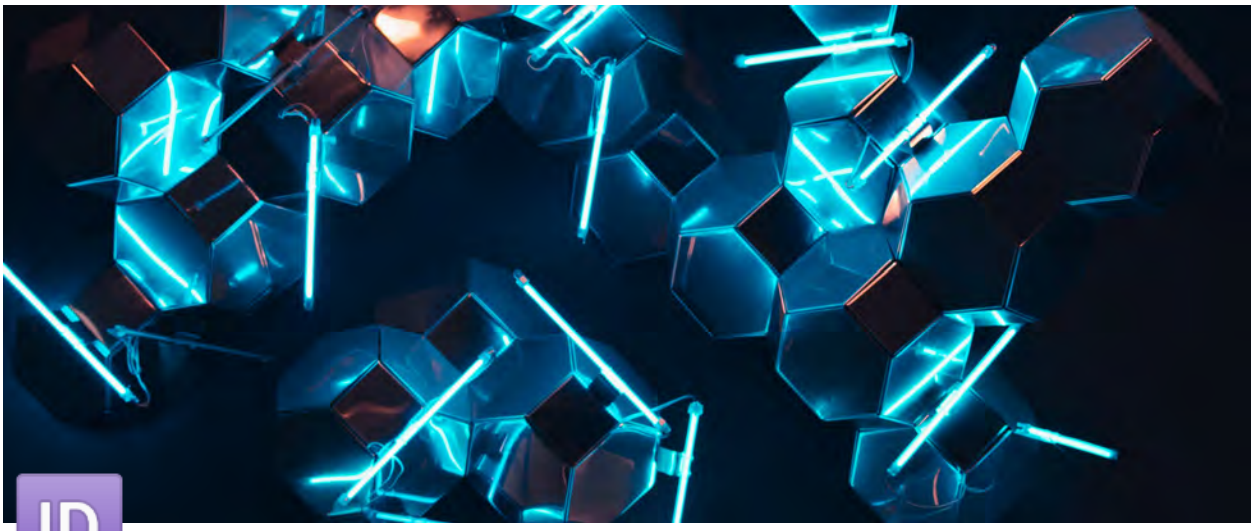
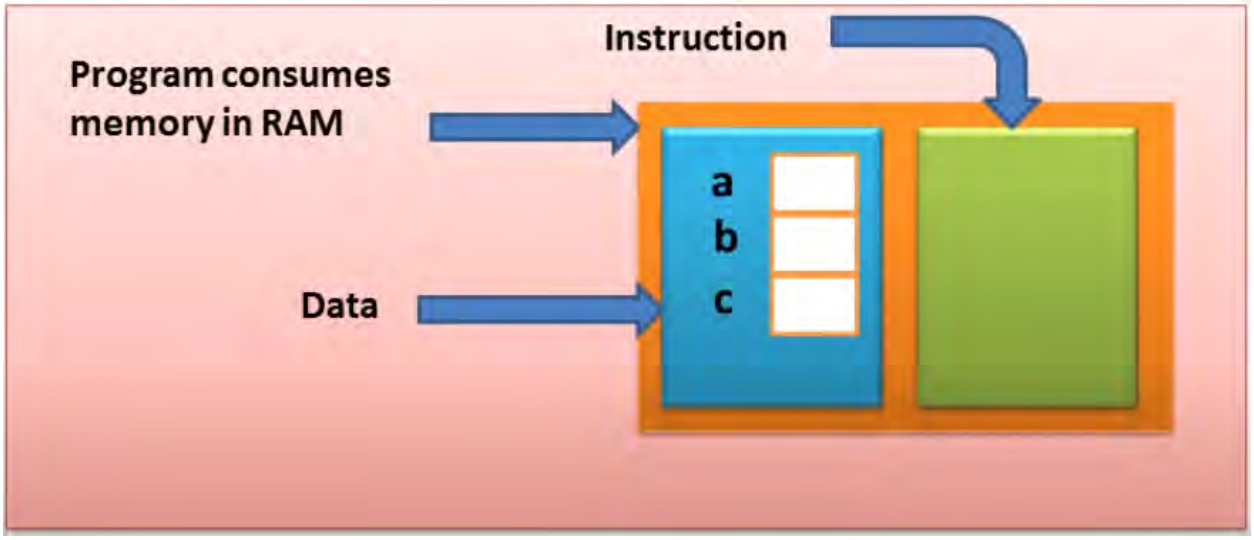


Identifiers

Constant:

- Any information is constant
- Data = Information = Constant
- Primary: integer, real, character
- Secondary: array, string, pointer, union, structure, enumerator, class
- Integer Constant: 23, -341, 0, 5
- Real Constant: 3.4, -0.6, 3.0
- Character Constant: ‘a’, ‘A’, ‘+’, ‘2’, ‘ ‘



Variables:

- Variables are the names of memory locations where we store data
- Variable name is any combination of alphabet, digit, and underscore
- Variable name can not start with digit

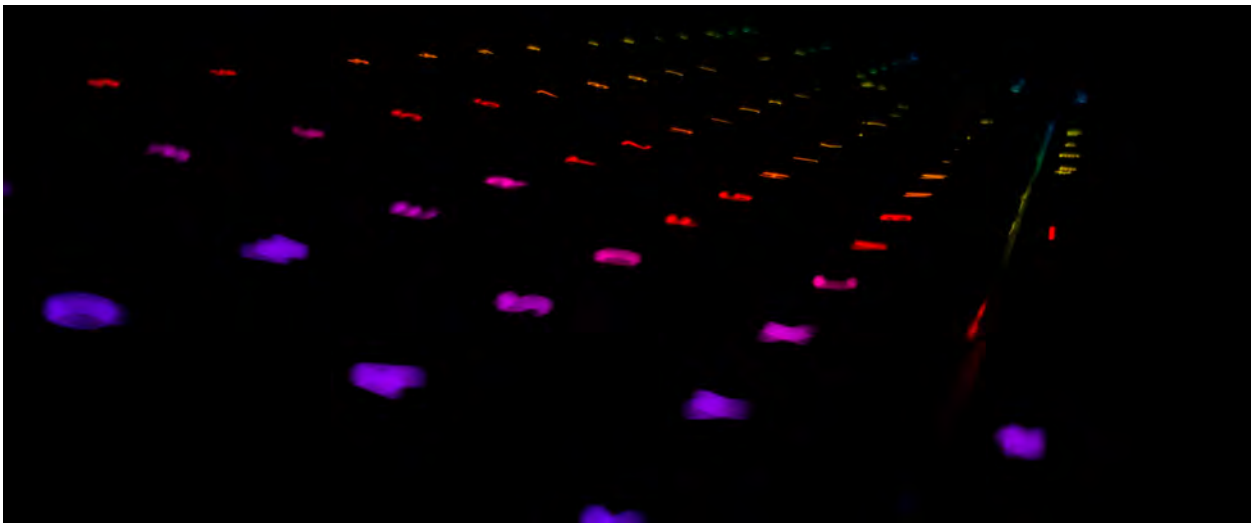
Keywords:

auto	double	int	struct	asm	private
break	else	long	switch	catch	public
case	enum	register	typedef	class	protected
char	extern	return	union	delete	template
const	float	short	unsigned	friend	this
continue	for	signed	void	inline	throw
default	goto	sizeof	volatile	new	try
do	if	static	while	operator	virtual

Data Types:

- `int`
- `char`
- `float`
- `double`
- `void`

- Unlike C, you can declare variables even after action statements



↔

Input/Output

Output Instruction:

- In C, standard output device is monitor and `printf()` is used to send data to monitor
- `printf()` is predefined function
- In C++, we can use `cout` to send data to monitor
- `cout` is a predefined object
- The operator `<<` is called the insertion or put to operator

- C: `printf("Hi")`;
- C++: `cout << "Hi";`

- C: `printf("Sum of %d and %d is %d", a, b, c);`
- C++: `cout << "Sum of " << a << " and " << b << " is " << c;`

- C: `printf(”%d”, a+b);`
- C++: `cout << a+b;`

Input Instruction:

- In C, standard input device is keyboard and `scanf()` is used to receive data from keyboard
- `scanf()` is a predefined function
- In C++, we can use `cin` to input data from keyboard
- The identifier `cin` is a predefined object in C++
- The operator `>>` is known as extraction or get from operator

- C: `scanf(”%d”, &a);`
- C++: `cin >> a;`

- C: `scanf(”%d %d”, &a, &b);`
- C++: `cin >> a >> b;`

Remember:

- According to the ANSI standards for C language, explicit declaration of function is recommended but not mandatory
- ANSI standards for C++ language says explicit declaration of function is compulsory

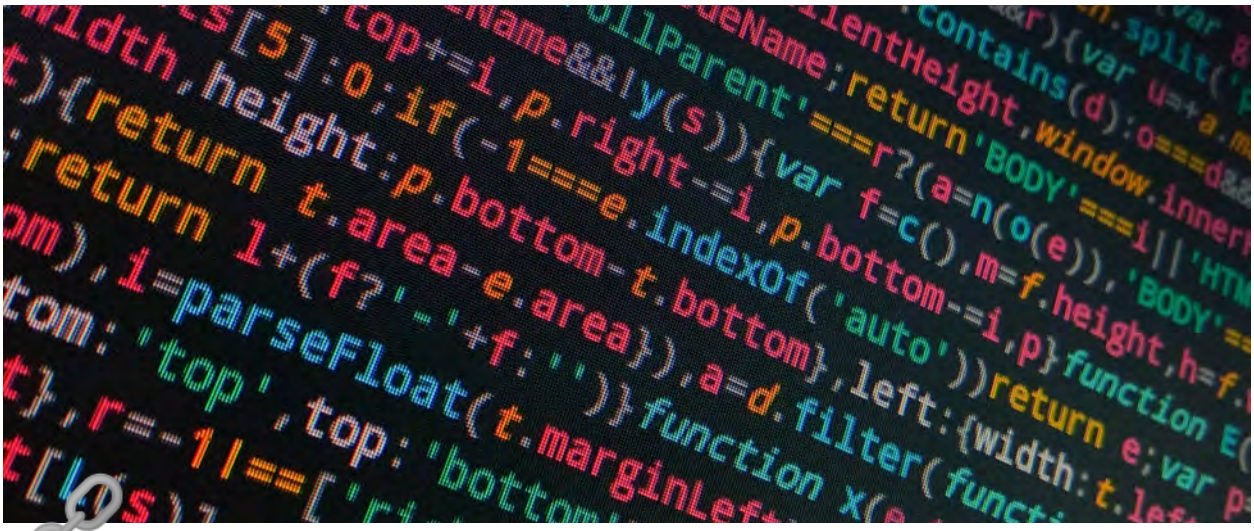
`iostream`:

- We need to include header file `iostream.h`, it contains declarations for the identifier `cout` and the operation `<<`. And also for the identifier `cin` and operator `>>`

- Inserting `endl` into the output stream causes the screen cursor to move to the beginning of the next line
- `endl` is a manipulator and it is declared in `iostream.h`

