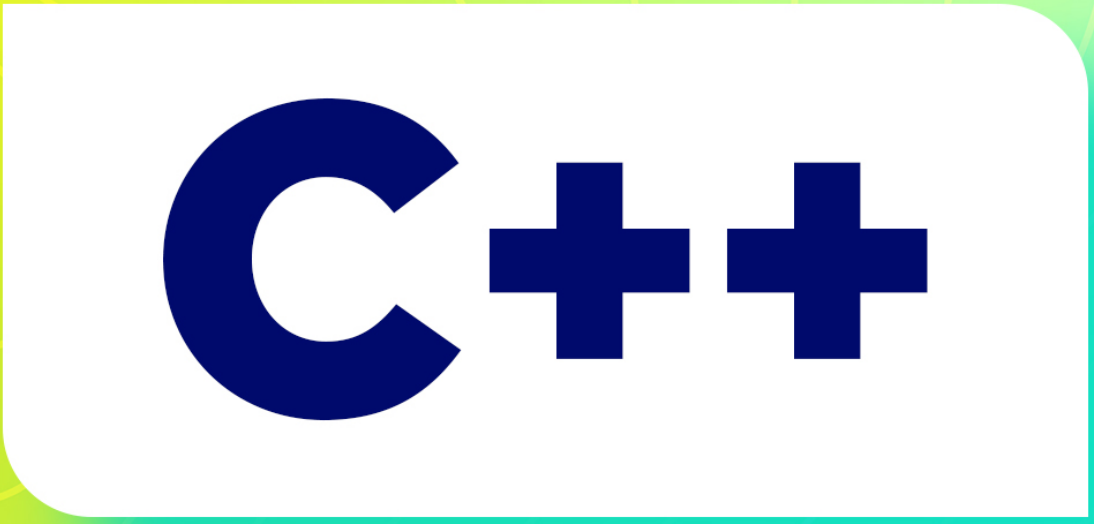
The Ultimate



Part 1: The Fundamentals

Mosh Hamedani
codewithmosh.com

Table of Content

Getting Started

Terms

C++ Standard Library
Compiling
Console application
Function
Graphical User Interface (GUI)
Integrated Development Environment
Machine code
Statement
Syntax
Terminal

Summary

- C++ is one of the oldest yet most popular programming languages in the world due to its performance and efficiency.
- It's often used in building performance-critical applications, video games (especially with Unreal Engine), servers, operating systems, etc.
- To learn C++, you need to learn the syntax (grammar) of the language as well as C++ Standard Library, which is a collection of pre-written C++ code for solving common problems.
- To write C++ applications, we often use an Integrated Development Environment (IDE). The most popular IDEs are MS Visual Studio, XCode, and CLion.
- To run C++ applications, first we have to compile our C++ code to machine code.
- The main() function is the starting point of a C++ program.

Your First C++ Program

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World";
    return 0;
}
```

The Basics

Terms

Camel case
Comment
Constant
Directive
Expression
Hungarian notation
Mathematical expression
Operand
Operator
Pascal case
Snake case
Standard input stream
Standard output stream
Stream extraction operator
Stream insertion operator
Variable

Summary

- We use *variables* to temporarily store data in the computer's memory.
- To declare a variable, we should specify its type and give it a meaningful name.
- We should initialize variables before using them. Using an uninitialized variable can lead to unexpected behavior in our programs since these variables hold garbage values.
- Unlike variables, the value of *constants* don't change.
- The common naming conventions used in C++ applications are: **PascalCase**, **camelCase**, and **snake_case**.
- An *expression* is a piece of code that produces a value. A mathematical (arithmetic) expression consists of an operator (+, -, *, /, %) and two operands.
- Multiplication and division operators have a higher priority than addition and subtraction operators. So, they're applied first.

- We can use parentheses to change the order of operators.
- We use **cout** (pronounced sea-out) to write characters to the *Standard Output Stream* which represents the *terminal* or *console* window.
- We use **cin** (pronounced sea-in) to read data from the *Standard Input Stream* which represents the keyboard.
- We use the *Stream Insertion Operator* (<<) to write data to a stream.
- We use the *Stream Extraction operator* (>>) to read data from a stream.
- The *Standard Template Library* (STL) consists of several files each containing functions for different purposes.
- To use functions in the Standard Library, we should include the corresponding files using the **#include** *directive*.
- Using *comments* we can explain what cannot be expressed in code. This includes why's, how's, and any assumptions we made while writing code.

