Fundamentals

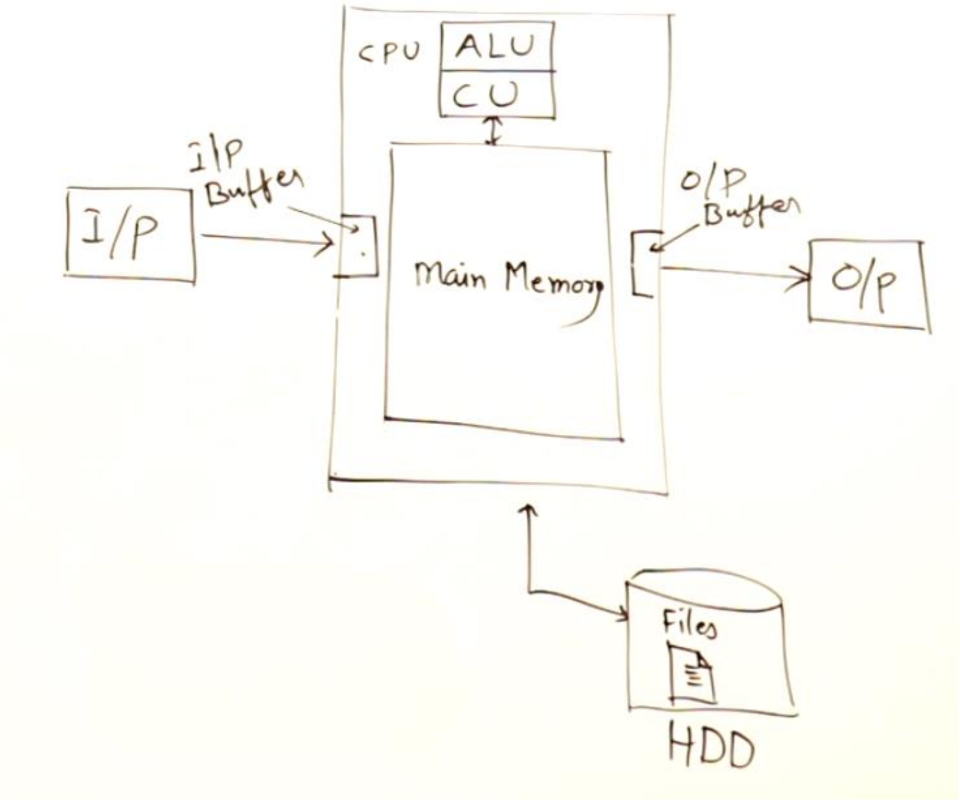
12 August 2024 20:01

What is a Program?

A **Program** is a set of instructions given by a programmer to a computer/ machine to perform certain tasks.

What is an Operating System?

A **Operating System** is a program that help us manage the computer resources, especially the allocation of those Resources for another Programs.
Examples of Operating Systems are Windows, Linux, MacOS.



How Computer Works?

A computer works on binary language (aliased as Machine language). Firstly, any input or a program is checked for an error and after checking and removing errors it got converted to a machine language which later got performed or read
by a computer.

Main Components in main memory are:

1. **CPU** : It is a hardware component that performs data I/O, processing for a Computer System.

    - **CU** : it is Called Control Unit which directs operations, basically instruct the memory logic unit and both I/O devices of the computer on how to respond to program's Instructions.
    - **ALU** : Arithmetic Logic Unit is an combinational circuit that performs arithmetic and bitwise operations on integer binary number.

2. **INPUT BUFFER** : It is a temporary storage used in Computing to hold Data being received From the input devices such as Keyboard or a mouse.
   Whenever a input is passed via a keyboard or any other input device, the data is first get Stored here and then passed to the main memory.
3. **OUTPUT BUFFER** : It is a memory cache location where data is held until an output device is ready to receive/ read it.

What is a Low-level and High Level Languages?

A **Low-Level language** is a language that is more closer to a Computer than any other language such as Machine Language or Assembly Language. They allow us to directly Manipulate the memory, register, etc.

- Machine language basically is a Binary number combination (only 0's and 1's). They are directly operated by CPU.
- Assembly language is a way to write Programs that are very closer to how a computer works. It is at much higher level than binary language but it is still very hard to understand and write.

A **High-level Language** is a language that allow a programmer to write programs in much easier way than low level languages and increases readability for humans. They have syntax Closer to human understandable languages such as English. Examples, are C++, Java, Python, etc.

What is the difference Between a Compiler and Interpreter?

A **Compiler** is a Translator which takes a program written in high level language (such as C++,etc.) as an input and produces an output of Low-Level Language which is understandable by machine.

Compiler compiles whole program in one go. That is, if there's an error present in program it will not compile and throw an error. It will only Compile when there are no syntax errors are present.

Advantages of Compiler

- Compiled code runs faster in comparison to Interpreted code.
- Compilers help in improving the security of Applications.
  As Compilers give Debugging tools, which help in fixing errors easily.

Disadvantages of Compiler

- The compiler can catch only syntax errors and some semantic errors
- Compilation can take more time in the case of bulky code.

An **Interpreter** is a translator which translates the program into machine language line by line. If there's an error present in a program at 3rd line then the interpreter will run the first 2 lines and then throw an error at 3rd line.

Advantages of Interpreter

- Programs written in an Interpreted language are easier to debug.
- Interpreters allow the management of memory automatically, which reduces
memory error risks.
- Interpreted Language is more flexible than a Compiled language.

Disadvantages of Interpreter

- The interpreter can run only the corresponding Interpreted program.
- Interpreted code runs slower in comparison to Compiled code.

Basics

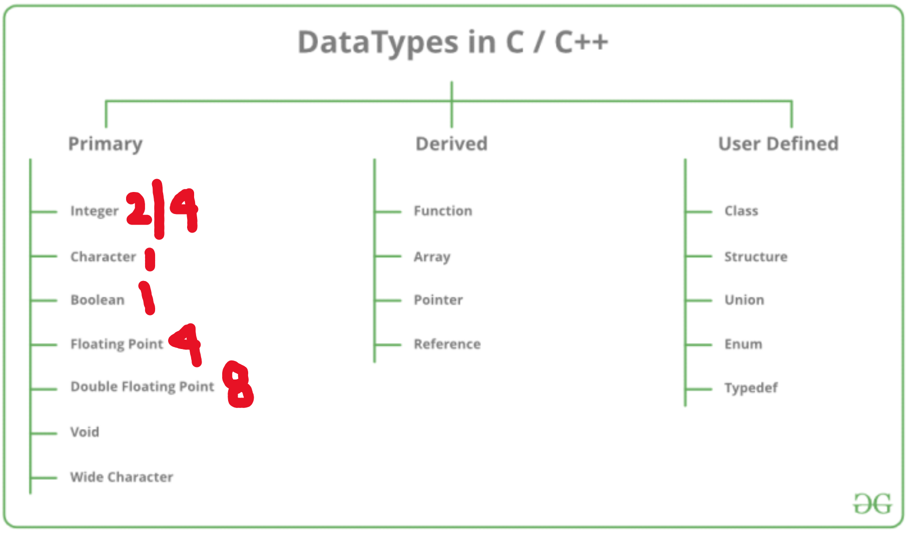
12 August 2024 20:01

C++ was created by Bjarne Stroustrup in 1985 at AT&T Bell labs.
C++ is an Object Orientated and Compiled High level Language which is used for the development of OS, Compilers, Games, etc.

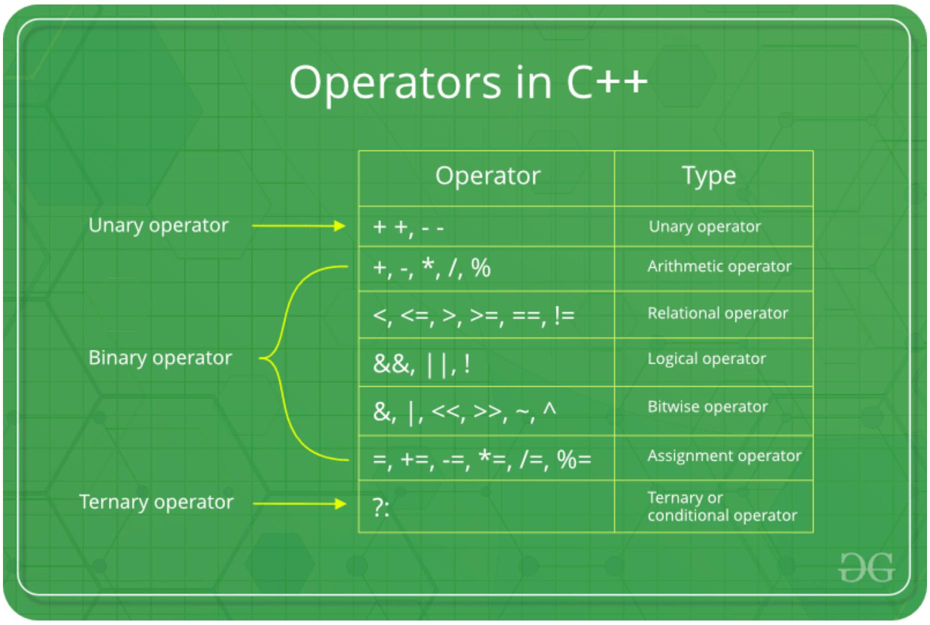
Data Types

A data types signifies 2 things that is

1. Representation of Data that is how a data can be represented in a program.
2. What are the operations are allowed to perform on that data.



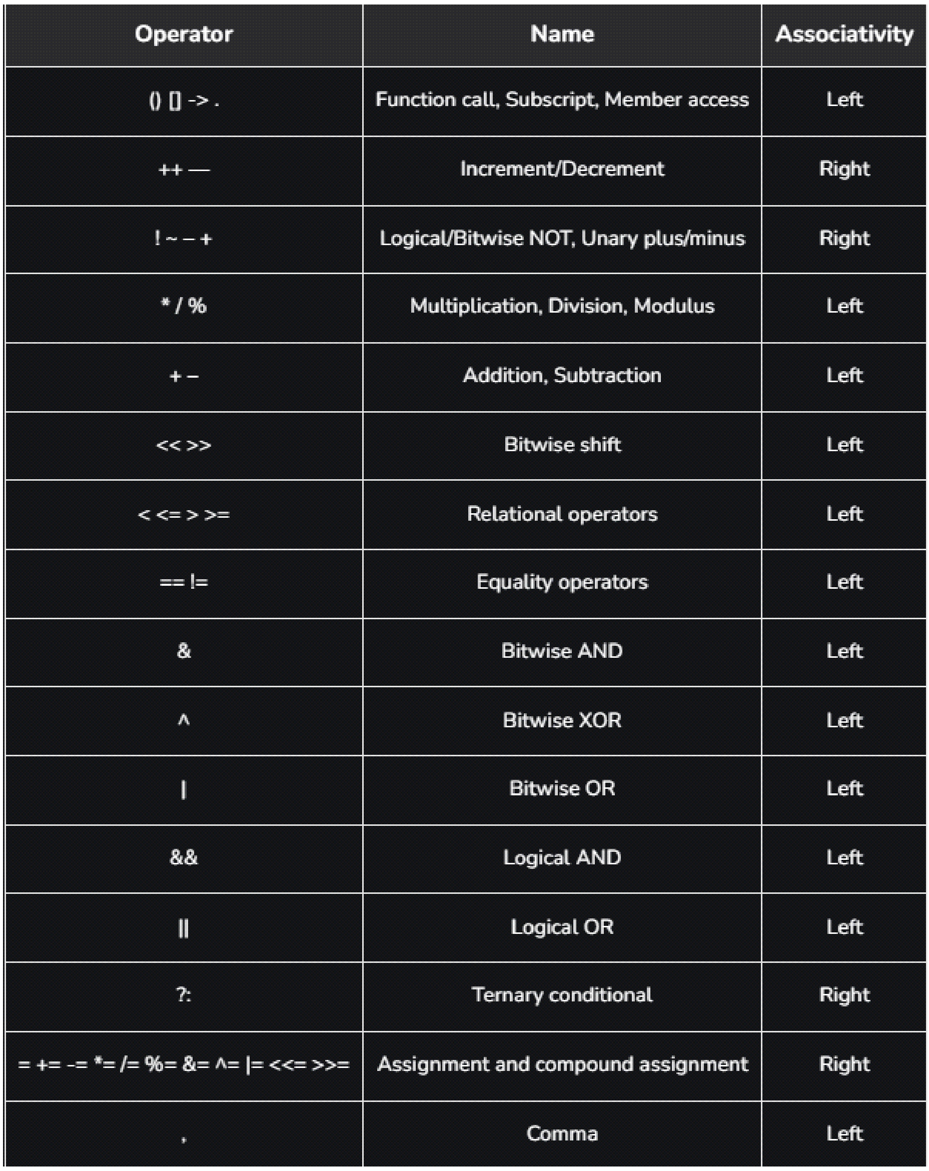
Variables are the names given to the data. example **Int id = 21**; **char s = "s"**;


Unary operator
Binary operator
Ternary operator

Operators in C++

Operator	Type
+ +, - -	Unary operator
+, -, *, /, %	Arithmetic operator
<, <=, >, >=, ==, !=	Relational operator
&&, | |, !	Logical operator
&, |, <<, >>, ~, ^	Bitwise operator
=, +=, -=, *=, /=, %=	Assignment operator
?:	Ternary or conditional operator

Order Precedence is the order in which an expression is evaluated. when there are multiple operators in a single line, then the part of expression having highest precedence get evaluated first.



Operator	Name	Associativity
() [] -> .	Function call, Subscript, Member access	Left
++ —	Increment/Decrement	Right
! ~ – +	Logical/Bitwise NOT, Unary plus/minus	Right
* / %	Multiplication, Division, Modulus	Left
+ –	Addition, Subtraction	Left
<< >>	Bitwise shift	Left
< <= > >=	Relational operators	Left
== !=	Equality operators	Left
&	Bitwise AND	Left
^	Bitwise XOR	Left
|	Bitwise OR	Left
&&	Logical AND	Left
||	Logical OR	Left
?:	Ternary conditional	Right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment and compound assignment	Right
,	Comma	Left

Practice Problems

Q1. Write a Program to take inputs from user and find the area of the triangle? Also, show difference in answer if wrong Order Precedence is followed in the Expression.

```
#include <iostream>
int main() {
    float height, base;
    std::cout << "Enter the height of the Triangle: ";
    std::cin >> height;
    std::cout << "Enter the base of the Triangle: ";
    std::cin >> base;
    // Correct calculation
    float correct = (height * base) / 2;
    // Incorrect calculation (integer division causes the error)
    float incorrect = (1 / 2) * height * base;  // This will be zero
    std::cout << "Answer with correct precedence is: " << correct
<< std::endl;
    std::cout << "Answer with wrong precedence is: " << incorrect
<< std::endl;
    return 0;
```

```
        }
```

Q2. Write a program to find the sum of N Natural number.

```
        #include<iostream>
        using namespace std;
        int main() {
            float n;
            cout<<"Enter n : ";
            cin>>n;
            float ans = n * ( n + 1) / 2;
            cout<<"Sum of First N natural Number is "<<ans;
            return 0;
        }
```

Q3. Write a program to find the area of Circle.

```
        #include<iostream>
        #define pi 3.14159f
        using namespace std;
        int main() {
            float radius;
            cout<<"Enter Radius : ";
            cin>>radius;
            float area = pi * radius * radius;
            float ar = (float) 22 / 7 * radius * radius;
            cout<<"way 1 : area of triangle is "<<area<<endl;
            cout<<"way 2 : area of triangle is "<<ar;
            return 0;
        }
```

Compound Assignment

+=

-=

*=

/=

%=

Increment and Decrement

Increment operator (++) as name suggest it increments/ increases the value by 1 whereas decrement operator (--) decreases the value by 1.

Types :

Increment

1. **Prefix increment (++value)** : first increments then assign the value if assigned to a variable.
2. **Post increment (value++)** : first assign the value if assigned to a variable than increments.

Decrement :

1. **Prefix decrement (--value)** : first decrements then assign the value if assigned to a variable.
2. **Post decrement (value--)** : first assign the value if assigned to a variable than decrements.

```
#include<iostream>
using namespace std;
int main() {
    int i,j,k,l;
    i = 1;
    j = 1;
    k = 1;
    l = 1;
    cout<<"prefix increment of "<<i<<" is "<<++i<<endl; // 2
    cout<<"post increment of "<<j<<" is "<<j++<<endl; // 1
    cout<<"prefix decrement of "<<k<<" is "<<--k<<endl; // 0
    cout<<"post decrement of "<<l<<" is "<<l--<<endl; // 1
    return 0;
}
```

Note : In C++, integer are allocated with certain no. of bits that is a range. If an integer value, takes more bits than the allocated number of bits, then we may encounter an overflow or underflow.

- **Overflow** : if number is greater than the number it can be, then this causes integer Overflow.
- **Underflow** : if number is less than the number it can be, then it causes a underflow.

Example, let us assume a range of int data type is between -128 to 127. if we try to increment the value of variable beyond 127 it will cause Integer overflow where as if we try to decrement beyond the minimum value of -128 it will cause integer underflow and value will become 127.

−128 to 127

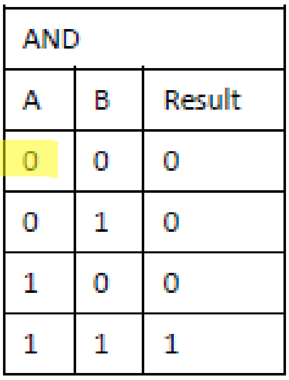
Program :

```
#include<iostream>
using namespace std;
int main() {
    // overfloe shows cyclic behaviour
    int min, max;
    min = -2147483648;
    max = 2147483647;
    cout<<"incrementing max : "<<++max<<endl; //-2147483648
    cout<<"decrementing min : "<<--min<<endl; //2147483647
    return 0;
}
```

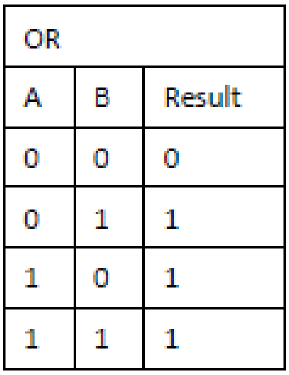
Bitwise Operators

1. **Bitwise AND (&)**

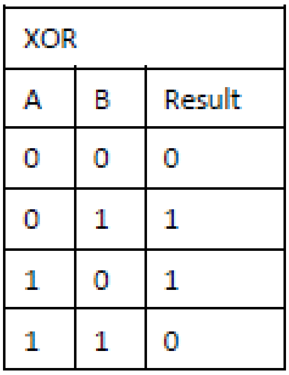
2. **Bitwise OR (|)**
3. **Bitwise XOR (^)**
4. **Bitwise NOT (~)**



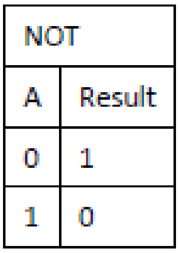
AND		
A	B	Result
0	0	0
0	1	0
1	0	0
1	1	1



OR		
A	B	Result
0	0	0
0	1	1
1	0	1
1	1	1



XOR		
A	B	Result
0	0	0
0	1	1
1	0	1
1	1	0



NOT	
A	Result
0	1
1	0

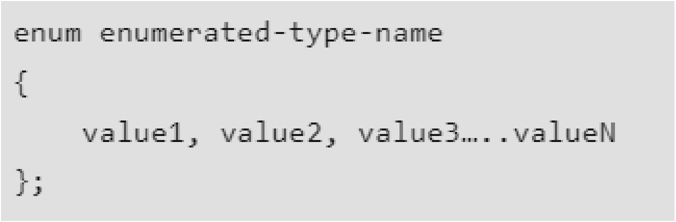
5. **Left Shift (<<) :** Shifts bits to left side.
6. **Right Shift (>>) :** shifts bits to right Side.

Program :

```
#include<iostream>
using namespace std;
int main() {
    int left, right;
    left = 2;
    right = 4;
    cout<<"Right shift : "<<(right >> 1)<<endl; // 2
    cout<<"left shift : "<<(left << 1)<<endl; // 4
    return 0;
}
```

Enum is a keyword used to define enumerated values. It is an user defined data type that can be assigned with some values. These values are defined at the time of declaration by the programmer.

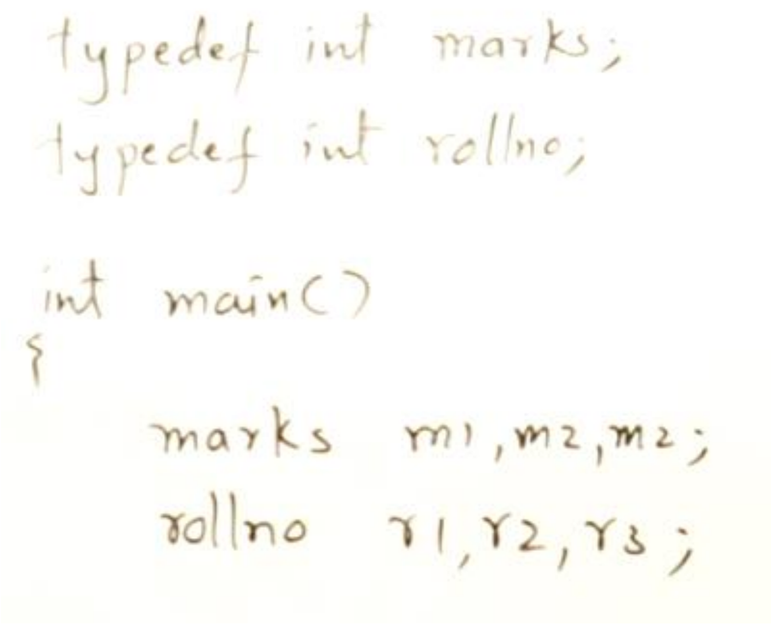
A variable defined using the enum can only have a value which is declared inside enum otherwise compiler will give an error.



```
enum enumerated-type-name
{
    value1, value2, value3…..valueN
};
```

```
#include<iostream>
using namespace std;
enum week {Monday, tuesday, wednesday, thursday, friday, saturday,
sunday};
int main() {
    week day;
    day = Monday;
    cout<<day<<endl; // 0 i.e. its index
    day = wednesday;
    cout<<day<<endl; // 2
    return 0;
}
```

A **typedef** declaration lets you define your own identifiers that can be used in place of type specifiers such as int , float , and double . A typedef declaration does not reserve storage.



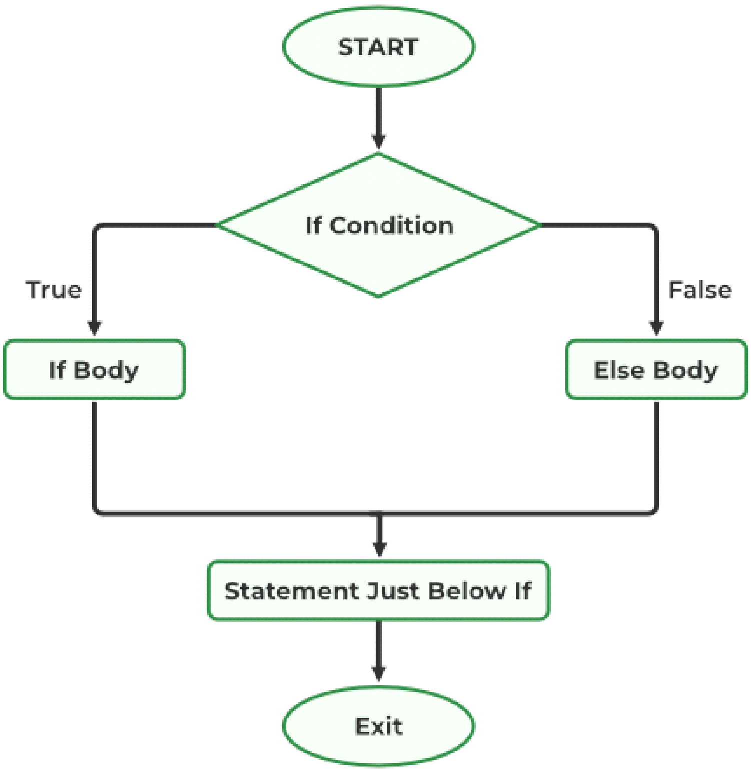
```
typedef int marks;
typedef int rollno;

int main()
{
    marks m1,m2,m2;
    rollno r1,r2,r3;
```

Conditionals

13 August 2024 20:48

Conditionals check for specific conditions to execute code inside them if they match they work else they won't.

1. If - else if - else Conditionals



Syntax :

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

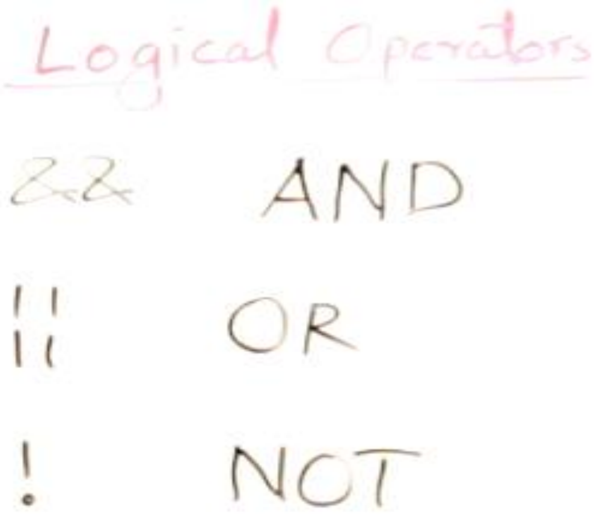
Questions

1. Write a program to find if number exists in array or not.

Ans.

