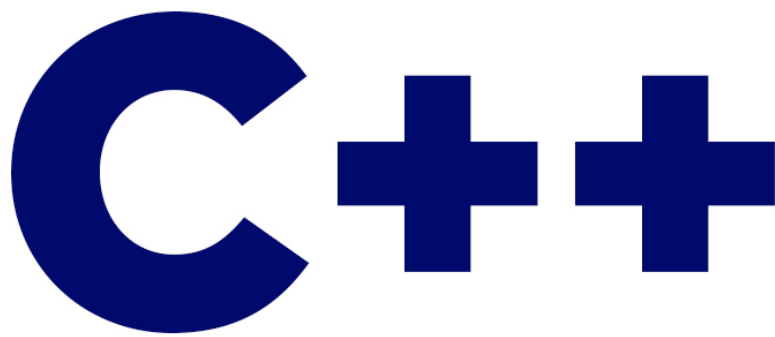


The Ultimate

The C++ logo is displayed in a dark blue color. It consists of a large, bold, sans-serif capital letter 'C' followed by two bold, sans-serif plus signs ('++'). The logo is centered within a white rectangular box that has rounded corners.

Part 3: Object-oriented Programming



Hi! I am Mosh Hamedani. I'm a software engineer with over 20 years of experience and I've taught millions of people how to code and become professional software engineers through my YouTube channel and coding school (Code with Mosh).

This PDF is part of my **Ultimate C++ course** where you will learn everything you need to know from the absolute basics to more advanced concepts. You can find the full course on my website.

<https://codewithmosh.com>

<https://www.youtube.com/c/programmingwithmosh>

<https://twitter.com/moshhamedani>

<https://www.facebook.com/programmingwithmosh/>

Table of Content

Classes.....	4
Operator Overloading.....	11
Inheritance.....	14
Exceptions.....	18
Template.....	20

Classes

Terms

Access modifiers	Header (interface) file
Classes	Immutable objects
Constructors	Implementation file
Copy constructor	Member initializer list
Data hiding	Object-oriented programming
Default constructor	Objects
Destructors	Programming paradigms
Getters and setters	Static members
Encapsulation	Unified Modeling Language (UML)

Summary

- A *programming paradigm* is a style or way of writing software. The two most popular programming paradigms are *object-oriented* and *functional programming*.
- An *object* is a software entity that contains *attributes (properties)* and *functions (methods)*. In C++, these are called *member variables* and *member functions* respectively.
- A *class* is a blueprint for creating objects. An object is an instance of a structure or class.
- We often use structures as simple data containers and classes for creating objects that can do things.
- *Encapsulation* means combining the data and functions that operate on the data into a single unit (class / object). The words class and object are often used interchangeably.
- UML is short for *Unified Modeling Language*. It's a visual language for representing the classes of an application.

- To create a class, we need two files: a *header (interface) file* with **.h** extension, and an *implementation file* with **.cpp** extension.
- *Access modifiers* control how members of a class are accessed. *Public* members are accessible everywhere. *Private* members are only accessible within a class.
- Once we create an object from a class, that object has a *state* (the data that it stores). As our program runs, the state of an object may change.
- Objects should protect their internal state and provide functions for accessing the state. This is referred to as *data* or *information hiding* in object-oriented programming.
- *Getters (accessors)* allow us to read the values stored in member variables.
- *Setters (mutators)* allow us to change the values of member variables.
- A *constructor* is a special function inside a class that is used for initializing objects. It gets automatically called when an instance of a class is created.
- Member variables can be initialized in the constructor and/or using the *member initializer list*. Initializing member variables using a member initializer list is more efficient because variables are created and initialized in a single operation.
- A *default constructor* is a constructor with zero parameters. The C++ compiler automatically generates a default constructor for a class with no constructors. This allows us to create instances of that class without providing an argument.
- A *copy constructor* is used to create an object as a copy of an existing object. It's called when we declare and initialize an object as well as when we pass an object to a function (by value) and return it (by value). The C++ compiler automatically generates a copy constructor for our classes unless we define one.

- A *destructor* is another special function inside classes that is used for releasing system resources (eg memory, file handles, etc). Destructors are automatically called when objects are destroyed. C++ automatically destroyed objects declared on the stack when they go out of scope. Objects declared on the heap (free store) should be explicitly released using the delete operator.
- *Static* members of a class are shared by all objects of the class. Static functions of a class cannot access instance members because they don't know about the existence of any instances.
- If we declare an object using the **const** keyword, all its member variables will become constant as well. We refer to this object as *immutable* (unchangeable).

