**class:**

It is a blueprint or template that defines the structure and behavior of objects. A class encapsulates data (known as member variables or attributes) and the functions (known as member functions or methods) that operate on that data.

During the declaration of the class, no memory is assigned, but when you create an object, then the memory is allocated.

Class class-name

{

Access-specifier: ///provide or grant access for the members.

Data\_variable

Member-function

{

body

}

};

return\_type class\_name::functionName(arguments){

// function working here

}

**Object:**

instance of the class

classname objectname;

Combination of data (known as properties or properties) and the function(knows as meths) that operate on that data.

- same class can have multiple objects

- object can belong to one class.

-Each object has its own set of member variables, and changes to one object do not affect the others.

**Access Specifier:**

Private: Only members of the same class have access to private members.

Public: You can access the public members from within, as well as from outside of the class.

Protected: You can access the protected members from the same class members and members of the derived class.

**Example:**

#include <iostream>

class MyClass {

private:

int privateVar;

protected:

int protectedVar;

public:

int publicVar;

// Constructor to initialize class members

MyClass(int privateVal, int protectedVal, int publicVal)

: privateVar(privateVal), protectedVar(protectedVal), publicVar(publicVal) {}

// Public method to access privateVar

void setPrivateVar(int value) {

privateVar = value;

}

// Public method to access protectedVar

void setProtectedVar(int value) {

protectedVar = value;

}

// Public method to access class members

void displayValues() const {

std::cout << "PrivateVar: " << privateVar << ", ProtectedVar: " << protectedVar

<< ", PublicVar: " << publicVar << std::endl;

}

};

int main() {

// Create an object of MyClass

MyClass myObject(10, 20, 30);

// Access publicVar directly

myObject.publicVar = 40;

// Access setPrivateVar and setProtectedVar methods

myObject.setPrivateVar(50);

myObject.setProtectedVar(60);

// Access displayValues method

myObject.displayValues();

// Accessing privateVar and protectedVar directly (would result in a compilation error)

// myObject.privateVar = 70; // Error: privateVar is private

// myObject.protectedVar = 80; // Error: protectedVar is protected

return 0;

}

**1.example:**

class MyClass { //MyClass class name

private: //default access specifier

int data; // member variable

public:

void setData(int value) {

data = value;

}

int getData() {

return data;

}

};

int main() {

MyClass myObject; // create an instance of the class

myObject.setData(42); // call member functions

int value = myObject.getData();

return 0;

}

The member functions of a class are shared among all instances (objects) of that class.

Each object contains its own set of data members (attributes), but the functions are shared among all objects of the same class.

==========================================

**how to find the content of the object:**

#include <iostream> //cin,cout,clog,cerr

#include <string>

class Person {

private:

int employeeid;

int age;

int height;

public:

// Constructor to initialize the object

Person(const int& employeeid, int a, double h) : employeeid(employeeid), age(a), height(h) {}

// Member function to display information about the person

void displayInfo() const {

std::cout << "Name: " << employeeid << ", Age: " << age << ", Height: " << height << " inches" << std::endl;

}

// Member function to have a birthday

void celebrateBirthday() {

age++;

std::cout << employeeid << " is now " << age << " years old. Happy birthday!\n";

}

};

int main() {

// Creating an object of the Person class

Person person1(775, 66, 106);

// Displaying the memory layout of the object

std::cout << "Memory layout of Person 1 object:\n";

// Using reinterpret\_cast to treat the object as an array of bytes

const int\* objectMemory = reinterpret\_cast<const int\*>(&person1);

// Displaying the content of each 4 bytes

for (std::size\_t i = 0; i < sizeof(Person) / sizeof(int); ++i) {

std::cout << "Int " << i << ": " << objectMemory[i] << std::endl;

}

// Displaying information about the person

std::cout << "\nInformation about Person 1:\n";

person1.displayInfo();

return 0;

}

========================

Memory layout of Person 1 object:

Int 0: 775

Int 1: 66

Int 2: 106

Information about Person 1:

Name: 775, Age: 66, Height: 106 inches

========================

**access specifier:**

Access specifiers (public, private, protected) in C++ classes control the visibility of class members.

public members are accessible from outside the class, private members are only accessible within the class, and protected members

are accessible within the class and its derived classes.

==========================================================

**How to define the function outside the class?**

best for for incremental compilation and larger project reduce the compilation time:

class Student {

public:

int a; // member variable

void function(int); // member function declaration

};

// member function definition

void Student::function(int value) {

// implementation of the function

}

How to pass the default argument to the function?  
class MyClass {

public:

// Function declaration with default values

void myFunction(int x = 0, int y = 0);

// Other class members...

};

// Function definition

void MyClass::myFunction(int x, int y) {

// Function implementation...

}

// Other class member definitions...

**inline function:**

functions defined inside a class are implicitly considered inline functions in C++. Defining functions as inline allows the compiler

to replace the function call with the actual function code, which can potentially lead to faster code execution and reduced stack usage.

Compile-time Overhead:

Code Bloat:

Optimization Decisions:

#include <iostream>

class Calculator {

public:

// Inline function to add two numbers

inline int add(int a, int b) {

return a + b;

}

// Inline function to multiply two numbers

int multiply(int a, int b);

};

// Definition of the multiply function as an inline function

inline int Calculator::multiply(int a, int b) {

return a \* b;

}

int main() {

Calculator calculator;

// Using inline functions

int result\_add = calculator.add(5, 3);

return 0;

}

==================

Polymorphism

1. compile time polymorphism (function overloading or static polymorphism)
2. runtime time polymorphism (function overriding or runtime or static polymorphism)

**Function overloading: (achieved at compile time)**

Function overloading is the ability to define multiple functions in the same scope with the same name but different parameters. The compiler determines which function to call based on the number and types of arguments provided during the function invocation.

