arguments);    // for a global function
	or
friend return_type class_name::function_name (arguments);    // for a member function of
another class
```

```
// C++ program to create a global function as a friend
// function of some class
#include <iostream>
using namespace std;

class base {
private:
	int private_variable;

protected:
	int protected_variable;

public:
	base()
	{
		private_variable = 10;
		protected_variable = 99;
	}
	// friend function declaration
	friend void friendFunction(base& obj);
};


// friend function definition
void friendFunction(base& obj)
{
	cout << "Private Variable: " << obj.private_variable
		<< endl;
	cout << "Protected Variable: " << obj.protected_variable;
}

// driver code
int main()
{
	base object1;
	friendFunction(object1);

	return 0;
}


// C++ program to create a member function of another class
// as a friend function
#include <iostream>
using namespace std;

class base; // forward definition needed
// another class in which function is declared
class anotherClass {
public:
	void memberFunction(base& obj);
};

// base class for which friend is declared
class base {
private:
	int private_variable;

protected:
	int protected_variable;

public:
	base()
```

```
        {
                private_variable = 10;
                protected_variable = 99;
        }

        // friend function declaration
        friend void anotherClass::memberFunction(base&);
};

// friend function definition
void anotherClass::memberFunction(base& obj)
{
        cout << "Private Variable: " << obj.private_variable
                << endl;
        cout << "Protected Variable: " << obj.protected_variable;
}

// driver code
int main()
{
        base object1;
        anotherClass object2;
        object2.memberFunction(object1);

        return 0;
}
```

A Function Friendly to Multiple Classes

```
// C++ Program to demonstrate
// how friend functions work as
// a bridge between the classes
#include <iostream>
using namespace std;

// Forward declaration
class ABC;

class XYZ {
        int x;

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

```
          friend void max(XYZ, ABC);
};

class ABC {
          int y;

public:
          void set_data(int a)
          {
          y = a;
          }

          friend void max(XYZ, ABC);
};

void max(XYZ t1, ABC t2)
{
          if (t1.x > t2.y)
                    cout << t1.x;
          else
                    cout << t2.y;
}

// Driver code
int main()
{
          ABC _abc;
          XYZ _xyz;
          _xyz.set_data(20);
          _abc.set_data(35);

          // calling friend function
          max(_xyz, _abc);
          return 0;
}
```

```
#include<iostream>
using namespace std;
class B; //forward declaration.
class A
```

```
{
   int x;
   public:
       void setdata (int i)
         {
           x=i;
         }
   friend void max (A, B); //friend function.
} ;
class B
{
    int y;
    public:
        void setdata (int i)
          {
            y=i;
          }
    friend void max (A, B);
};
void max (A a, B b)
{
  if (a.x >= b.y)
       std:: cout<< a.x << std::endl;
  else
       std::cout<< b.y << std::endl;
}
 int main ()
{
  A a;
  B b;
   a. setdata (10);
   b. setdata (20);
   max (a, b);
   return 0;
}
```

Advantages of Friend Functions

- A friend function is able to access members without the need of inheriting the class.
- The friend function acts as a bridge between two classes by accessing their private data.
- It can be used to increase the versatility of overloaded operators.
- It can be declared either in the public or private or protected part of the class.