```
// Declaring a variable
int number = 1;

// Declaring a constant
const double pi = 3.14;

// Mathematical expressions
int x = 10 + 3;

// Writing to the console
cout << "x = " << x;

// Reading from the console
cin >> number;
```


Data Types

Terms

Array
Binary system
Boolean values
Casting
Characters
Compile-time error
Data type
Decimal system
Floating-point number
Hexadecimal system
Index
Overflow
Run-time error
Stream manipulator
String
Underflow

Summary

- C++ has several built-in *data types* for storing *integers* (whole numbers), *floating-point numbers* (numbers with a decimal point), characters, and Boolean values (true/false).
- Floating-point numbers are interpreted as **double** by default. To represent a **float**, we have to add the **F** suffix to our numbers (eg 1.2F).
- Whole numbers are interpreted as **int** by default. To represent a **long,** we have to use the **L** suffix (eg 10L).
- Using the **auto** keyword, we can let the compiler infer the type of a variable based on its initial value.
- Numbers can be represented using the *decimal*, *binary*, and *hexadecimal systems*.
- If we store a value larger or smaller than a data type's limits, *overflowing* or *underflowing* occurs.

- Using the **sizeof()** function, we can see the number of bytes taken by a data type.
- We can use *stream manipulators* to format data sent to a stream. The most common manipulators are **setw, fixed, setprecision, boolalpha, left,** and **right.**
- The Boolean **false** is represented as 0. Any non-zero number is interpreted as the Boolean **true**.
- In C++, characters should be surrounded with single quotes.
- Characters are internally represented as numbers.
- A *string* is a sequence of characters and should be surrounded by double quotes.
- We use *arrays* to store a sequence of items (eg numbers, characters, etc).
- Array elements can be accessed using an *index*. The index of the first element in an array is 0.
- When we store a smaller value in a larger data type, the value gets automatically *cast* (converted to) the larger type. When storing a large value in a smaller data type, we have to explicit cast the value.
- C-style casting involves prefixing a variable with the target data type in parentheses. In C++, we use the **static_cast** operator.
- C++ casting is safer because conversion problems can be caught at the *compile-time*. With C-style casting, we won't know about conversion issues until the *run-time*.


```cpp
// Data types
double price = 9.99;
float interestRate = 3.67F;
long fileSize = 90000L;
char letter = 'a';
string name = "Mosh";
bool isValid = true;
auto years = 5;

// Number systems
int x = 255;
int y = 0b111111;
int z = 0xFF;


// Data types size and limits
int bytes = sizeof(int);
int min = numeric_limits<int>::min();
int max = numeric_limits<int>::max();

// Arrays
int numbers[] = { 1, 2, 3 };
cout << numbers[0];

// C-style casting
double a = 2.0;
int b = (int) a;

// C++ casting
int c = static_cast<int>(a);
```


Decision Making

Terms

Boolean expression
Comparison operators
Conditional operator
If statement
Logical operators
Nesting if statements
Switch statement

Summary

- We use *comparison operators* to compare values.
- A *Boolean expression* is a piece of code that produces a Boolean value.
- With *Logical operators* we can combine Boolean expressions and represent more complex conditions.
- With the *logical AND operator* (&&) both operands should be true. If either of them is false, the result will be false.
- With the *logical OR operator* (||), the result is true if either of the operands is true.
- The *logical NOT operator* (!) reverses a Boolean value.
- Using ***if*** and ***switch*** *statements*, we can control the logic of our programs.
- An if statement can have zero or more **else if** clauses for evaluating additional conditions.

- An if statement can optionally have an **else** clause.
- We can code an if statement within another. This is called *nesting* if statements.
- Using the *conditional operator* we can simplify many of the if/else statements.
- We use **switch** statements to compare a variable against different values.
- A switch block often has two or more *case labels* and optionally a *default label*.
- Case labels should be terminated with a **break** statement; otherwise, the control moves to the following case label.
- Switch statements are not as flexible as if statements but sometimes they can make our code easier to read.