Creating a Class

```
// Declaring a class (in Rectangle.h)
class Rectangle {
public:
    int getArea();
    void draw();
private:
    int width;
    int height;
};

// Implementation of the class (in Rectangle.cpp)
#include "Rectangle.h"

int Rectangle::getArea() {
    return width * height;
}

void Rectangle::draw() {
    cout << "Drawing a Rectangle";
}
```

Getters and setters

```
class Rectangle {  
public:  
    int getWidth() const;  
    void setWidth(int width);  
private:  
    int width;  
};  
  
int Rectangle::getWidth() {  
    return width;  
}  
  
void Rectangle::setWidth(int width) {  
    if (width < 0)  
        throw invalid_argument("width");  
    this->width = width;  
}
```


Constructors and destructor

```
class Rectangle {  
public:  
    // Default constructor  
    Rectangle() = default;  
    // Copy constructor  
    Rectangle(const Rectangle& source);  
    // Constructor with parameters  
    Rectangle(int width, int height);  
    // Destructor  
    ~Rectangle();  
};
```

Member initializer list

```
Rectangle::Rectangle(int width, int height)  
    : width{width}, height{height} {  
  
}
```

Constructor delegation

```
Rectangle::Rectangle(int width, int height, const string& color)  
    : Rectangle(width, height) {  
    this->color = color;  
}
```

Static members

```
class Rectangle {  
public:  
    static int getObjectCount();  
private:  
    static int objectsCount;  
};  
  
int main() {  
    cout << Rectangle::getObjectCount();  
    return 0;  
}
```

Operator Overloading

Terms

Binary operators

Friends of classes

Inline functions

Operator overloading

Spaceship operator

Subscript operator

Three-way comparison operator

Unary operators

Summary

- By overloading the built-in operators, we can make them work with our custom types (classes and structures). Most operators can be overloaded.
- If we overload `==` for a class, a C++ 20 compiler makes sure `!=` works as well.
- The *spaceship operator* `<=>` (also called *the three-way comparison operator*) determines, in a single expression, if X is less than, equal to, or greater than Y.
- If we overload `<=>` for a class, a C++ 20 compiler will generate all four relational operators (`<`, `<=`, `>`, `>=`) for us. It does not, however, generate `==` and `!=` operators as this can lead to suboptimal performance. So, the only two operators we need to overload are `==` and `<=>`.
- *Friend* functions have access to private members of a class despite not being members of the class. Friend functions undermine data hiding and should be used only when absolutely necessary.
- When overloading arithmetic operators (`+`, `-`, `*`, `/`), it's often best to overload their corresponding compound assignment operators (`+=`, `-=`, `*=`, `/=`).

- The copy constructor is called when creating a new object as a copy of an existing one. The assignment operator is used when assigning to an existing object.
- If we overload the assignment operator for a class, we often need to define a copy constructor for that class as well.
- The subscript operator [] is used to get access to individual elements in an object that behaves like an array or a collection.
- Functions defined in a class header file are known as *inline*. We can explicitly make functions defined in an implementation file inline using the **inline** keyword. Inline functions hint the compiler to optimize the executable by replacing each function call with the code in the function itself. Whether this happens or not is up to the compiler.

```
class Length {  
public:  
    // Comparison operators  
    bool operator==(const Length& other) const;  
    std::strong_ordering operator<=>(const Length& other) const;  
  
    // Arithmetic operators  
    Length operator+(const Length& other) const;  
    Length& operator+=(const Length& other);  
  
    // Assignment operator  
    Length& operator=(const Length& other);  
  
    // Increment operator  
    Length& operator++(); // prefix  
    Length operator++(int); // postfix  
  
    // Type conversion  
    operator int() const;  
};
```

Inheritance

Terms

Abstract classes

Base class

Child class

Derived class

Dynamic binding

Early binding

Final classes and methods

Inheritance

Late binding

Overriding a method

Parent class

Polymorphism

Protected members

Pure virtual methods

Redefining a method

Static binding

Summary

- *Inheritance* allows us to create a new class based on an existing class. The new class automatically inherits all the members of the base class (except constructors and destructor).
- A class that another class inherits is called a *base* or *parent class*.
- A class that inherits another class is called a *derived* or *child class*.
- *Protected members* of a class are accessible within a class and the derived classes, but not outside of these classes.
- Instances of a derived class can be implicitly converted to their base type because they contain everything the base class expects.