Disadvantages of Friend Functions

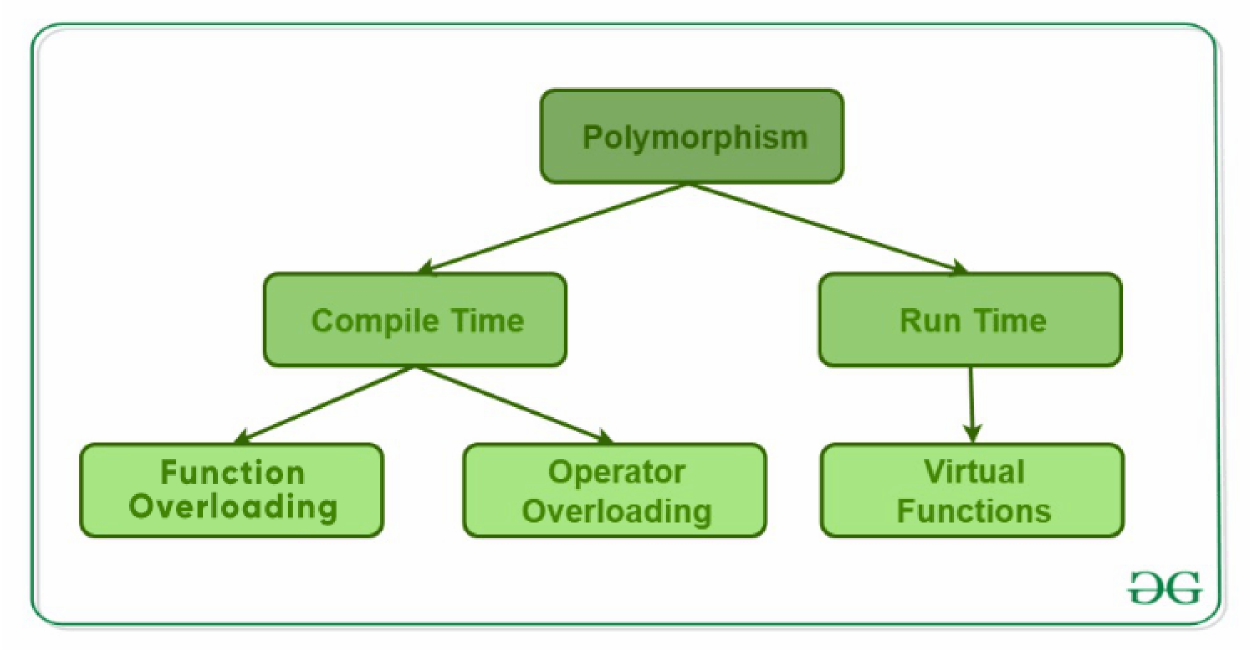
- Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
- Friend functions cannot do any run-time polymorphism in their members.

Polymorphism

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee.

Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**



A. Function Overloading

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Geeks {
public:
	// Function with 1 int parameter
	void func(int x)
	{
		cout << "value of x is " << x << endl;
	}

	// Function with same name but
	// 1 double parameter
	void func(double x)
	{
		cout << "value of x is " << x << endl;
	}

	// Function with same name and
	// 2 int parameters
	void func(int x, int y)
	{
		cout << "value of x and y is " << x << ", " << y
			<< endl;
	}
};

// Driver code
int main()
{
	Geeks obj1;

	// Function being called depends
	// on the parameters passed
	// func() is called with int value
	obj1.func(7);

	// func() is called with double value
	obj1.func(9.132);

	// func() is called with 2 int values
	obj1.func(85, 64);
	return 0;
}
```

B. Operator Overloading

```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#include <iostream>
using namespace std;

class Complex {
private:
	int real, imag;
public:
	Complex(int r = 0, int i = 0)
	{
		real = r;
		imag = i;
	}

	// This is automatically called
	// when '+' is used with between
	// two Complex objects
	Complex operator+(Complex const& obj)
	{
		Complex res;
		res.real = real + obj.real;
		res.imag = imag + obj.imag;
		return res;
	}
	void print() { cout << real << " + i" << imag << endl; }
};

// Driver code
int main()
{
	Complex c1(10, 5), c2(2, 4);

	// An example call to "operator+"
	Complex c3 = c1 + c2;
	c3.print();
}
```

B. Virtual Function

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
	// virtual function
	virtual void display()
	{
		cout << "Called virtual Base Class function"
			<< "\n\n";
	}

	void print()
	{
		cout << "Called GFG_Base print function"
			<< "\n\n";
	}
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {

public:
	void display()
	{
		cout << "Called GFG_Child Display Function"
			<< "\n\n";
	}

	void print()
	{
		cout << "Called GFG_Child print Function"
			<< "\n\n";
	}
};

// Driver code
int main()
{
	// Create a reference of class GFG_Base
	GFG_Base* base;

	GFG_Child child;

	base = &child;

	// This will call the virtual function
	base->GFG_Base::display();

	// this will call the non-virtual function
	base->print();
}
```

```
// C++ program for virtual function overriding
#include <bits/stdc++.h>
using namespace std;

class base {
public:
	virtual void print()
	{
		cout << "print base class" << endl;
	}

	void show() { cout << "show base class" << endl; }
};

class derived : public base {
public:
	// print () is already virtual function in
	// derived class, we could also declared as
	// virtual void print () explicitly
	void print() { cout << "print derived class" << endl; }

	void show() { cout << "show derived class" << endl; }
};

// Driver code
int main()
{
	base* bptr;
	derived d;
	bptr = &d;

	// Virtual function, binded at
	// runtime (Runtime polymorphism)
	bptr->print();

	// Non-virtual function, binded
	// at compile time
	bptr->show();

	return 0;
}
```

Write a C++ code to find the height CGPA obtainer of two departments using friend function.