Sample Program

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

void main(){
  clrscr();
  int x;
  cout << "Enter a number: " << endl;
  cin >> x;
  int s = x * x;
  cout << "Square of " << x << "is" << s;
  getch();
}
```



Reference Variables

Types of variables:

- Ordinary (`int x = 5;`)
- Pointer (`int *p;`)
- Reference

Reference Variable:

- `int &y = x;` - this is not address of operator as it is to the left of the equation, not right
- It must be initialized
- y++ is same as x++
- y is another name for x
- A little different than pointer variables
- Reference means address

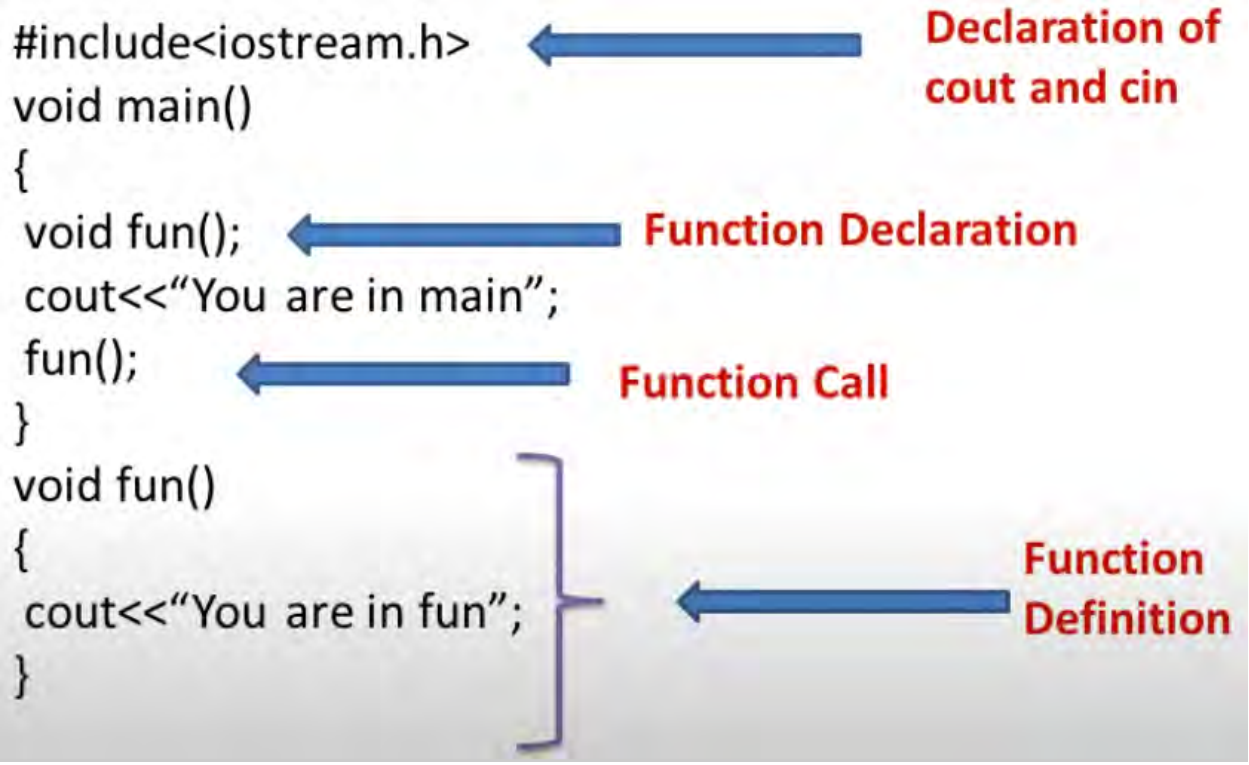
- Reference variable is an internal pointer
- Declaration of Reference variable is preceded with & symbol (but do not read it as address of)
- Reference variable must be initialized during declaration
- It can be initialized with already declared variables only
- Reference variables can not be updated, unlike pointer variables

Functions

What is Function?

- Function is a block of code performing a unit task
- Function has a name, return type, and arguments
- Function is a way to achieve modularization
- Function are pre-defined and user-defined
- Pre-defined functions are declared in header files and defined in library files

Definition, Declaration, and Call:



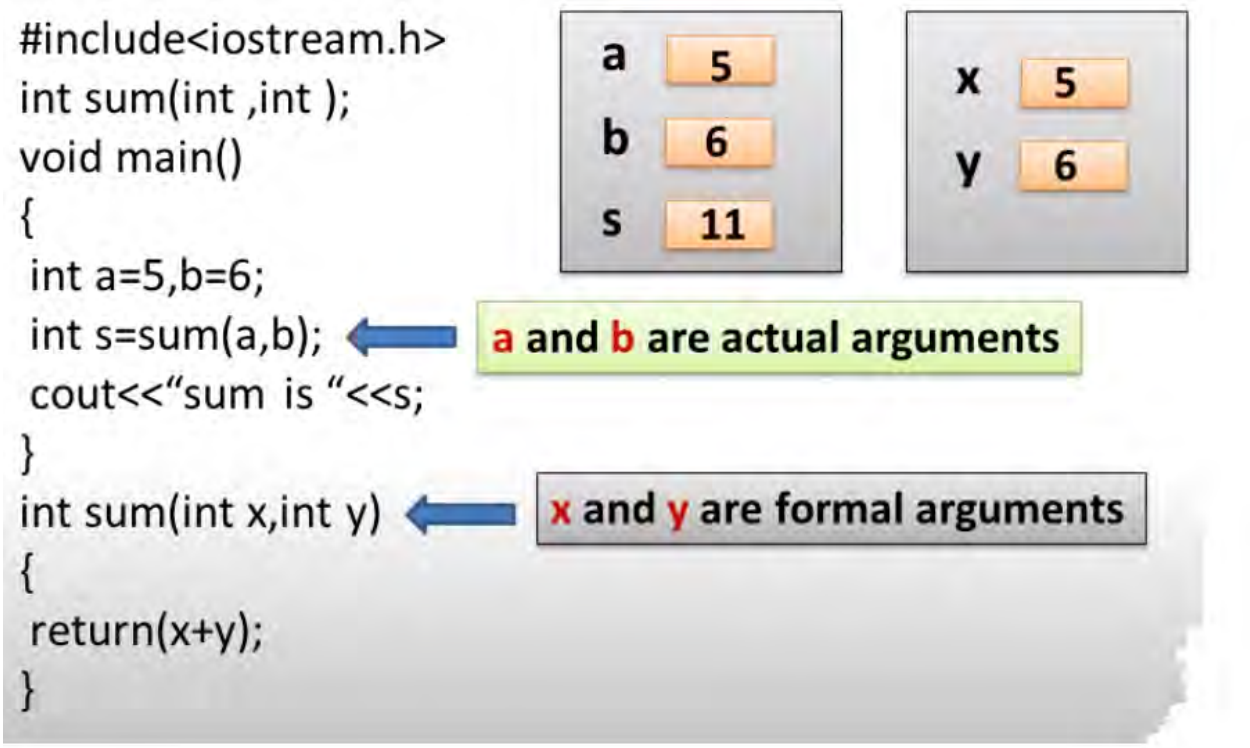
Declaration:

- Function Declaration is also known as function prototype
- Functions need to be declared before use (just like variables)
- Functions can be declared locally or globally
- `return_type functionName(argument_list);`
- Function definition is a block of code

Ways to define a function:

- takes nothing, returns nothing
- takes something, returns nothing
- takes nothing, return something
- takes something, returns something

Formal and Actual Arguments:



Types of formal arguments:

- Ordinary variables of any type
- Pointer variables
- Reference variables

Call by value:

```
#include<iostream.h>
int sum(int ,int );
void main()
{
 int a=5,b=6;
 int s=sum(a,b);
 cout<<"sum is "<<s;
}
int sum(int x,int y)
{
 return(x+y);
}
```

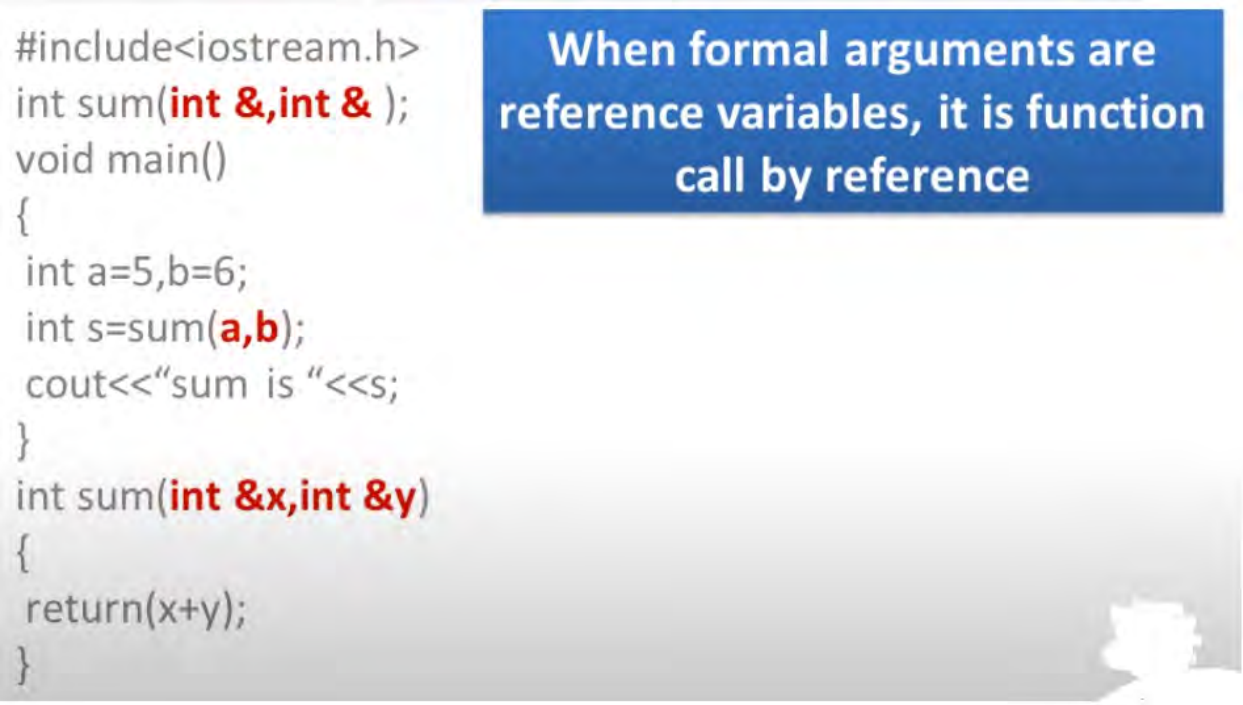
When formal arguments are ordinary variables, it is function call by value

Call by address:

```
#include<iostream.h>
int sum(int *,int * );
void main()
{
 int a=5,b=6;
 int s=sum(&a,&b);
 cout<<"sum is "<<s;
}
int sum(int *p,int *q)
{
 return(*p+*q);
}
```

When formal arguments are pointer variables, it is function call by address

Call by reference:



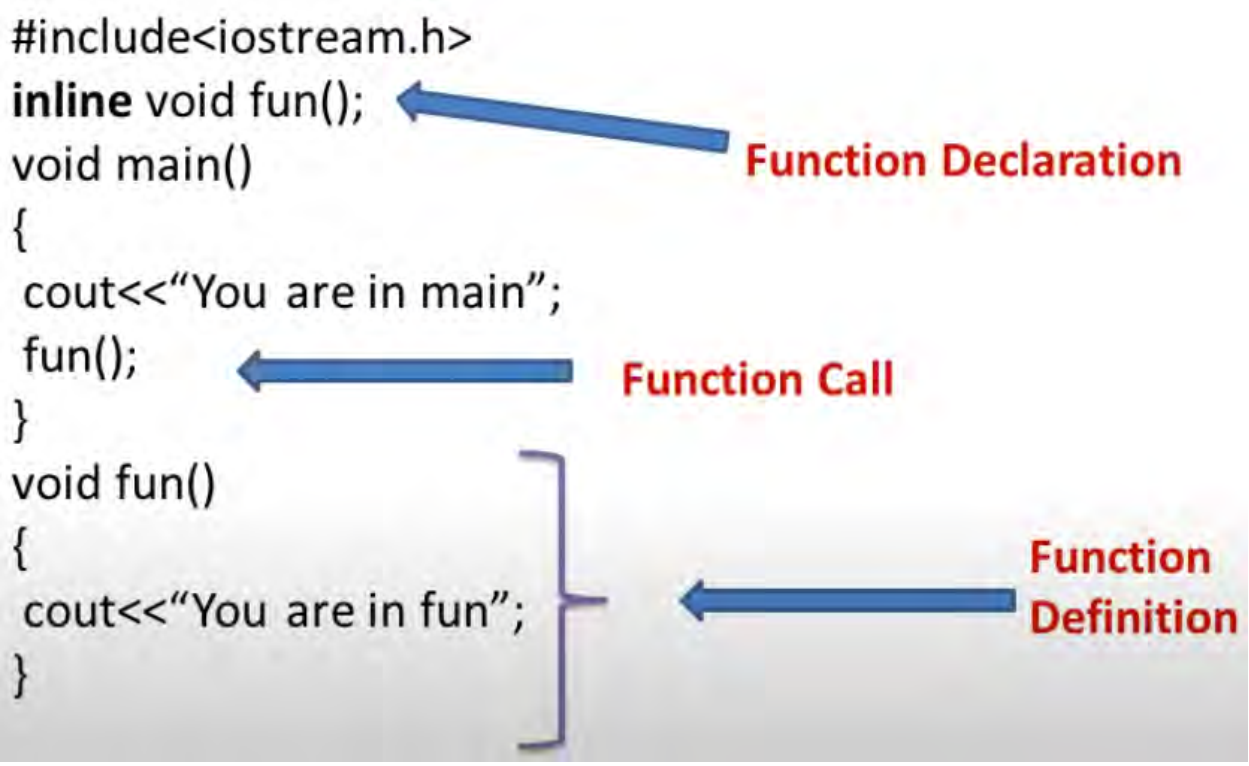
- Using functions saves memory but it consumes time
- But when function is small, it is worthless to spend so much extra time in such tasks in cost of saving comparatively small space

Inline Function:

- To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function*
- An inline function is a function that is expanded in line when it is invoked
- Compiler replaces the function call with the corresponding function code
- Inline is a request, not a command

- The benefit of speed of inline functions reduces as the function grows in size
- So the compiler may ignore the request in some situations such as function containing loops, switch, goto; function with recursion, and function containing static variable

Example



Default Arguments:

```
/* Default Arguments */
#include<conio.h>
#include<iostream.h>
int add(int,int,int=0);
void main()
{
 clrscr();
 int a,b;
 cout<<"Enter two numbers";
 cin>>a>>b;
 cout<<"Sum is "<<add(a,b);
 int c;
 cout<<"\nEnter three numbers";
 cin>>a>>b>>c;
 cout<<"Sum is "<<add(a,b,c);
 getch();
}
int add(int x,int y,int z)
{
 return(x+y+z);
}
```

Function Overloading:

- It is when multiple functions performing similar operation can have same name
- Example: Different functions calculating area of different shapes can have same name
- It is a way to implement **Polymorphism**