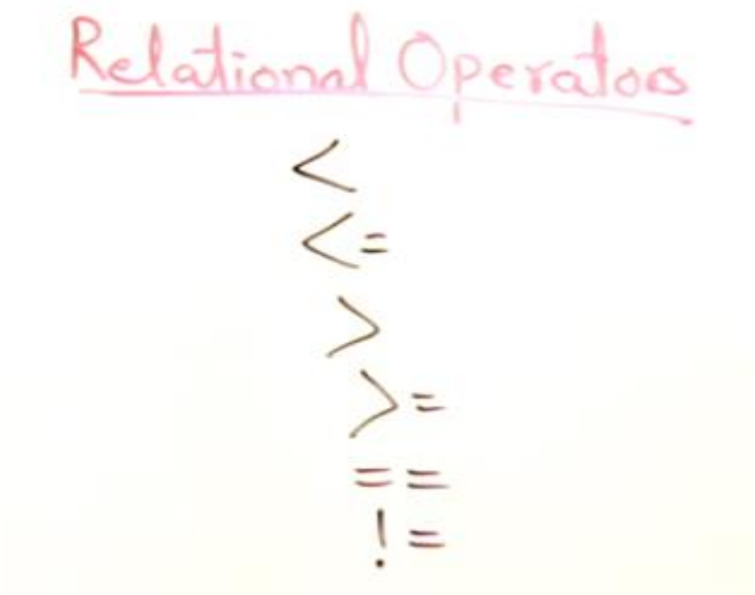
```
#include<iostream>
using namespace std;
```

```
bool find(int arr[], int n, int target) {
    for(int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return true;  // Return true if target is found
        }
    }
    return false;  // Return false if target is not found 
after checking all elements
}
int main() {
    int arr[5] = {1,2,3,4,5};
    int x;
    cin>>x;
    if (find(arr,5,x)){
        cout<<"True"<<endl;
    } else {
        cout<<"False"<<endl;
    }
    return 0;
}
```



Logical Operators

&&	AND
||	OR
!	NOT



Relational Operators

<
<=
>
>=
==
!=

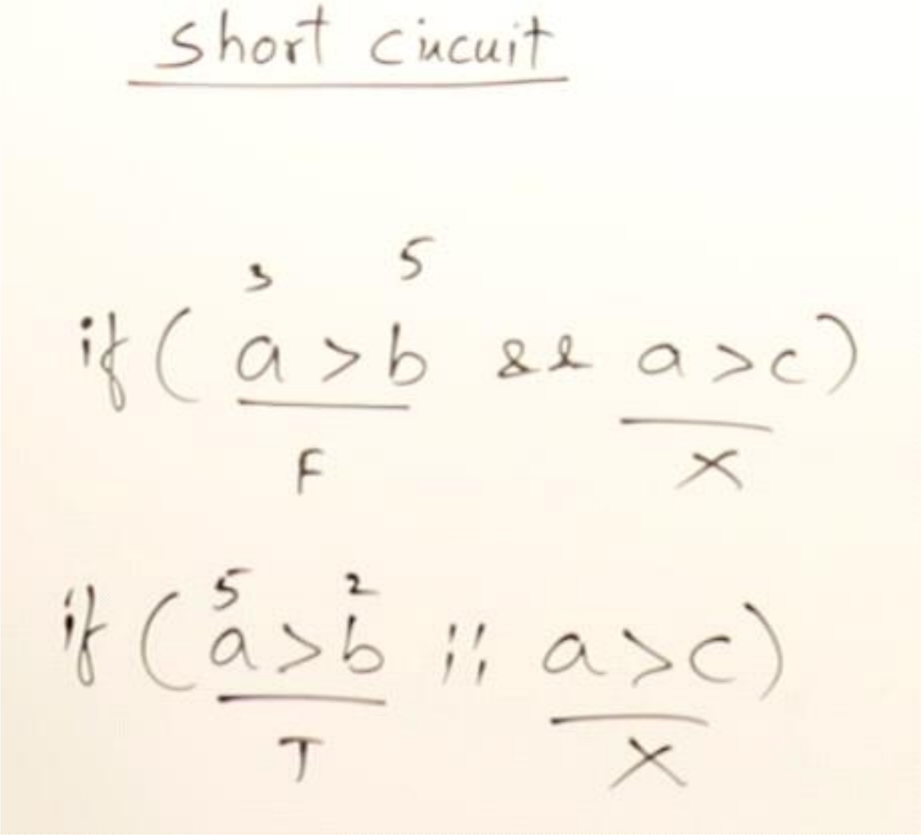
Compound conditionals are a way to test two conditions in just one statement.

```
#include <iostream>
int main() {
    int age = 25;
    bool hasID = true;
    bool isMember = false;
    // Compound conditional statement using AND (&&) and OR (||)
    if ((age >= 18 && hasID) || isMember) {
        std::cout << "Access granted." << std::endl;
    } else {
        std::cout << "Access denied." << std::endl;
    }
    return 0;
}
```

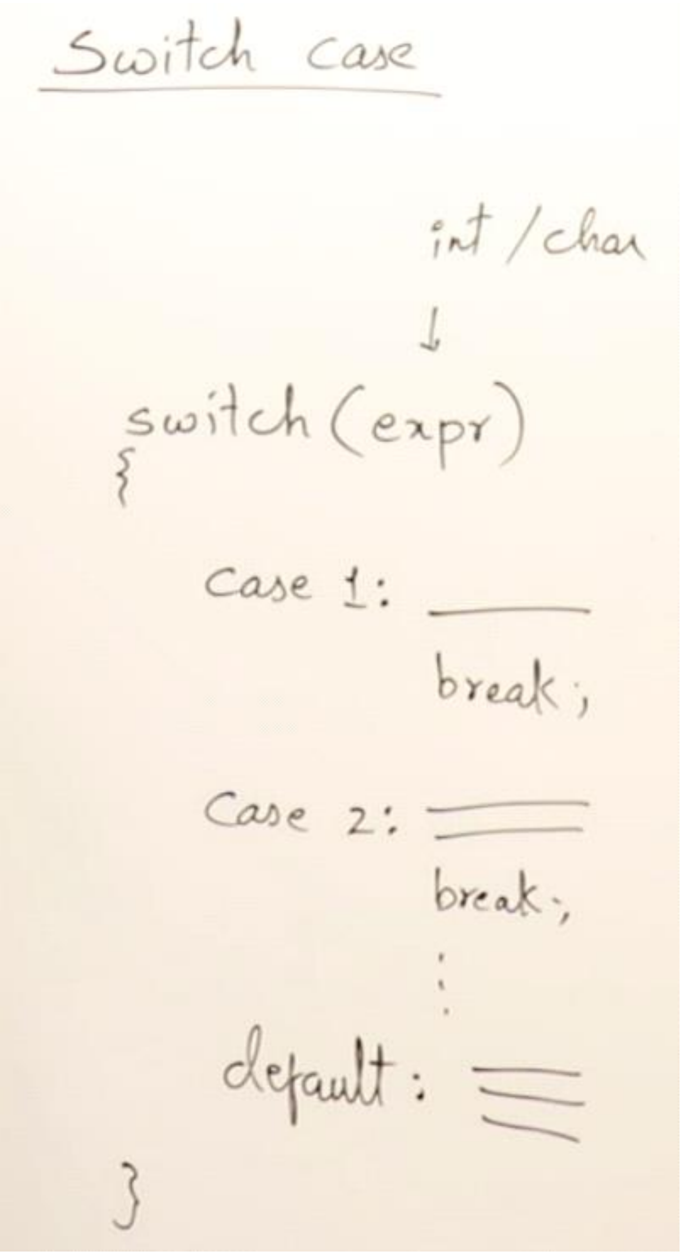
Note : if-else if-else conditions can also be nested.

Short Circuit

in And, if first condition is false it will not check second one and in OR, if first is correct then it will not check second one.



Switch-case Conditionals



Switch case

int / char

↓

```
switch (expr)
{
        case 1:
                        break;
        case 2:
                        break;
                        :
        default:
}
```

Q1. Write a switch case taking input between 1-7 and print the day of week.

Ans.

```
#include <iostream>
using namespace std;
int main() {
    int day;
    cout << "Enter a number (1-7): ";
    cin >> day;
    switch(day) {
        case 1:
            cout << "Monday" << endl;
            break;
        case 2:
            cout << "Tuesday" << endl;
            break;
        case 3:
            cout << "Wednesday" << endl;
            break;
        case 4:
            cout << "Thursday" << endl;
            break;
        case 5:
            cout << "Friday" << endl;
            break;
        case 6:
            cout << "Saturday" << endl;
            break;
        case 7:
            cout << "Sunday" << endl;
            break;
        default:
            cout << "Invalid input! Please enter a number
between 1 and 7." << endl;
    }
    return 0;
}
```

Ques. Write a program to find if a leap year or not.

Ans.

```
#include<iostream>
using namespace std;
int main() {
    int n;
    cout<<"Enter a year : ";
    cin>>n;
    if (n % 4 == 0){
        if (n % 100 == 0){
            if (n % 400 == 0){
                cout<<"Leap Year"<<endl;
            } else {
                cout<<"Not a Leap Year"<<endl;
            }
        } else {
            cout<<"Leap Year"<<endl;
        }
    } else {
        cout<<"Not a Leap Year"<<endl;
    }
    return 0;
}
```

Loops

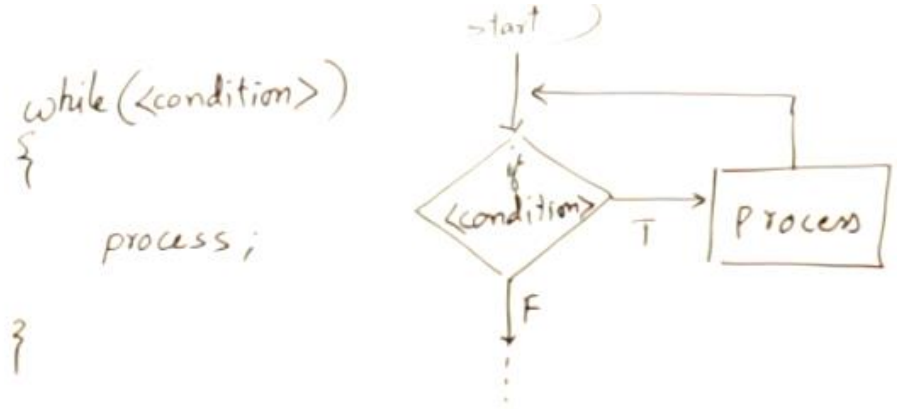
14 August 2024 22:47

loop is used to perform certain tasks multiple times.

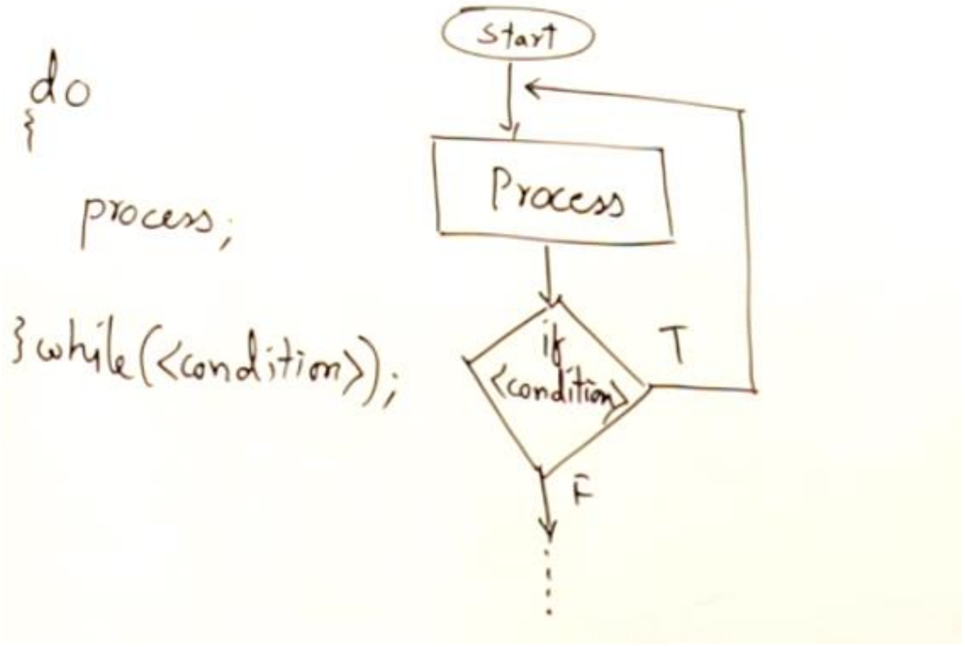
Types of Loops

1. While
2. Do….while
3. For
4. For Each

1. **While Loop** is used to run a loop for unlimited no. of times until a specific condition is met.
2. **Do….While Loop** is a variant of the While Loop but the only difference is that it will run exactly once before checking the condition.



```
while(<condition>)
{
    process;
}
```



```
do
{
    process;
}while(<condition>);
```

3. **For Loop** is used to execute certain tasks for a specific number of times. It is also known as counter controlled loop.

for(initialization; condition; updation)

Q1. Write a program to check if a number is Armstrong number or not.

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    int num, originalNum, remainder, n = 0;
    double result = 0.0;
    cout << "Enter an integer: ";
    cin >> num;
    originalNum = num;
    while (originalNum != 0) {
        originalNum /= 10;
        ++n;
    }
    originalNum = num;
    while (originalNum != 0) {
        remainder = originalNum % 10;
        result += pow(remainder, n);
        originalNum /= 10;
    }
    if ((int)result == num)
        cout << num << " is an Armstrong number." << endl;
    else
        cout << num << " is not an Armstrong number."
<< endl;
    return 0;
}
```

Arrays

15 August 2024 21:27

Array is collection of data of similar type.
Code:

```
#include<iostream>
using namespace std;
int main() {
    int number[5] = {1,2,3,4,4};
    char character[5] = {'A','B','C','D','E'};
    float f[2] = {1.0, 2.0};
    return 0;
}
```

<u>For Each loop on Arrays</u> :

```
#include<iostream>
using namespace std;
int main() {
    int  arr[5] = {1,2,3,4,5};
    for (int i : arr)
    cout<<i<<endl;

    return 0;
}
```

<u>Linear Search on Arrays</u>

```
#include<iostream>
using namespace std;
int main() {
    int arr[20] = {35, 98, 98, 100, 117, 202, 215, 278, 281, 
415, 420, 532, 685, 707, 868, 896, 905, 926, 936, 947};
    int target;
    cout<<"Enter the number : ";
    cin>>target;
    for (int x : arr){
        if (x == target){
            cout<<"Found"<<endl;
            return 0;
        }
    }
    cout<<"not Found"<<endl;
    return 1;
}
```

<u>Binary Search on Arrays</u>

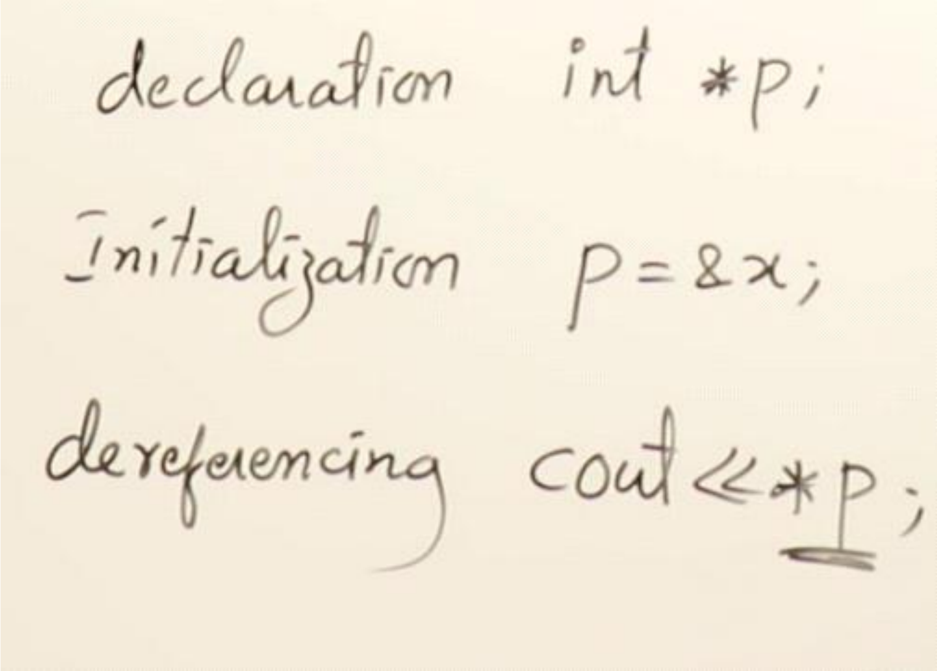
```
#include<iostream>
using namespace std;
int main() {
```

```cpp
    int arr[100] = {1, 35, 53, 54, 60, 63, 82, 83, 101, 107,
114, 115, 121, 124, 133, 137, 139, 154, 169, 172, 177, 192, 202,
228, 232, 232, 258, 258, 260, 261, 266, 267, 267, 274, 292, 353,
364, 389, 395, 419, 435, 436, 475, 479, 497, 500, 503, 504, 517,
518, 537, 558, 561, 566, 572, 578, 580, 581, 602, 603, 604, 616,
639, 643, 650, 652, 656, 669, 673, 686, 698, 702, 703, 748, 749,
777, 784, 801, 814, 817, 820, 838, 861, 867, 871, 872, 892, 904,
913, 915, 919, 920, 925, 932, 947, 955, 957, 959, 990, 991};
    int start = 0, end = 99;
    int target;
    cout<<"Enter the number : ";
    cin>>target;
    while (start <= end){
        int mid = (start + end) / 2;
        if (arr[mid] == target){
            cout<<"Found at "<<mid<<endl;
            return 0;
        } else if (arr[mid] > target){
            end = mid - 1;
        } else {
            start = mid + 1;
        }
    }
    cout<<"not found"<<endl;
    return 1;
}
```

Pointers

15 August 2024 22:20

Pointers are basically a variable that stores the address of another variable as a data.



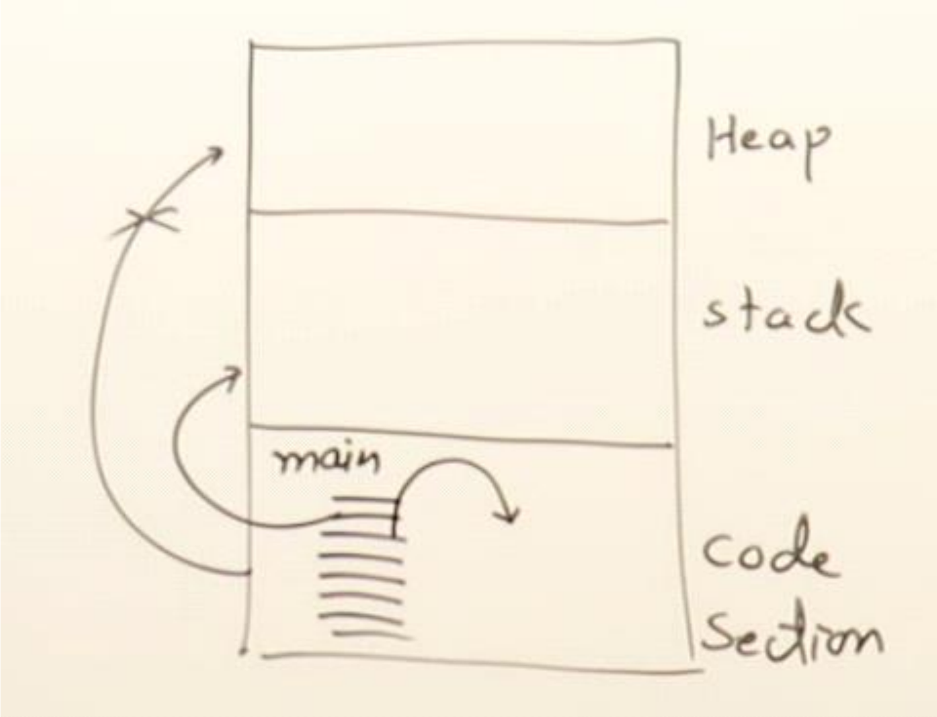
declaration int *p;

Initialization p=&x;

dereferencing cout<<*p;

Why Pointers?

As we know our memory is divided into 3 part i.e. main Code section, stack and heap section. Our program can access main and stack section but it cannot access heap section directly. So, we access it indirectly using pointers.



Pointers are also used to access files in program as they cannot be directly accessed by the program.

Even different devices in computers (Like Keyboard, mouse, printer, etc.) are accessed with the help of pointers in Operating System.

Even different devices in computers (Like Keyboard, mouse, printer, etc.) are accessed with the help of pointers in Operating System.

Note : we cannot use System programming with Java, C# as they don't have pointers but we can use System with JVM in Java and common language runtime in C# or .Net .

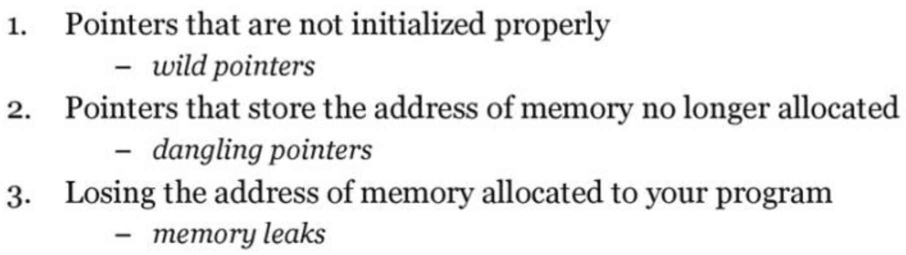
Heap Memory Allocation

```
#include<iostream>
using namespace std;
int main() {
    int* p, *q;
    p = (int*)malloc(sizeof(int) * 5); // C-Style
    q = new int[5]; // C++ Style
    for (int i = 0; i < 5; i++){
        p[i] = i*i;
        q[i] = i;
    }
    for (int i = 0; i < 5; i++){
        cout<<p[i]<<" "; // 0 1 4 9 16
    }
    free(p);
    delete[] q;
    return 0;
}
```

Pointer Arithmetic

```
#include<iostream>
using namespace std;
int main() {
    int arr[5] = {1,2,3,4,5};
    int *p = arr;
    for (int i = 0; i < 5; i++){
        cout<<*p + i<<endl;
    }
    return 0;
}
```

Problems with pointer



1. Pointers that are not initialized properly
   - *wild pointers*
2. Pointers that store the address of memory no longer allocated
   - *dangling pointers*
3. Losing the address of memory allocated to your program
   - *memory leaks*

Pointer to a Function W/O Arguments

```
#include<iostream>
```

```
using namespace std;
void diplay(){
    cout<<"Hello World"<<endl;
}
int main() {
    void (*fn)(); // initialization
    fn = diplay; //declaration
    (*fn)(); // Calling
    return 0;
}
```

Pointer to a Function With Arguments

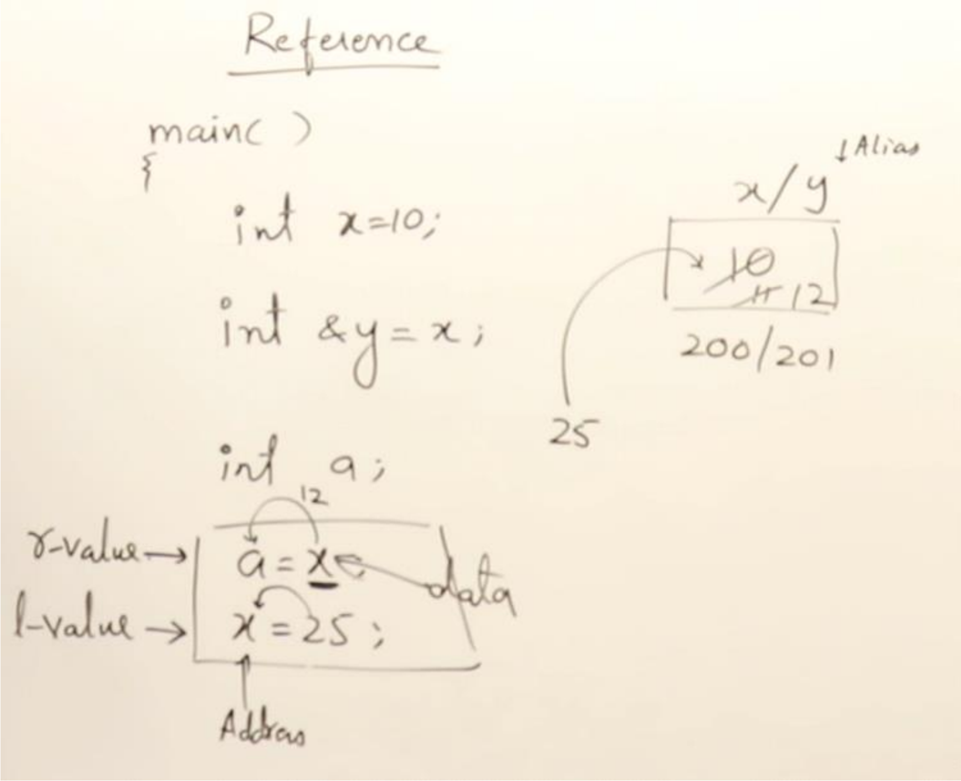
```
#include<iostream>
#include<string>
using namespace std;
void diplay(int age, string name){
    cout<<"Hello "<<name<<endl;
}
int main() {
    void (*fn)(int, string); // initialization Don't give x, y arg name
    fn = diplay; //declaration
    (*fn)(20, "Sachin"); // Calling
    return 0;
}
```

References

17 August 2024 13:34

reference is nothing but an another name/ alias for an another variable. changing it also affect the original variable.

```
#include<iostream>
using namespace std;
int main() {
    int a = 4;
    int &ref = a;
    cout << "original values of a = "<<a<<" and ref = 
"<<ref<<endl;
    ref /= 2;
    cout << "Values after dividing ref by 2 of ref;  a = "<<a<<" 
and ref = "<<ref<<endl;
    return 0;
}
```

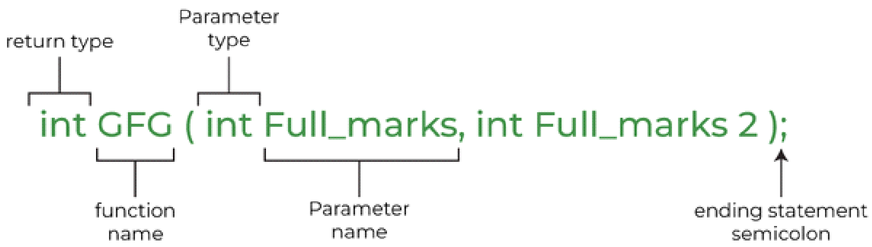


Reference doesn’t consumes any memory as it is an alias for original.

Functions

17 August 2024 20:33

Functions are a block of code which do some certain/specific tasks.

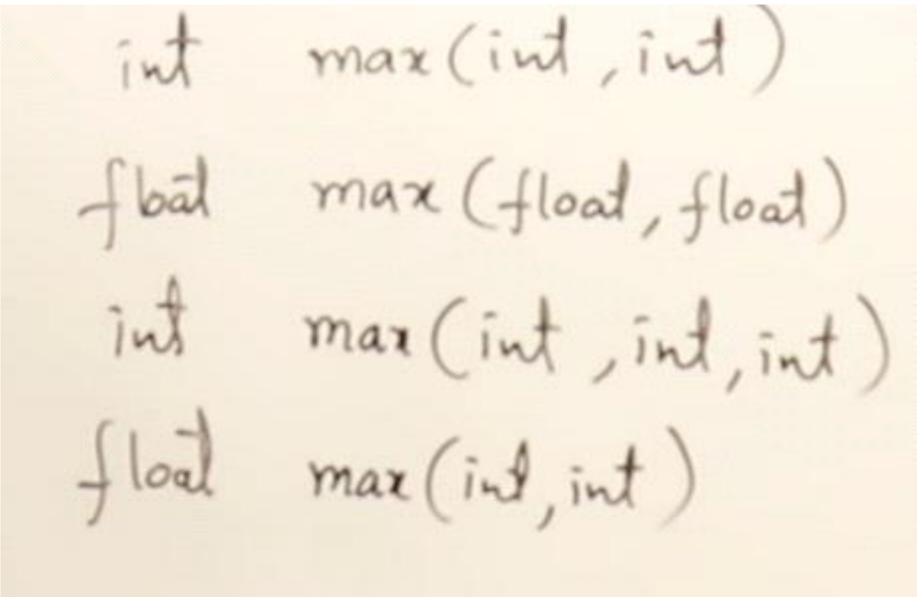

return type
Parameter type
int GFG (int Full_marks, int Full_marks 2);
function name
Parameter name
ending statement semicolon

Function Overloading

We can write functions with same name with different parameters and this is called function Overloading.