```
// Comparison operators
bool a = 10 > 5;
bool b = 10 == 10;
bool c = 10 != 5;


// Logical operators
bool d = a && b; // Logical AND
bool e = a || b; // Logical OR
bool f = !a;     // Logical NOT


if (temperature < 60) {
    // ...
}
else if (temperature < 90) {
    // ...
}
else {
    // ...
}


// Conditional operator
double commission = (sales < 10'000) ? .05 : .1;
```



```
switch (menu) {
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    default:
        // ...
}
```


Loops

Terms

Break statement
Continue statement
Do-while statement
For statement
Infinite loop
Iteration
Loops
Loop variable
Nested loop
While statement

Summary

- We use *loops* to repeat a set of statements.
- In C++, we have four types of loops: **for** loops, range-based **for** loops, **while** loops, and **do-while** loops.
- **For** loops are useful when we know ahead of time how many times we want to repeat something.
- Range-based **for** loops are useful when iterating over a list of items (eg an array or a string).
- **While** and **do-while** loops are often used when we don't know ahead of time how many times we need to repeat something.
- Using the **break** statement we can break out of a loop.
- Using the **continue** statement we can skip an iteration.


```cpp
// For loop
for (int i = 0; i < 5; i++)
    cout << i << endl;

// Same algorithm using a while loop
int i = 0;
while (i < 5) {
    cout << i << endl;
    i++;
}

// Same algorithm using a do-while loop
i = 0;
do {
    cout << i << endl;
    i++;
} while (i < 5);


// Range-based for loop
int numbers[] = { 1, 2, 3 };
for (int number: numbers)
    cout << number << endl;
```


Functions

Terms

Debugging
Functions
Function arguments
Function declaration
Function definition
Function parameters
Function prototype
Function signature
Global variables
Invoking a function
Header files
Local variables
Namespaces
Overloading functions

Summary

- A *function* is a group of one or more statements that perform a task. Each function should have a clear responsibility. It should do one and only one thing.
- A function can have zero or more *parameters*
- *Arguments* are the values passed to a function.
- To *call* (or *invoke*) a function, we type its name, followed by parenthesis, and the arguments (if any).
- Function parameters can have a default value. This way, we don't have to provide arguments for them.
- The *signature* of a function includes the function name, and the number, order, and type of parameters.

- *Overloading* a function means creating another variation with a different signature. By overloading functions, we can call our functions in different ways.

- Arguments of a function can be passed by value or reference. When passed by value, they get copied to the parameters of the function.

- To pass an argument by a reference, we should add an & after the parameter type.

- *Local variables* are only accessible within the function in which they are defined. *Global variables* are accessible to all functions.

- Global variables can lead to hard-to-detect bugs and should be avoided as much as possible.

- A *function declaration* (also called a *function prototype*) tells the compiler about the existence of a function with a given signature. A *function definition* (or implementation) provides the actual body (or code) for the function.

- As our programs grow in more complexity, it becomes critical to split our code into separate files.

- A *header file* ends with "**.h**" or "**.hpp**" extension and consists of function declarations and constants. We can import header files using the **#include** directive.

- An *implementation file* ends with ".**cpp**" extension and consists of function definitions.

- Using *namespaces* we can prevent name collisions in our programs.

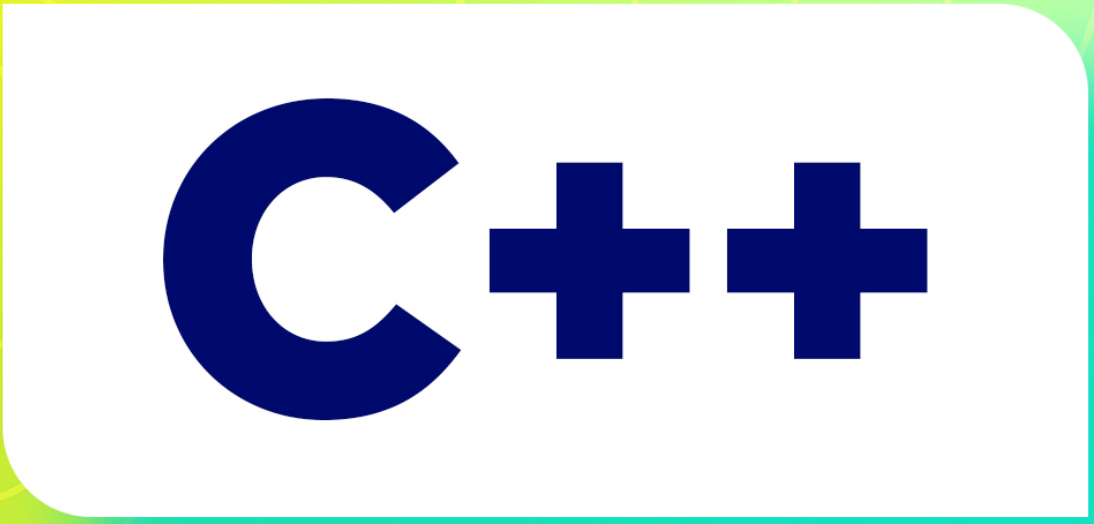
- *Debugging* is a technique for executing a program line by line and identifying potential errors.