Member function cab e overlaoaded based on the const type.

void area(int a);

void area(int a, int b);

**void** fun() **const**  { cout << "fun() const " << endl; }

**void** fun()        {  cout << "fun() " << endl;     }

* Improve the code readability and maintainability
* Functionoverlading cannot be achieved for the different return type alone, static and non static, pointer and array and default argument.
* All the default argument of the function overloading needs to in the left hand size.  
  **int** fun(**int** x , **int** y = 0, **int** z=0)

#include <iostream>

class MathOperations {

public:

// Overloaded functions for adding integers and doubles

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

// Overloaded function for concatenating strings

std::string add(const std::string& str1, const std::string& str2) {

return str1 + str2;

}

};

int main() {

MathOperations math;

std::cout << math.add(5, 7) << std::endl; // Calls int add(int a, int b)

std::cout << math.add(3.14, 2.7) << std::endl; // Calls double add(double a, double b)

std::cout << math.add("Hello", " World") << std::endl; // Calls string add(const string& str1, const string& str2)

return 0;

}

a. int fun(int x, int y);

void fun(float x, int y);

b. int fun(int x, int y);

void fun(int x, int y);

c. int fun(int x, int y);

static int fun(int x, int y);

d. int fun(int \*ptr, int n);

int fun(int ptr[], int n);

e. int fun( int x, int y);

int fun( int x, int y = 10);

Function overriding:

Function overriding is a concept in object-oriented programming where a derived class provides a specific implementation for a method that is already defined in its base class. The overridden method in the derived class has the same signature (name, return type, and parameters) as the method in the base class.

**Pure Virtual function**: Only declaration but no definition. A pure virtual function is declared by assigning 0 in declaration.

// An abstract class

class Test {

// Data members of class

public:

// Pure Virtual Function

virtual void show() = 0;

/\* Other members \*/

};

**Abstract Class:**

Class containing atleast one pure virtual function is called abstract class. Abstract class cannot have the object.// pure virtual functions make a class abstract

//if the class is derived from the abstract class but not implement the virtual function then that class also become abstract class and hence cannot have the object.

#include <iostream>

using namespace std;

class Test {

int x;

public:

virtual void show() = 0;

int getX() { return x; }

};

int main(void)

{

Test t;

return 0;

}

1. Can the constructor be virtual: No, we cannot have constructor as Virtual. The reason is that constructor is called before any of the function in the class.

b. Does virtual needs to be overridden: No, not every virtual function needs to be overridden. However, if a virtual function is not overridden in a derived class, then the base class implementation will be called.

c. Virtual Destructors?:Yes, virtual destructors can be used to ensure that the correct destructor is called when an object is destroyed.

#include <iostream>

// Base class with a virtual function and a virtual destructor

class Shape {

public:

virtual void draw() const {

std::cout << "Drawing a shape." << std::endl;

}

// Virtual destructor

virtual ~Shape() { //if it is not virtual then the deriver class destructor is not called.

std::cout << "Base class destructor." << std::endl;

}

};

// Derived class overriding the virtual function

class Circle : public Shape {

public:

void draw() const override {

std::cout << "Drawing a circle." << std::endl;

}

// Destructor (not virtual, but still gets called through the base class pointer/reference)

~Circle() {

std::cout << "Derived class destructor (Circle)." << std::endl;

}

};

int main() {

// Example demonstrating dynamic dispatch using a pointer

Shape\* shapePtr = new Circle(); // Polymorphism: base class pointer pointing to a derived class object

shapePtr->draw(); // Calls the overridden method in the derived class (Circle)

delete shapePtr; // Calls the virtual destructor in the base class and releases memory

// Example demonstrating dynamic dispatch using a reference

Circle circle;

Shape& shapeRef = circle; // Polymorphism: base class reference referring to a derived class object

shapeRef.draw(); // Calls the overridden method in the derived class (Circle)

return 0;

}

========================================

Data Types:  
C++ is a statically-typed programming language, which means that the data types of variables must be declared before they are used.  
  
**Integer Types:**

**int**

**short**

**long**

**long long**

**unsigned int, unsigned, unsigned long, unsigned long long**

**floating point Type:**

**float**

**double**

**long double**

**character Type:**char: Character type.

wchar\_t: Wide character type.

char16\_t, char32\_t: Unicode character types.

**Boolean Type:**

**User-Defined Types: Struct, class and union**

**unScoped enum**

**enum color { black, blue, green, cyan, red, magenta, yellow, white };**

**int c1{yellow};**

**color c2{ color(c1 + 1) };**

**color c3{ static\_cast<color>(c2 + 1) };**

**limiting the size of the enum**

**enum sign : char {**

**negative,**

**positive**

**};**

**enum class ColorClass { black, red, green, yellow, blue, magenta, cyan, white }; //scoped enum**

**enum ColorEnum { black, red, green, yellow, blue, magenta, cyan, white }; //unscoped enum**

**ColorClass classColor = ColorClass::red;**

char h = 'a';

wchar\_t i = L'b';

char16\_t j = u'c';

char32\_t k = U'd';

cout << "Character data types: " << endl;

cout << "char: " << h << endl;

wcout << "wchar\_t: " << i << endl;

cout << "char16\_t: " << j << endl;

cout << "char32\_t: " << k << endl;

char: a

wchar\_t: b

char16\_t: 99

char32\_t: 100

wchar\_t wideString[] = L"Hello, \xe7\x9a\x84 wchar\_t!"; //Hello, 的 wchar\_t!

char Vs String?