```
#include<iostream>
using namespace std;
int add(int x, int y){
    return x + y;
}
int add(double x, double y){
    return x + y;
}
int main() {
    cout<<add(5,4)<<endl; // 9
    cout<<add(5.25,4.37)<<endl; // 9
    return 0;
}
```

Other Examples:

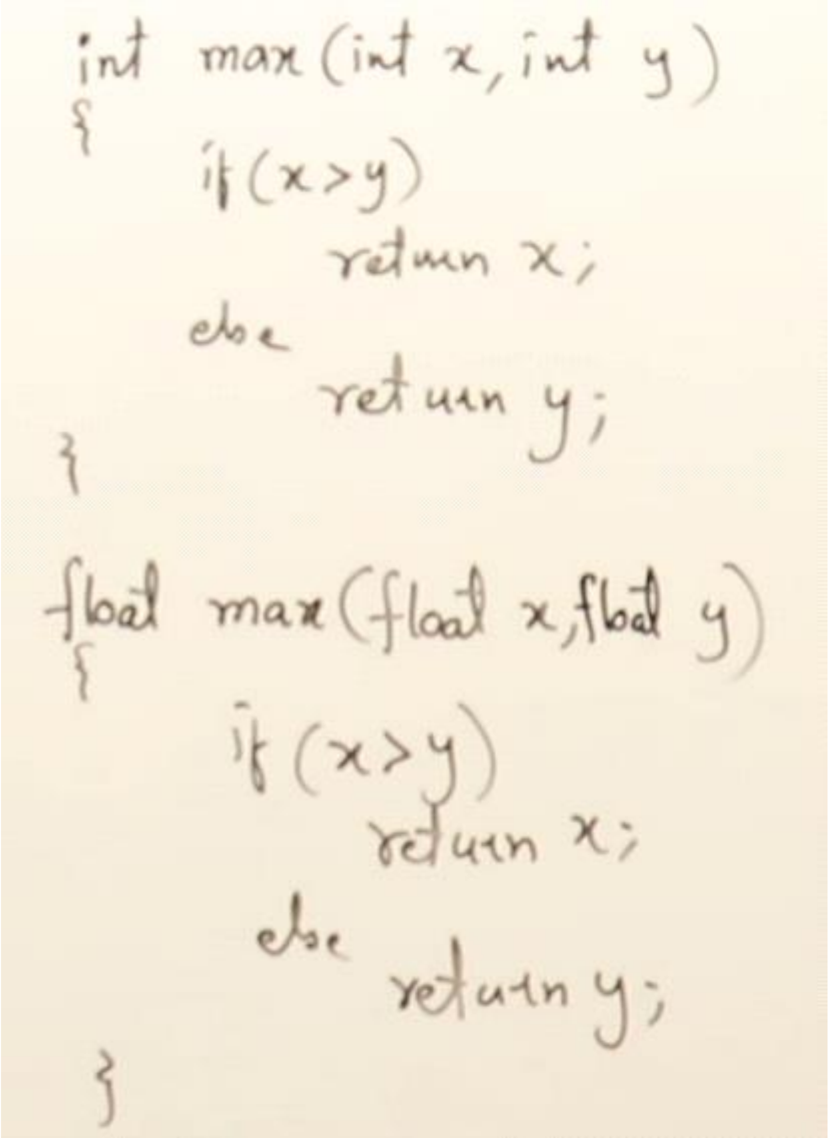


```
int   max(int, int)
float max(float, float)
int   max(int, int, int)
float max(int, int)
```

Function Templates

What are function templates? They are functions that are generic i.e. generalized.

What Problems function templates solve?



```
int max(int x, int y)
{
    if(x>y)
        return x;
    else
        return y;
}

float max(float x, float y)
{
    if (x>y)
        return x;
    else
        return y;
}
```

As you can see that these two functions have same logic but we have to write them again and again. And so, to reduce this redundancy we use function templates.

```
#include<iostream>
using namespace std;
template <class T>
T maximum(T x, T y){
    if (x > y){
        return x;
    } else {
        return y;
    }
}
int main() {
    cout<<maximum(4,5)<<endl;
    cout<<maximum(9.84,5.45)<<endl;
    return 0;
}
```

Default Arguments

```
#include<iostream>
using namespace std;
int add(int x, int y, int z = 0){
    return x + y + z;
}
int main() {
    cout<<add(5,4)<<endl;
    cout<<add(5,4,7)<<endl;
    return 0;
}
```

Parameter Passing

1. Pass by Value

In pass by value, a copy of values are passed and any change inside the funcion will only happen locally and will have no effect on actual arguments.

```
#include<iostream>
using namespace std;
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
    cout<<"Swapped Values of a = "<<a<<" & b = "<<b<<endl;
}
int main() {
    int a,b;
    a = 10;
    b = 20;
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" Before
Swapping"<<endl;
    swap(a,b);
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" after
Swapping"<<endl;
    return 0;
}
```

2. Pass by reference

In pass by reference, original variables are passed instead of their copies hence any change to them will change the value of original Arguments. No additional space is created.

```
#include<iostream>
using namespace std;
void swap(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
    cout<<"Swapped Values of a = "<<(a)<<" & b = "<<(b)<<endl;
}
int main() {
    int a,b;
    a = 10;
    b = 20;
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" Before
Swapping"<<endl;
    swap(a,b);
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" after
Swapping"<<endl;
    return 0;
}
```

3. Pass by Pointer

In pass by pointer, addresses of Actual arguments are passed as arguments and any change inside the function will affect the Actual Arguments.

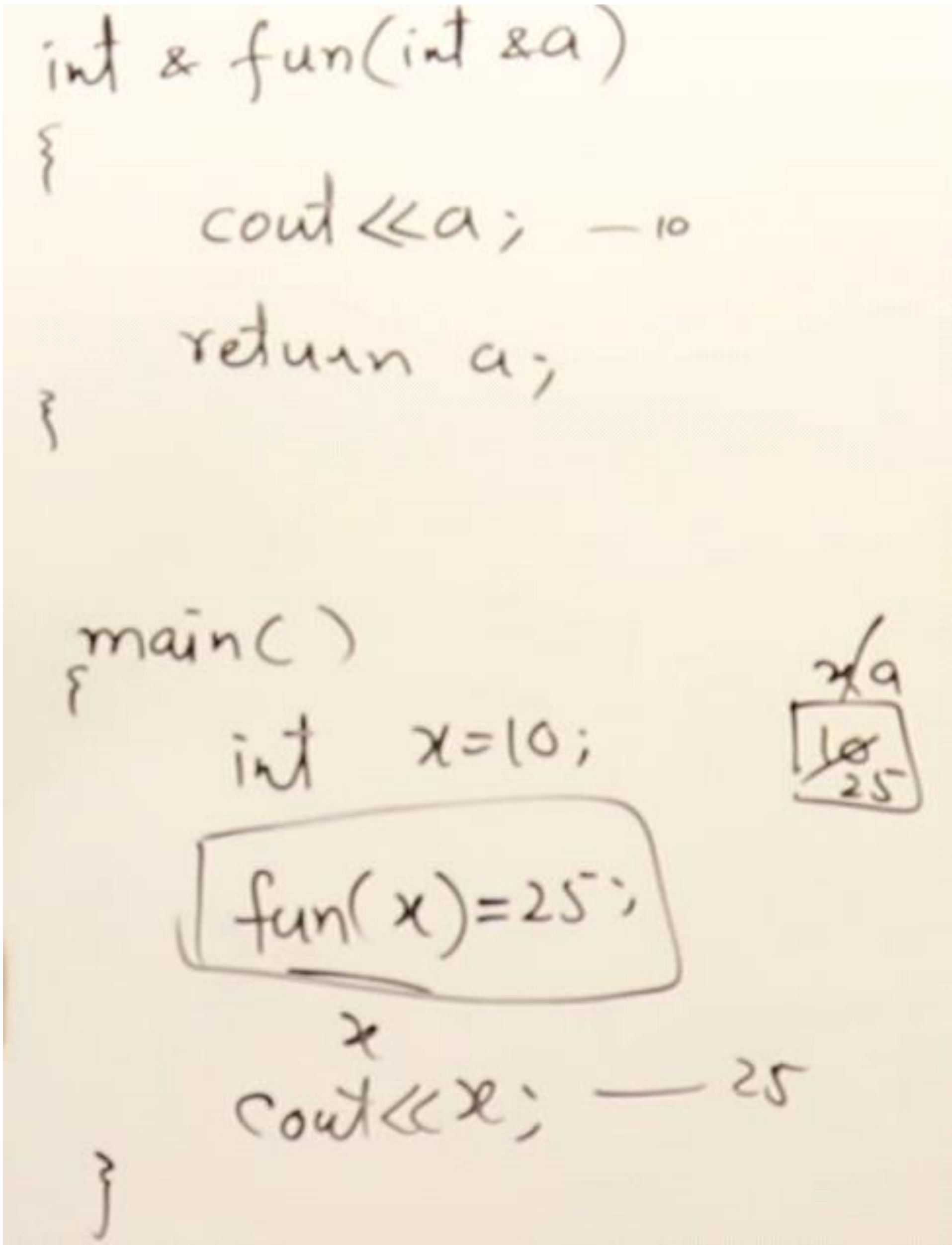
```
#include<iostream>
using namespace std;
void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
    cout<<"Swapped Values of a = "<<(*a)<<" & b = "<<(*b)<<endl;
}
int main() {
    int a,b;
    a = 10;
    b = 20;
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" Before
Swapping"<<endl;
```

```
    swap(&a,&b);
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" after Swapping"<<endl;
    return 0;
}
```

Return by Pointer

```
#include<iostream>
using namespace std;
int* array(int size){
    int* p = new int[size];
    for (int i = 0; i < size; i++){
        p[i] = i * 5;
    }
    return p;
}
int main() {
    int* ptr = array(5);
    for (int i = 0; i < 5; i++){
        cout<<*(ptr + i)<<endl;
    }
    return 0;
}
```

Return by Reference



```
int & fun(int &a)
{
    cout<<a;  — 10
    return a;
}

main()
{
    int x=10;
    fun(x)=25;
    cout<<x;  — 25
}
```

Scope means to which extent a variable can be worked with.

1. **Local Scope Variables** are variables that are defined inside some Methods, functions including parameters and can only be used inside that function/ method.
2. **Global Scope Variables** are variables that are defined outside every Methods, function and can be used everywhere.

Static Variables in Functions

When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call.

```
#include<iostream>
using namespace std;
int gl = 52; // Global Variable
```

```
void fun(){
    int a = 5; // local Variable
    static int x = 5; // Static Variable
    x++;
    cout<<a<<" "<<x<<endl;
}
int main() {
    fun(); // 5 6
    fun(); // 5 7
    fun();  // 5 8
    return 0;
}
```

Inline Functions are the functions that are expanded inline instead of calling a function call stack overhead.

OOPs

19 August 2024 22:25

The **Object oriented programming** is a programming paradigm based on the concepts of Objects and classes.

Principles of OOPS

1. **Abstraction** is the ability to hide complex / important/ internal logic hidden from normal users. For e.g. In a vehicle, only seats, steering , chassis frame are visible and its mechanism, engines are well hidden in the frame.
2. **Encapsulation** is the ability to encapsule/ Wrap the data as a single unit.
3. **inheritance** allows a child class to inherit from the parent class.
4. **Polymorphism** is the ability of object to take on different forms or behave in different ways depending on the context they are used. e.g. if we want to learn driving then do you learn BMW, Maruti, Suzuki? No, we learn Car and thereafter we can manage to drive any car.

Q. Write a class Quadrilateral.

```
#include<iostream>
using namespace std;
class Quadrilateral
{
private:
    int side1, side2, side3, side4, angle;
public:
    // Constructor
    Quadrilateral(int side1, int side2, int side3, int side4, int angle){
        this->side1 = side1;
        this->side2 = side2;
        this->side3 = side3;
        this->side4 = side4;
        this->angle = angle;
    }
    bool is_rect(){
        if (side1 == side3 && side2 == side4 && angle == 90){
            cout << "true" << endl;
            return true;
        }
        cout << "False" << endl;
        return false;
    }
    bool is_square(){
        if (side1 == side2 && side2 == side3 && side3 == side4 && angle ==
90){
            cout << "true" << endl;
            return true;
        }
        cout << "False" << endl;
        return false;
    }
};
int main() {
    Quadrilateral rectangle(40, 50, 40, 50, 90);
    cout << rectangle.is_rect() << endl;
    cout << rectangle.is_square() << endl;
    Quadrilateral square(50, 50, 50, 50, 90);
    cout << square.is_rect() << endl;
    cout << square.is_square() << endl;
    return 0;
}
```

Pointer to an Object

```
#include<iostream>
using namespace std;
class Rectangle{
public:
    int length, breadth;
    int perimeter(){
        return 2 * (length + breadth);
    }
    int area(){
        return length * breadth;
    }
};
int main() {
    Rectangle *r = new Rectangle(); // Allocate memory for the Rectangle
object in heap memory
    r->length = 6; // -> is arrow operator to visit length var space at
address r, so basically a dereferencing operator
    r->breadth = 4;
    cout << "Perimeter: " << r->perimeter() << endl;
    cout << "Area: " << r->area() << endl;
    delete r; // Free the allocated memory
    return 0;
}
```

Constructors

Constructor is something that is called during the initialization of an object.

- Constructor is a member function of a class
- It will have same name as class name
- It will not have return type
- Its should be public
- It can be declared as private also in some cases
- It is called when object is created
- It is used for initializing an object
- It can be overloaded
- If its not defined then class will have a default constructor
- Constructor can take default arguments

Destructors

Destructor is an instance member function that is invoked automatically whenever an objects is going to be deleted.

```
class Rectangle{
private:
    int length;
    int width;
public:
    // Constructor: Called when an object of the class is created
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }
    // Destructor: Called when an object of the class is destroyed
    ~Rectangle() {
        // Clean-up code, if needed, would go here
        cout << "Rectangle object is being deleted" << endl;
    }
};
```

Types of Constructors:

1. **Default / Built-in Constructors** are the constructors that are provided by the compiler that takes no arguments or all arguments having default values.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Default constructor
    Rectangle() {
        length = 0;
        width = 0;
    }
};
```

2. **Parameterized Constructors** are the user defined constructors that takes one or more arguments.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Parameterized constructor
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }
};
```

3. **Non - parameterized Constructors** are the type of constructors as the name suggest do

not take any arguments.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Non-parameterized (default) constructor
    Rectangle() {
        length = 1; // Default length
        width = 1;  // Default width
    }
};
```

4. Copy Constructers is a constructor that takes a reference to an object of the same class as a parameter.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Copy constructor
    Rectangle(const Rectangle &rect) {
        length = rect.length;
        width = rect.width;
    }
};
```

Types of Data Copying

1. **Shallow Copy** is type of copy where any change to copied variable may lead to change in Original passed Data.

```
#include<iostream>
using namespace std;
class Shallow {
public:
    int* data;
    Shallow(int value) {
        data = new int(value);
    }
    // Shallow copy constructor
    Shallow(const Shallow &source) : data(source.data) {
        cout << "Shallow copy constructor - shallow copy made" << endl;
    }
    void displayData() const {
        cout << "Data: " << *data << endl;
    }
    // Destructor is called, leading to potential double deletion
    ~Shallow() {
        delete data;
        cout << "Destructor freeing data" << endl;
    }
};
```

```
int main() {
    Shallow obj1(10);
    Shallow obj2 = obj1; // Shallow copy
    obj1.displayData();
    obj2.displayData();
    return 0;
}
```

2. **Deep Copy** is type of copy where any change to copied variable will not lead to change in Original passed Data.

```
#include<iostream>
using namespace std;
class Deep {
public:
    int* data;
    Deep(int value) {
        data = new int(value);
    }
    // Deep copy constructor
    Deep(const Deep &source) {
        data = new int(*source.data);
        cout << "Deep copy constructor - deep copy made" << endl;
    }
    void displayData() const {
        cout << "Data: " << *data << endl;
    }
    ~Deep() {
        delete data;
        cout << "Destructor freeing data" << endl;
    }
};
int main() {
    Deep obj1(10);
    Deep obj2 = obj1; // Deep copy
    obj1.displayData();
    obj2.displayData();
    return 0; // Destructor is called, but no issues as each object manages
its own resource
}
```

All types of Member Functions

1. **Constructors** - called when object is created
2. **Accessors** - used for knowing the value of data members
3. **Mutators** - used for changing value of data member
4. **Facilitator** - actual functions of class
5. **Enquiry** - used for checking if an object satisfies some condition
6. **Destructor** - used for releasing resources used by object

Scope Resolution Operator

:: is a Scope Resolution Operator and it is used to access a global variable when there is a local variable with the same name, or to define a function outside a class or namespace. It helps in distinguishing between different scopes and namespaces in your code.

```
class Rectangle{
private:
    int length;
    int width;
public:
    Rectangle(int l, int w){
        length = l;
        width = w;
    }
    void area();
    void change_length(int l){ // this will automatically become inline Function so its better to write it inside with scope resolution operator
        length = l;
    }
};
void Rectangle::area(){
    printf("%d \n", length * width);
}
```

Function inside Classes will automatically become inline Function so its better to write it inside with scope resolution operator.

Operator Overloading

24 August 2024 23:31

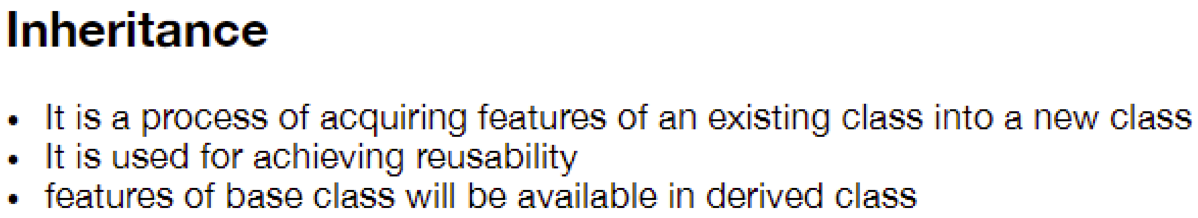
a manner in which Object Oriented systems allow the same operator name or symbol to be used for multiple operations.

```
#include<iostream>
using namespace std;
class Complex
{
private:
    int real, img;
public:
    Complex(int r = 0, int i = 0){
        real = r;
        img = i;
    }
    Complex add(const Complex& num) const {
        return Complex(real + num.real, img + num.img);
    }
    Complex operator+(const Complex& num) const {
        return Complex(real + num.real, img + num.img);
    }
    void display() const {
        std::cout << real << " + " << img << "i" << std::endl;
    }
};

int main() {
    Complex a(5,4);
    Complex b(5,4);
    Complex c = a + b;
    c.display();
    Complex d = a.add(b);
    d.display();
    return 0;
}
```

Inheritance

15 September 2024 16:20

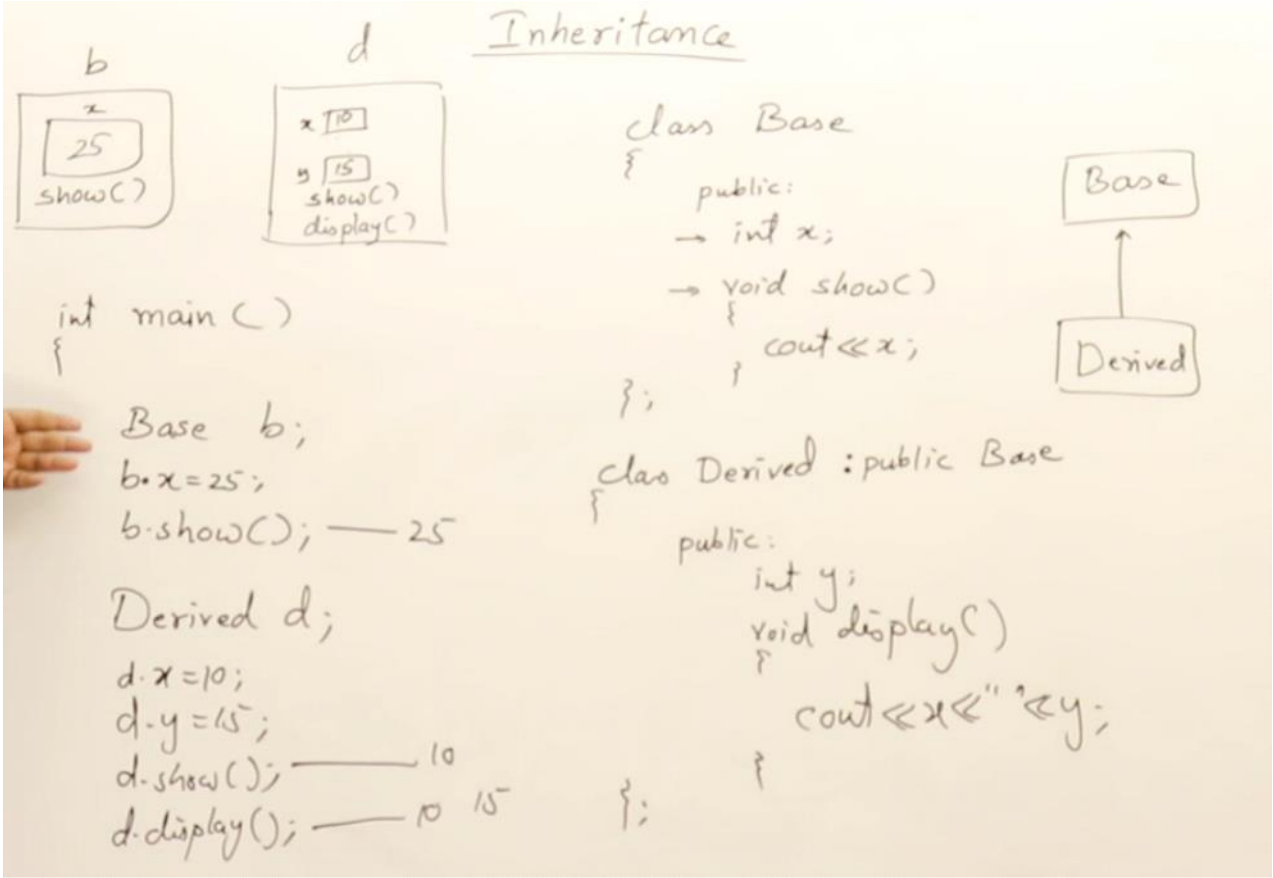
Inheritance is a way in which one class inherits the properties from another class.



Inheritance

- It is a process of acquiring features of an existing class into a new class
- It is used for achieving reusability
- features of base class will be available in derived class

```
#include<iostream>
using namespace std;
class Base{
public:
    int x;
    void print() {
        cout<<x<<endl;
    }
};
class Derived : public Base {
public:
    int y;
    void print(){
        cout<<x + y<<endl;
    }
};
int main() {
    Derived d;
    d.y = 5;
    d.x = 22;
    d.print();
    return 0;
}
```



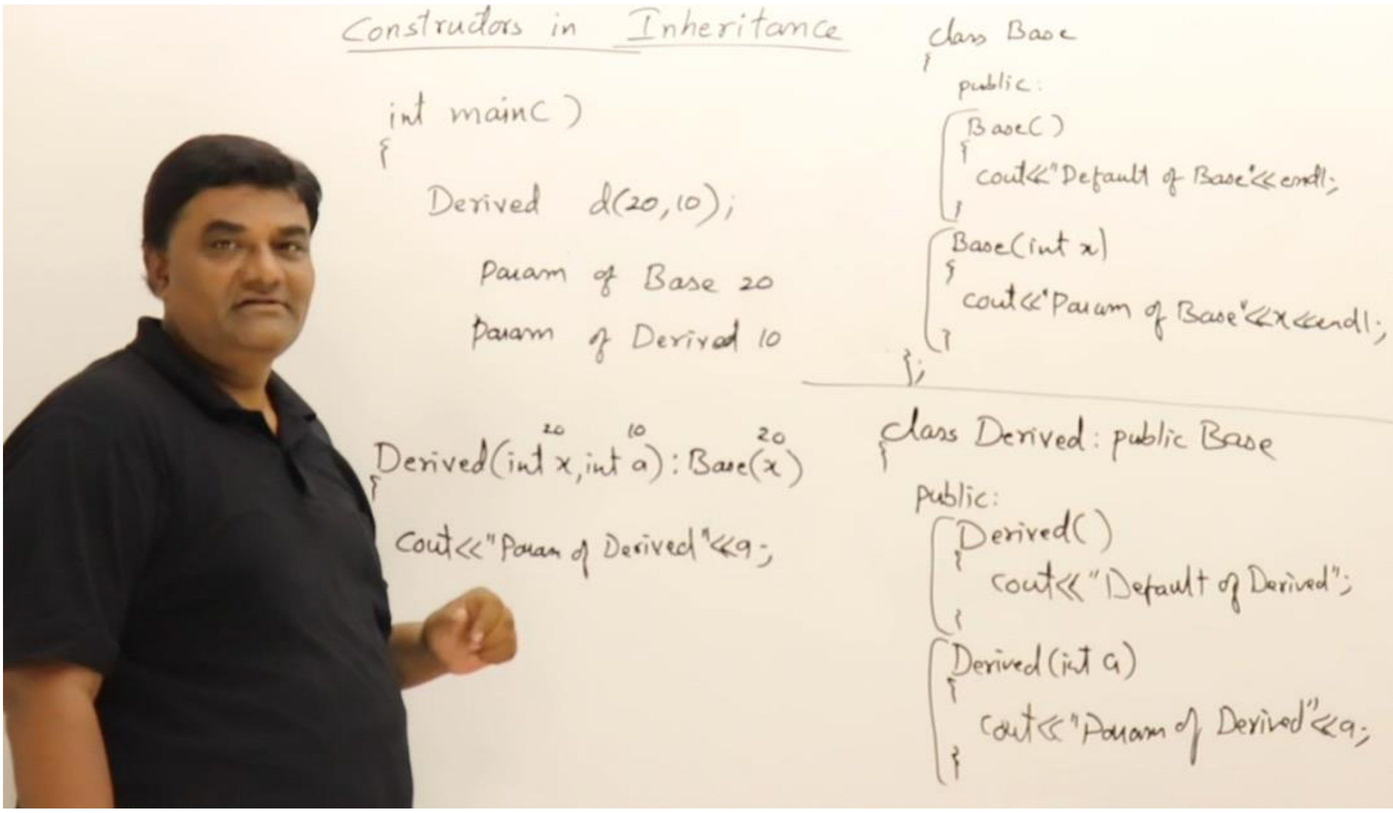
Inheritance

```
class Base
{
    public:
    → int x;
    → void show()
      {
         cout<<x;
      }
};

class Derived :public Base
{
    public:
       int y;
       void display()
       {
          cout<<x<<" "<<y;
       }
};

int main()
{
    Base b;
    b.x=25;
    b.show(); —— 25

    Derived d;
    d.x=10;
    d.y=15;
    d.show(); —— 10
    d.display(); —— 10 15
```

Q. Write a inherited Cuboid class from Base Rectangle Class.