```cpp
// Defining functions
void greet(string name) {
    cout << "Hello " << name;
}

string fullName(string firstName, string lastName) {
    return firstName + " " + lastName;
}


// Parameters with a default value
double calculateTax(double income, double taxRate = .3) {
    return income * taxRate;
}


// Overloading functions
void greet(string name) {
}

void greet(string title, string name) {
}

// Reference parameters
void increase(double& number) {
    number++;
}
```



```
// Function declaration
void greet(string name);


// Defining a namespace
namespace messaging {
    void greet(string name) {}
}


// Using a namespace
using namespace messaging;
// or
using messaging::greet;
```


The Ultimate



Part 2: Intermediate

Mosh Hamedani
codewithmosh.com

Table of Content

Arrays

Terms

Arrays
Array elements
Bubble sort algorithm
Linear search algorithm
Pointers
Unpacking arrays

Summary

- An *array* is a collection of items stored in sequence.
- We can access the items (also called *elements*) in an array using an index. The index of the first element in an array is 0.
- An array can be unpacked into separate variables.
- Using the **size()** function in the C++ Standard Library (STL) we can determine the size of an array.
- When passing an array to a function, we should always pass its size as well. The reason for this is that array parameters are interpreted as *pointers* (memory addresses). We cannot use a memory address to determine how many elements exist in an array.
- In C++, we cannot assign an array to another. To copy an array, we need to copy individual elements using a loop.
- Similarly, arrays cannot be compared for equality. To compare two arrays, we have to compare their individual elements.
- There are many algorithms for searching and sorting arrays. These algorithms differ based on their performance and memory usage.


```cpp
// Creating and initializing an array
int numbers[] = { 1, 2, 3 };
string names[5];

// Accessing elements in an array
numbers[0] = 10;
cout << numbers[0];

// Determining the size of an array
auto size : = std::size(numbers);

// Unpacking arrays
auto [x : int , y : int , z : int ] = numbers;

// Multi-dimensional arrays
int matrix[2][3] = {
        { 11, 12, 13 },
        { 21, 22, 23 }
};

matrix[0][0] = 10;
```


Pointers

Terms

Address-of operator
Dereference operator
Free store
Heap memory
Indirection operator
Memory leak
Null pointer
Pointer
Reference parameters
Shared pointers
Smart pointers
Stack memory
Unique pointers

Summary

- Using the *address-of operator* (**&**) we can get the address of a variable.
- A *pointer* is a variable that holds the memory address of another variable.
- Using the *indirection* or *dereference operator* **(*)** we can get the content of a memory address stored in a pointer.
- One of the applications of pointers is to efficiently pass objects between function calls. Reference parameters are a safer and simpler alternative for the same purpose.
- A *null pointer* is a pointer that doesn't point to any objects.
- Local variables are stored in a part of memory called the *stack memory*. The memory allocated to these variables is automatically released when they go out of scope.

- We can use the **new** operator to dynamically allocate memory on a different part of memory called the *heap* (or *free store*).

- When allocating memory on the heap, we should always deallocate it using the **delete** operator. If we don't, our program's memory usage constantly increases. This is known as a *memory leak*.

- *Smart pointers* in the STL are the preferred way to work with pointers because they take care of releasing the memory when they go out of scope.

- There are two types of smart pointers: *unique* and *shared*.

- A unique pointer owns the memory address it points to. So we cannot have two unique pointers pointing to the same memory location.

- If we need multiple pointers pointing to the same memory location, we have to use shared pointers.