```
#include <iostream>

int area(int, int);
float area(int);

int main(){
  int r;
  cout << "Enter radius of circle: ";
  cin >> r;
  float c = area(r);
  cout << "Area of circle: " << c << endl;
```

```
  int l, b, a;
  cout << "\nEnter length and breadth of rectangle: ";
  cin >> l >> b;
  a = area(l, b);
  cout << "Area of rectangle: " << a;
  return 0;
}

float area(int r){
  return(3.14 * r * r);
}

int area(int l, int b){
  return(l * b);
}
```

How Function Overloading is resolved?

- First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions
- If no exact match is found, C++ tries to find a match through promotion:
  - char, unsigned char, and short is promoted to an int
  - float is promoted to double
- If no promotion is found, C++ tries to find a match through standard conversion

Structure

- Structure is a collection of dissimilar elements
- Structure is a way to group variables
- Structure is used to create data type
- In C, we used to write `struct` whenever we defined a new variable of that structure type
  - Example: `struct BookInfo book;`
- However, in C++, writing `struct` is optional
  - Example: `BookInfo book;`
- In C, we can only define variables inside structure
- In C++, we can also define functions inside definition of structure
- This is used to achieve *Encapsulation*

Example:

```
#include <iostream>

struct book{
  int bookid;
  char title[20];
  float price;
  void input(){
    cout << "Enter bookid, title, price: ";
    cin >> bookid >> title >> price;        // no need to use dot '.' operator
  }
  void display(){
    cout << bookid << title << price;
  }
};

int main(){
  book b1;        // here, b1 is an object
  b1.input();
  b1.display();
  return 0;
}
```

Data Security:

Example:

```
#include <iostream>

struct book{
  int bookid;
  char title[20];
  float price;
  void input(){
    cout << "Enter bookid, title, price: ";
    cin >> bookid >> title >> price;        // no need to use dot '.' operator
    if(bookid < 0){
      bookid = -1 * bookid;
    }
  }
  void display(){
    cout << bookid << title << price;
  }
};

int main(){
```

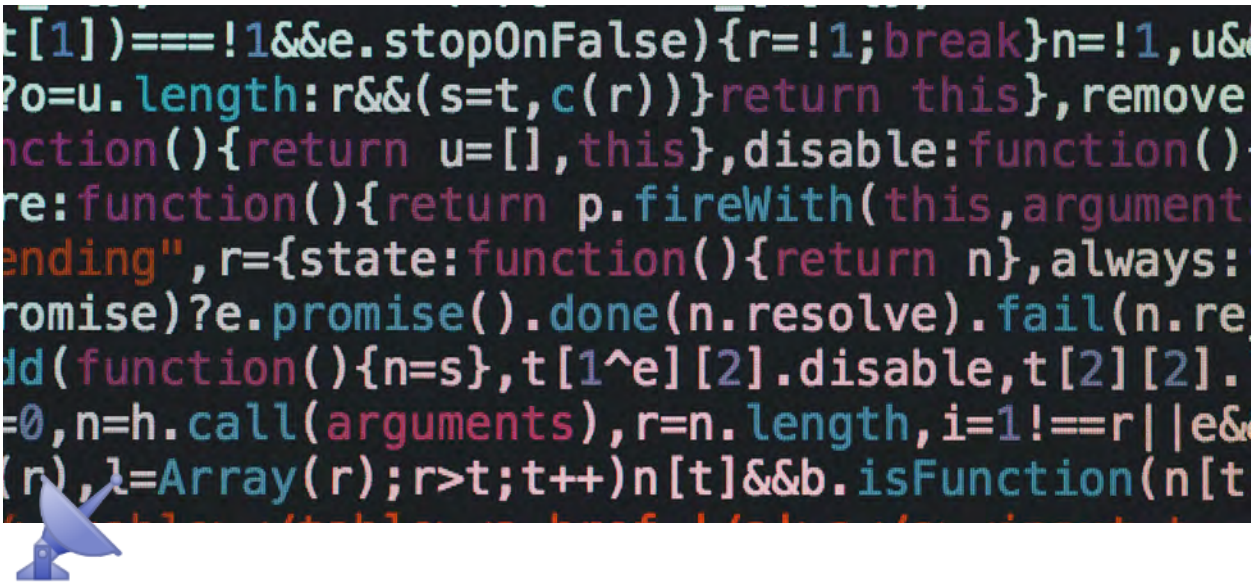
```
  book b1;        // here, b1 is an object
  b1.input();
  b1.display();
  return 0;
}
```

- In this example, if user enters a negative value of bookid, it will be converted to positive
- But what if coder does not calls input function and himself define book id by writing: `b1.bookid = -100;`
- This will store negative value in bookid
- To prevent this, we can define access rights of each variable and functions
- There are three types: `public` , `private` , and `protected`
- In this example, we will use `private` to ensure variables are only accessed inside this structure
- However, functions will we accessed by user, so they must be defined `public`

Example:

```
#include <iostream>

struct book{
  private:
  int bookid;
  char title[20];
  float price;

  public:
  void input(){
    cout << "Enter bookid, title, price: ";
    cin >> bookid >> title >> price;        // no need to use dot '.' operator
    if(bookid < 0){
      bookid = -1 * bookid;
    }
  }
  void display(){
    cout << bookid << title << price;
  }
```

```
};

int main(){
  book b1;        // here, b1 is an object
  b1.input();
  b1.display();
  return 0;
}
```



Classes and Objects

Class and Structure:

- The only difference between structure and class is that:
  - the members of structure are by default public
  - the members of class are by default private

Example:

- Using Structure

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

struct Complex{
  private:
    int a, b;
  public:
    void set_data(int x, int y){
```

```
      a = x;
      b = y;
    }
    void show_data(){
      cout << a << b;
    }
};

int main(){
  Complex c1;       // here, c1 is a variable
  c1.set_data(3, 4);
  c1.show_data();
  return 0;
}
```

- If we didn’t used private, all members were public and thus can be accessed anywhere

- Using Class

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

class Complex{
    int a, b;
  public:
    void set_data(int x, int y){
      a = x;
      b = y;
    }
    void show_data(){
      cout << a << b;
    }
};

int main(){
  Complex c1;       // here, c1 is an object
  c1.set_data(3, 4);
  c1.show_data();
  return 0;
}
```

- Here, all members are by default private
- But only above functions are public as we have explicitly done that

- When functions are defined inside a class, it is by default *inline*
- So, to not make it inline, these functions can also be defined outside the class but its prototype must be defined inside class with a little difference

Example:

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

class Complex{
    int a, b;
  public:
    void set_data(int, int);
    void show_data(){
      cout << a << b;
    }
};

void Complex::set_data(int x, int y){
      a = x;
      b = y;
    }

int main(){
  Complex c1;       // here, c1 is an object
  c1.set_data(3, 4);
  c1.show_data();
  return 0;
}
```

- The `::` is called membership operator

Passing and Returning Objects in a function:

Example:

- addition of two complex numbers
- c1 = 3 + 4i
- c2 = 5 + 6i

- c1 + c2 = 8 + 10i

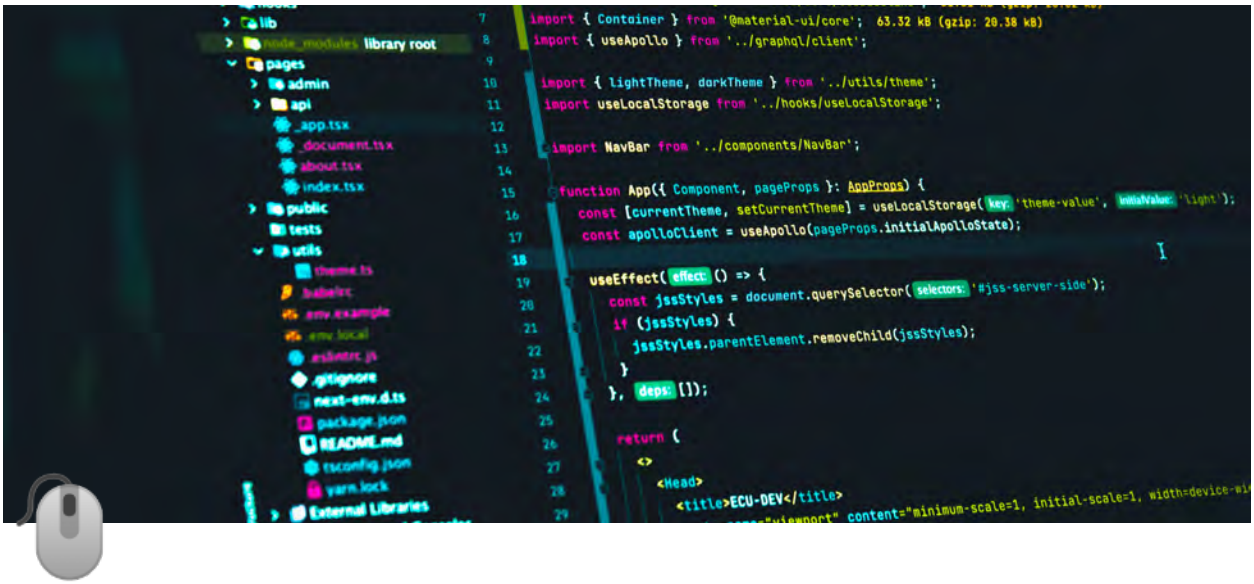
```
#include <iostream>
using namespace std;

class Complex{
  int a, b;

  public:
  void set_data(int x, int y){
    a = x;
    b = y;
  }
  void show_data(){
    cout << a << " " << b << endl;
  }
  Complex add(Complex c){
    Complex temp;
    temp.a = a + c.a;
    temp.b = b + c.b;
    return temp;
  }
};

int main(){
  Complex c1, c2, c3;
  c1.set_data(3, 4);
  c2.set_data(5, 6);
  c3 = c1.add(c2);
  c3.show_data();
  return 0;
}
```

Technical Description:

- Class is the description of an object
- Object is an instance of a class

- Instance member variable:
  - Attributes, data members, fields, properties
- Instance member function:

- Methods, procedures, actions, operations, services



Static Members

Static Local Variable:

- Concept as it is taken from C
- They are by default initialized to zero
- Their lifetime is throughout the program

Static Member Variable:

- Declared inside the class body
- Also known as class member variable
- They must be defined outside the class
- Static member variable does not belong to any object, but to the whole class
- There will be only one copy of static member variable for the whole class
- Any object can use the same copy of class variable
- They can also be used with class name

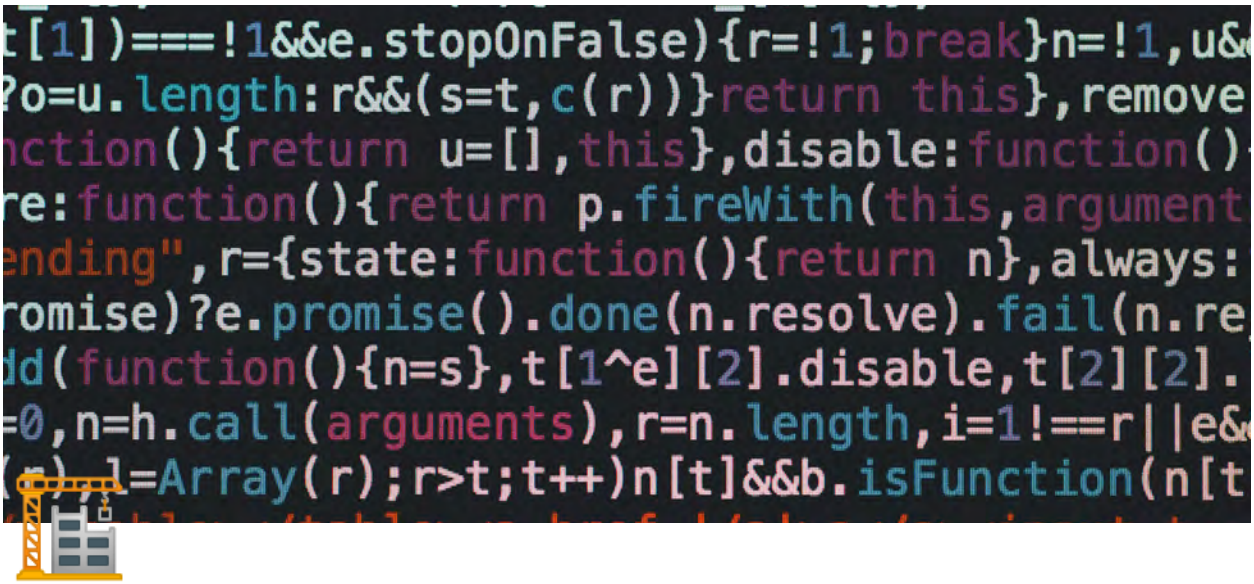
- This does not depends on object, but it depends on class
- Even if object was not declared, it will occupy space in memory
- This variable will be initialized once for all objects

Example:

```
class Account{
  private:  // no need to write this as private by deafult
    int balance;  // instance member variable
    static float roi;  // static member variable/class variable
  public:
    void setBalance(int);
    static void setRoi(float); // static member function
};

void Account::setBalance(int b){
  balance = b;
}

static void Account::setRoi(float r){
  roi = r;
}

float Account::roi;  // writing this line is important

int main(){
  Account a1;
  Account::setRoi(4.5f);
  return 0;
}
```

Static Member Function:

- Refer to code above
- These functions are used to access static member variables
- They are qualified with the keyword `static`
- They are also called class member functions
- They can be invoked with or without objects
- They can only access static members of the class



Constructor

- Constructor is a member function of a class
- The name of the constructor is same as the name of the class
- It has no return type, so can't use `return` keyword
- It must be an instance member function, that is, it can never be static
- Constructor is implicitly invoked when an object is created (called automatically when an object is created)
- Constructor is used to solve problem of initialization

Example:

```
class Complex{
  private:
    int a, b;
  public:
    Complex(){
      cout << "Hello constructor" << endl; // just for testing
    }
};

int main(){
  Complex c1;
```

```
  return 0;
}
```

What is problem of initialization?