```
class Rectangle{
protected:
    int length, width; // Protected to allow access in derived class
public:
    Rectangle(int l, int w){
        this->length = l;
        this->width = w;
    }
    void perimeter() {
        cout<<"Perimeter : "<< 2 *(length + width)<<endl;
    }
    void area() {
        cout<<"Area : "<< (length * width)<<endl;
    }
    void setLength(int l){
        this->length = l;
    }
    void setWidth(int w){
        this->width = w;
    }
};
class Cuboid : public Rectangle{
private:
int height;
public:
    Cuboid(int l, int w, int h) : Rectangle(l, w) { // Call Rectangle
constructor
        this->height = h;
    }
    void volume() {
        cout << "Volume: " << (length * width * height) << endl;
    }
    void setHeight(int h){
        this->height = h;
    }
};
```

Whenever we are calling a class object first the constructor from the base class is called and then the constructor from derived class is executed.



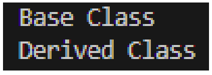
Constructors in Inheritance

```
int main( )
{
    Derived d(20,10);

        Param of Base 20
        Param of Derived 10

                 20       10           20
Derived(int x,int a):Base(x)
{
    Cout<<"Param of Derived"<<a;
```

```
class Base
{
  public:
  Base( )
  {
  cout<<"Default of Base"<<endl;
  }
  Base(int x)
  {
  cout<<"Param of Base"<<x<<endl;
  }
};
```

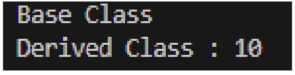
```
class Derived: public Base
{
  public:
  Derived( )
  {
    cout<<"Default of Derived";
  }
  Derived(int a)
  {
    cout<<"Param of Derived"<<a;
  }
```

```
#include<iostream>
using namespace std;
class Base{
public:
Base(){
    cout<<"Base Class"<<endl;
}
Base(int a){
    cout<<"Base Class : "<<a<<endl;
}
};
class Derived : public Base {
public:
Derived(){
    cout<<"Derived Class"<<endl;
}
Derived(int b){
    cout<<"Derived Class : "<<b<<endl;
}
Derived(int a, int b) : Base (a) {
    cout<<"Derived Class : "<<b<<endl;
}
};
int main() {
    Derived d;
```



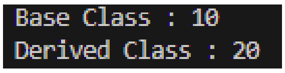
```
Base Class
Derived Class
```

```
    Derived d(10);
```



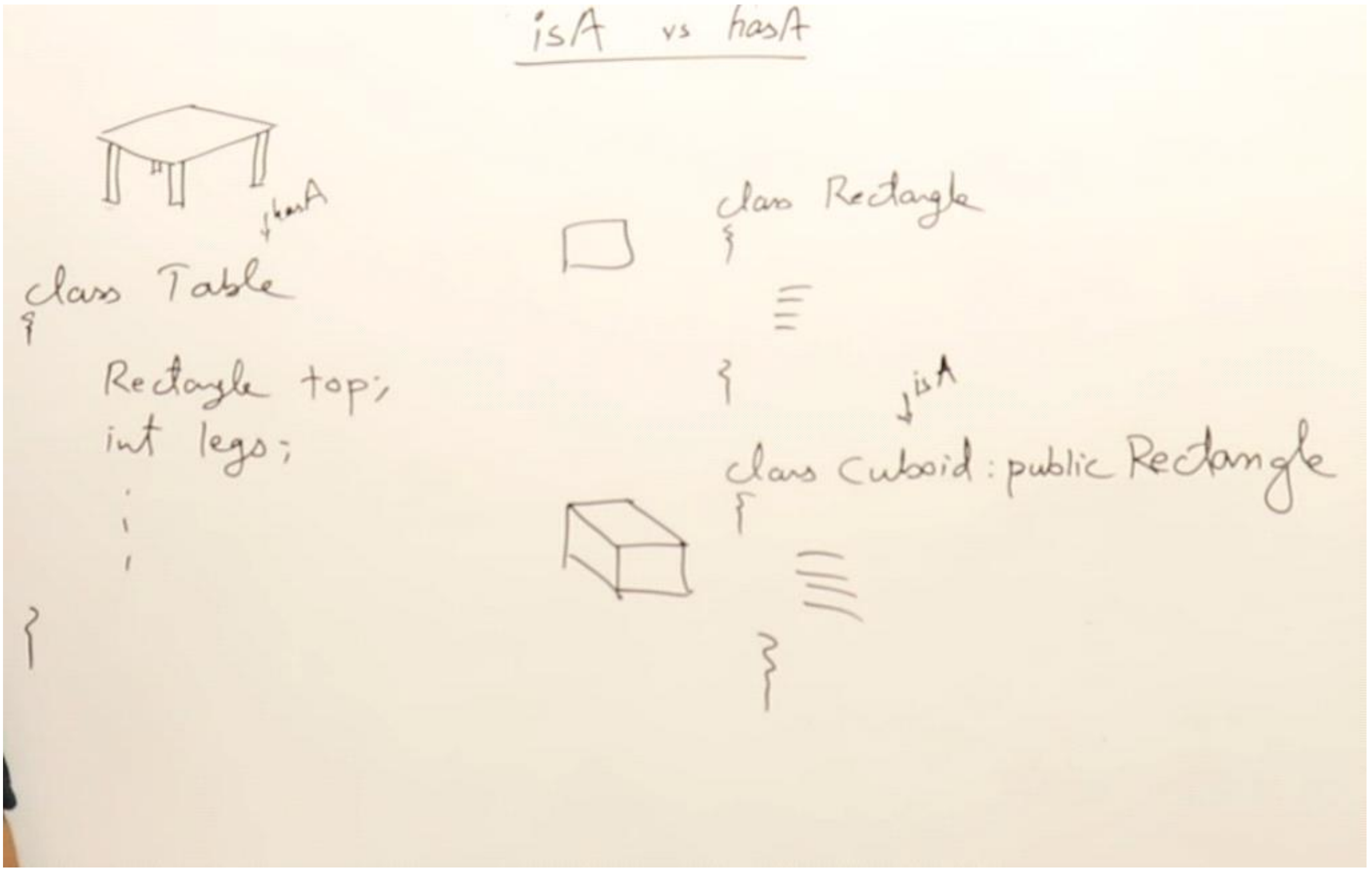
```
Base Class
Derived Class : 10
```

```
    Derived d(10,20);
```



```
Base Class : 10
Derived Class : 20
```

```
    return 0;
}
```



```
class Table
{
    Rectangle top;
    int legs;
    :
}

class Rectangle
{
    ≡
}

class Cuboid : public Rectangle
{
    ≡
}
```

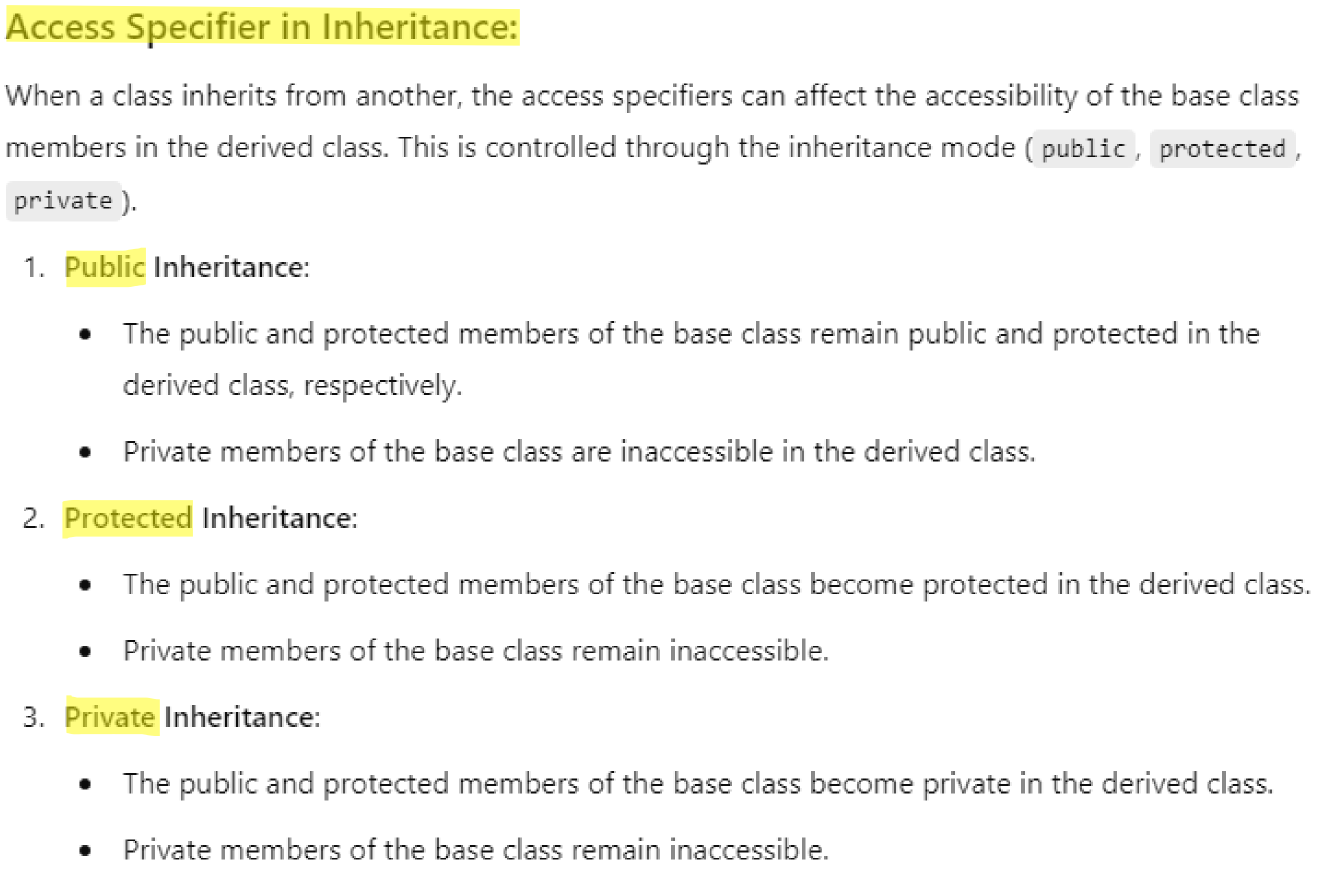
isA is relationship refers to **inheritance**, where one class is a more specific version of another class. The derived (or subclass) is said to be a type of the base (or superclass).

```
class Animal {
public:
    void makeSound() {
        cout << "Animal sound!" << endl;
    }
};
class Dog : public Animal {  // Dog isA Animal
public:
    void bark() {
        cout << "Bark!" << endl;
    }
};
```

hasA

- relationship refers to **composition** or **aggregation**, where a class contains an object of another class as one of its members.
- This relationship describes ownership or usage, where one class has a reference to another class but does not inherit from it.

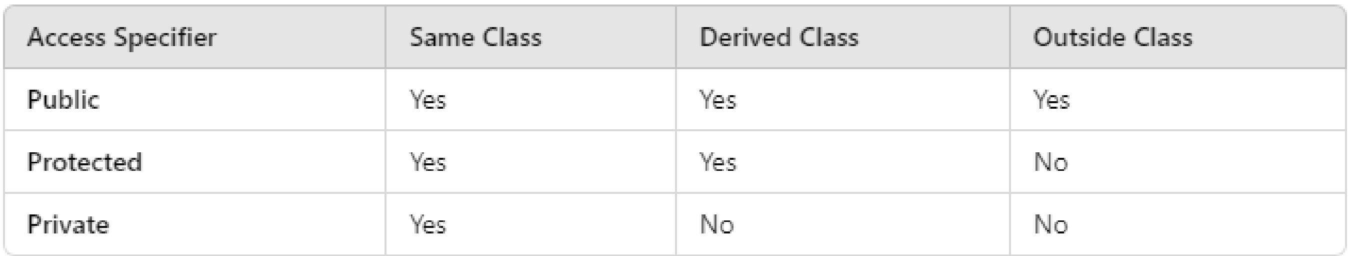
```
class Engine {
public:
    void start() {
        cout << "Engine started" << endl;
    }
};
class Car {
private:
    Engine engine;  // Car hasA Engine
public:
    void startCar() {
        engine.start();
        cout << "Car is running" << endl;
    }
};
```



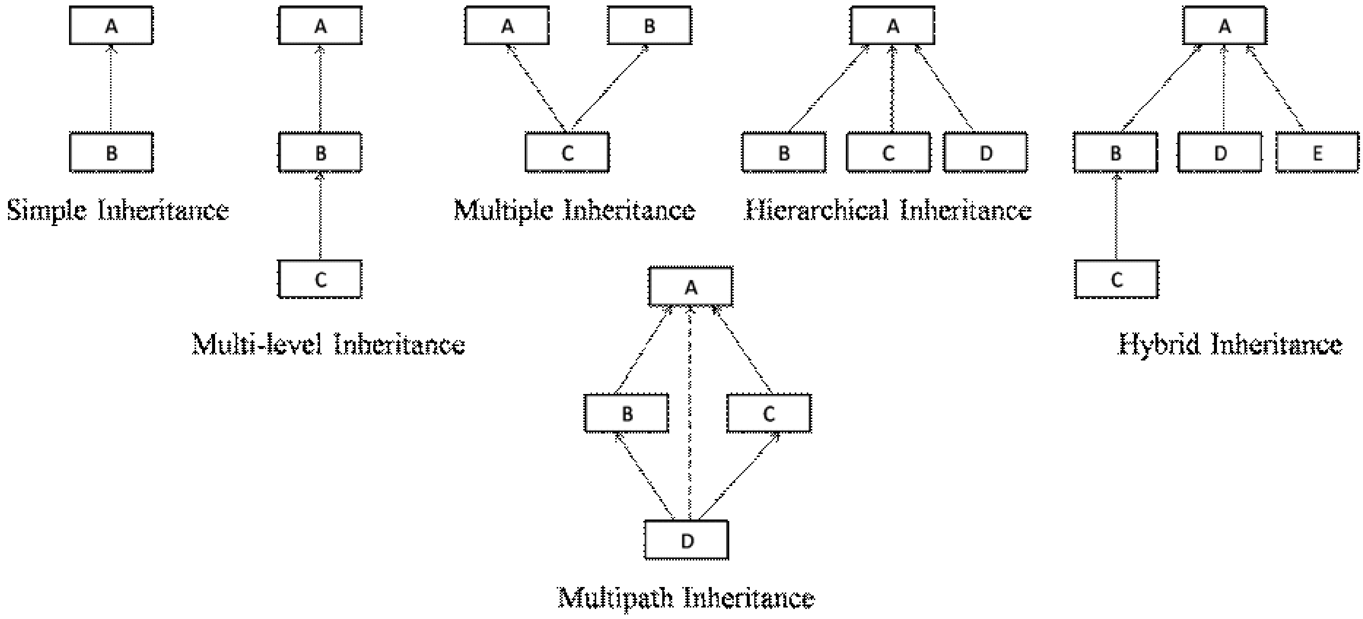
Access Specifier in Inheritance:

When a class inherits from another, the access specifiers can affect the accessibility of the base class members in the derived class. This is controlled through the inheritance mode (`public`, `protected`, `private`).

1. Public Inheritance:
   - The public and protected members of the base class remain public and protected in the derived class, respectively.
   - Private members of the base class are inaccessible in the derived class.
2. Protected Inheritance:
   - The public and protected members of the base class become protected in the derived class.
   - Private members of the base class remain inaccessible.
3. Private Inheritance:
   - The public and protected members of the base class become private in the derived class.
   - Private members of the base class remain inaccessible.



Access Specifier	Same Class	Derived Class	Outside Class
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No



1. Single Inheritance

```
class Animal {
public:
    void sound() {
        cout << "Animal makes a sound" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
```

2. Multiple Inheritance

```
class Engine {
public:
    void startEngine() {
        cout << "Engine started" << endl;
    }
};
class Transmission {
public:
    void startTransmission() {
        cout << "Transmission started" << endl;
    }
};
class Car : public Engine, public Transmission {
    // Car inherits from both Engine and Transmission
};
```

3. Multilevel Inheritance

```
class LivingBeing {
public:
    void grow() {
        cout << "Living being grows" << endl;
    }
};
class Animal : public LivingBeing {
public:
    void move() {
        cout << "Animal moves" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
```

4. Hierarchical Inheritance

```
class Animal {
public:
    void breathe() {
        cout << "Animal breathes" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
class Cat : public Animal {
public:
    void meow() {
        cout << "Cat meows" << endl;
    }
};
```

5. Hybrid Inheritance

```
class Animal {
public:
    void breathe() {
        cout << "Animal breathes" << endl;
    }
};
class Mammal : public Animal {
public:
    void feedMilk() {
        cout << "Mammal feeds milk" << endl;
    }
};
class Bird : public Animal {
public:
    void layEggs() {
        cout << "Bird lays eggs" << endl;
    }
};
class Bat : public Mammal, public Bird {
public:
    void fly() {
        cout << "Bat flies" << endl;
    }
};
```

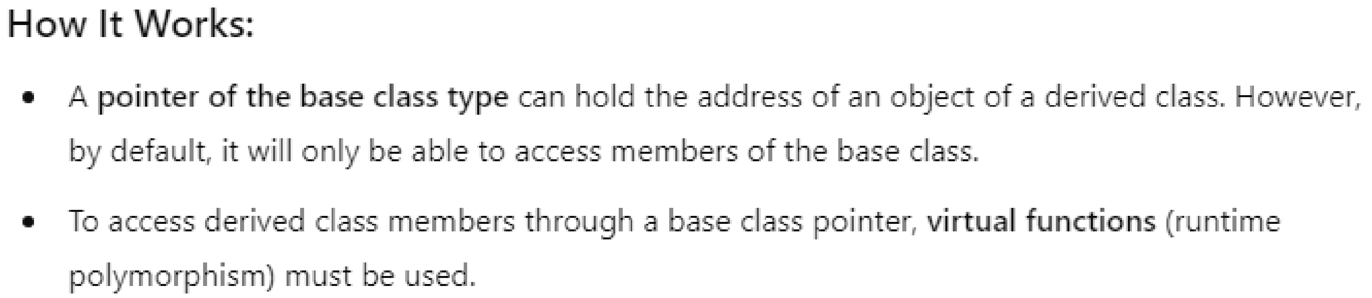
6. Virtual/ Multipath/ Diamond Inheritance

```
class Animal {
public:
    void speak() {
        cout << "Animal speaks" << endl;
    }
};
class Mammal : virtual public Animal {
    // Virtual inheritance ensures only one copy of Animal is inherited
};
class Bird : virtual public Animal {
    // Virtual inheritance ensures only one copy of Animal is inherited
};
class Bat : public Mammal, public Bird {
    // Only one copy of Animal is inherited, no ambiguity
};
```

Base Class Pointer

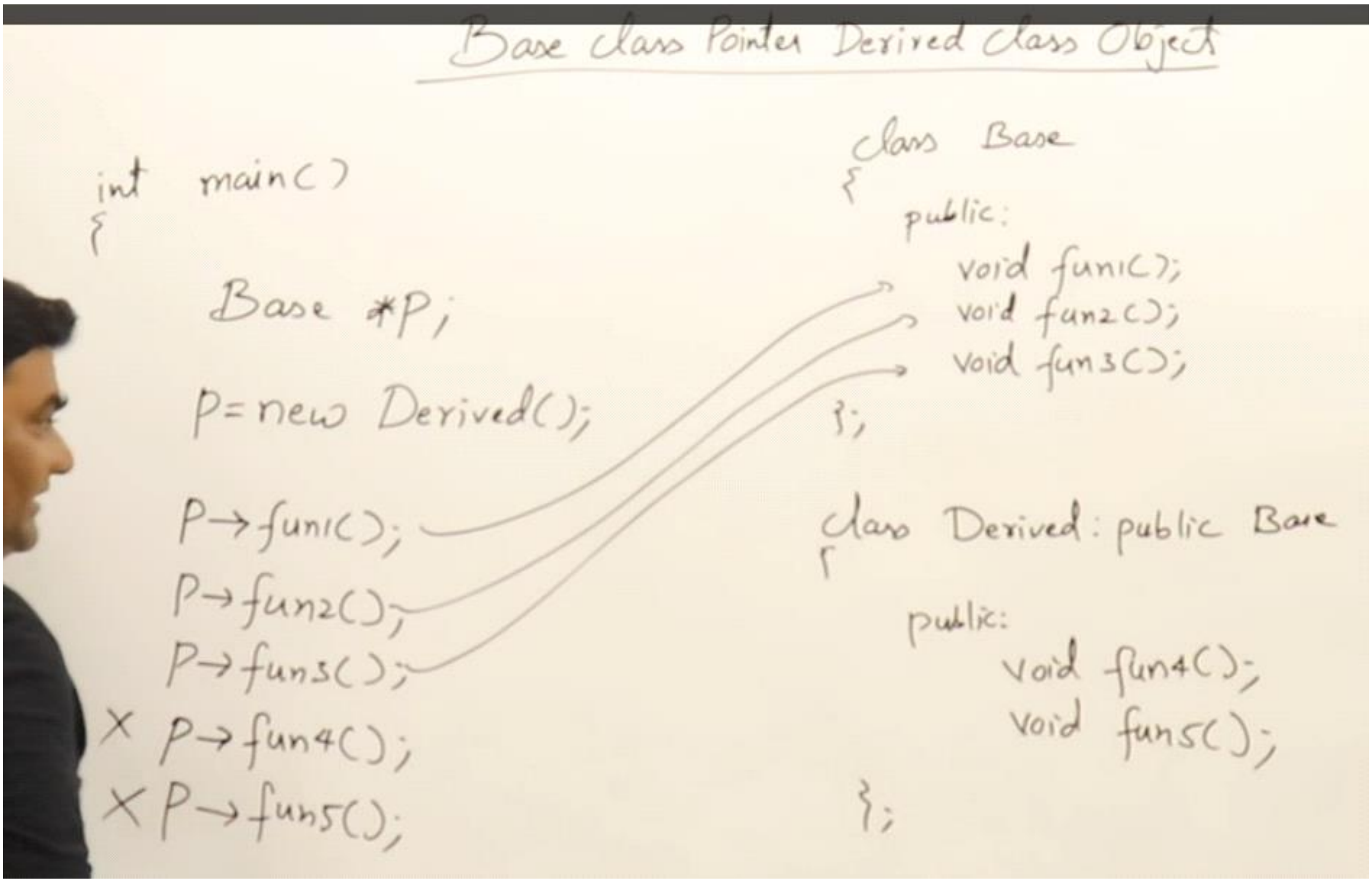
22 September 2024 20:12

base class pointers can point to **derived class objects**. This is a fundamental concept in **polymorphism** in object-oriented programming, where a base class pointer or reference can be used to refer to any object derived from that base class.



How It Works:

- A **pointer of the base class type** can hold the address of an object of a derived class. However, by default, it will only be able to access members of the base class.
- To access derived class members through a base class pointer, **virtual functions** (runtime polymorphism) must be used.



Base class Pointer Derived class Object

```
int main()
{
    Base *P;

    P=new Derived();

    P→fun1();
    P→fun2();
    P→fun3();
 × P→fun4();
 × P→fun5();
```

```
class Base
{
 public:
    void fun1();
    void fun2();
    void fun3();
};

class Derived : public Base
{
 public:
    void fun4();
    void fun5();
};
```

Base class Pointer pointing to derived class object

- Base class pointer can point on derived class object
- But only those functions which are in base class, can be called
- If derived class is having overrides functions they will not be called unless base class functions are declared as virtual
- Derived class pointer cannot point on base class object

Example 1

```
class Base
{
public:
    void fun1()
    {
        cout<<“fun1 of Base "<<endl;
    }
};
```

Example 2

```
class Derived: public Base
{
public:
    void fun2()
    {
        cout<<“fun2 of Derived"<<endl;
    }
};

class Rectangle
{
public:
    void area()
    {
        cout<<“Area of Rectangle"<<endl;
    }
};

class Cuboid: public Rectangle
{
public:
    void volume()
    {
        cout<<“Volume of Cuboid"<<endl;
    }
};
```

```
#include <iostream>
using namespace std;
class Base {
public:
    void display() {
        cout << "Base class display function." << endl;
    }
    virtual void show() {
        cout << "Base class show function." << endl;
    }
};
class Derived : public Base {
public:
    void display() {
        cout << "Derived class display function." << endl;
    }
    void show() override {
        cout << "Derived class show function." << endl;
    }
};
int main() {
    Base* basePtr;        // Base class pointer
    Derived derivedObj;   // Derived class object
    basePtr = &derivedObj;  // Base class pointer pointing to
derived class object
    basePtr->display();    // Calls Base class version (no
polymorphism)
    basePtr->show();       // Calls Derived class version
(polymorphism through virtual functions)
    return 0;
}
```

Polymorphism

22 September 2024 20:45

Polymorphism is the ability of object to take on different forms or behave in different ways depending on the context they are used. e.g. if we want to learn driving then do you learn BMW, Maruti, Suzuki? No, we learn Car and thereafter we can manage to drive any car.

Runtime polymorphism

Runtime polymorphism (also known as **dynamic polymorphism**) is a concept in OOPs where the method to be invoked is determined at runtime, rather than at compile time. It allows for flexibility in how objects interact, enabling the same piece of code to handle different types of objects and behave differently depending on the object's actual type.

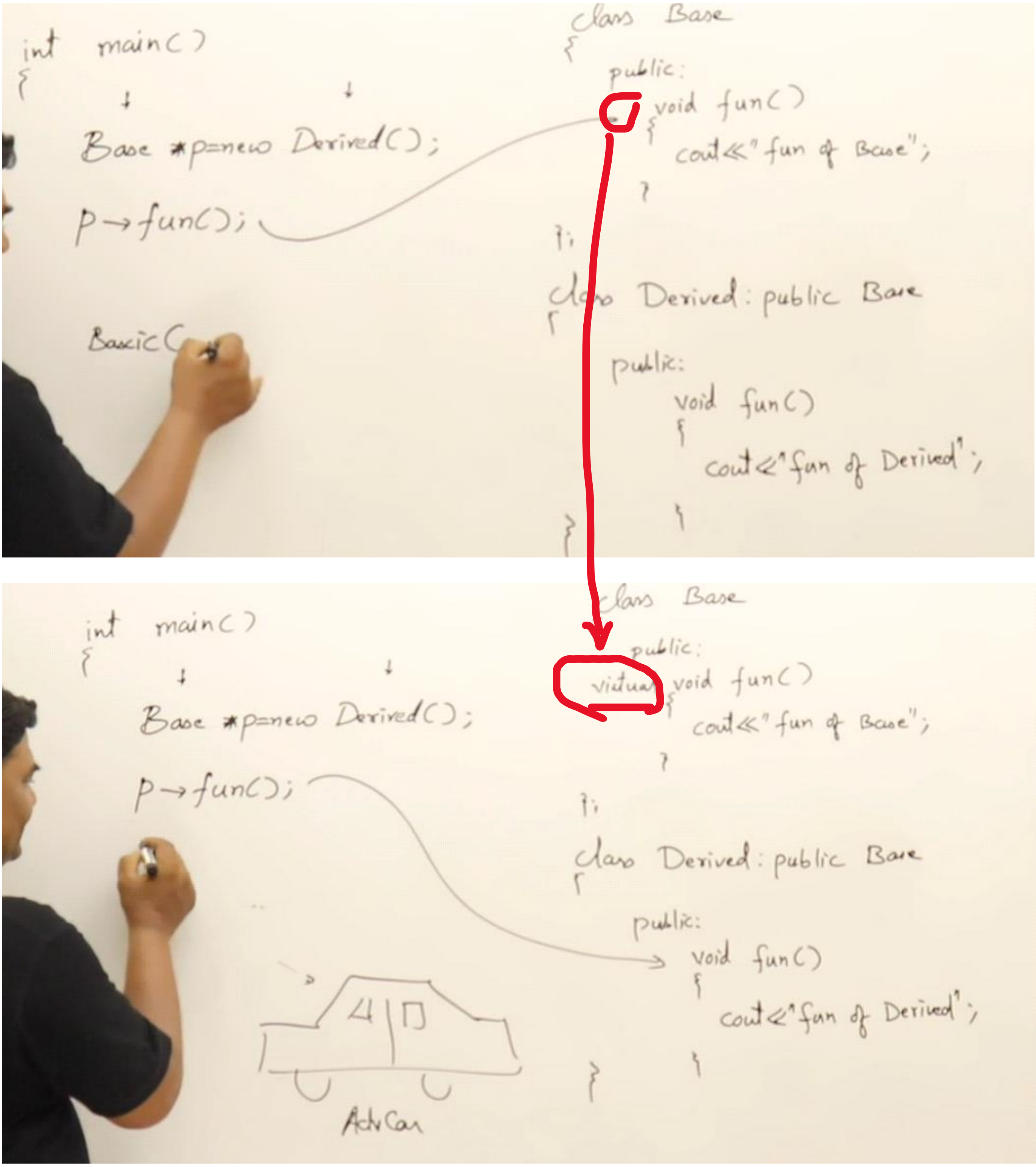
Function Overriding

Function overriding occurs when a derived class provides a specific implementation for a method that is already defined in its base (or parent) class. This allows the derived class to customize or completely replace the behaviour of the inherited method.

```
#include<iostream>
using namespace std;
class Base {
public:
    virtual void display(){
        cout<<"Base Class"<<endl;
    }
};
class Child1 : public Base {
};
class Child2 : public Base {
public:
    void display() override{
        cout<<"Child Class"<<endl;
    }
};
int main() {
    Base a;
    Child1 b;
    Child2 c;
    a.display();
    b.display();
    c.display();
    return 0;
}
```

Virtual Functions

A **virtual function** in C++ is a member function in a base class that can be overridden in derived classes. This feature enables **runtime polymorphism**.