```cpp
// Declaring and using pointers
int number = 10;
int* ptr = &number;
*ptr = 20;

// Pointer to constant data (const int)
const int x = 10;
const int* ptr = &x;

// Constant pointer
int x = 10;
int* const ptr = &x;

// Constant pointer to constant data
int x = 10;
const int* const ptr = &x;

// Dynamic memory allocation using raw pointers
int* numbers = new int[10];
delete[] numbers;

// Dynamic memory allocation using smart pointers
#include <memory>
auto numbers : unique_ptr<int[]> = make_unique<int[]>(10);
```


Strings

Terms

C String
C++ String
Character literal
Concatenate
Escape sequences
Raw string
String literal
Substring

Summary

- A *C-string* (also called *C-style String*) is an array of characters terminated with the null terminator (**\0**).
- C-strings cannot be copied, compared, or concatenated (combined). That's why you should avoid them in new code and prefer to use C++ strings.
- C++ strings are represented using the **string** class in the STL. This class internally uses a character array to hold a string. But it hides away all the complexity around C-strings. It dynamically resizes the array when needed and provides useful methods for working with strings.
- A *string literal* is a sequence of characters enclosed with double quotation marks.
- A *character literal* is a character enclosed with single quotation marks.
- We use escape sequences to represent special characters within string and character literals. Examples of escape sequences are `\\`, `\"`, `\'`, `\n` and `\t`.
- In *raw strings*, escape sequences are not processed.


```
// Working with C-strings
char name[5] = "Mosh";
char copy[5];

cout << strlen(name);

strcpy(copy, name);

if (strcmp(name, copy) == 0)
    cout << "Equal";


// Working with C++ strings
string name = "Mosh";

cout << name.length();

string copy = name;

if (name == copy)
    cout << "Equal";


// Modifying strings
string name = "Mosh";
name.append(" Hamedani");
name.insert(0, "I am ");
name.erase(0, 2);
name.clear();
name.replace(0, 2, "**");
```

```
// Searching strings
string name = "Mosh";
int index;
index = name.find('a');
index = name.rfind('a');
index = name.find_first_of(",.;");
index = name.find_last_of(",.;");
index = name.find_first_not_of(",.;");

// Extracting substrings
string name = "Mosh Hamedani";
string substr;
substr = name.substr();
substr = name.substr(3);
substr = name.substr(3, 5);


// Working with characters
string name = "Mosh Hamedani";
bool b;
b = isupper(name[0]);
b = islower(name[0]);
b = isdigit(name[0]);
b = isalpha(name[0]);

name[0] = toupper(name[0]);
name[0] = tolower(name[0]);
```



```cpp
// Conversions
string str = "10";
int i = stoi(str);
double d = stod(str);
string s = to_string(10);

// Escape sequences
string message = "Hello\nWorld";
string columns = "first\tlast";
string path = "c:\\folder\\file.txt";

// Raw strings
string path = R"(c:\folder\file.txt)";
```


Structures

Terms

Enumerations
Enumerators
Methods
Nested structures
Operator overloading
Strongly-typed enums
Structures
Structure members

Summary

- We use *structures* to define custom data types.
- Members of a structure can be variables or functions (also called *methods*).
- Structures can be nested to represent more complex types.
- To compare two structures, we have to compare their individual members.
- We can provide operators for our structures using a technique called *operator overloading*.
- Just like the built-in data types, structures can be used as function parameters or their return type.
- Using an *enumeration*, we can group related constants into a single unit. Members of this unit are called *enumerators*.
- *Strongly-typed enumerations* define a scope for their members. This allows two enums having members with the same name.


```cpp
// Defining a structure
struct Movie {
    string title;
    int releaseYear = 1900;
};


// Creating an instance of a structure
Movie movie = { "Terminator 1", 1984 };

// Unpacking a structure
auto [title : string, releaseYear : int] = movie;


// Operator overloading
bool operator==(const Movie& first, const Movie& second) {
    return (
        first.title == second.title &&
        first.releaseYear == second.releaseYear
    );
}

// Pointer to a structure
void showMovie(Movie* movie) {
    // Structure pointer operator
    cout << movie->title;
}
```



```cpp
// Classic (unscoped) enumeration
enum Action {
    list,
    add,
    update
};


// Strongly-typed enumeration
enum class Operation {
    list,
    add,
    update
};