- A derived class can *redefine* a non-virtual method in the base class by providing its own version of that method.
- A derived class can *override* a virtual function in the base class by providing its own implementation of that function.
- *Polymorphism* refers to the situation where an object can take many different forms. This happens when we pass a child object to a function that expects an object of the base class.
- Redefining a function prevents polymorphic behavior because it forces the compiler to decide which function to call at compile time. This is known as *static or early binding*.
- Overriding a function enables polymorphism. It allows the compiler to determine which function to call at runtime. This is known as *dynamic or late binding*.
- A *pure virtual method* is a method that has to be overridden in a derived class. We can declare a method as a pure virtual method by coding “= 0” in its declaration.
- An *abstract class* is a class with at least one pure virtual method.
- Abstract classes cannot be instantiated. They exist just to provide common code to derived classes.
- *Final methods* cannot be overridden.
- *Final classes* cannot be inherited.
- *Multiple inheritance* allows a class to inherit multiple classes.
- While inheritance is a great technique for code reuse, deep inheritance hierarchies often result in complex, unmaintainable applications and should be avoided.

```
// Inheritance
class TextBox : public Widget {};

// Calling the constructor of the base class
TextBox::TextBox(bool enabled) : Widget(enabled) {}

// Implicit conversion to the base type.
// Object slicing happens here.
TextBox textBox;
Widget widget = textBox;

// No object slicing happens here because
// we're using a reference variable.
TextBox textBox;
Widget& widget = textBox;

// Method overriding
class Widget {
public:
    virtual void draw() const;
};

class TextBox : public Widget {
public:
    void draw() const override;
};
```



```
// Abstract class
class Widget {
public:
    // Pure virtual method
    virtual void draw() const = 0;
};

class TextBox : public Widget {
public:
    // Final method
    void draw() const override final;
};

// Final class
class TextBox final : public Widget {
public:
};

// Multiple inheritance
class DateTime : public Date, public Time {};
```

Exceptions

Terms

Exception

Catch an exception

Throw an exception

Rethrow an exception

Call stack

Summary

- We use *exceptions* to report errors that occur while our program is running. An exception is an object that contains information about an error.
- The C++ STL offers a hierarchy of predefined exceptions. All these exceptions derive from the **exception** class.
- We *throw* an exception using the **throw** statement.
- We *catch* exceptions using a **try** statement. A try statement has a try block, followed by one or more catch blocks, each responsible to handle a certain type of exception.
- When using multiple catch blocks, we should order them from the most specific to more generic ones.
- Sometimes we need to catch an exception and re-throw it so it can be handled in a different part of the program. To do that, we use the **throw** keyword without any arguments.
- To create a custom exception, we create a class that inherits the **exception** class in the STL.
- The *call stack* lists the functions that have been called in the reverse order. When an exception is thrown, the runtime looks for a catch block through the call stack. If no catch block is found, the program crashes.

```
try {  
    // Code that may throw an exception  
    doWork();  
}  
  
// Catch blocks ordered from the most specific  
// to more generic ones  
catch (const invalid_argument& ex) {  
    cout << ex.what();  
}  
  
catch (const logic_error& ex) {  
    cout << ex.what();  
}  
  
catch (const exception& ex) {  
    cout << ex.what();  
}
```

Templates

Terms

Class templates

Function templates

Generics

Template argument

Template declaration

Template parameter

Summary

- *Templates* allow us to create flexible functions and classes that can work with all data types. They're called *generics* in other programming languages like C#, Java, TypeScript, etc.
- To create a function/class template, we code a *template declaration* before the function/class. The declaration consists of the **template** keyword followed by a pair of angle brackets (<>).
- Within the angle brackets, we type one or more type parameters. The type of these parameters can be **class** or **typename**.
- By convention, we name these parameters **T**, **U**, **V**, but we can name them whatever that makes more sense.
- The compiler generates instances of a function/class template based on the usages. If we don't use a function/class template, it's not included in the executable.
- Methods of class templates should be implemented in the header file.
- When using templates, most of the time the compiler can guess the type of parameters. If not, we have to explicitly supply type arguments.

```
// Function template
template<typename K, typename V>
void display(K key, V value) {}

// Class template
template<typename K, typename V>
class Pair {
public:
    Pair(K key, V value);
private:
    K key;
    V value;
};

template<typename K, typename V>
Pair<K, V>::Pair(K key, V value): key{key}, value{value} {
}
```