```
int main()
{
    Base *p=new Derived();
    p->fun();

    BasicC
```

```
class Base
{
    public:
        void fun()
        {
            cout<<"fun of Base";
        }
};
class Derived: public Base
{
    public:
        void fun()
        {
            cout<<"fun of Derived";
        }
}
```

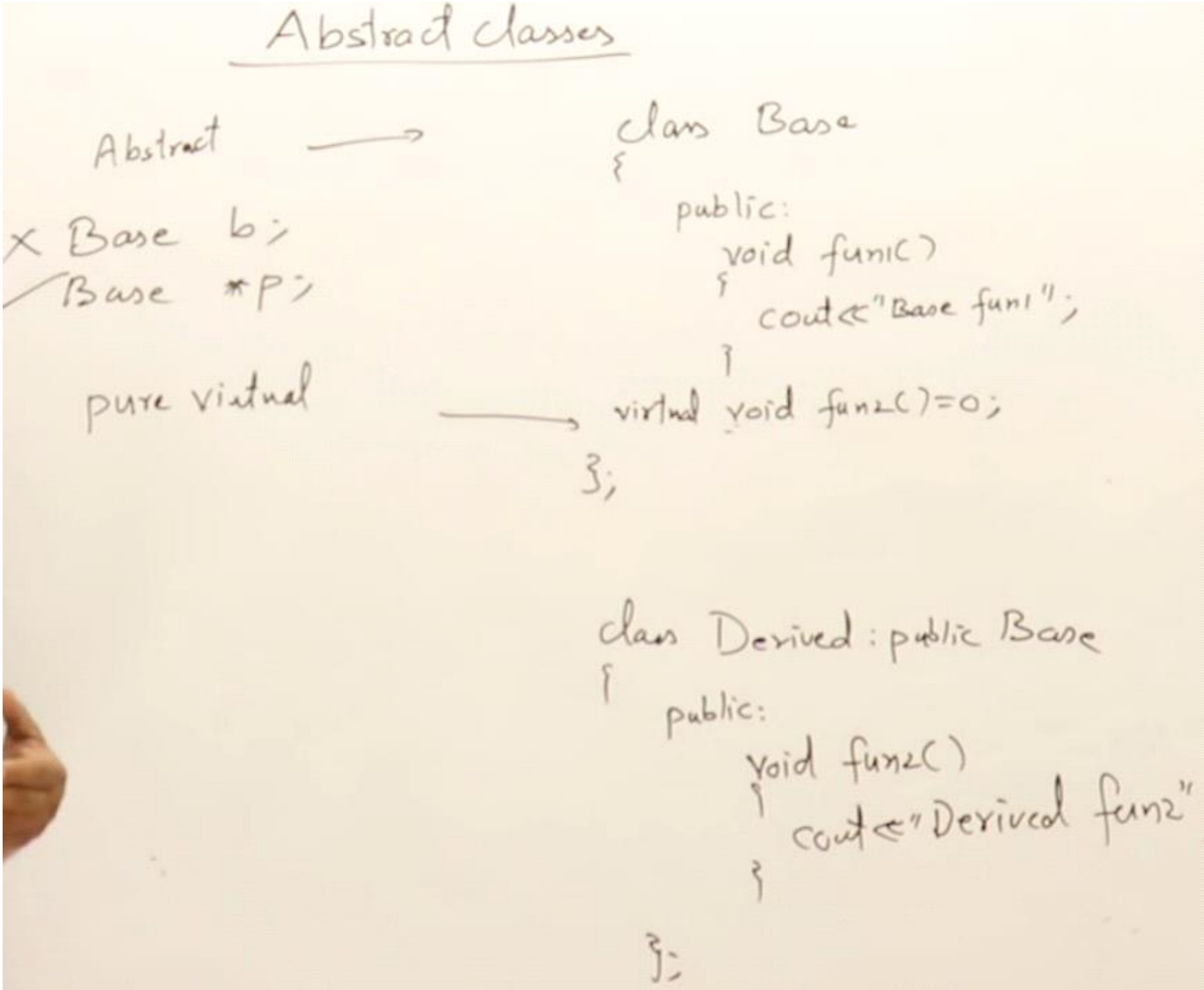
```
int main()
{
    Base *p=new Derived();
    p->fun();
```

```
class Base
{
    public:
        virtual void fun()
        {
            cout<<"fun of Base";
        }
};
class Derived: public Base
{
    public:
        void fun()
        {
            cout<<"fun of Derived";
        }
}
```

Abstract Classes

an abstract class is a class that contains at least one pure virtual function. A pure virtual function is a function that is declared in the base class and has no implementation in that class.

Classes that inherit from an abstract class must provide an implementation for all pure virtual functions, otherwise, they too become abstract classes.



Abstract classes

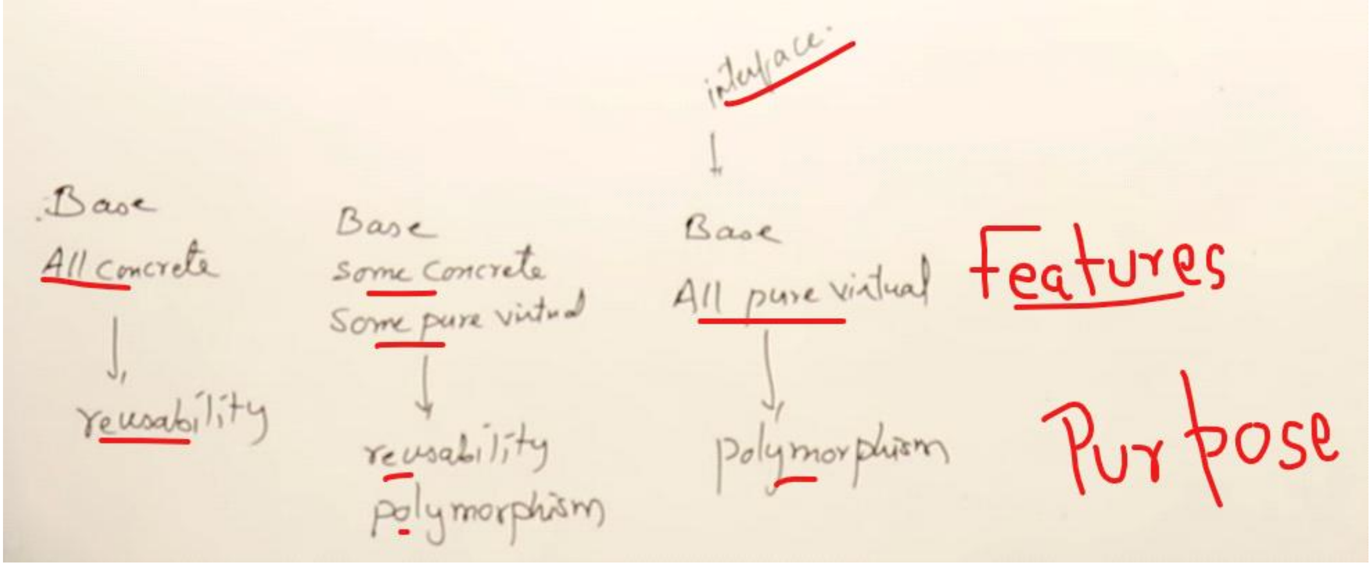
Abstract → class Base

× Base b;

✓ Base *p;

pure virtual → virtual void fun2()=0;

```
class Base
{
  public:
    void fun1()
    {
      cout<<"Base fun1";
    }
  virtual void fun2()=0;
};


class Derived : public Base
{
  public:
      void fun2()
      {
        cout<<"Derived fun2"
      }

};
```



Base
All concrete
↓
reusability

Base
Some concrete
Some pure virtual
↓
reusability
polymorphism

interface
↓
Base
All pure virtual
↓
polymorphism

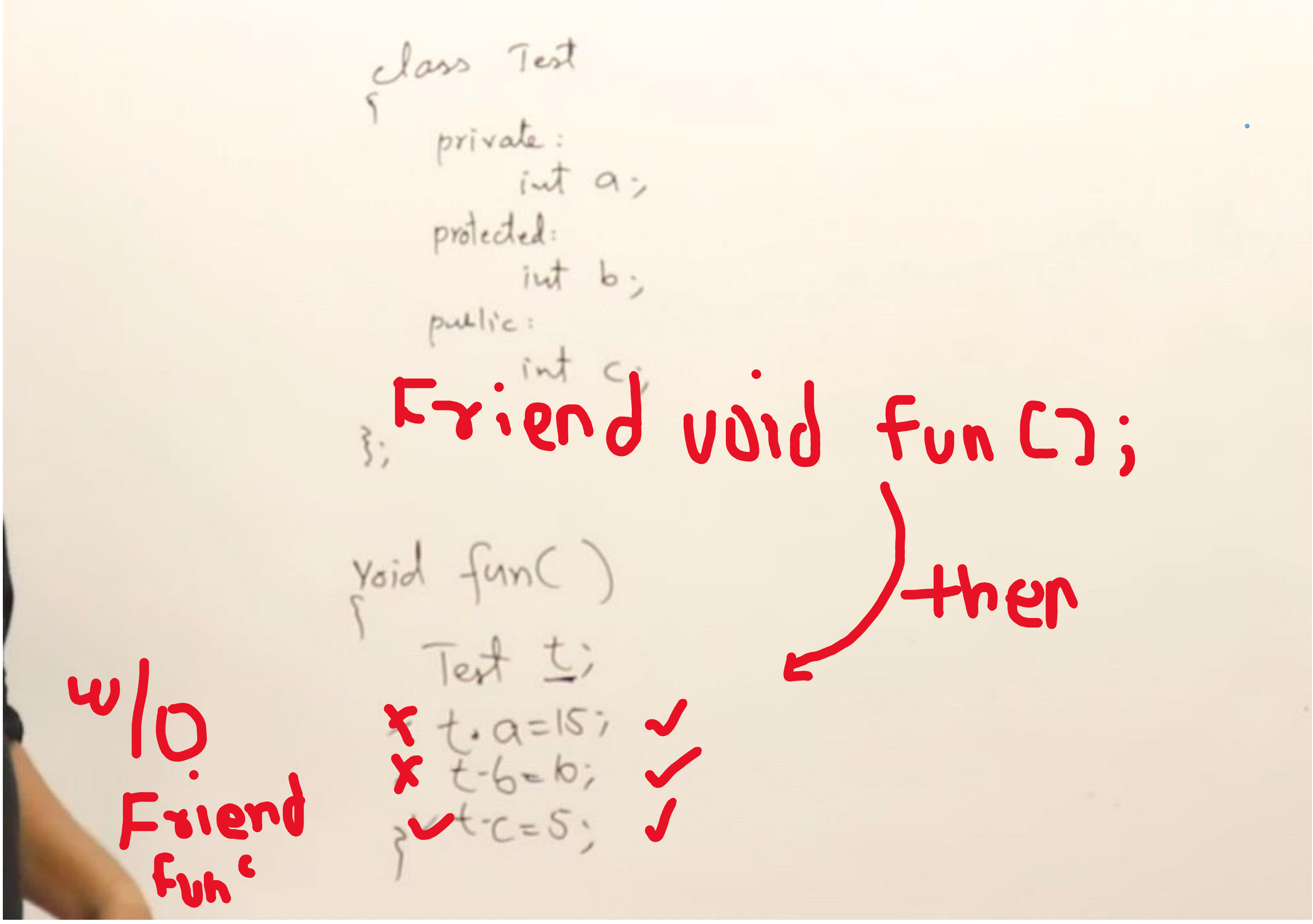
Features

Purpose

Friend and Static Members Inner Class

23 September 2024 11:03

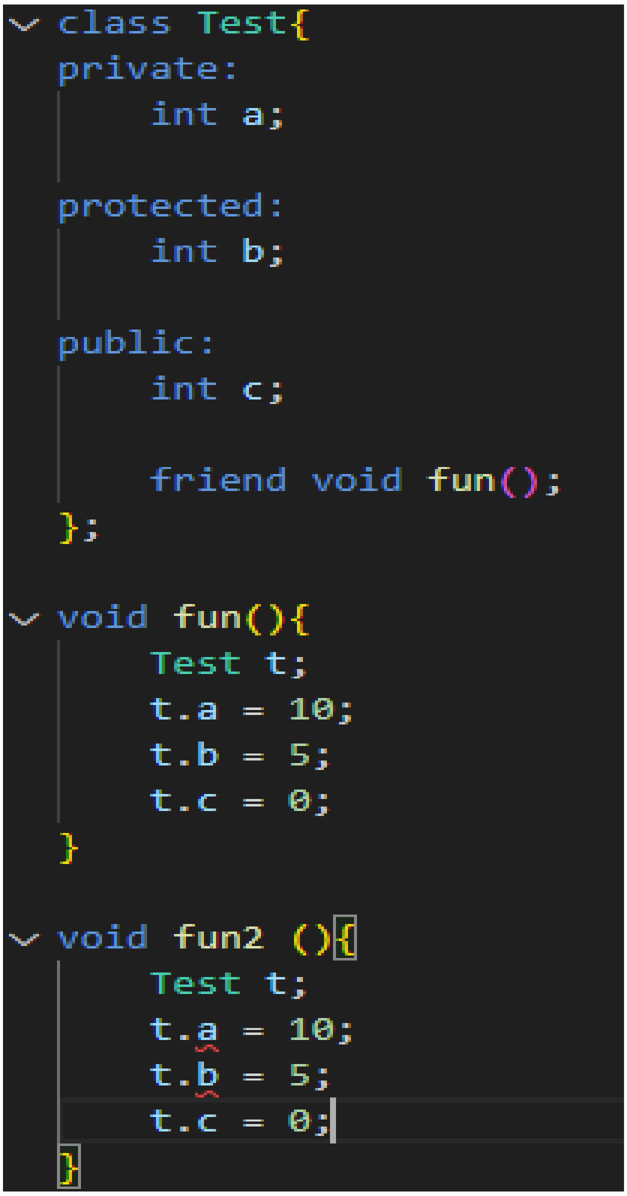
In cpp a function outside the class is not allowed its private and protected members, hence they require a **friend function** inside the class.



```
class Test
{
    private:
        int a;
    protected:
        int b;
    public:
        int c;
    Friend void fun();
};

void fun()
{
    Test t;
  ✗ t.a=15;   ✓
  ✗ t.b=10;   ✓
  ✓ t.c=5;    ✓
}
```

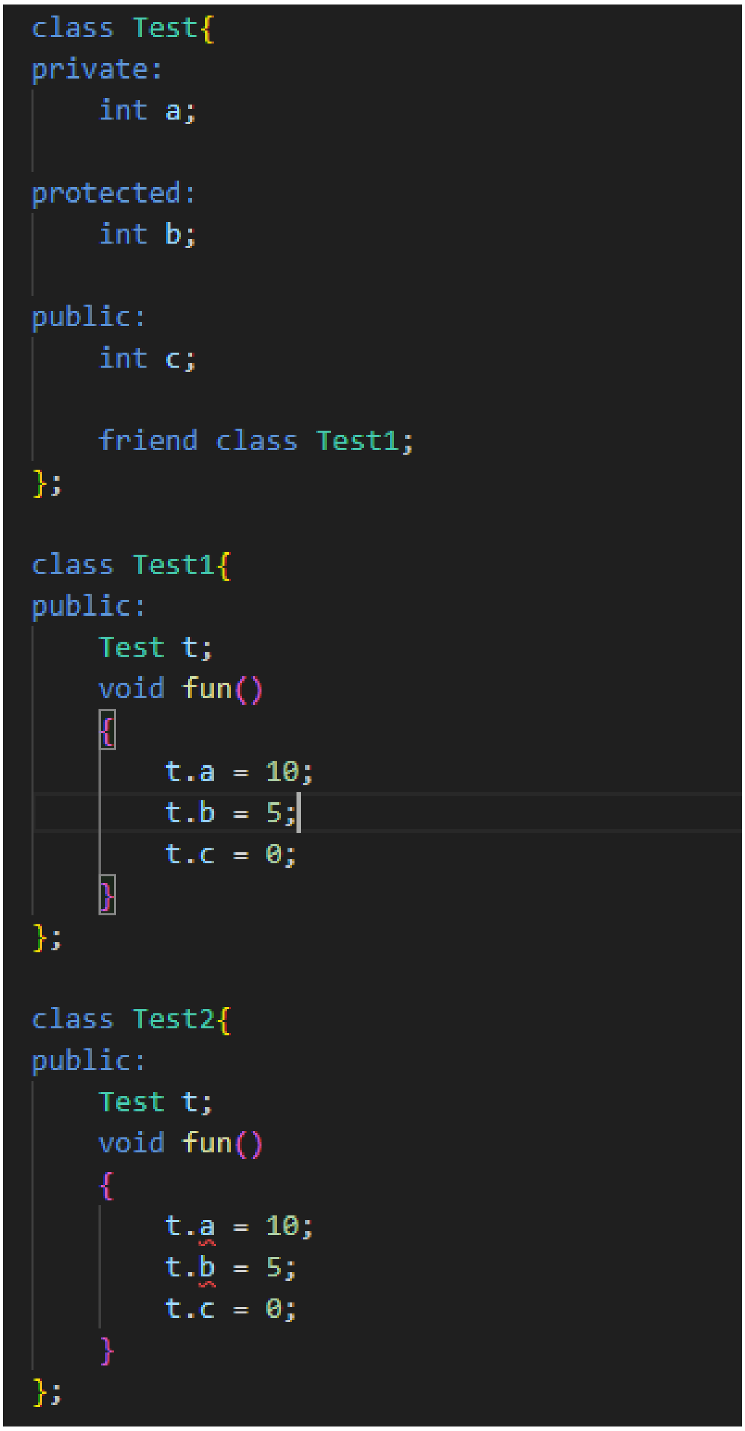
then

w/o Friend fun^c



```
class Test{
private:
    int a;

protected:
    int b;

public:
    int c;

    friend void fun();
};

void fun(){
    Test t;
    t.a = 10;
    t.b = 5;
    t.c = 0;
}

void fun2 (){
    Test t;
    t.a = 10;
    t.b = 5;
    t.c = 0;
}
```

Similarly Friend classes, another class cannot utilize members of another classes without inheriting. Hence, we require a **friend classes** to do the same.



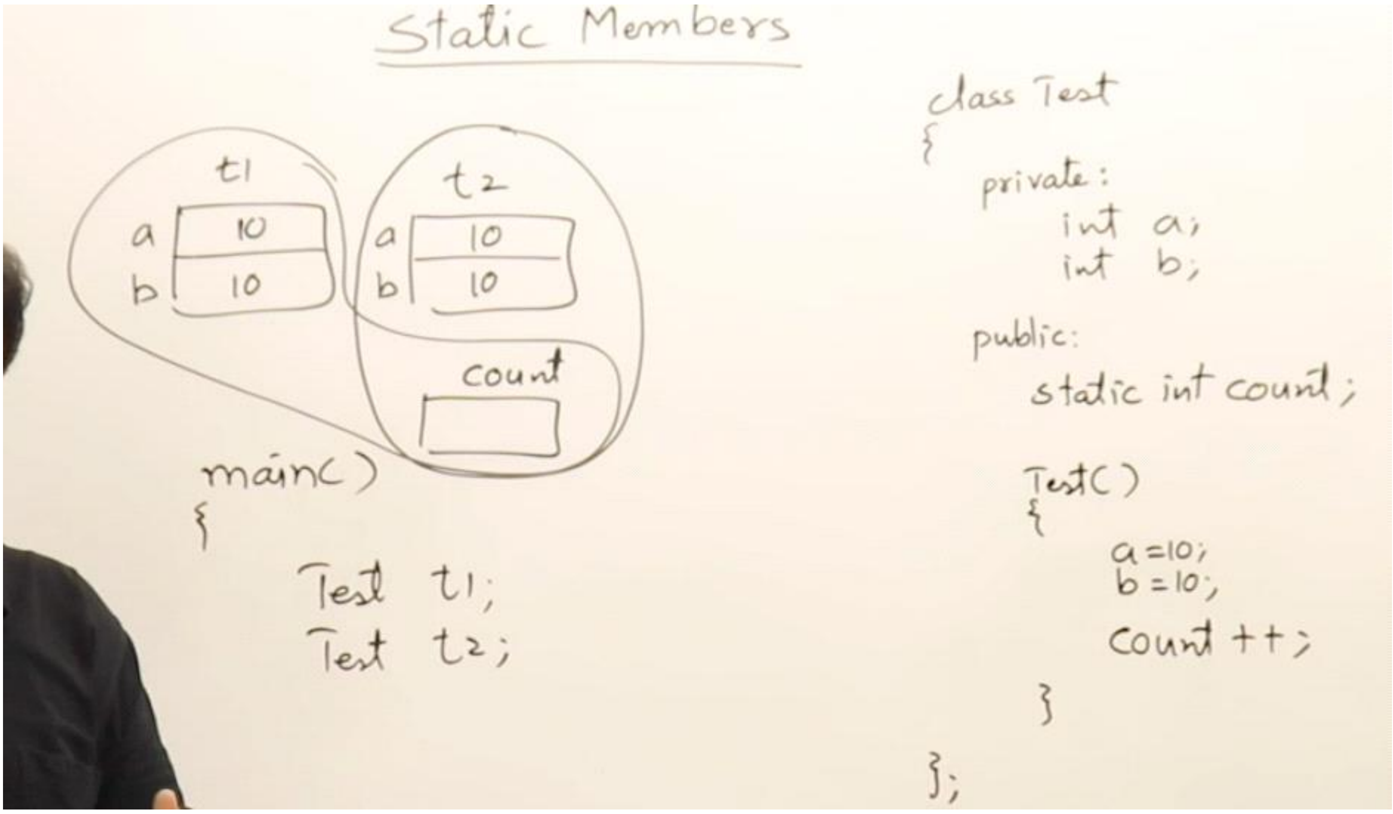
```cpp
class Test{
private:
    int a;

protected:
    int b;

public:
    int c;

    friend class Test1;
};

class Test1{
public:
    Test t;
    void fun()
    {
        t.a = 10;
        t.b = 5;
        t.c = 0;
    }
};

class Test2{
public:
    Test t;
    void fun()
    {
        t.a = 10;
        t.b = 5;
        t.c = 0;
    }
};
```

Static Member of a class

The static members belong to a class not to an object. Hence, every object share the same static memory location for static variable/ function/ member.

Static Members

- Static data members are members of a class
- Only one instance of static members is created and shared by all objects
- They can be accessed directly using class name
-
- Static members functions are functions of a class, they can be called using class name, without creating object of a class.
- They can access only static data members of a class, they cannot access non-static members of a class.

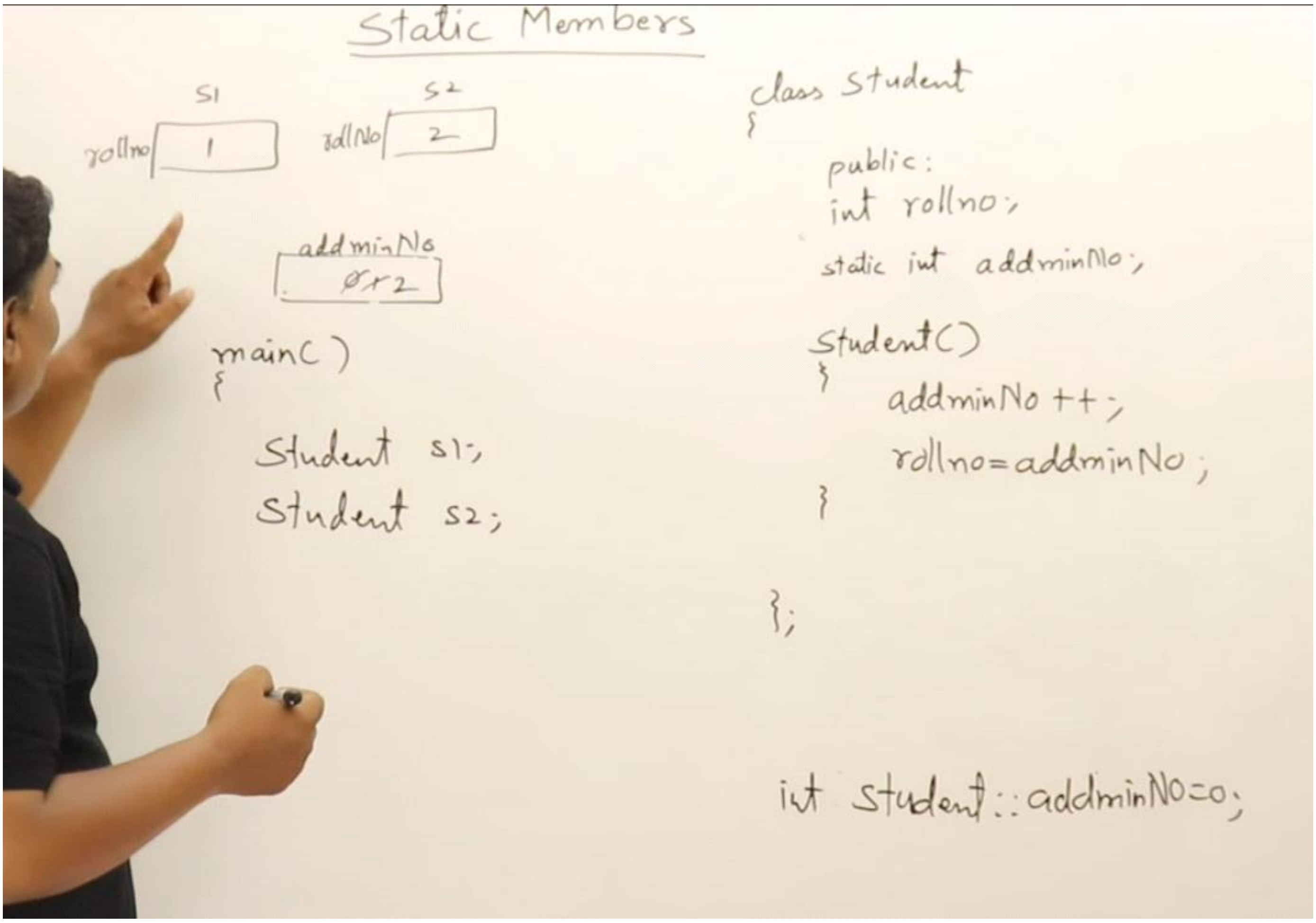


Static Members

```
class Test
{
  private:
     int a;
     int b;

  public:
    static int count;

    Test()
    {
       a=10;
       b=10;
       count++;
    }

};
```

```
main()
{
   Test t1;
   Test t2;
```

They are sharable members of a class.



Static Members

```
class Student
{
  public:
  int rollno;
  static int addminNo;