// Using enumerations
void doSomething(Operation operation) {
    if (operation == Operation::add) {
        // ...
    }
    else if (operation == Operation::list) {
        // ...
    }
}
```


Streams

Terms

Binary files
Buffer
File streams
Input streams
Output streams
Streams
String streams
Text files

Summary

- A *stream* is an abstraction for a data source or destination. Using streams, we can read data or write it to a variety of places (eg terminal, files, network, etc) in the same way.
- In the C++ STL, we have many stream classes for different purposes. All these classes inherit their functionality from **ios_base**.
- A *buffer* is a temporary storage in memory used for reading or writing data to streams.
- If an error occurs while reading data from a stream, the invalid data stays in the buffer and will be used for subsequent reads. In such situations, first we have to put the stream into a clean state using the **clear()** method. Then, we should clear the data in the buffer using the **ignore()** method.
- In the C++ STL, we have three stream classes for working with files. (**ifstream** for reading from files, **ofstream** for writing to files, and **fstream** for reading and writing to files).
- *Binary files* store data the same way it is stored in memory. They are more efficient for storing large amount of numeric data but they're not human readable.
- Using *string streams* we can convert data to a string or vice versa.


```cpp
// Writing to a stream
cout << "Hello World";

// Reading from a stream
int number;
cin >> number;

// Handling read errors
if (cin.fail()) {
    cout << "Error";
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

// Writing to a text file
ofstream file;
file.open("file.txt", ios::app);
if (file.is_open()) {
    file << "Hello World";
    file.close();
}
```

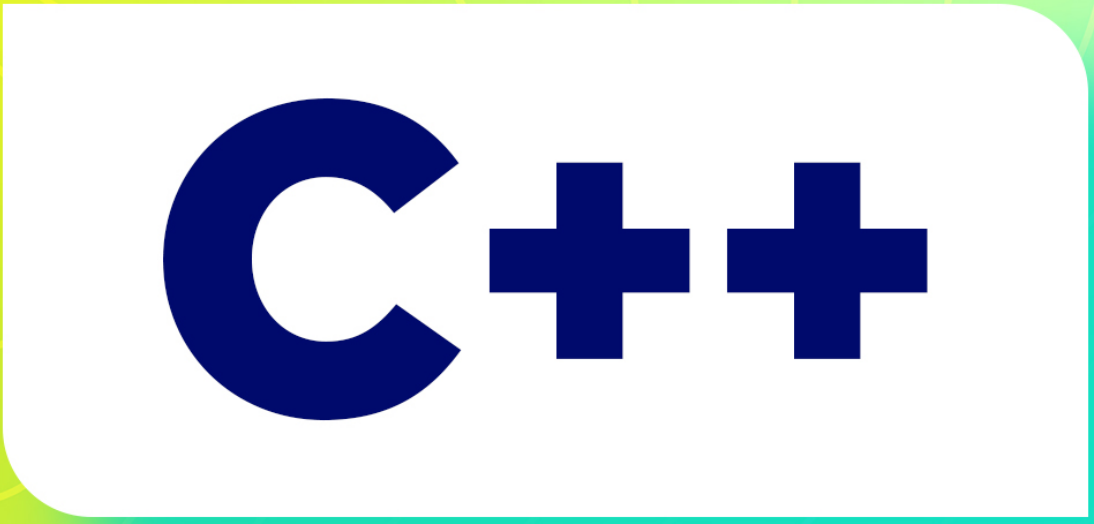


```cpp
// Reading from a text file
ifstream file;
file.open("file.txt");
if (file.is_open()) {
    string str;
    while(!file.eof()) {
        getline(file, str);
    }
    file.close();
}


// Writing to a binary file
int numbers[3] = { 1, 2, 3 };

ofstream file;
file.open("file.bin", ios::app | ios::binary);
if (file.is_open()) {
    file.write(
        reinterpret_cast<char*>(&numbers),
        sizeof(numbers));
    file.close();
}
```



```cpp
// Reading from a binary file
ifstream file;
file.open("file.bin", ios::binary);
if (file.is_open()) {
    int number;
    while (file.read(
        reinterpret_cast<char*>(&number),
        sizeof(number))) {
        cout << number << endl;
    }
    file.close();
}
```


The Ultimate



Part 3: Object-oriented Programming

Mosh Hamedani
codewithmosh.com

Table of Content

Classes

Terms

Access modifiers
Classes
Constructors
Copy constructor
Data hiding
Default constructor
Destructors
Getters and setters
Encapsulation
Header (interface) file
Immutable objects
Implementation file
Member initializer list
Object-oriented programming
Objects
Programming paradigms
Static members
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 getObjectsCount();
private:
    static int objectsCount;
};


int main() {
    cout << Rectangle::getObjectsCount();
    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;
};
```



```cpp
// 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 the 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.


```cpp
// 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} {
}
```