- When an object is created, its member variables contains a garbage value
- So this garbage value does not represent the real world entity
- So we need to make object an object (just like insaan ko bolna kai insaan ban jao)

Constructor Overloading:

- Just like function overloading

```
class Complex{
  private:
    int a, b;
  public:
    Complex(int x, int y){ // 1
      a = x;
      b = y;
    }
    Complex(int k){ // 2
      a = k;
    }
    Complex(){ // 3
      a = 0;
      b = 0;
    }
};

int main(){
  Complex c1(3, 4), c2(5), c3; // c1:1, c2:2, c3:3
  // another way to write above things
  Complex c4 = Complex(3, 4);
  return 0;
}
```

Default Constructor:

- If you don’t make constructor, compiler makes two constructors by itself which are known as **default constructor** and **copy constructor**
- **Default constructor** takes nothing and returns nothing (every constructor returns nothing)
- If you make constructor, now compiler will not make default constructor but will still make copy constructor

Copy Constructor Example:

```
class Complex{
  .
  .
  .
  Complex(Complex &c){  // reference varible to avoid recursion of making infinite objects
    a = c.a;
    b = c.b;
  }
};

int main(){
  Complex c4(c1);
  return 0;
}
```

Destructor

- Destructor is an instance member function of a class
- The name of the destructor is same as the name of a class but preceded by tilde (~) symbol
- Destructor can never be static
- Destructor has no return type
- Destructor takes no arguments (no overloading is possible)
- It is invoked implicitly when object is going to destroy

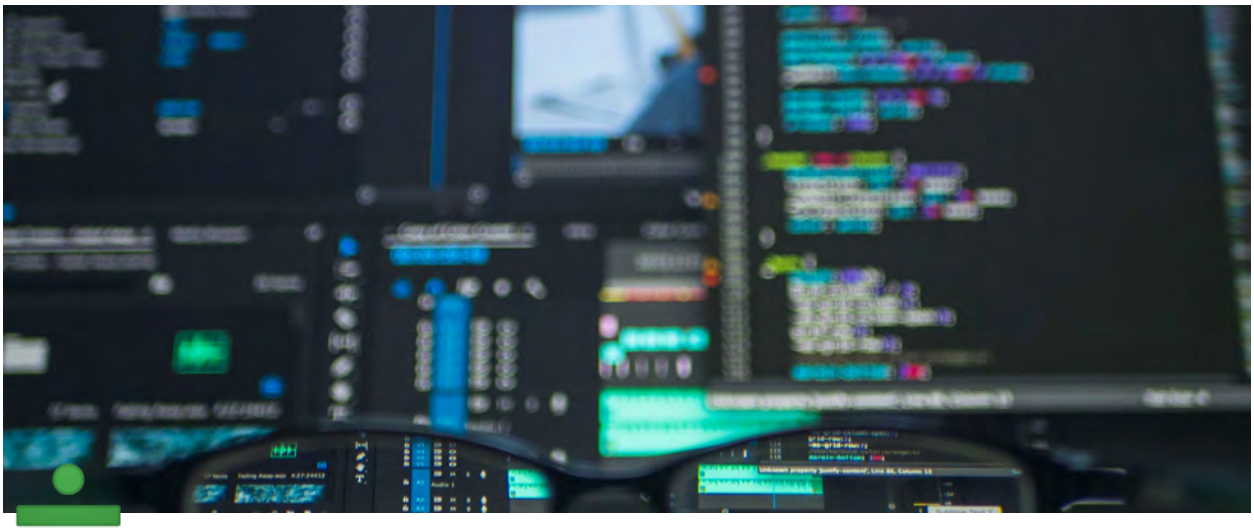
Example:

```
class Complex{
  int a, b;
  public:
    ~Complex(){
        cout << "Destructor"; // just for testing
    }
};

void fun(){
  Complex obj;
}
```

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

Why destructor?

- It should be defined to release resources allocated to an object
- For example, if you have a class that has a pointer and that pointer points to a location by keyword `new`, we need to free that resource once our task is done. To do that, we can use `delete` keyword in destructor because object will be destroyed by the program itself, but any pointer inside it pointer to another location will remain in the memory so we can use destructor to free that location



Operator Overloading

Example of Binary Operator:

```
class Complex{
  private:
    int a, b;
  public:
    void setData(int, int);
    void showData();
    Complex operator +(Complex c){
      Complex temp;
      temp.a = a + c.a;
      temp.b = b + c.b;
      return temp;
    }
};

void Complex::setData(int x, int y){
  a = x;
  b = y;
}
void Complex::showData(){
  cout << "a = " << a << ", b = " << b << endl;
}

int main(){
  Complex c1, c2, c3;
  c1.setData(3, 4);
```

```
  c2.setData(5, 6);
  c3 = c1 + c2;
  c3.showData();
  return 0;
}
```

- When an operator is overloaded with multiple jobs, it is known as **operator overloading**
- It is a way to implement compile time polymorphism
- Any symbol can be used as a function name
  - If it is a valid operator in C
  - If it is preceded by `operator` keyword
- You can not overload `sizeof` and `?:` operator

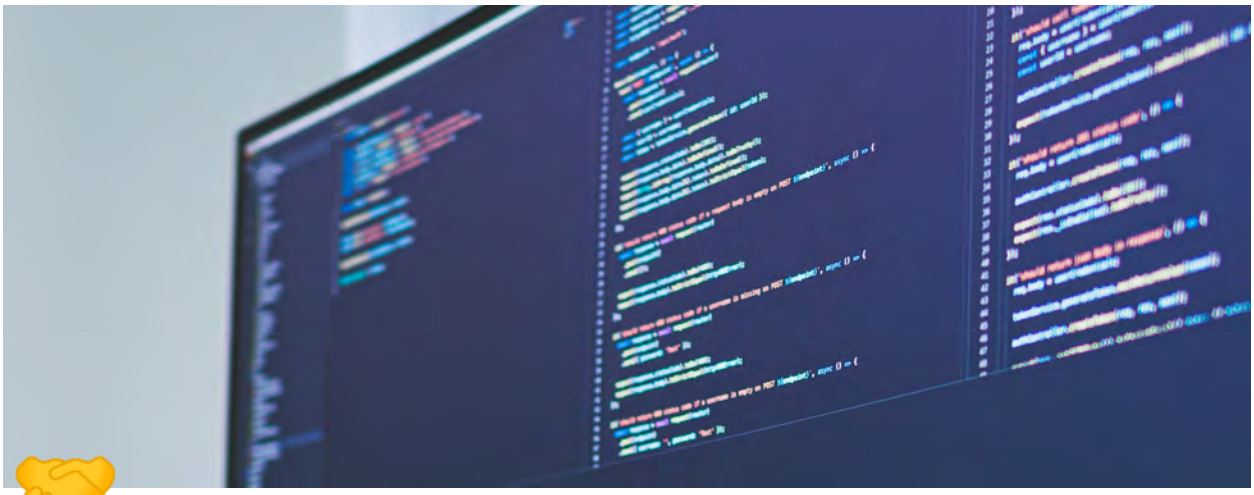
Example of Unary Operator:

```
class Complex{
  private:
    int a, b;
  public:
    void setData(int, int);
    void showData();
    Complex operator -(){
      Complex temp;
      temp.a = -1 * a;
      temp.b = -1 * b;
      return temp;
    }
};

void Complex::setData(int x, int y){
  a = x;
  b = y;
}
void Complex::showData(){
  cout << "a = " << a << ", b = " << b << endl;
}

int main(){
  Complex c1, c2, c3;
  c1.setData(3, 4);
  c2 = -c1;   // c2 = c1.operator-();
```

```
  c2.showData();
  return 0;f
}
```

Overloading of increment operator:

```
class Integer{
  private:
    int x;
  public:
    void setData(int);
    void showData();
    //  overloading
    Integer operator ++(){  // pre-increment
      Integer i;
      i.x = ++x;
      return i;
    }
    Integer operator ++(int){  // post-increment (argument is int to differ from pre and post)
      Integer i;
      i.x = x++;
      return i;
    }
};

void Integer::setData(int a){
  x = a;
}

void Integer::showData(){
  cout << x;
}

int main(){
  Integer i1;
  i1.setData(3);
  i1.showData();
  Integer i2;
  i2 = ++i1;  // i2 = i1.operator ++();
  i1.showData();
  i2.showData();
  return 0;
}
```



Friend Function

- Friend function is not a member function of a class to which it is a friend
- Friend function is declared in the class with friend keyword
- It must be defined outside the class to which it is friend
- Friend function can access any member of the class to which it is a friend
- Friend function cannot access members of the class directly
- It has no caller object
- It should not be defined with membership label

Example:

```
class Complex{
  private:
    int a, b;
  public:
    void setData(int x, int y){
        a = x, b = y;
    }
    void showData(){
      cout << a << b;
    }
    friend void fun(Complex);
};

void fun(Complex c){
  cout << "Sum is: " << c.a + c.b;
}
```

```
int main(){
  Complex c1;
  fun(c1); // not c1.fun() as it is not member of that class, only a friend
  return 0;
}
```

- Friend function can become friend to more than one class

Example:

```
class B; // this is formal declaration of B because B is defined after A so A don't knows what is B

class A{
  private:
    int a;
  public:
    void setData(int x) { a = x; }
    friend void fun(A, B);
};

class B{
  private:
    int b;
  public:
    void setData(int y) { b = y; }
    friend void fun(A, B);
};

void friend(A o1, B o2){
  cout << "Sum is: " << o1.a + o2.b;
}

int main(){
  A obj1;
  B obj2;
  obj1.setData(2);
  obj2.setData(3);
  fun(obj1, obj2);
  return 0;
}
```

Overloading of operators as a friend function:

Example:

```
class Complex{
  private:
    int a, b;
  public:
    void setData(int x, int y){
      a = x, b = y;
    }
    void showData(){
      cout << a << b;
    }
    friend Complex operator +(Complex, Complex);
};

Complex operator +(Complex X, Complex Y){
  Complex temp;
  temp.a = X.a + Y.a;
  temp.b = X.b + Y.b;
}

int main(){
  Complex c1, c2, c3;
  c1.setData(3, 4);
  c2.setData(5, 6);
  c3 = c1 + c2;
  c3.showData();
  return 0;
}
```

Overloading of unary operator as a friend function:

Example:

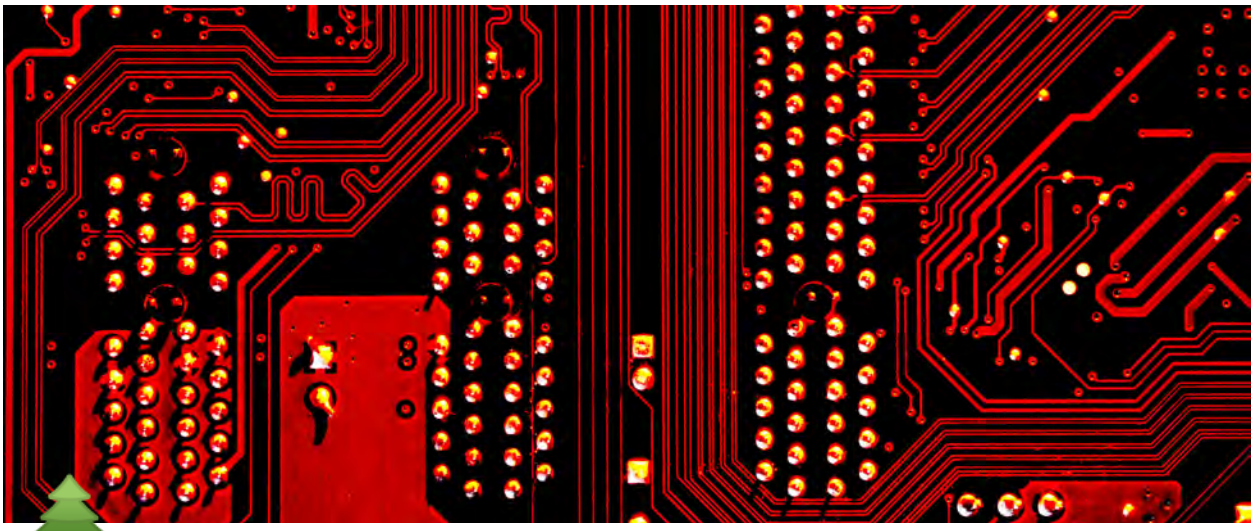
```
class Complex{
  private:
    int a, b;
  public:
    void setData(int x, int y){
      a = x, b = y;
    }
    void showData(){
      cout << a << b;
    }
    friend Complex operator-(Complex);
};

Complex operator -(Complex c){
  Complex temp;
  temp.a = -1 * c.a;
  temp.b = -1 * c.b;
  return temp;
}
```

```
int main(){
  Complex c1, c2;
  c1.setData(3, 4);
  c2 = -c1 // c2 = operator-(c1);
  c2.showData();
  return 0;
}
```