  Student()
  {
     addminNo++;
     rollno=addminNo;
  }

};

int student::addminNo=0;
```

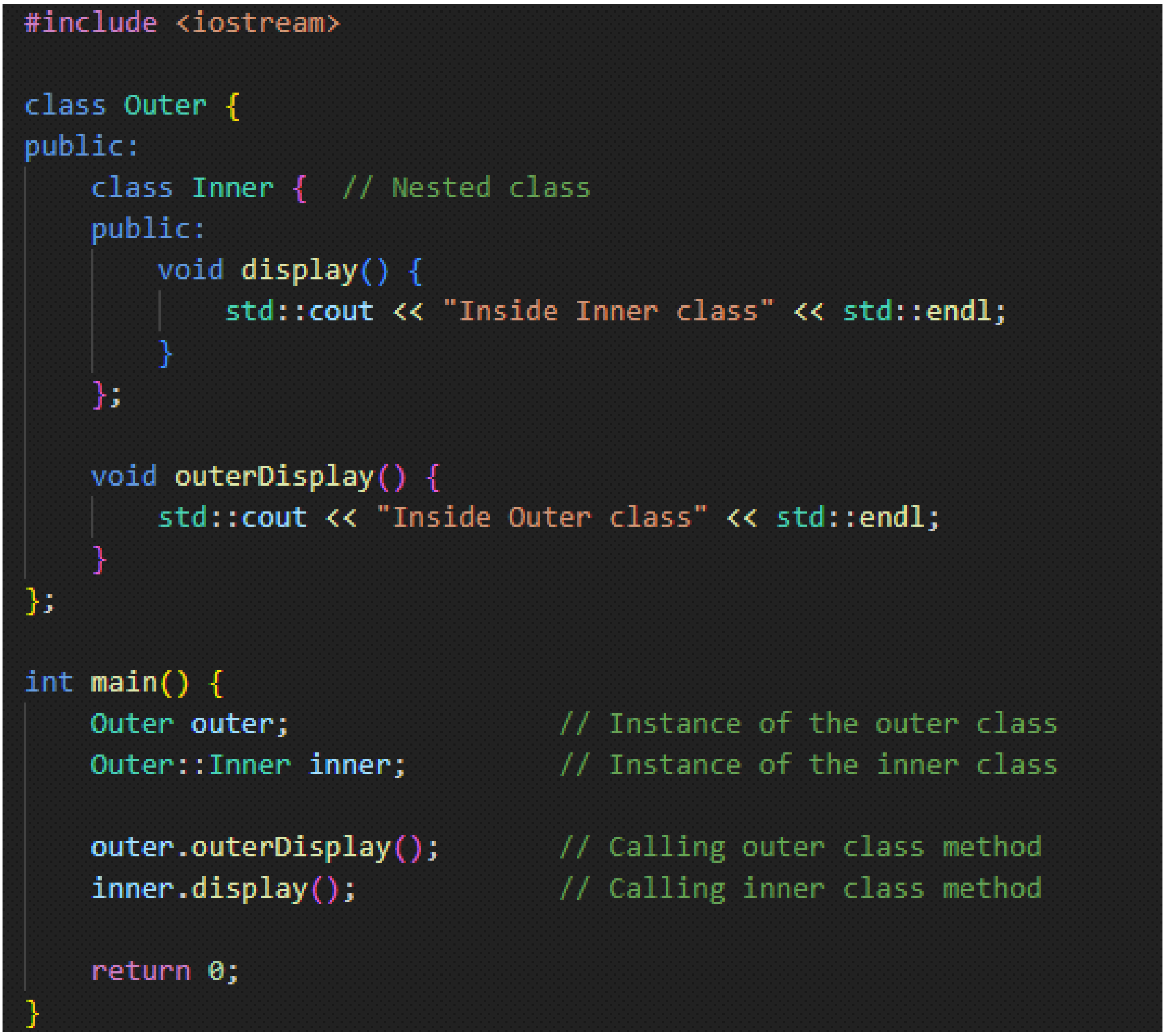
```
main()
{
  Student s1;
  Student s2;
```

Example :

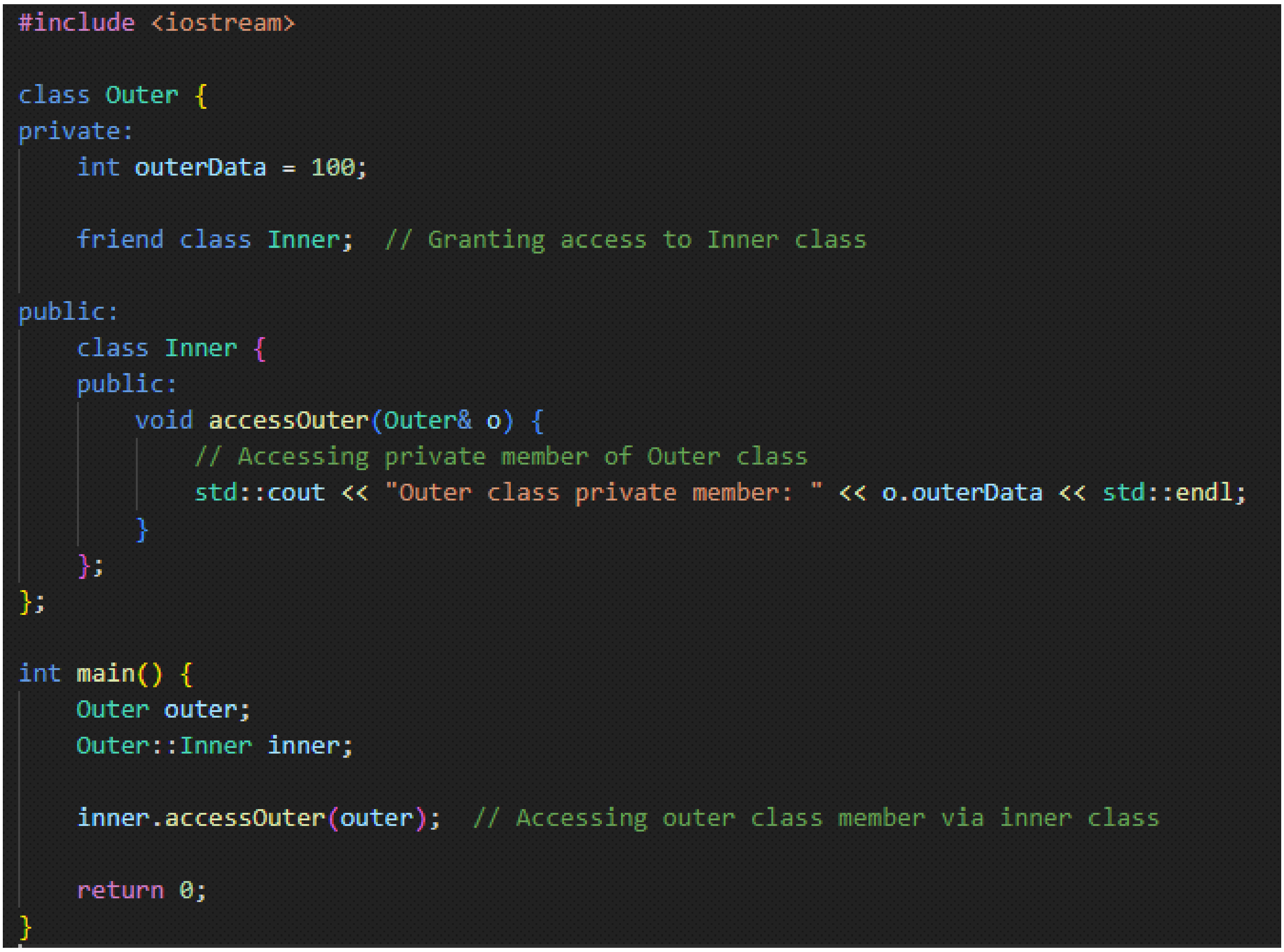
```cpp
#include <iostream>
using namespace std;
class Student{
public:
    int roll;
    string name;
    static int addNo;
    Student(string n)
    {
        addNo++;
        roll = addNo;
        name = n;
    }
    void display()
    {
        cout << "Name " << name << endl
             << "Roll " << roll << endl;
    }
};
int Student::addNo = 0;
int main(){
    Student s1("John");
    Student s2("Ravi");
    Student s3("Khan");
    Student s4("Khan");
    Student s5("Khan");
    Student s6("Khan");
    s1.display();
    s6.display();
    cout << "Number Admission " << Student::addNo << endl;
}
```

Inner Class

an **inner class** (or nested class) is a class declared inside another class.
Nested classes can also access the enclosing class’s members (private or public) if declared as friend. But without friend they cannot.



```cpp
#include <iostream>

class Outer {
public:
    class Inner {  // Nested class
    public:
        void display() {
            std::cout << "Inside Inner class" << std::endl;
        }
    };

    void outerDisplay() {
        std::cout << "Inside Outer class" << std::endl;
    }
};

int main() {
    Outer outer;                // Instance of the outer class
    Outer::Inner inner;         // Instance of the inner class

    outer.outerDisplay();       // Calling outer class method
    inner.display();            // Calling inner class method

    return 0;
}
```



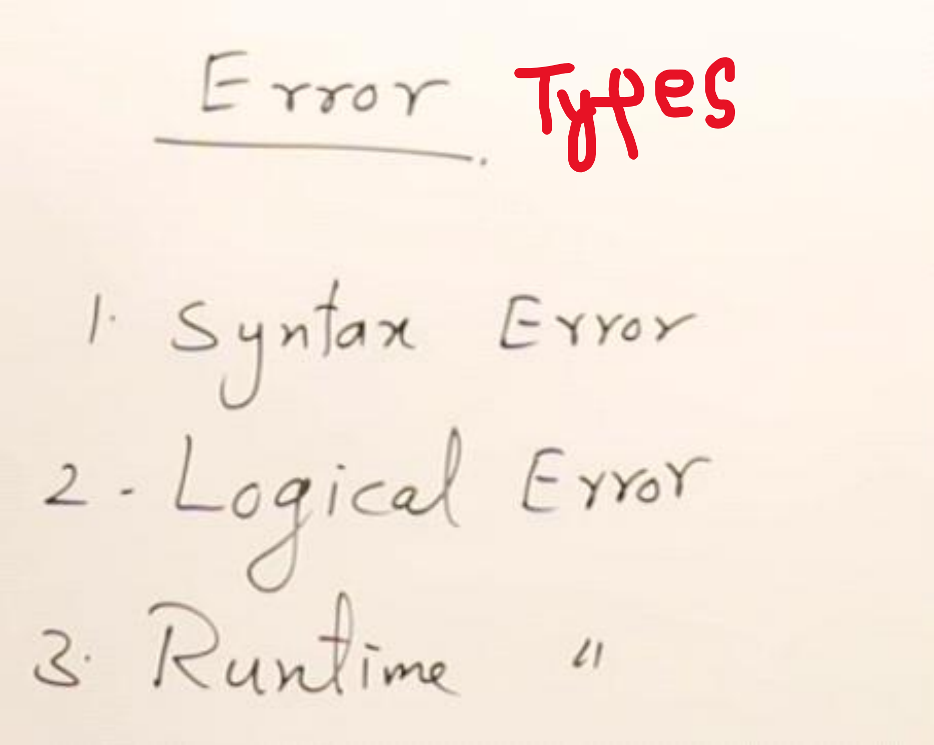
```cpp
#include <iostream>

class Outer {
private:
    int outerData = 100;

    friend class Inner;  // Granting access to Inner class

public:
    class Inner {
    public:
        void accessOuter(Outer& o) {
            // Accessing private member of Outer class
            std::cout << "Outer class private member: " << o.outerData << std::endl;
        }
    };
};

int main() {
    Outer outer;
    Outer::Inner inner;

    inner.accessOuter(outer);  // Accessing outer class member via inner class

    return 0;
}
```

Summary of Nested Classes:

1. **Inner (Nested) classes** are declared inside another class.
2. They are useful for logically grouping classes and encapsulating related logic.
3. Inner classes can be declared `public`, `private`, or `protected` based on access requirements.
4. Inner classes can access private members of the outer class only if explicitly allowed (e.g., via `friend` declaration).
5. An instance of an inner class is typically declared as `Outer::Inner inner;`.

Exception Handling

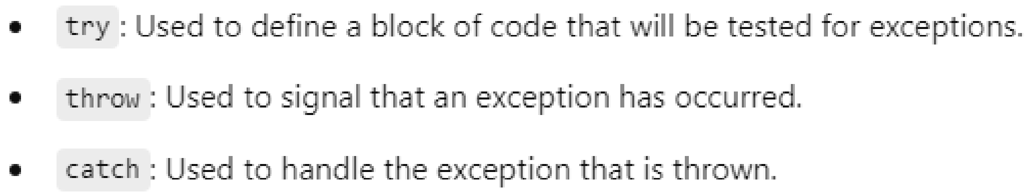
23 September 2024 19:33



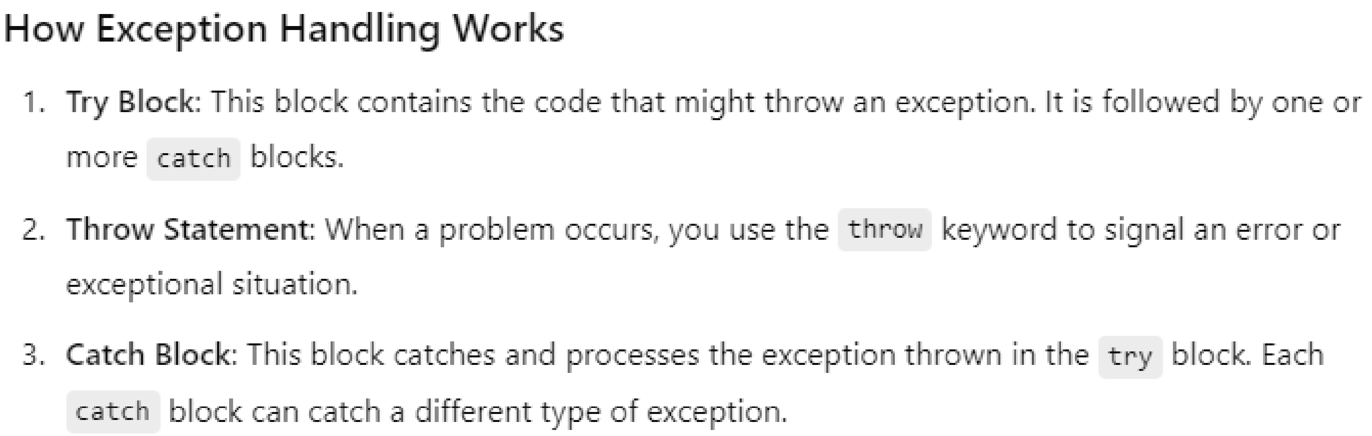
Error Types

1. Syntax Error
2. Logical Error
3. Runtime "

1. **Syntax Error :** it happens when a keyword or defined words are misspelled or mistyped. It is detected by Compiler
2. **Logical Error :** error in logic of program.
3. **Runtime Error :** error faced during the usage of program when user is using. It due to bad input by user like providing integer where string is needed.

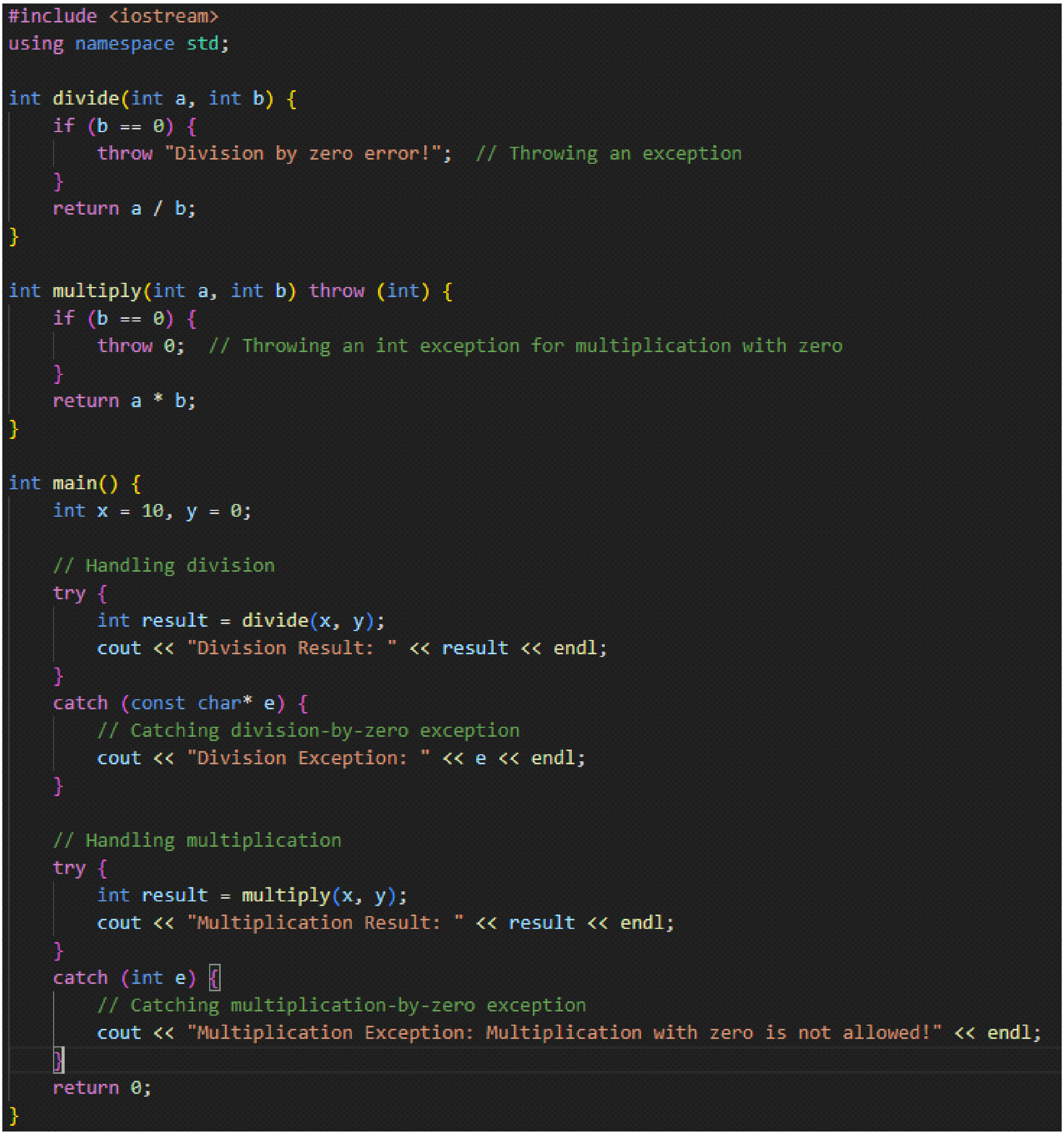


- `try` : Used to define a block of code that will be tested for exceptions.
- `throw` : Used to signal that an exception has occurred.
- `catch` : Used to handle the exception that is thrown.

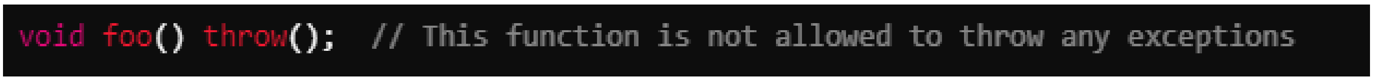


How Exception Handling Works

1. **Try Block**: This block contains the code that might throw an exception. It is followed by one or more `catch` blocks.
2. **Throw Statement**: When a problem occurs, you use the `throw` keyword to signal an error or exceptional situation.
3. **Catch Block**: This block catches and processes the exception thrown in the `try` block. Each `catch` block can catch a different type of exception.



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

int divide(int a, int b) {
    if (b == 0) {
        throw "Division by zero error!";  // Throwing an exception
    }
    return a / b;
}

int multiply(int a, int b) throw (int) {
    if (b == 0) {
        throw 0;  // Throwing an int exception for multiplication with zero
    }
    return a * b;
}

int main() {
    int x = 10, y = 0;

    // Handling division
    try {
        int result = divide(x, y);
        cout << "Division Result: " << result << endl;
    }
    catch (const char* e) {
        // Catching division-by-zero exception
        cout << "Division Exception: " << e << endl;
    }

    // Handling multiplication
    try {
        int result = multiply(x, y);
        cout << "Multiplication Result: " << result << endl;
    }
    catch (int e) {
        // Catching multiplication-by-zero exception
        cout << "Multiplication Exception: Multiplication with zero is not allowed!" << endl;
    }
    return 0;
}
```



```
void foo() throw();  // This function is not allowed to throw any exceptions
```

Modern Replacement: `noexcept`

Instead of using `throw()`, modern C++ uses the `noexcept` specifier to indicate that a function does not throw exceptions. It has two forms:

- `noexcept`: The function is guaranteed not to throw any exceptions.
- `noexcept(expression)`: The function is conditionally noexcept, based on the truth value of the expression.

Example with `noexcept`:

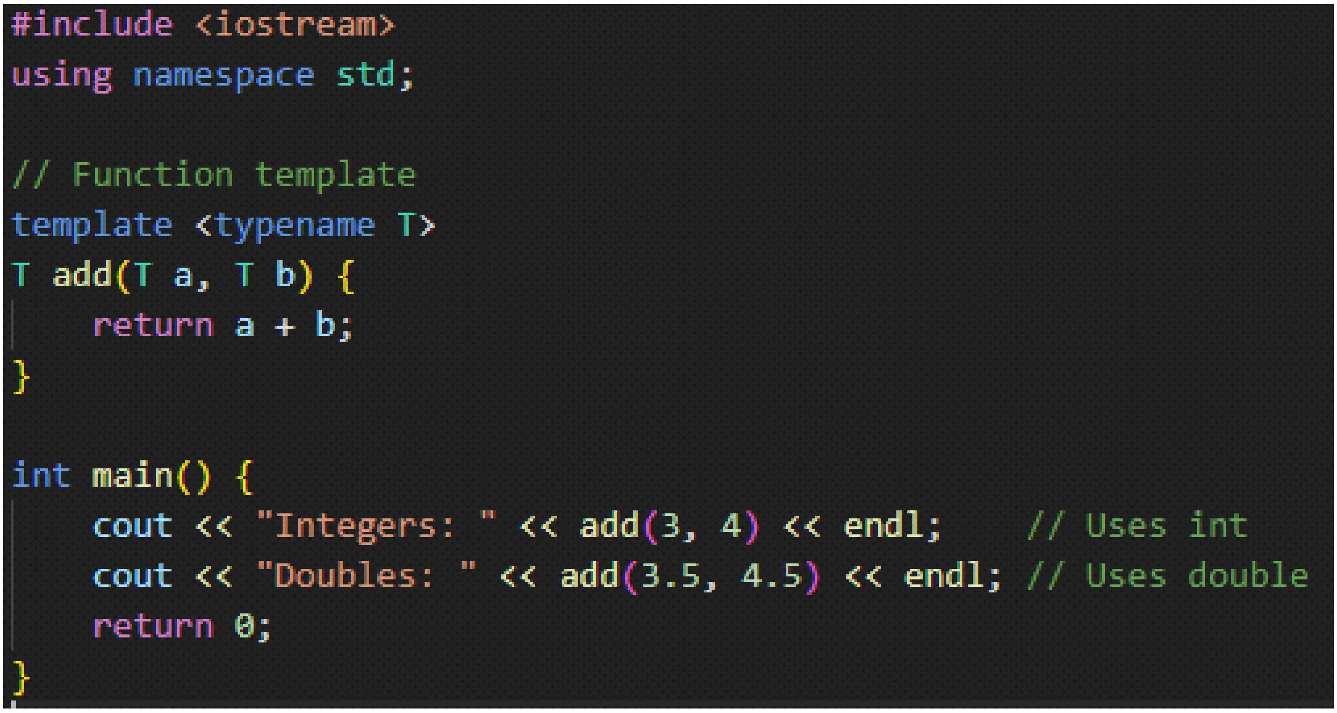
```cpp
void foo() noexcept {  // This function will not throw any exceptions
    // Function body
}

void bar() noexcept(false) {  // This function can throw exceptions
    throw "Some error";  // This is allowed because noexcept(false) allows exceptions
}
```

Templates

23 September 2024 22:03

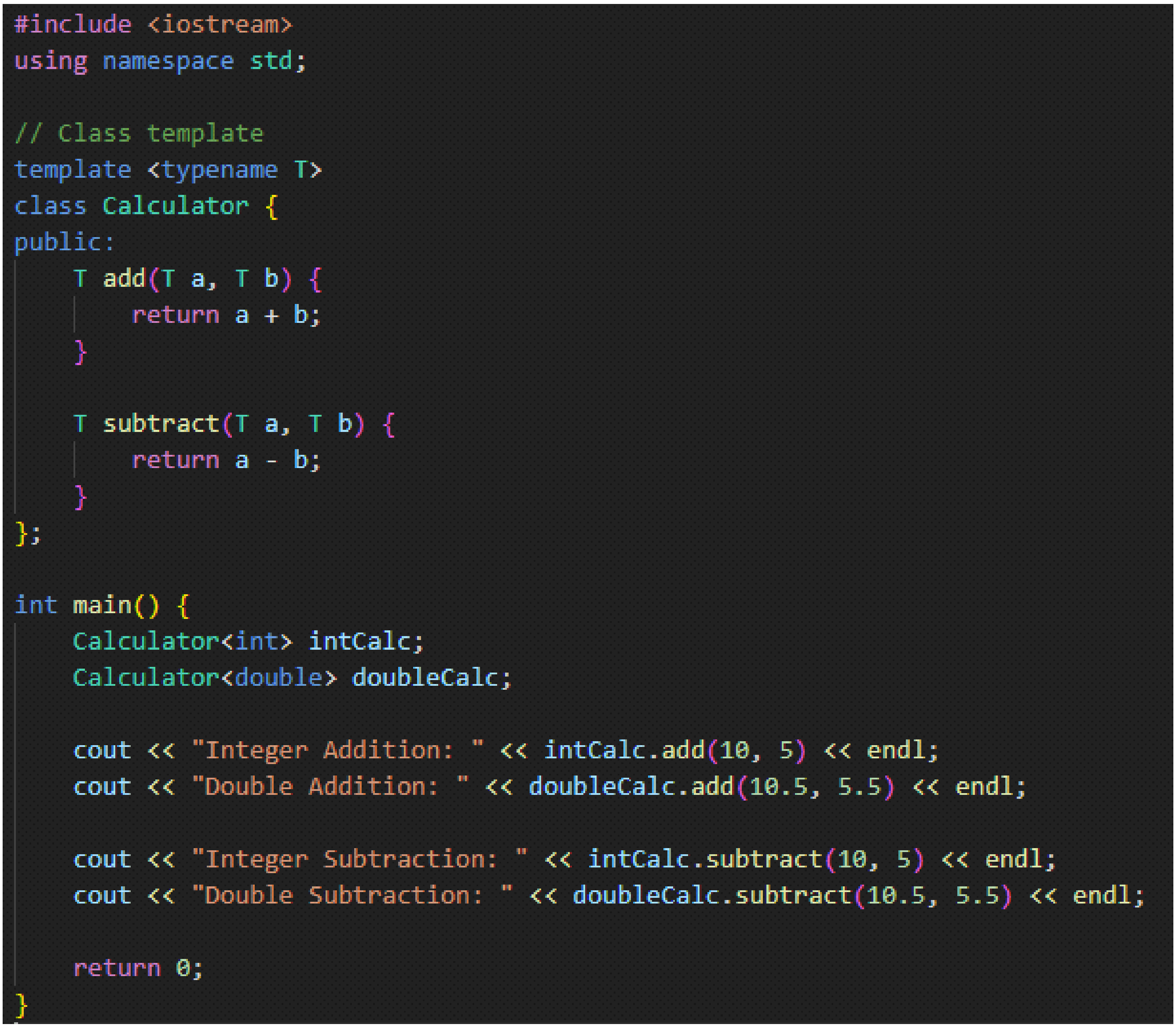
They are used for generic programming to create generic function or classes.

Function Templates



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

// Function template
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << "Integers: " << add(3, 4) << endl;     // Uses int
    cout << "Doubles: " << add(3.5, 4.5) << endl; // Uses double
    return 0;
}
```

Class Templates



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

// Class template
template <typename T>
class Calculator {
public:
    T add(T a, T b) {
        return a + b;
    }

    T subtract(T a, T b) {
        return a - b;
    }
};

int main() {
    Calculator<int> intCalc;
    Calculator<double> doubleCalc;

    cout << "Integer Addition: " << intCalc.add(10, 5) << endl;
    cout << "Double Addition: " << doubleCalc.add(10.5, 5.5) << endl;

    cout << "Integer Subtraction: " << intCalc.subtract(10, 5) << endl;
    cout << "Double Subtraction: " << doubleCalc.subtract(10.5, 5.5) << endl;

    return 0;
}
```

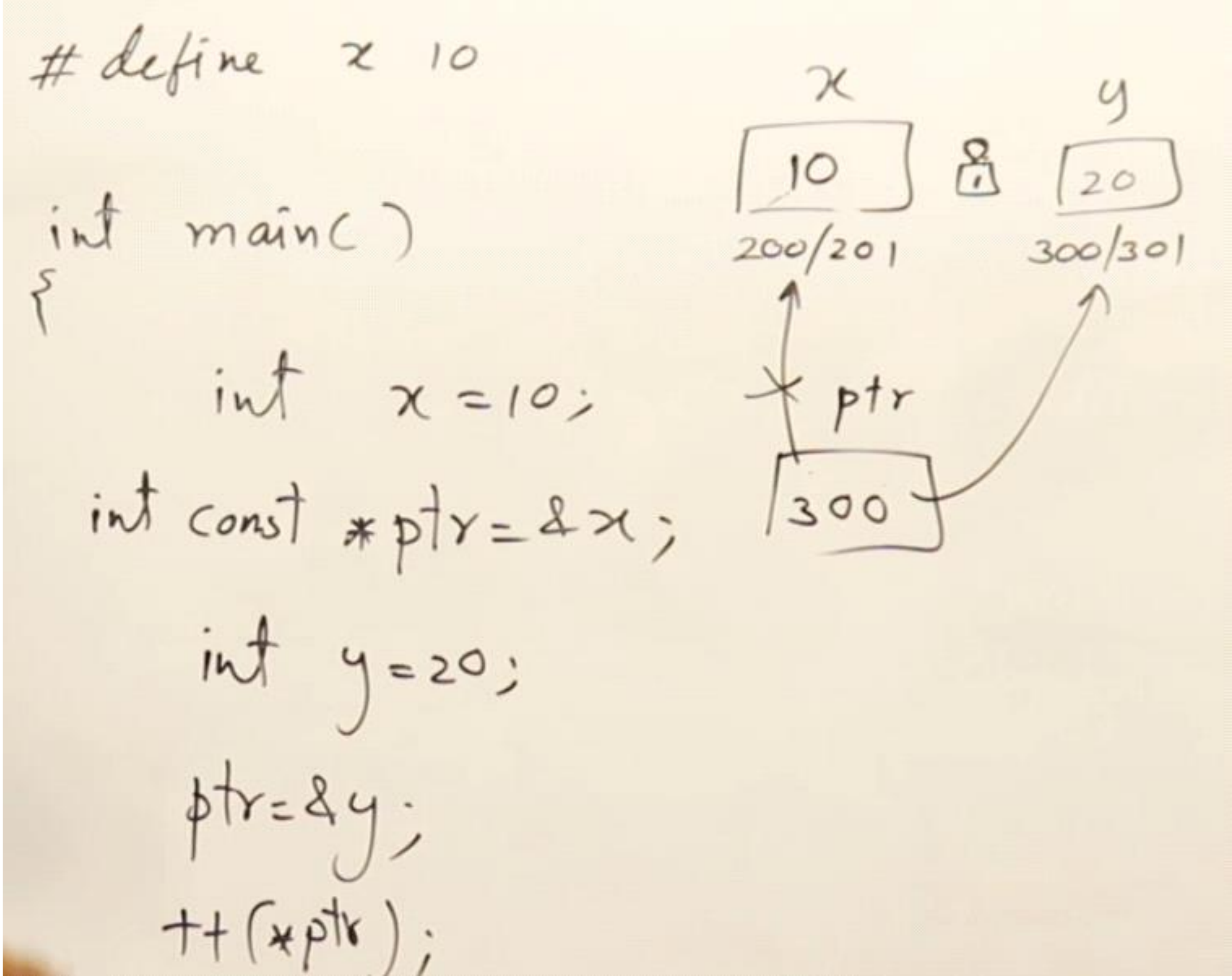
Constants, Preprocessor directives and Namespaces

24 September 2024 19:14

Constants Qualifier

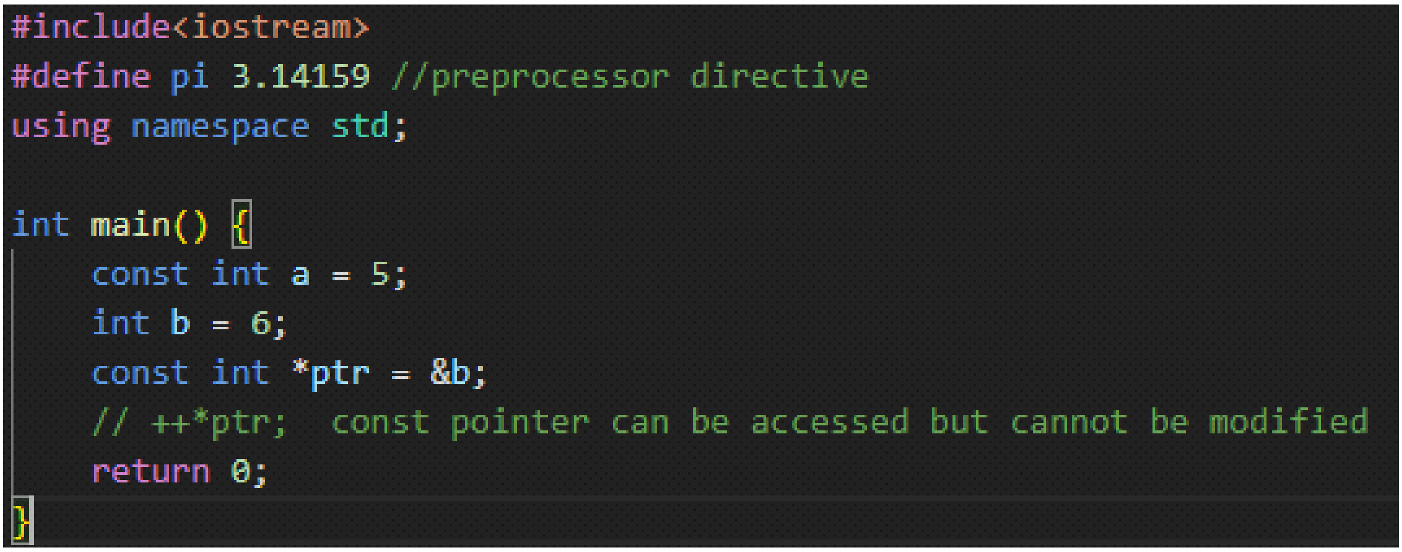
It refer to the use of the const keyword to mark variables, pointers, references, or member functions as immutable.

Const pointer can be read but cannot be modified. The const ptr can point at any data but cannot modify that data.

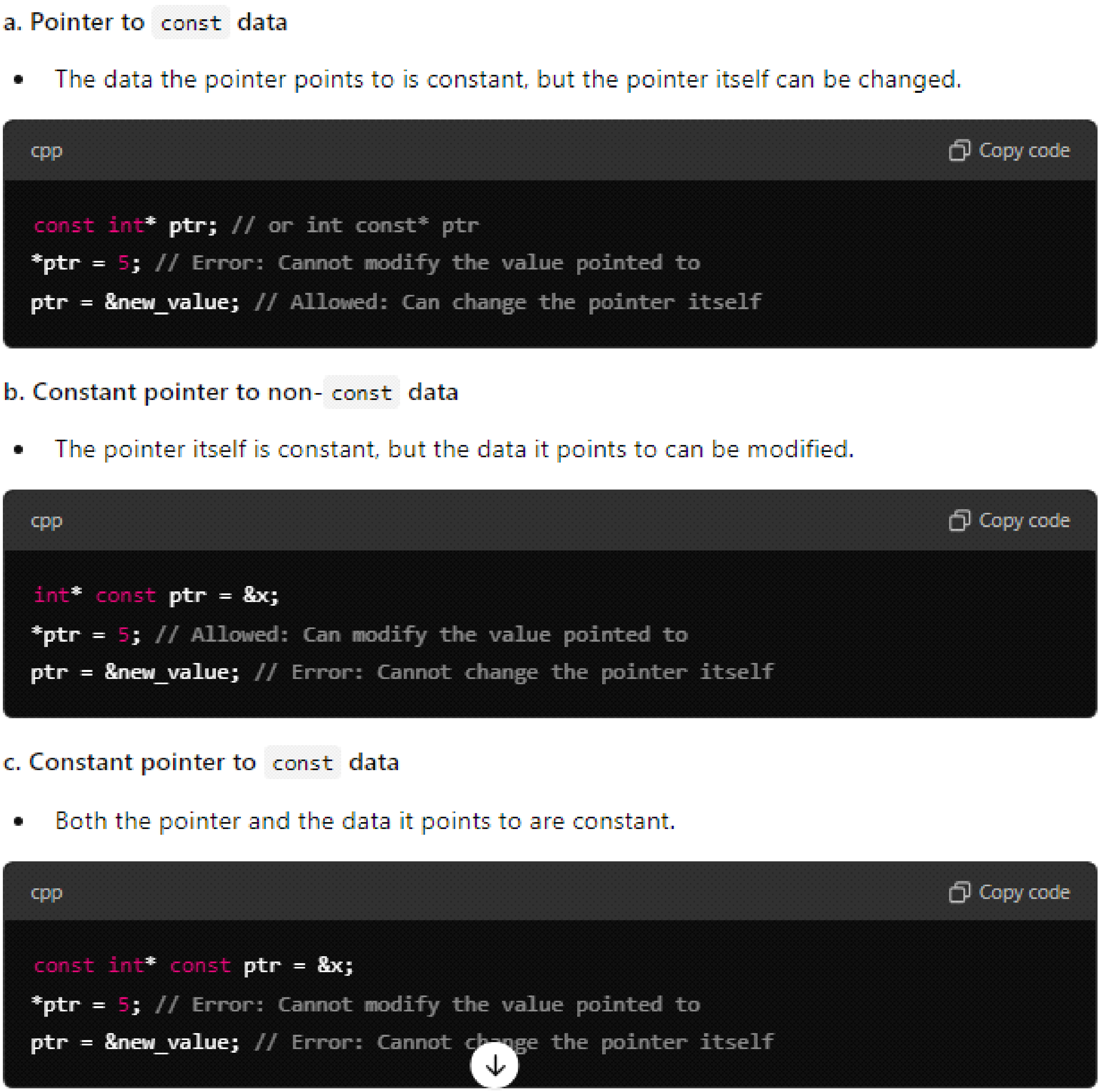


```
#define x 10

int main()
{
      int x=10;
  int const *ptr=&x;
      int y=20;
     ptr=&y;
    ++(*ptr);
```

x
10
200/201
y
20
300/301
ptr
300



```cpp
#include<iostream>
#define pi 3.14159 //preprocessor directive
using namespace std;

int main() {
    const int a = 5;
    int b = 6;
    const int *ptr = &b;
    // ++*ptr;  const pointer can be accessed but cannot be modified
    return 0;
}
```



a. Pointer to `const` data

- The data the pointer points to is constant, but the pointer itself can be changed.

```cpp
const int* ptr; // or int const* ptr
*ptr = 5; // Error: Cannot modify the value pointed to
ptr = &new_value; // Allowed: Can change the pointer itself
```

b. Constant pointer to non-`const` data

- The pointer itself is constant, but the data it points to can be modified.

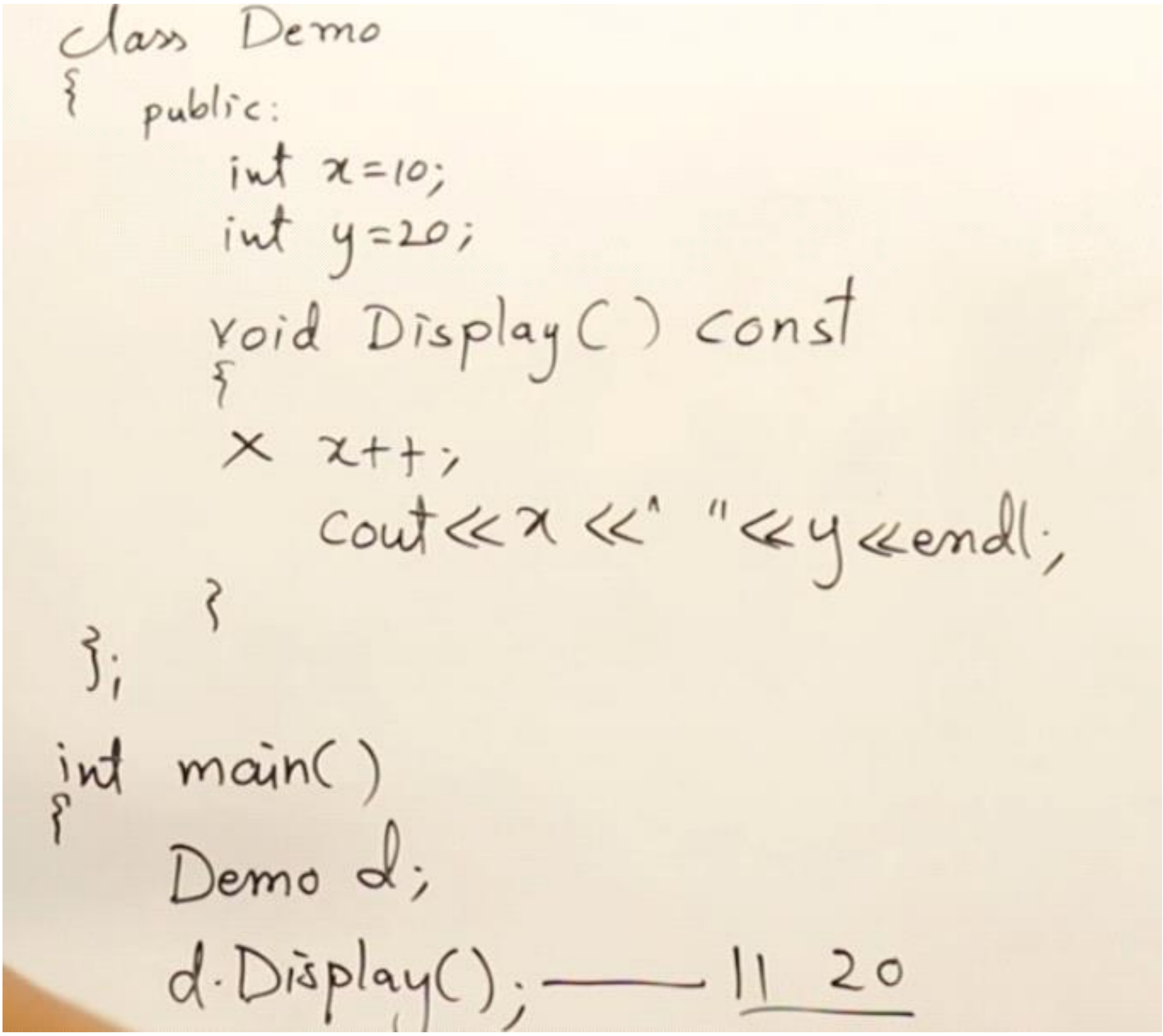
```cpp
int* const ptr = &x;
*ptr = 5; // Allowed: Can modify the value pointed to
ptr = &new_value; // Error: Cannot change the pointer itself
```

c. Constant pointer to `const` data

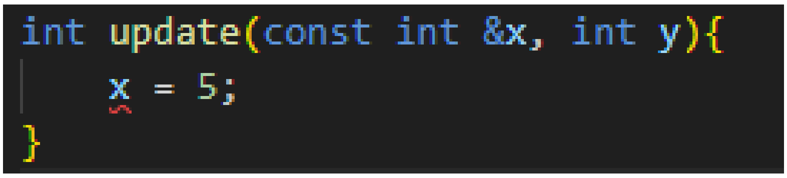
- Both the pointer and the data it points to are constant.

```cpp
const int* const ptr = &x;
*ptr = 5; // Error: Cannot modify the value pointed to
ptr = &new_value; // Error: Cannot change the pointer itself
```

If we know that we are not going to update any data we can use const in a function. And if we by mistake update any data the compiler will give us an error.



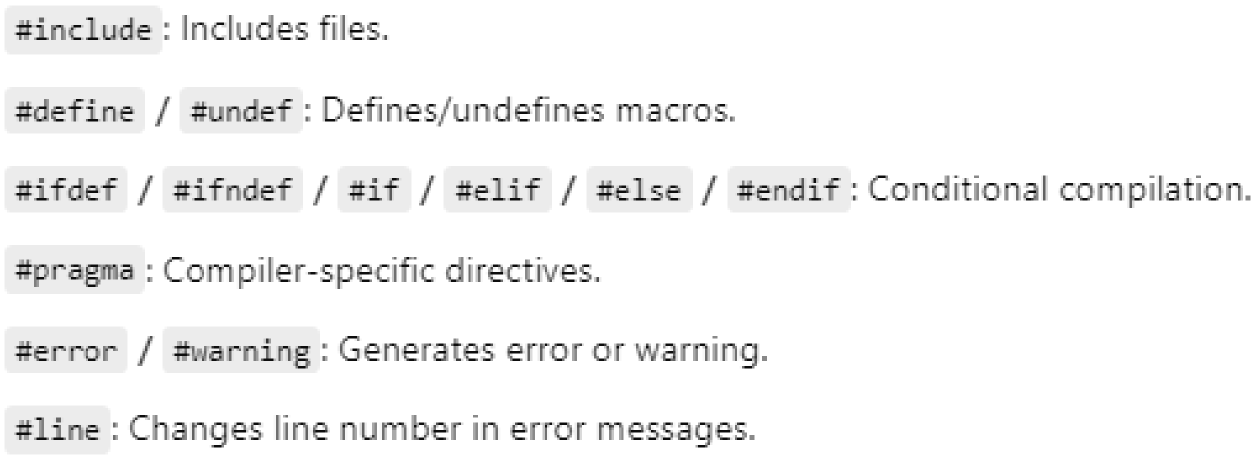
```
class Demo
{  public:
       int x=10;
       int y=20;
      void Display() const
      {
       ×  x++;
          cout<<x<<" "<<y<<endl;
      }
 };
int main()
{
    Demo d;
    d.Display(); ——— 11 20
```



```
int update(const int &x, int y){
    x = 5;
}
```

Preprocessor directives

They are the commands that we give to our compiler.



`#include` : Includes files.

`#define` / `#undef` : Defines/undefines macros.

`#ifdef` / `#ifndef` / `#if` / `#elif` / `#else` / `#endif` : Conditional compilation.

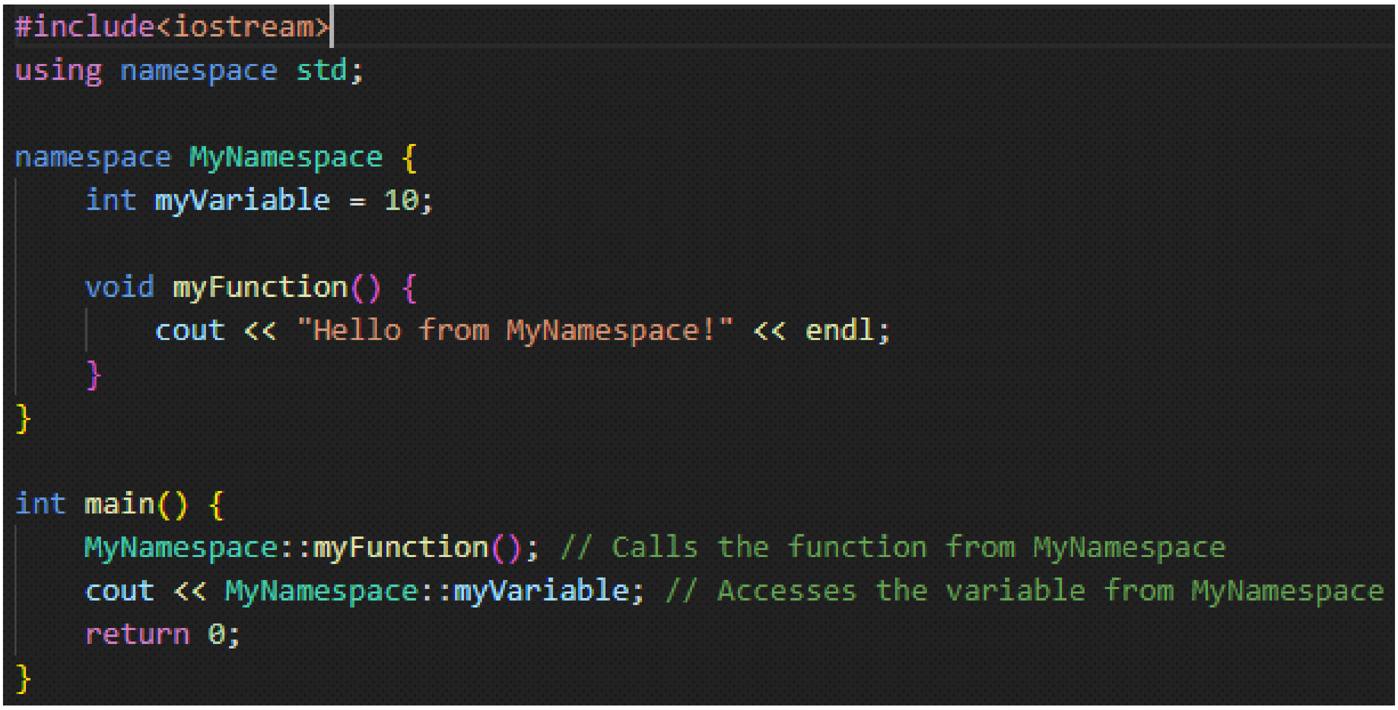
`#pragma` : Compiler-specific directives.

`#error` / `#warning` : Generates error or warning.

`#line` : Changes line number in error messages.

Namespace

namespace is a feature that allows us to organize code into logical groups and avoid name collisions, especially when your code base includes multiple libraries. Namespaces provide a way to group related classes, functions, variables, and other identifiers.

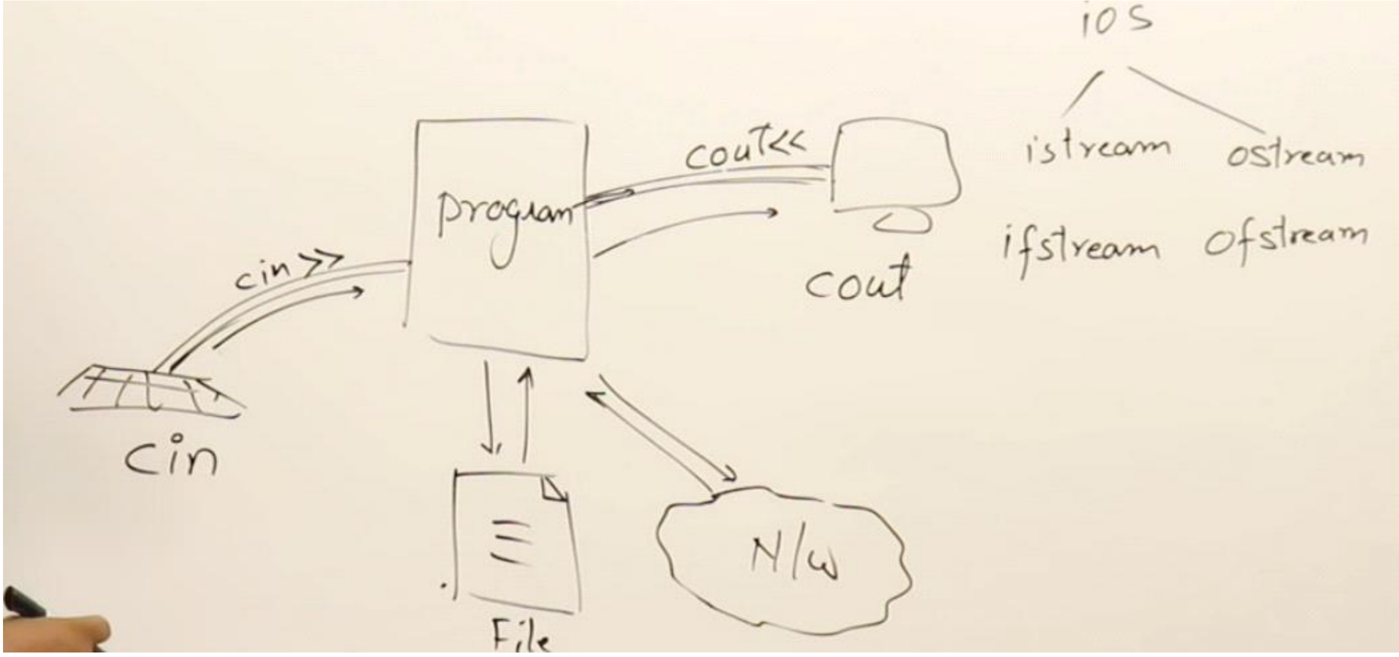


```cpp
#include<iostream>
using namespace std;

namespace MyNamespace {
    int myVariable = 10;

    void myFunction() {
        cout << "Hello from MyNamespace!" << endl;
    }
}

int main() {
    MyNamespace::myFunction(); // Calls the function from MyNamespace
    cout << MyNamespace::myVariable; // Accesses the variable from MyNamespace
    return 0;
}
```

IO Streams

24 September 2024 19:55

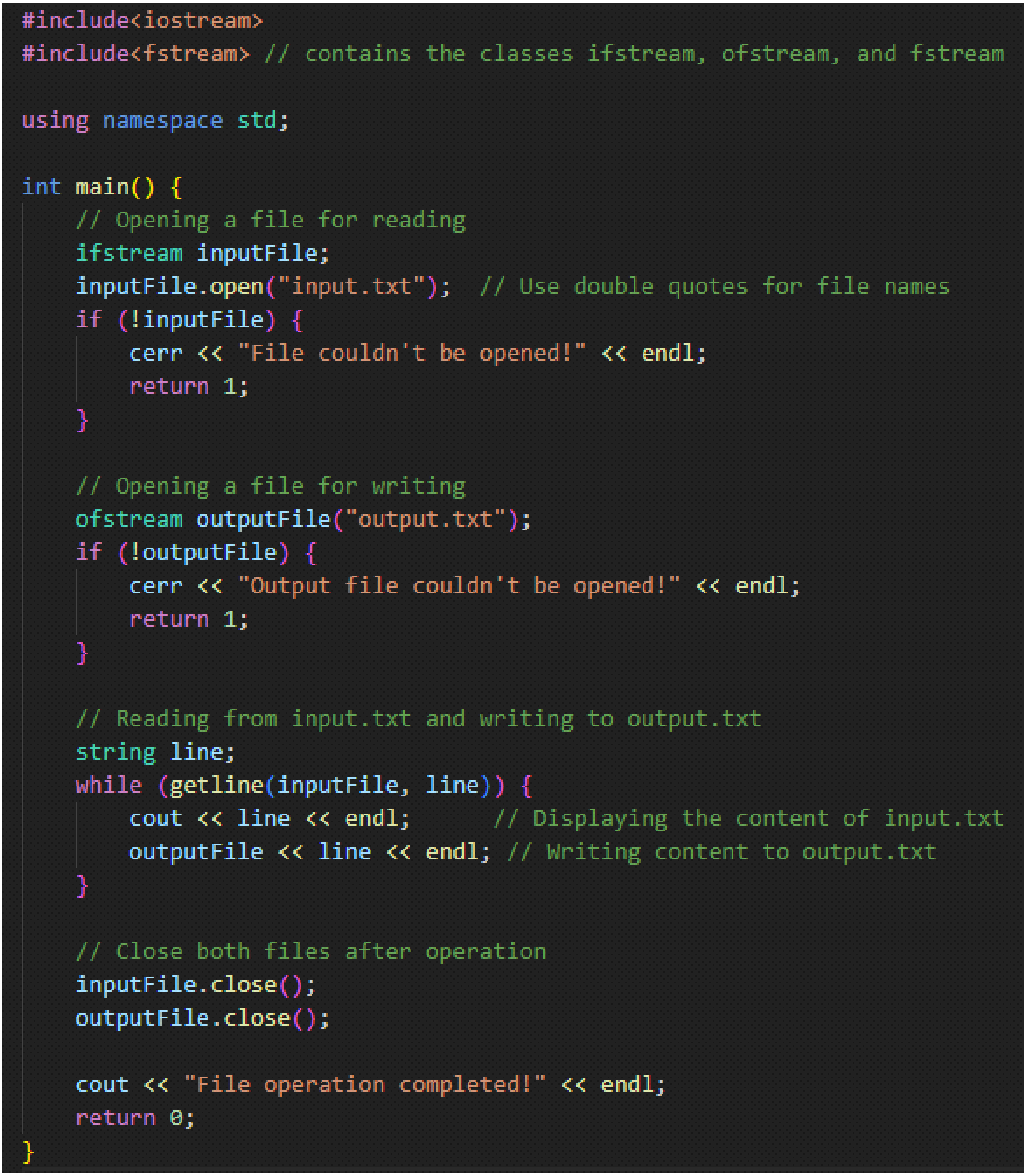
Streams are flow of character or should we say flow of data.