WILL GET BACK TO THIS

Example:

```
class Complex{
  private:
    int a, b;
  public:
    void setData(int x, int y){
      a = x, b = y;
    }
    void showData(){
      cout << a << b;
    }
    friend Complex operator>>(Complex);
};

Complex operator >>(Complex c){

}

int main(){
  Complex c1;
  cin >> c1;
  c1.showData();
  return 0;
}
```



Inheritance

- Class is used to describe properties and behavior of an object
- Property names and values
- Behavior means actions (methods)

Example:

- Let us assume that we want to represent a car
- Properties:
  - Price
  - Fuel Type
  - Engine
  - Color
  - Capacity
- Methods:
  - setPrice()

    - setFuelType()
    - setEngine()
    - setColour()
    - setCapacity()
    - getPrice()
    - getFuelType()
    - getEngine()
    - getColour()
    - getCapacity()
- Now we want to represent a sports car
- Things in purple color are extra for sports car
- Properties:
    - Price
    - Fuel Type
    - Engine
    - Color
    - Capacity
    - Alarm
    - Navigator
    - safeGuard
- Methods;
    - setAlarm()
    - setNavigator()
    - setSafeGuard()
    - getAlarm()
    - getNavigator()

    - getSafeGuard()
    - setPrice()
    - setFuelType()
    - setEngine()
    - setColour()
    - setCapacity()
    - getPrice()
    - getFuelType()
    - getEngine()
    - getColour()
    - getCapacity()
- So to achieve this, we will first make class of normal class and then of sports class and will link them so that sports class can **inherit** properties and behaviors of normal class

Inheritance:

- It is a process of inheriting properties and behaviors of existing class into a new class
- Existing Class = Parent Class = Base Class
- New Class = Child Class = Derived Class

Syntax:

```
class Base_Class
{

};

class Derived_Class : Visibility_Mode Base_Class
{
```

```
};
```

Example:

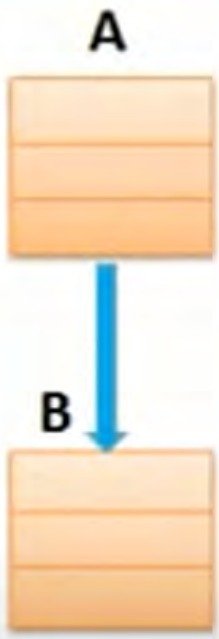
```
class Car{

};

class SportsCar : public Car{

};
```

Types of Inheritance:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

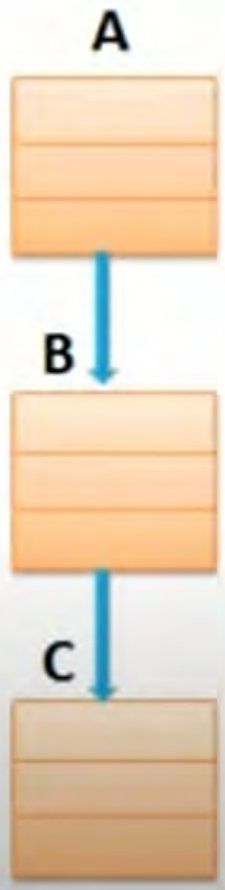
Single Inheritance:

```
class A
{
};
class B:public A
{
};
```



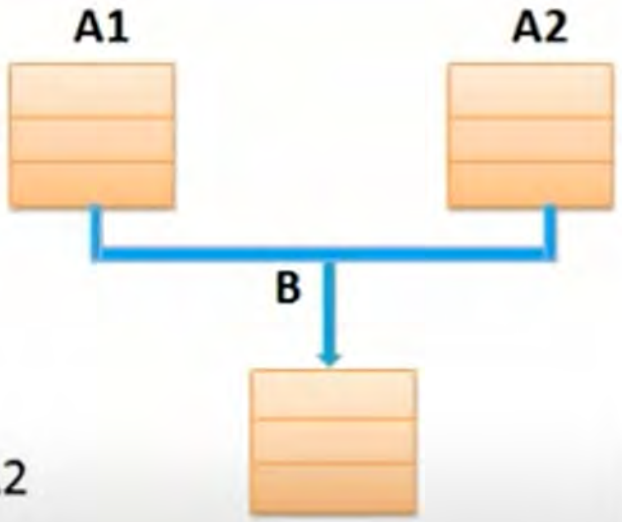
Multilevel Inheritance:

```
class A
{
};
class B:public A
{
};
class C:public B
{
};
```



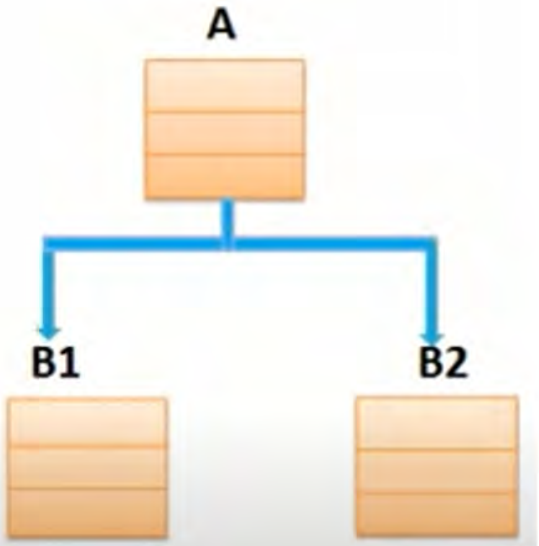
Multiple Inheritance:

```
class A1
{
};
class A2
{
};
class B:public A1,public A2
{
};
```

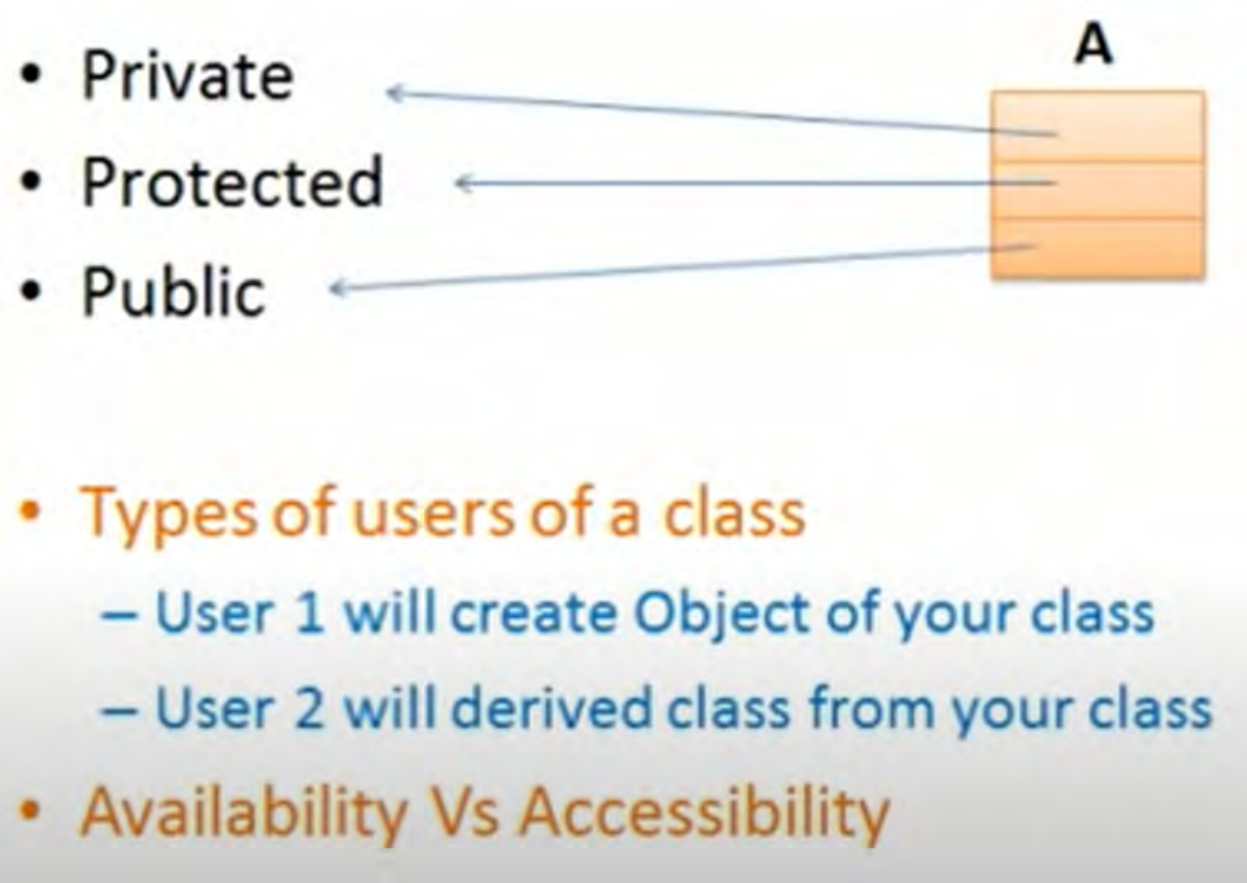


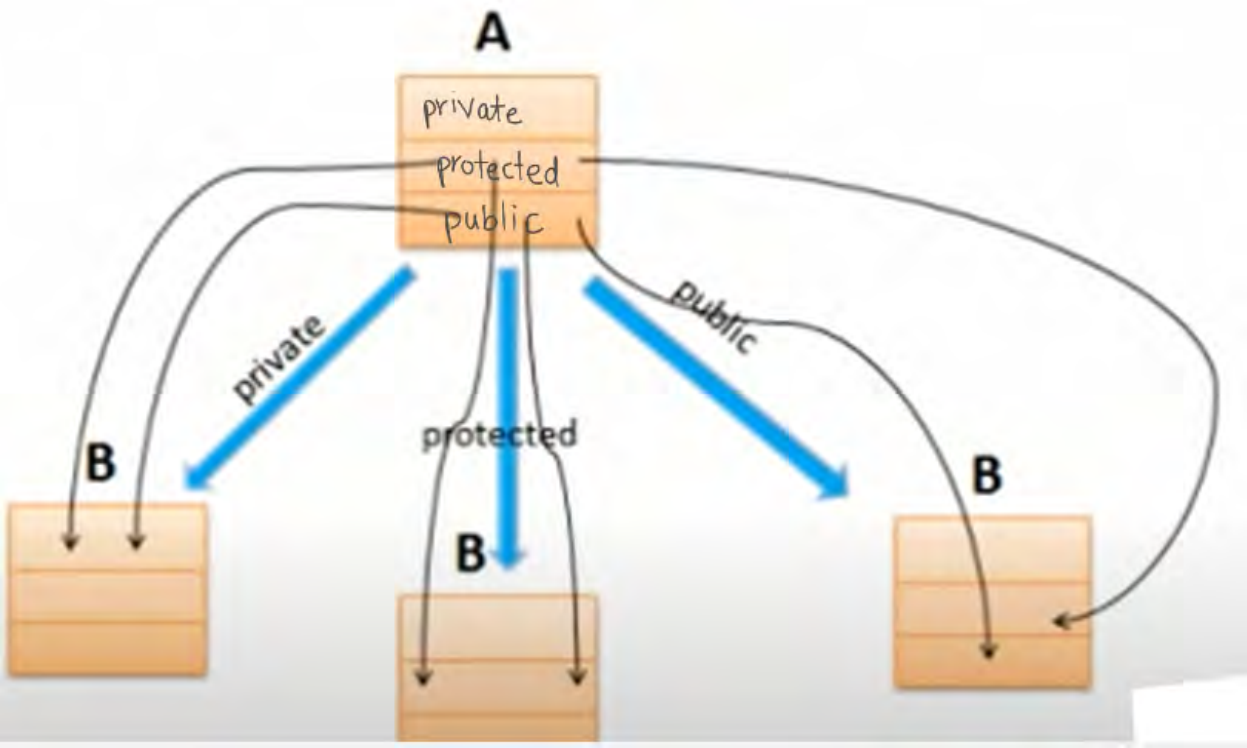
Hierarchical Inheritance:

```
class A
{
};
class B1:public A
{
};
class B2:public A
{
};
```



Visibility Modes:



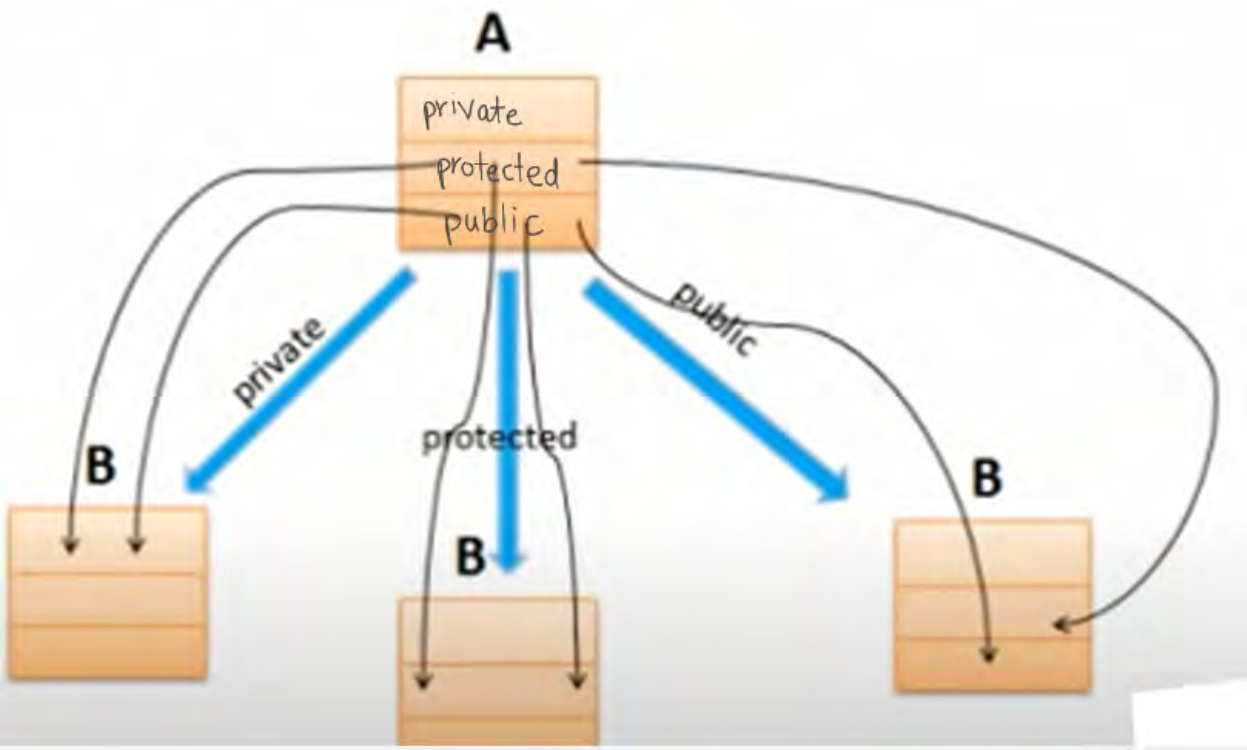


Is-a relationship:

- Banana **is a** fruit
- When this relationship exists, it is called Association:
  - Aggregation
  - Composition
  - Inheritance
- Jahan **is-a** relationship banta hai wahan inheritance hoti hai
  - One class becomes child class and other becomes parent class
  - Banana is a fruit: banana-child(specific), fruit-parent(generalized)
- **is-a** relationship is always implemented as a public inheritance

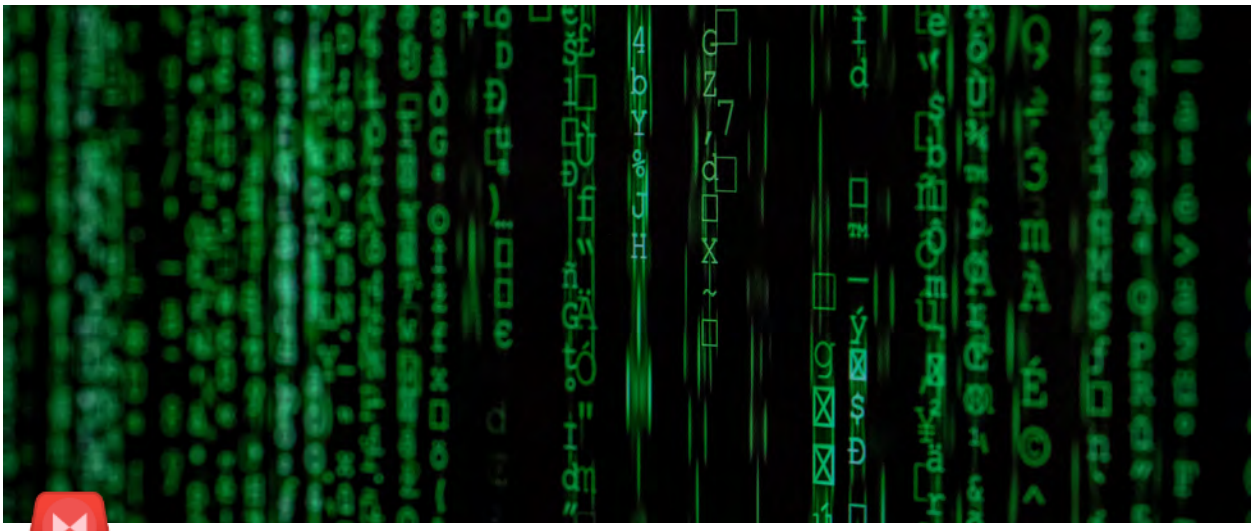
Example:

```
class Car{
  private:
    int gear;
  public:
    void incrementGear(){
      if(gear < 5){
        gear++;
      }
    }
};

// sports car is a car: so we can use public inheritance as shown below

class SportsCar: public Car{

};
```



Constructor and Destructor in Inheritance:

- We know that constructor is invoked implicitly when an object is created
- In inheritance, when we create object of derived class, what will happen?



Constructor and Destructor in Inheritance

Example:

```
class A
{
  public:
    A(){

    }
};

class B: public A
{
  public:
    B():A(){ // the constructor B calls constructor A (as B is a child of A)

    }
};

void main(){
  B obj;
}
```

- Child class constructor calls parent class constructor

```
class A
{
  int a;
  public:
    A(int k){
      a = k;
    }
    ~A(){

    }
};

class B: public A
{
  public:
    B(int x, int y):A(x){ // the constructor B calls constructor A (as B is a child of A)
      b = y;
    }
    ~B(){

    }
};

void main(){
  B obj(2, 3);
}
```

this Pointer

- A pointer that contains address of an object is called Object Pointer

Using previous method:

```
class Box{
  private:
    int l, b, h;
  public:
    void setDimension(int x, int y, int z){
      l = x; b = y; h = z;
    }
    void showDimension(){
      cout << l << b << h;
    }
};

int main(){
  Box smallBox;
  smallBox.setDimension(12, 10, 5);
  smallBox.showDimension();
  return 0;
}
```

Using pointer:

```
class Box{
  private:
    int l, b, h;
  public:
    void setDimension(int x, int y, int z){
      l = x; b = y; h = z;
    }
    void showDimension(){
      cout << l << b << h;
```

```
    }
};

int main(){
  Box smallBox;
  Box *p;
  p = &smallBox;
  p->setDimension(12, 10, 5);
  p->showDimension();
  return 0;
}
```

this pointer:

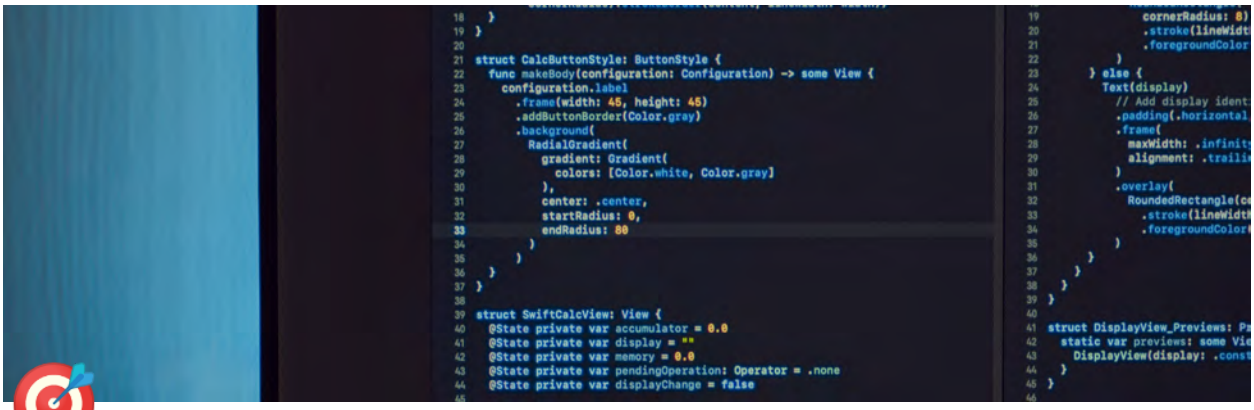
- `this` is a keyword
- `this` is a local object pointer in every instance member function containing address of the caller object
- `this` pointer can not be modified
- It is used to refer caller object in member function

```
class Box{
  private:
    int l, b, h;
  public:
    void setDimension(int l, int b, int h){  // these l, b, h are not instance member variable, they are different
      this->l = l;
      this->b = b;
      this->h = h;
    }
    void showDimension(){
      cout << this->l << this->b << this->h;
    }
};

int main(){
  Box smallBox;
  smallBox.setDimension(12, 10, 5);
  smallBox.showDimension();
  return 0;
}
```

new and delete

- SMA: static memory allocation
- DMA: dynamic memory allocation

```
int *p = new int;
float *q = new float;
Complex *ptr = new Complex;
float *q = new float[5];


int x;
cin >> x;
int *p = new int[x];

delete p;
delete []p;
```



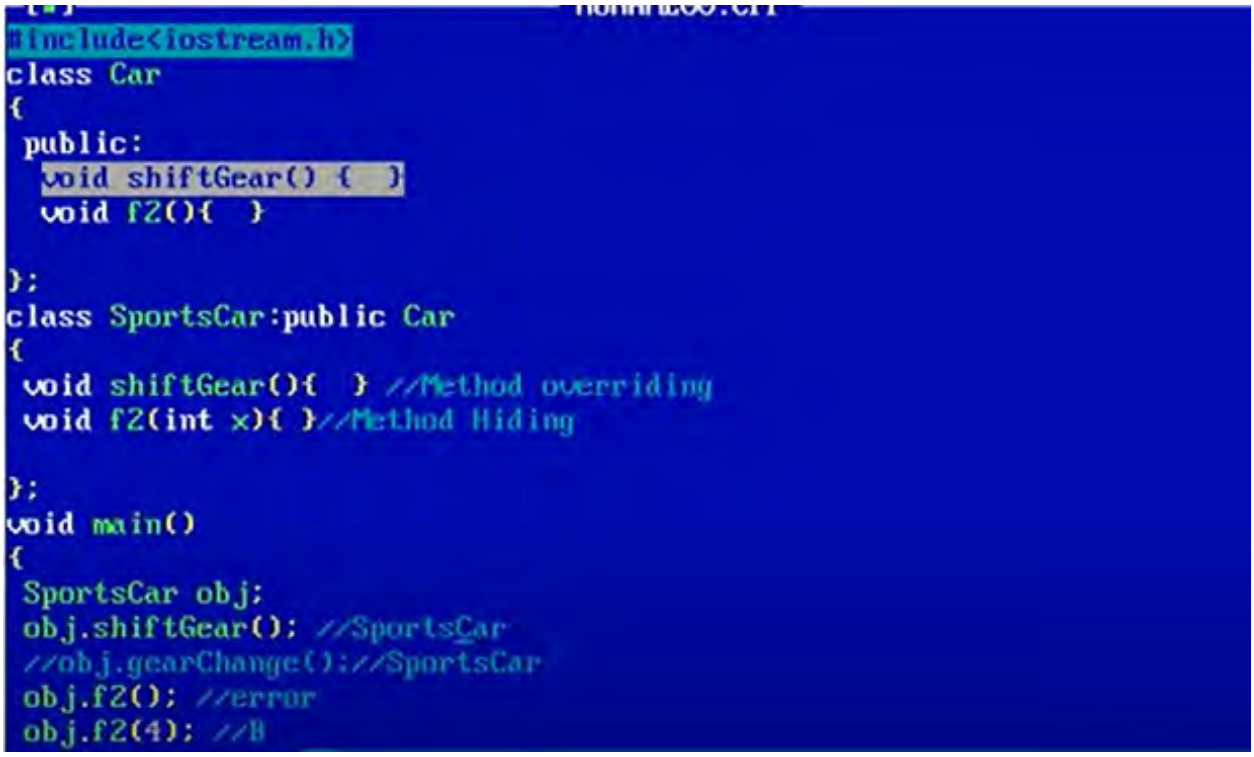
Method Overriding

```
class A{
  public:
    void f1(){ }
    void f2() { }
};

class B: public A{
  public:
    void f1(){ } // method overriding - same name and same arguments
    void f2(int){ } // method hiding - same name but different  arguments
};

void main(){
  B obj;
  obj.f1(); // this will be binded with method present in class B
  obj.f2(); // error: if exact name found in B, it will not go to A but as there is argument mismatch, it will give error
  obj.f2(4); // B
}
```

- Above example of function f2 is not function overloading because in function overloading, those functions must be defined in the same class, not different classes

Why and when to use method overriding:

- When one class inherits other class, it will inherit everything in parent class. There might be a case where a child class performs a certain method of parent class but in a different way, we use function overriding by using same name of method in both classes but child function will have different coding of it



```
#include<iostream.h>
class Car
{
 public:
  void shiftGear() {  }
  void f2(){  }

};
class SportsCar:public Car
{
 void shiftGear(){  } //Method overriding
 void f2(int x){ }//Method Hiding

};
void main()
{
 SportsCar obj;
 obj.shiftGear(); //SportsCar
 //obj.gearChange();//SportsCar
 obj.f2(); //error
 obj.f2(4); //H
```

Virtual Functions

- Base class pointer can point to the object of any of its descendant’s class
- But its opposite is not true

Example:

```
class A
{
  public:
    void f1() { }
};

class B: public A
{
  public:
    void f1() { }  // function overriding
};

void main(){
  A *p;
  B o2;
  p = &o2;
  p->f1(); // this will bind with A, not B which is the problem
}
```

- If we accessed f1() using object variable, f1() of class B will be called
- But if we called f1() using *p, it will call f1() of class A which is a problem
- This problem occurs due to early binding
- Late binding is the solution for this
- So, we can use virtual function

Example:

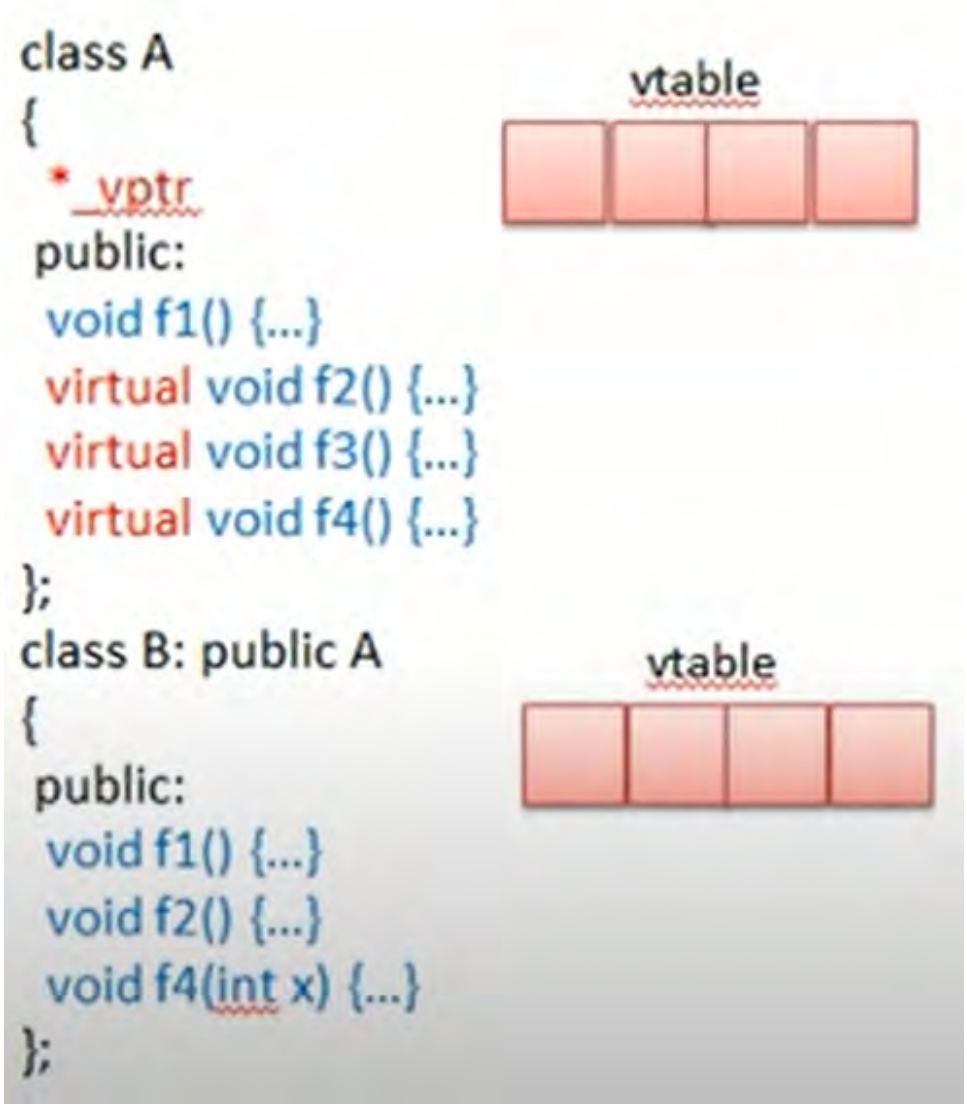
```
class A
{
  public:
    virtual void f1() { }
};

class B: public A
{
  public:
    void f1() { }  // function overriding
};

void main(){
  A *p;
  B o2;
  p = &o2;
  p->f1(); // now, this will bind with B, not A
}
```

Virtual Function working concept:

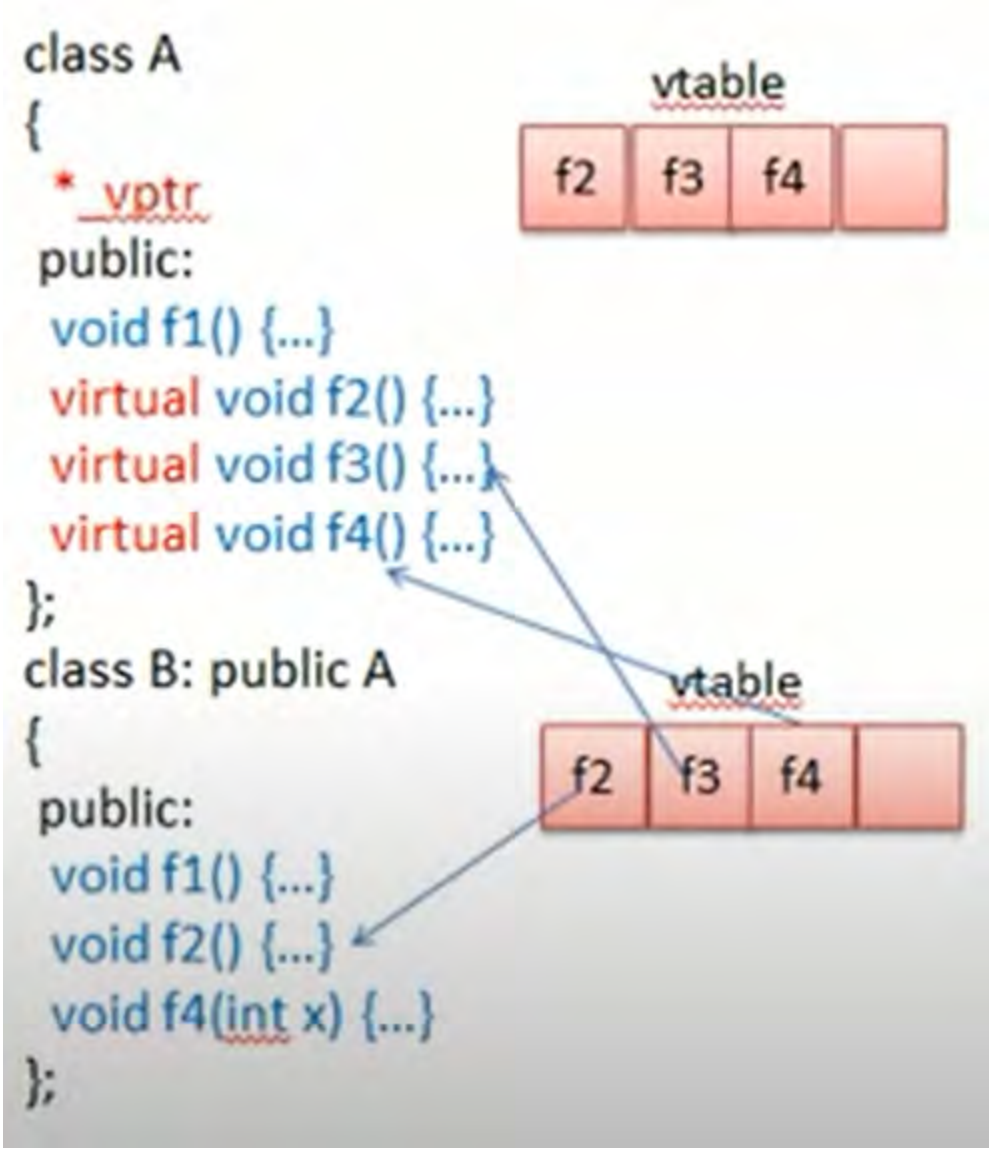
```
class A
{
  public:
    void f1() { }
    virtual void f2() { }
    virtual void f3() { }
    virtual void f4() { }
};

class B: public A
{
  public:
    void f1() { }
```

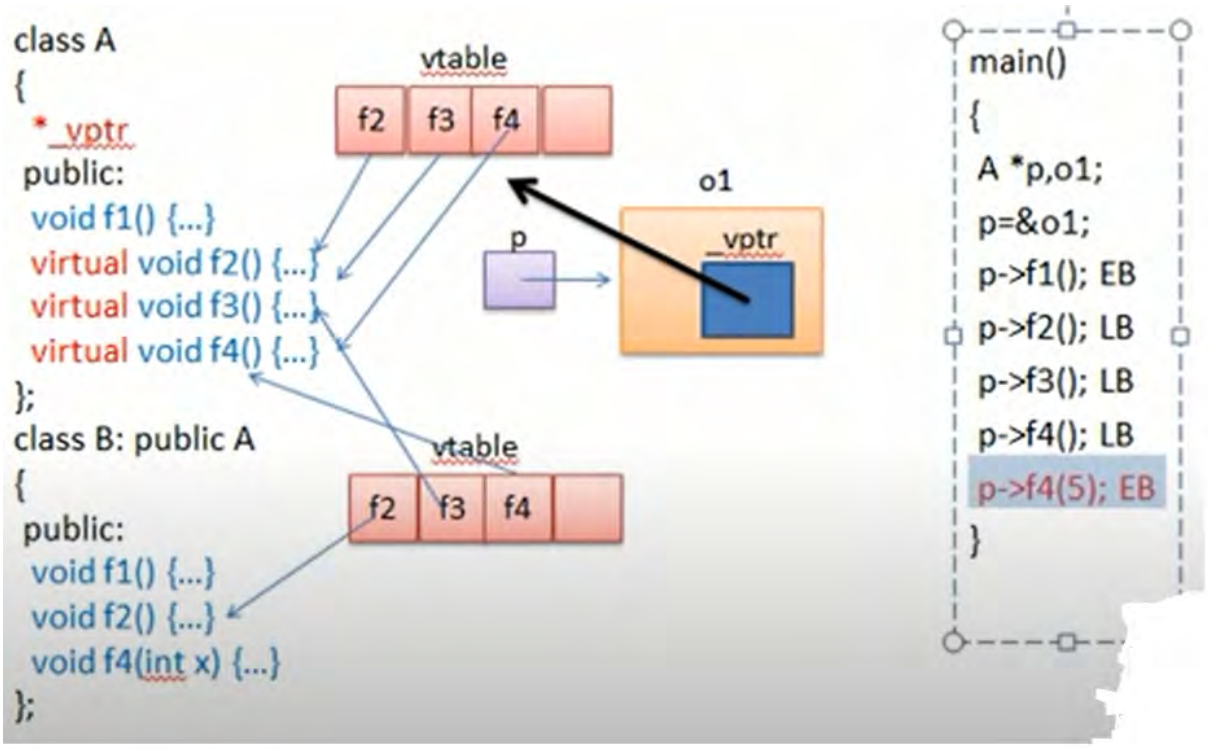
```
    void f2() { }
    void f4(int x) { }
};
```

- Kisi bhi class mai agar virtual function hoga to compiler khud us class kai liye apni taraf sai aik variable class kai andar as a member declare kardega called `*_vptr`
- Liken yeh variable child class mai alag sai nahi banay ga because yeh variable instance member variable hai because of inheritance (no need to make another variable as class B inherits properties and behaviors of class A)
- Second kaam compiler aik static array bana deta hai aur yeh pointer array hai. Is array mai functions ka address rakha jae ga
- This array is called `vtable`
- This time, unki descendants class mai bhi alag alag `vtable` wala array banay ga
- Liken `*_vptr` sirf aik hi banay ga us class kai liye jis mai virtual functions banay huay hai
- `*_vptr` `vtable` ka address contain karega
- Jis class ka bhi object banay ga, `*_vptr` us class kai `vtable` ka address store karay ga

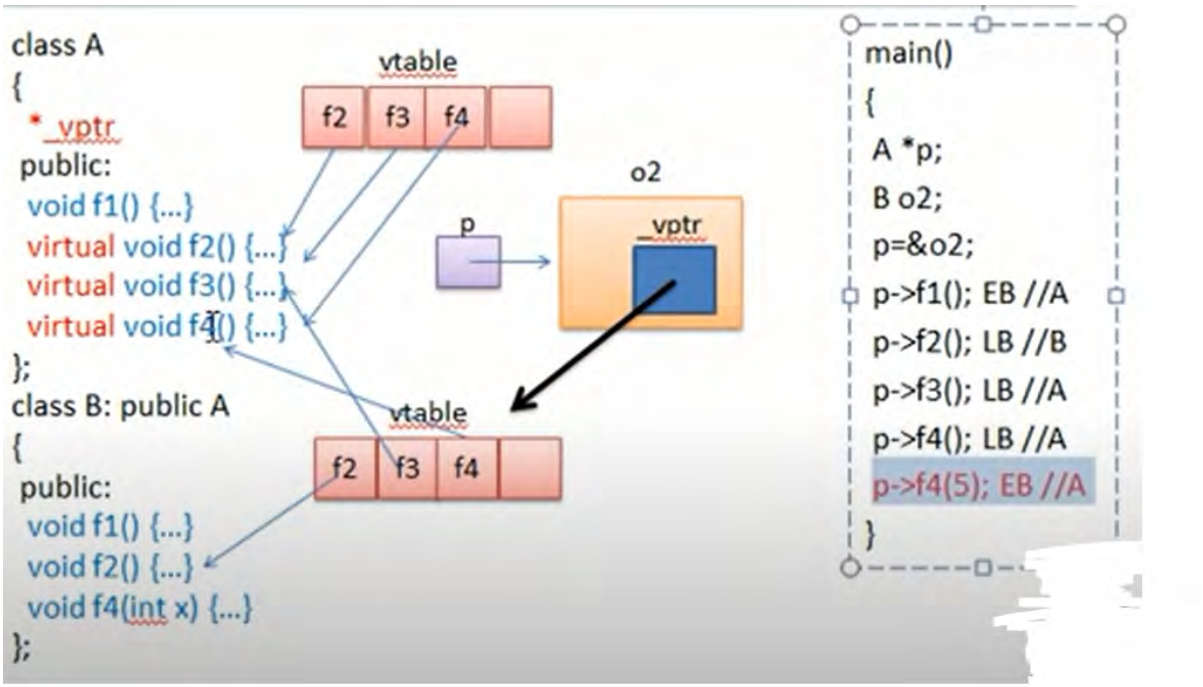


- `vtable` mai sirf virtual functions ka address hoga


class A
{
* vptr
public:
void f1() {...}
virtual void f2() {...}
virtual void f3() {...}
virtual void f4() {...}
};
class B: public A
{
public:
void f1() {...}
void f2() {...}
void f4(int x) {...}
};
vtable
f2
f3
f4
vtable
f2
f3
f4



- Normal function mai early binding hoti hai - early binding mai pointer kai type ko dekha jata hai
- Virtual functions mai late binding hoti hai - late binding mai pointer kisko point kar raha hai yeh dekha jata hai


class A
{
* _vptr
public:
void f1() {...}
virtual void f2() {...}
virtual void f3() {...}
virtual void f4() {...}
};
class B: public A
{
public:
void f1() {...}
void f2() {...}
void f4(int x) {...}
};
vtable
f2
f3
f4
o2
p
vptr
vtable
f2
f3
f4
main()
{
A *p;
B o2;
p=&o2;
p->f1(); EB //A
p->f2(); LB //B
p->f3(); LB //A
p->f4(); LB //A
p->f4(5); EB //A
}

Abstract Class

- A do nothing function is a pure virtual function

Example:

```
class Person
{
  public:
    virtual void fun() = 0;  // pure virtual function
    void f1() { }
};

class Student: public Person
{
  public:
    void fun() { } // method overriding - this should be defined
};
```

- In the above example, Person class ka object nahi ban sakta to avoid calling of fun() in Person as it is do-nothing
- However, Student ka object ban sakta hai and uskai andar fun() ki definition likhna paray gi

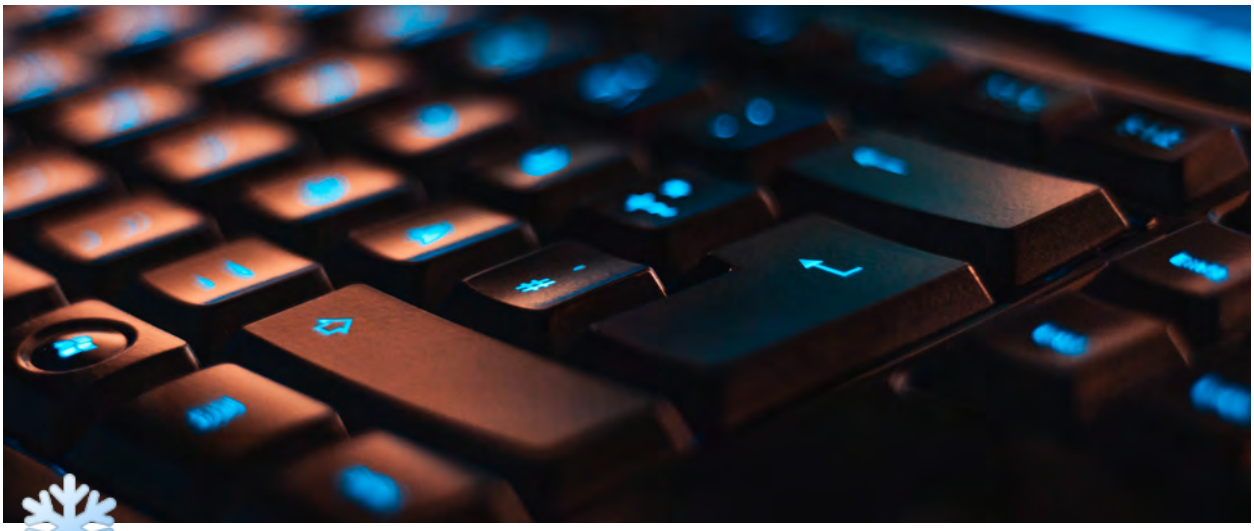
- So to call f1(), child class ka object banay ga aur yeh object f1() ko call karay ga

Abstract Class:

- A class containing pure virtual function is an abstract class
- We can not instantiate abstract class
- It is important to create child class of an abstract class and is also important to do method overriding of pure virtual function

Why Abstract Class?

- Example of student and faculty as a children of person
- Secondly, to force child class to define that function



Template

- The keyword `template` is used to define function template and class template
- It is a way to make your function or class generalize as far as data type is concerned

Example:

```
int big(int a, int b){
  if(a>b}{
    return a;
  }else{
    return b;
  }
}

int main(){
  cout << big(4, 5);
  cout << big(5.6, 3.4); // this function is not appropiate for real types
  return 0;
}
```

- The above function is not appropriate for real data type
- So we need to do function overloading

- But, agar hum function overloading sirf data types ki wajah sai kar rahein hai to hum sirf aik generalize function banana chahiye
- So, we can use template

```
template <class X>
X big(X a, X b){
  if(a>b}{
    return a;
  }else{
    return b;
  }
}

int main(){
  cout << big(4, 5);
  cout << big(5.6, 3.4); // this function is not appropiate for real types
  return 0;
}
```

Function Template:

- Function template is also known as generic function
- `template <class type> type func_name(type arg1, ...);`

For different data types:

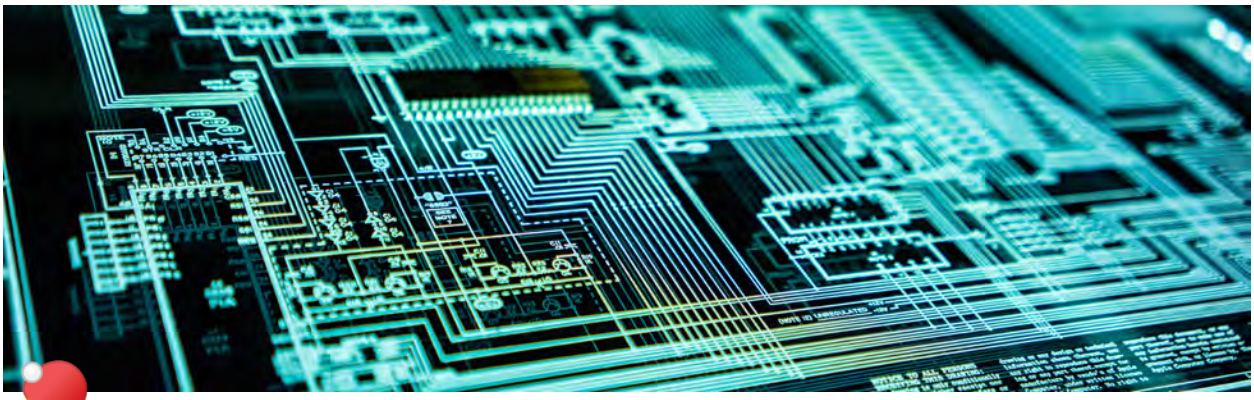
```
template <class X, class Y>
X big(X a, Y b){
  if(a>b}{
    return a;
  }else{
    return b;
  }
}

int main(){
  cout << big<int>(4, 5);
  cout << big<float>(5.6, 3.4); // this function is not appropiate for real types
```

```
  return 0;
}
```

Class Template:

- Class template is also known as generic class
- `template <class type> class class_name { };`

```
template <class X>
class ArrayList
{
  private:
    struct ControlBlock
    {
      int capacity;
      X *arr_ptr;
    };
    ControlBlock *s;
  public:
    ArrayList(int);
};

template <class T>
ArrayList<T>::ArrayList(int capacity){
  s = new ControlBlock
  s->capacity = capacity;
  s->arr_ptr = new X[s->capacity];
}

int main(){
  ArrayList<int> list1(4);
  return 0;
}
```



Initializers

- Initializer list is used to initialize data members of a class
- The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon

```
class Dummy{
  private:
    int a, b;
  public:
    Dummy();
};

int main(){

  return 0;
}

Dummy::Dummy():a(5),b(6)
{

}
```

- But when we can use constructor for it, then why use initializer?

```
class Dummy{
  private:
    int a, b;
    const int x; // it can not be initialized here as it is a member variable, ab kya karein?
  // constructor mai const instance member variable ko bhi nahi kar saktein, so now we are left with Initializer wala method
  // same applies for reference variable because usai bhi declare kartay waqt initialize karna hota hai
    int &y;
  public:
    Dummy(int);
};

int main(){
  int m = 6;
  Dummy d1(m);
  return 0;
```

```
}

Dummy::Dummy(int &n):x(5),y(n)
{

}
```

Why Initializers?

- There are situations where initializers of data members inside constructor doesn't work and initializers lists must be used
  - For initialization of non-static const data members
  - For initialization of reference members

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
using namespace std;

class Student{
    private:
        int rollNo;
        string name;
        float marks;
    public:
        void setData();
        void showData();
        void storeData();
        void readData();
        void searchStudent(int);
        void deleteStudent(int);
        void update(int, float);
};

int main(){
    Student s1;
    s1.readData();
    getch();
    s1.update(213195, 99.81);
    getch();
    s1.readData();
    return 0;
}

void Student::setData(){
    cout << "Enter your roll no: ";
    cin >> rollNo;
    cout << "Enter your name: ";
    fflush(stdin);
    getline(cin, name);
    cout << "Enter your marks: ";
    fflush(stdin);
    cin >> marks;
}

void Student::showData(){
    cout << endl << "Roll No: " << rollNo << endl;
    cout << "Name: " << name << endl;
    cout << "Marks: " << marks << endl << endl;
}
```

```
void Student::storeData(){
    ofstream fout;
    fout.open("file.dat", ios::app|ios::binary);
    fout.write((char*)this, sizeof(*this));
    fout.close();
}

void Student::readData(){
    ifstream fin;
    fin.open("file.dat", ios::in|ios::binary);
    fin.read((char*)this, sizeof(*this));
    while(!fin.eof()){
        showData();
        fin.read((char*)this, sizeof(*this));
    }
    fin.close();
}

void Student::searchStudent(int str){
    ifstream fin;
    fin.open("file.dat", ios::in|ios::binary);
    if(!fin){
        cout << "File not found" << endl;
    }else{
        fin.read((char*)this, sizeof(*this));
        while(fin.eof() == 0){
            if(this->rollNo == str){
                showData();
                break;
            }
            fin.read((char*)this, sizeof(*this));
        }
    }
    fin.close();
}

void Student::deleteStudent(int no){
    ifstream fin;
    ofstream fout;
    fin.open("file.dat", ios::in|ios::binary);
    fout.open("temp.dat", ios::out|ios::binary);

    if(!fin){
        cout << "File not found" << endl;
    }else{
        fin.read((char*)this, sizeof(*this));
        while(fin.eof() == 0){
            if(this->rollNo != no){
                fout.write((char*)this, sizeof(*this));
            }
```

```
            fin.read((char*)this, sizeof(*this));
        }
    }
    fin.close();
    fout.close();
    remove("file.dat");
    rename("temp.dat", "file.dat");
}

void Student::update(int id, float m){
    fstream file;
    file.open("file.dat", ios::in|ios::out|ios::ate|ios::binary);
    file.seekg(0);
    file.read((char*)this, sizeof(*this));
    while(file.eof() == 0){
        if(id == rollNo){
            marks = m;
            file.seekp(file.tellp() - sizeof(*this));
            file.write((char*)this, sizeof(*this));
        }
        file.read((char*)this, sizeof(*this));
    }
    file.close();
}
```