File Handling

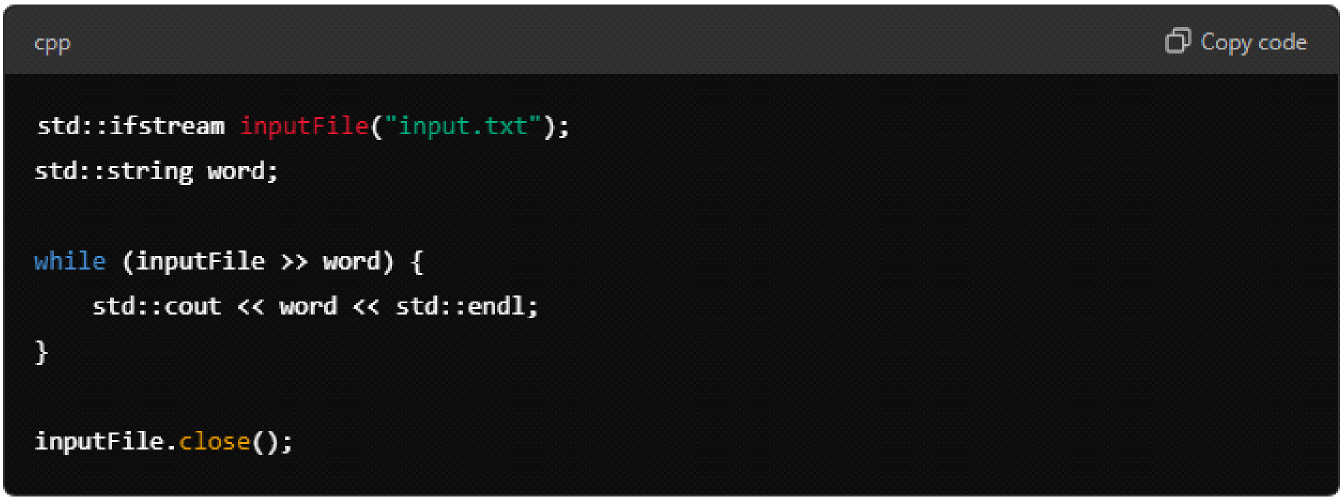
It involves reading to writing to files using streams. The standard Library provides :

- Ifstream : for input file handling
- Ofstream : for output file handling
- Fstream : for both input and output file handling



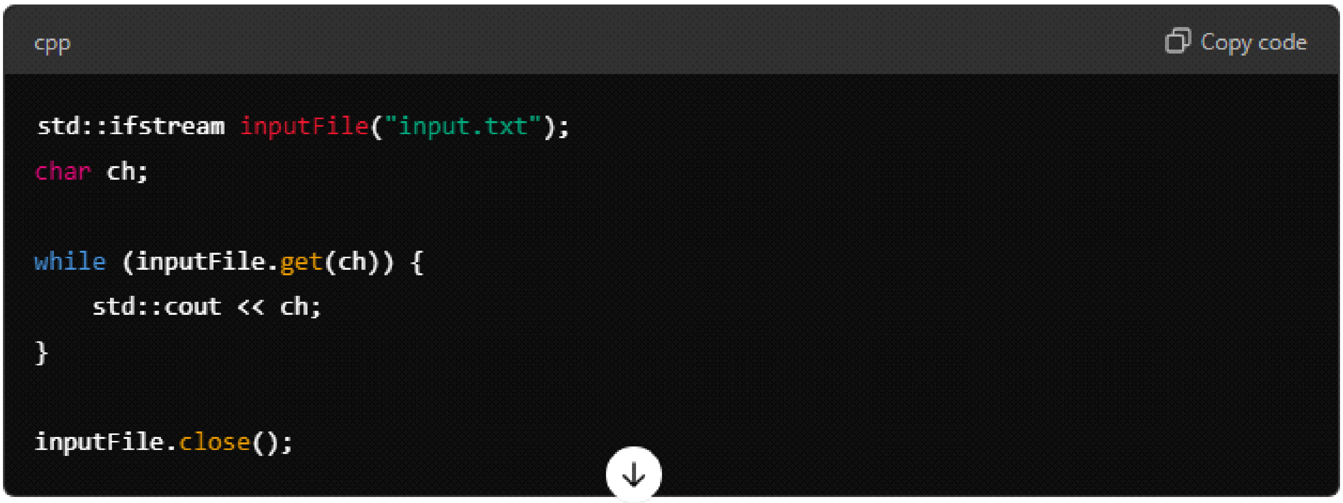
```cpp
#include<iostream>
#include<fstream> // contains the classes ifstream, ofstream, and fstream

using namespace std;

int main() {
    // Opening a file for reading
    ifstream inputFile;
    inputFile.open("input.txt");  // Use double quotes for file names
    if (!inputFile) {
        cerr << "File couldn't be opened!" << endl;
        return 1;
    }

    // Opening a file for writing
    ofstream outputFile("output.txt");
    if (!outputFile) {
        cerr << "Output file couldn't be opened!" << endl;
        return 1;
    }

    // Reading from input.txt and writing to output.txt
    string line;
    while (getline(inputFile, line)) {
        cout << line << endl;      // Displaying the content of input.txt
        outputFile << line << endl; // Writing content to output.txt
    }

    // Close both files after operation
    inputFile.close();
    outputFile.close();

    cout << "File operation completed!" << endl;
    return 0;
}
```

Reading Word by Word:

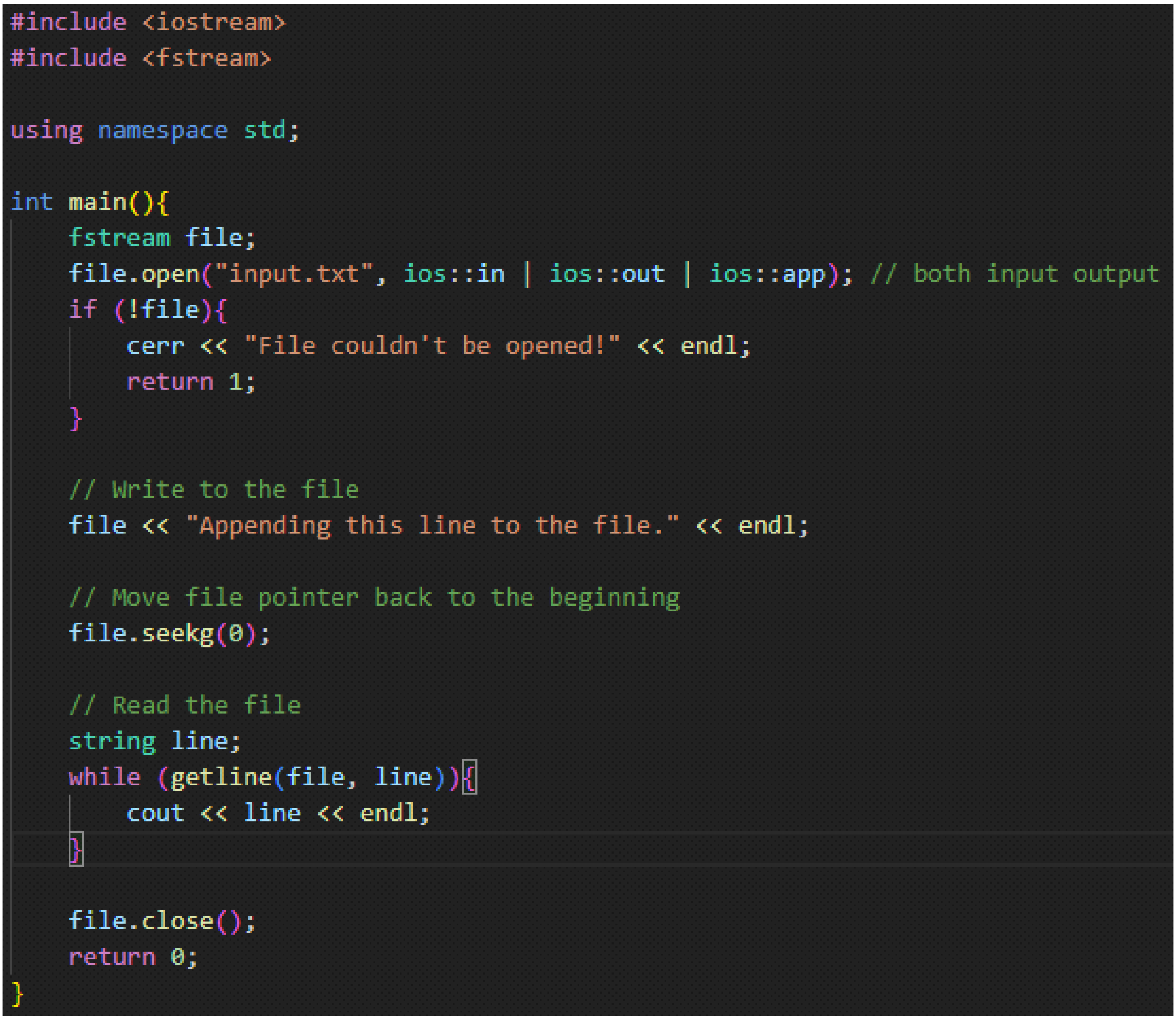


```cpp
std::ifstream inputFile("input.txt");
std::string word;

while (inputFile >> word) {
    std::cout << word << std::endl;
}

inputFile.close();
```

Reading Character by Character:



```cpp
std::ifstream inputFile("input.txt");
char ch;

while (inputFile.get(ch)) {
    std::cout << ch;
}

inputFile.close();
```



```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
    fstream file;
    file.open("input.txt", ios::in | ios::out | ios::app); // both input output
    if (!file){
        cerr << "File couldn't be opened!" << endl;
        return 1;
    }

    // Write to the file
    file << "Appending this line to the file." << endl;

    // Move file pointer back to the beginning
    file.seekg(0);

    // Read the file
    string line;
    while (getline(file, line)){
        cout << line << endl;
    }

    file.close();
    return 0;
}
```

STL

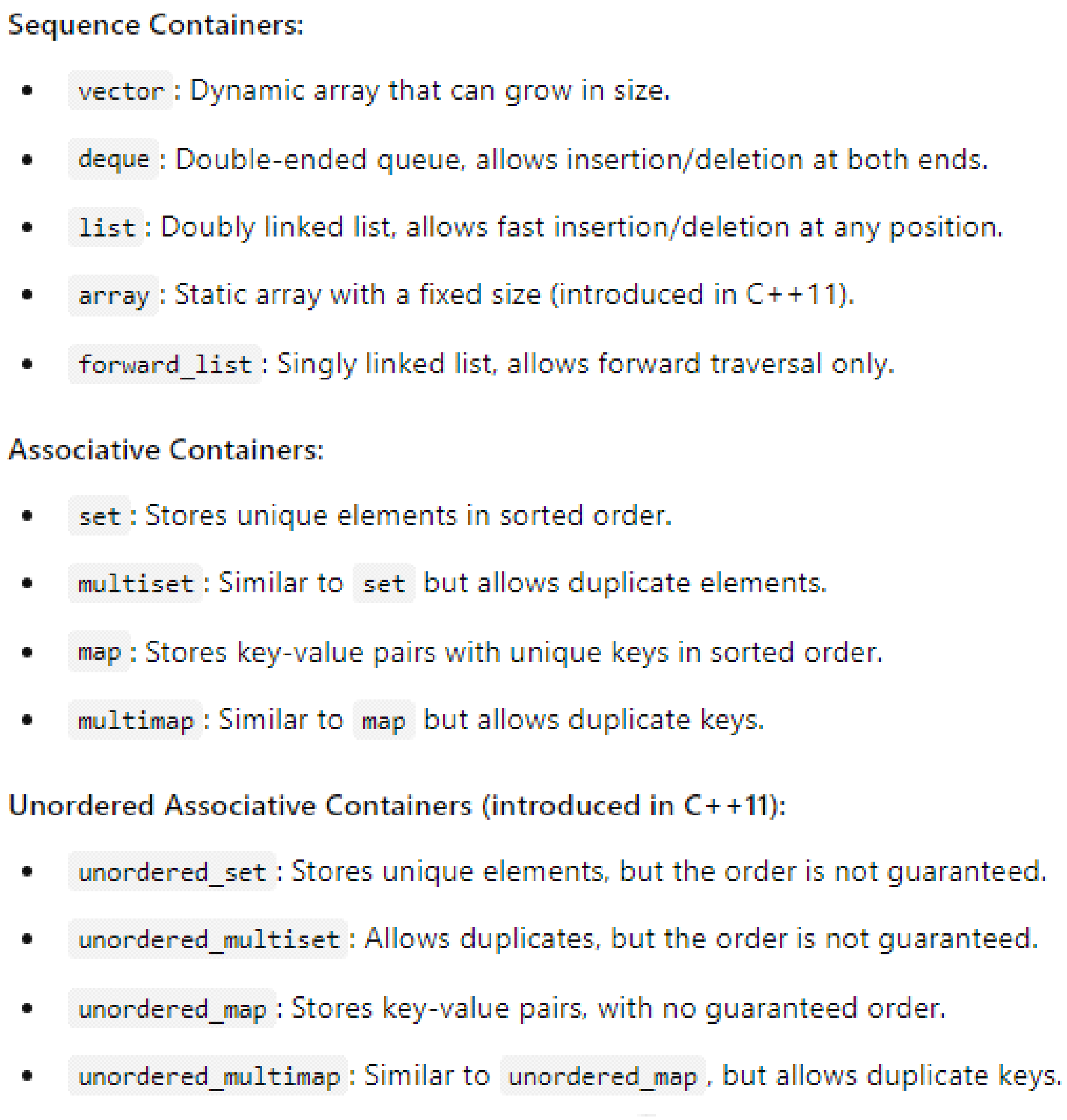
25 September 2024 19:59

STL stands for Standard Template Library. It is a powerful set of C++ template classes that provide general-purpose, reusable implementations of common data structures and algorithms.

Main Components of STL are:

Containers

These are data structures that store objects and data.



Sequence Containers:

- `vector` : Dynamic array that can grow in size.
- `deque` : Double-ended queue, allows insertion/deletion at both ends.
- `list` : Doubly linked list, allows fast insertion/deletion at any position.
- `array` : Static array with a fixed size (introduced in C++11).
- `forward_list` : Singly linked list, allows forward traversal only.

Associative Containers:

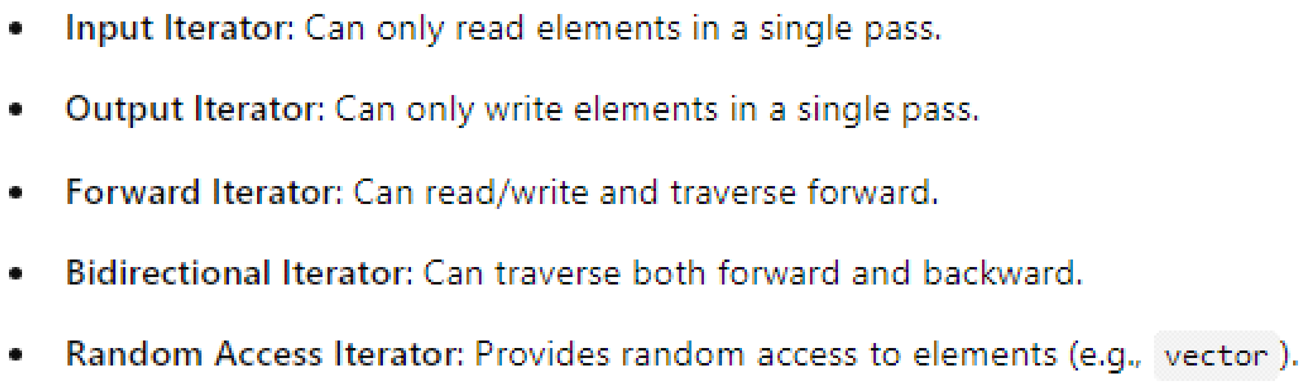
- `set` : Stores unique elements in sorted order.
- `multiset` : Similar to `set` but allows duplicate elements.
- `map` : Stores key-value pairs with unique keys in sorted order.
- `multimap` : Similar to `map` but allows duplicate keys.

Unordered Associative Containers (introduced in C++11):

- `unordered_set` : Stores unique elements, but the order is not guaranteed.
- `unordered_multiset` : Allows duplicates, but the order is not guaranteed.
- `unordered_map` : Stores key-value pairs, with no guaranteed order.
- `unordered_multimap` : Similar to `unordered_map` , but allows duplicate keys.

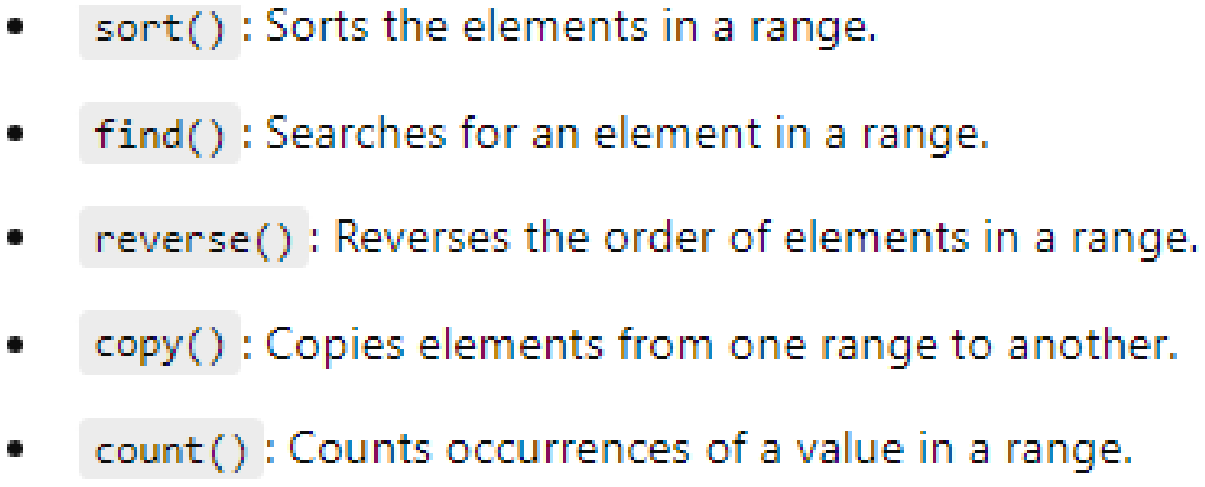
Iterators

Iterators are used to point to the elements of a container and traverse through them. They work like pointers.



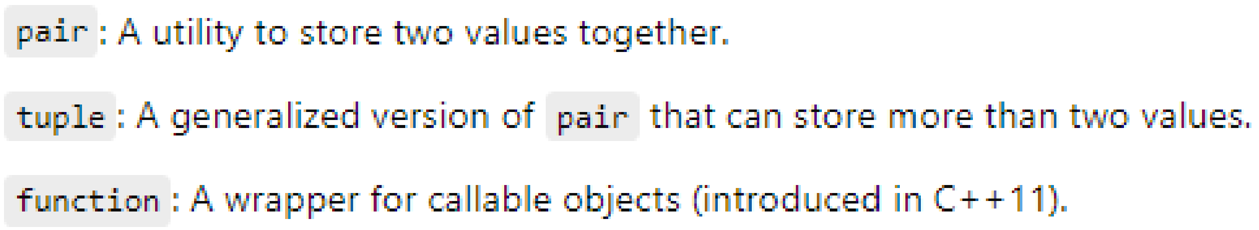
- **Input Iterator**: Can only read elements in a single pass.
- **Output Iterator**: Can only write elements in a single pass.
- **Forward Iterator**: Can read/write and traverse forward.
- **Bidirectional Iterator**: Can traverse both forward and backward.
- **Random Access Iterator**: Provides random access to elements (e.g., `vector`).

Algorithms



- `sort()` : Sorts the elements in a range.
- `find()` : Searches for an element in a range.
- `reverse()` : Reverses the order of elements in a range.
- `copy()` : Copies elements from one range to another.
- `count()` : Counts occurrences of a value in a range.

Utility Components



`pair` : A utility to store two values together.

`tuple` : A generalized version of `pair` that can store more than two values.

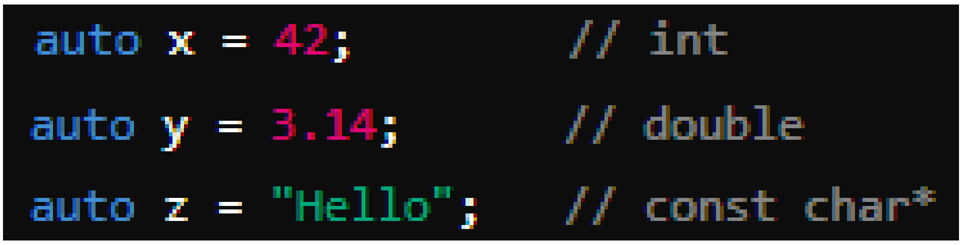
`function` : A wrapper for callable objects (introduced in C++11).

C++ 11

25 September 2024 20:29

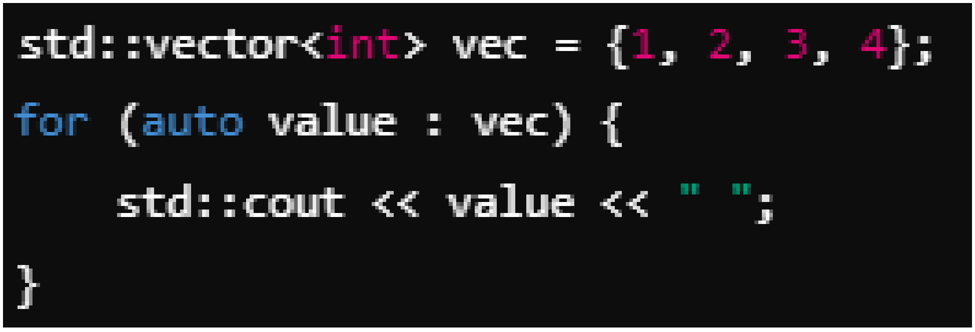
C++11 is a major update to the C++ language, introducing a wide range of features to improve the language's performance, usability, and safety.

Auto Keyword
It allows the compiler to automatically deduce the type of a variable based on its initializer.



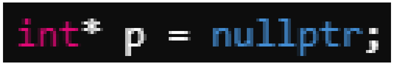
```
auto x = 42;          // int
auto y = 3.14;        // double
auto z = "Hello";     // const char*
```

Range-Based For Loop



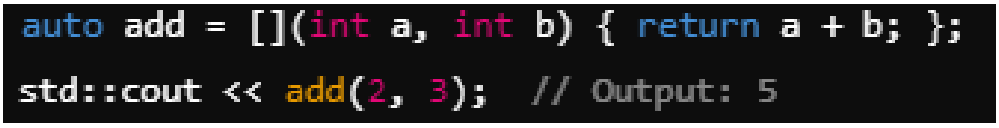
```
std::vector<int> vec = {1, 2, 3, 4};
for (auto value : vec) {
    std::cout << value << " ";
}
```

Nullptr
introduced as a new keyword to represent the null pointer, replacing the older NULL macro, making the distinction between pointers and integers clearer.

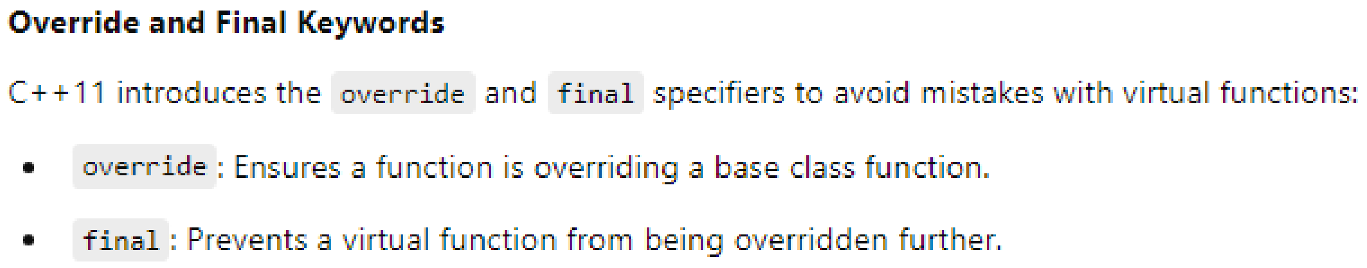


```
int* p = nullptr;
```

Lambda Functions
a convenient way to define anonymous functions inline, making it easier to pass functions as arguments.



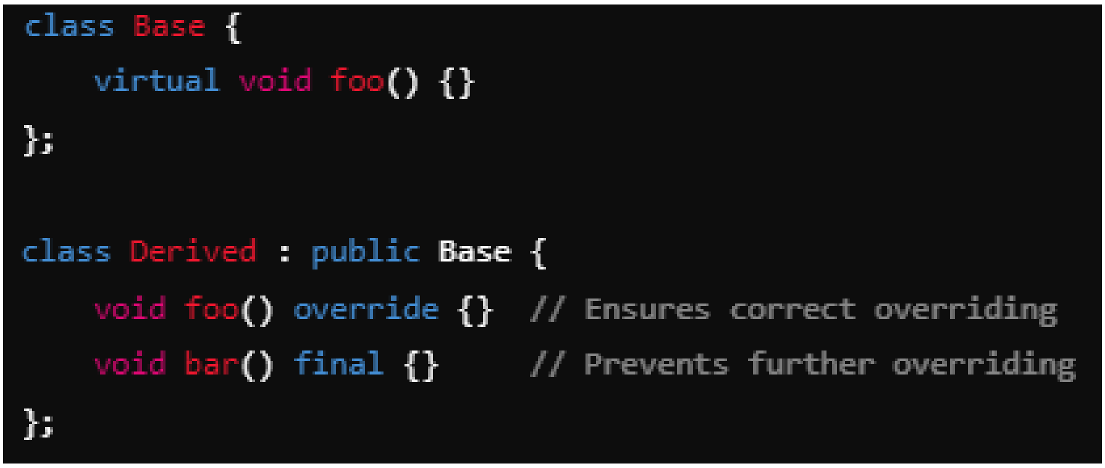
```
auto add = [](int a, int b) { return a + b; };
std::cout << add(2, 3);  // Output: 5
```



Override and Final Keywords

C++11 introduces the `override` and `final` specifiers to avoid mistakes with virtual functions:

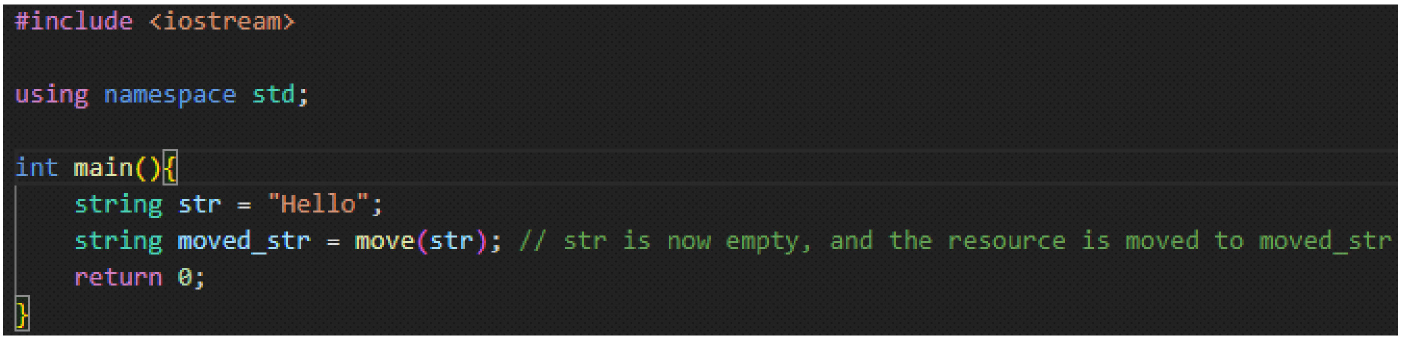
- `override`: Ensures a function is overriding a base class function.
- `final`: Prevents a virtual function from being overridden further.



```cpp
class Base {
    virtual void foo() {}
};

class Derived : public Base {
    void foo() override {}  // Ensures correct overriding
    void bar() final {}     // Prevents further overriding
};
```

Move Semantics and `std::move`

Move semantics improve the efficiency of handling temporary objects by allowing resources (like memory) to be moved rather than copied. This is particularly useful for avoiding expensive deep copies of large objects.



```cpp
#include <iostream>

using namespace std;

int main(){
    string str = "Hello";
    string moved_str = move(str); // str is now empty, and the resource is moved to moved_str
    return 0;
}
```