



C++ Tutorial 2

Instructor: Beom Heyn Kim

beomheyunkim@hanyang.ac.kr

Department of Computer Science



C++ Functions

Syntax

```
void myFunction() {           // declaration
    // code to be executed    // the body of the function (definition)
}
```

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

In the following example, myFunction() is used to print a text (the action), when it is called

```
// Create a function
void myFunction() {
    cout << "I just got executed!";
}

int main() {
    myFunction(); // call the function
    return 0;
}

// Outputs "I just got executed!"
```

Note: If a user-defined function, such as myFunction() is declared after the main() function, **an error will occur:**

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

void myFunction() {
    cout << "I just got executed!";
}

// Error
```

However, it is possible to separate the declaration and the definition of the function

```
// Function declaration
void myFunction();

// The main method
int main() {
    myFunction(); // call the function
    return 0;
}

// Function definition
void myFunction() {
    cout << "I just got executed!";
}
```



C++ Function Parameters

Syntax

```
void functionName(parameter1, parameter2, parameter3) {  
    // code to be executed  
}
```

// Information can be passed to functions as a parameter.
// Parameters act as variables inside the function.

Example

```
void myFunction(string fname) {  
    cout << fname << " Refsnes\n";  
}
```

```
int main() {  
    myFunction("Liam");  
    myFunction("Jenny");  
    myFunction("Anja");  
    return 0;  
}
```

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes

When a **parameter** is passed to the function,
it is called an **argument**.

You can also use a default parameter value, by using the equals sign (=).

If we call the function without an argument, it uses the default value ("Norway"):

```
void myFunction(string country = "Norway") {  
    cout << country << "\n";  
}
```

```
int main() {  
    myFunction("Sweden");  
    myFunction("India");  
    myFunction();  
    myFunction("USA");  
    return 0;  
}
```

// Sweden
// India
// Norway
// USA



C++ Function Parameters

Multiple Parameters

Inside the function, you can add as many parameters as you want:

```
void myFunction(string fname, int age) {  
    cout << fname << " Refsnes. " << age << "  
    years old. \n";  
}  
  
int main() {  
    myFunction("Liam", 3);  
    myFunction("Jenny", 14);  
    myFunction("Anja", 30);  
    return 0;  
}  
  
// Liam Refsnes. 3 years old.  
// Jenny Refsnes. 14 years old.  
// Anja Refsnes. 30 years old.
```

Note that when you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function:


```
int myFunction(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int z = myFunction(5, 3);  
    cout << z;  
    return 0;  
}  
// Outputs 8 (5 + 3)
```



C++ Function Parameters

Pass By Reference

In the examples from the previous page, we used normal variables when we passed parameters to a function. You can also pass a **reference** to the function. This can be useful when you need to change the value of the arguments:



```
void swapNums(int &x, int &y) {  
    int z = x;  
    x = y;  
    y = z;  
}  
  
int main() {  
    int firstNum = 10;  
    int secondNum = 20;  
  
    cout << "Before swap: " << "\n";  
    cout << firstNum << secondNum << "\n";  
  
    // Call the function, which will change the values of firstNum and secondNum  
    swapNums(firstNum, secondNum);  
  
    cout << "After swap: " << "\n";  
    cout << firstNum << secondNum << "\n";  
  
    return 0;  
}
```

Can you implement this and run? Then, try to remove ‘&’ before parameters x and y of swapNums and run it. How is the result different?



C++ Function Parameters


Pass Arrays as Function Parameters

You can also pass [arrays](#) to a function:

```
void myFunction(int myNumbers[5]) {  
    for (int i = 0; i < 5; i++) {  
        cout << myNumbers[i] << "\n"; // Outputs?  
    }  
}  
  
int main() {  
    int myNumbers[5] = {10, 20, 30, 40, 50};  
    myFunction(myNumbers);  
    return 0;  
}
```

When you call the function, you only need to use the **name of the array** when passing it as an argument. However, the full declaration of the array is needed in the function parameter (i.e., **int myNumbers[5]**).

Try this



```
#include <iostream>  
using namespace std;  
  
void myFunction(int myNumbers[5]) {  
    for (int i = 0; i < 5; i++) {  
        cout << myNumbers[i] << "\n";  
        myNumbers[i] += 500;  
    }  
}  
  
int main() {  
    int myNumbers[5] = {10, 20, 30, 40, 50};  
    myFunction(myNumbers);  
    for (int i = 0; i < 5; i++) {  
        cout << myNumbers[i] << "\n"; // Outputs?  
    }  
    return 0;  
}
```



C++ Function Overloading

Function Overloading

With **function overloading**, multiple functions can have the same name with different parameters:

```
int myFunction(int x)
float myFunction(float x)
double myFunction(double x, double y)
```

```
int plusFunc(int x, int y) {
    return x + y;
}

double plusFunc(double x, double y) {
    return x + y;
}

int main() {
    int myNum1 = plusFunc(8, 5);
    double myNum2 = plusFunc(4.3, 6.26);
    cout << "Int: " << myNum1 << "\n";
    cout << "Double: " << myNum2;
    return 0;
}
```



C++ Classes and Objects

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the `class` keyword:

```
class MyClass {           // The class
public:                   // Access specifier
    int myNum;            // Attribute (int variable)
    string myString;      // Attribute (string variable)
};
```

- The class keyword is used to create a class called MyClass.
- The public keyword is an **access specifier**, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
- Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called **attributes**.
- At last, end the class definition with a semicolon ;.



C++ Classes and Objects

Create an Object

In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name.

To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

```
class MyClass {    // The class
public:           // Access specifier
    int myNum;    // Attribute (int variable)
    string myString; // Attribute (string variable)
};

int main() {
    MyClass myObj; // Create an object of MyClass

    // Access attributes and set values
    myObj.myNum = 15;
    myObj.myString = "Some text";

    // Print attribute values
    cout << myObj.myNum << "\n";
    cout << myObj.myString;
    return 0;
}
```



C++ Methods

Class Methods

Methods are **functions** that belongs to the class.

There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

In the following example, we define a function inside the class, and we name it "myMethod".

Note: You access methods just like you access attributes; by creating an object of the class and using the dot syntax (.):

Inside Example

```
class MyClass {           // The class
public:                  // Access specifier
    void myMethod() {    // Method/function defined inside the class
        cout << "Hello World!";
    }
};

int main() {
    MyClass myObj;        // Create an object of MyClass
    myObj.myMethod();     // Call the method
    return 0;
}
```



C++ Methods

To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifying the name of the class, followed the scope resolution `::` operator, followed by the name of the function:

Outside Example

```
class MyClass {           // The class
public:                   // Access specifier
    void myMethod();      // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
    cout << "Hello World!";
}

int main() {
    MyClass myObj;        // Create an object of MyClass
    myObj.myMethod();     // Call the method
    return 0;
}
```



C++ Constructors

Constructors

A constructor in C++ is a **special method** that is automatically called when an object of a class is created.

To create a constructor, use the same name as the class, followed by parentheses ():

Example

```
class MyClass {           // The class
public:                   // Access specifier
    MyClass() {           // Constructor
        cout << "Hello World!";
    }
};

int main() {
    MyClass myObj;        // Create an object of MyClass (this will call the constructor)
    return 0;
}
```

Note: The constructor has the same name as the class, it is always public, and it does not have any return value.



C++ Constructors

Constructor Parameters

Constructors can also take parameters (just like regular functions), which can be useful for setting initial values for attributes.

```
class Car {    // The class
public:        // Access specifier
    string brand; // Attribute
    string model; // Attribute
    int year;     // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
        brand = x;
        model = y;
        year = z;
    }
};

int main() {
    // Create Car objects and call the constructor with different values
    Car carObj1("BMW", "X5", 1999);
    Car carObj2("Ford", "Mustang", 1969);

    // Print values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```



C++ Constructors

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by specifying the name of the class, followed by the scope resolution `::` operator, followed by the name of the constructor (which is the same as the class):

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

class Car {    // The class
public:        // Access specifier
    string brand; // Attribute
    string model; // Attribute
    int year;     // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
    brand = x;
    model = y;
    year = z;
}

int main() {
    // Create Car objects and call the constructor with different values
    Car carObj1("BMW", "X5", 1999);
    Car carObj2("Ford", "Mustang", 1969);

    // Print values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```



C++ Access Specifiers

Access Specifiers

By now, you are quite familiar with the public keyword that appears in all of our class examples:

```
class MyClass { // The class
public:         // Access specifier
               // class members goes here
};
```

The `public` keyword is an **access specifier**. Access specifiers define how the members (attributes and methods) of a class can be accessed. In the example above, the members are public - which means that they can be accessed and modified from outside the code.



C++ Access Specifiers

In C++, there are three access specifiers:

- **public** - members are accessible from outside the class
- **private** - members cannot be accessed (or viewed) from outside the class
- **protected** - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

```
class MyClass {  
    public:           // Public access specifier  
    int x;           // Public attribute  
    private:         // Private access specifier  
    int y;           // Private attribute  
};  
  
int main() {  
    MyClass myObj;  
    myObj.x = 25;     // Allowed (public)  
    myObj.y = 50;     // Not allowed (private)  
    return 0;  
}
```

Output:

```
error: y is private
```

Note: It is possible to access private members of a class using a public method inside the same class. (Encapsulation)

Also, by default, all members of a class are private if you don't specify an access specifier:



C++ Encapsulation

Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods.

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

```
class Employee {
```

```
private:
```

```
// Private attribute
```

```
int salary;
```

```
public:
```

```
// Setter
```

```
void setSalary(int s) {
```

```
    salary = s;
```

```
}
```

```
// Getter
```

```
int getSalary() {
```

```
    return salary;
```

```
}
```

```
};
```

```
int main() {
```

```
    Employee myObj;
```

```
    myObj.setSalary(50000);
```

```
    cout << myObj.getSalary();
```

```
    return 0;
```

```
}
```

The salary attribute is private, which have restricted access.

The public setSalary() method takes a parameter (s) and assigns it to the salary attribute (salary = s).

The public getSalary() method returns the value of the private salary attribute.



C++ Inheritance

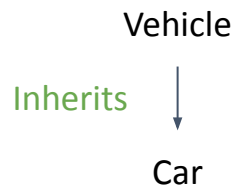
Inheritance

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **derived class** (child) - the class that inherits from another class
- **base class** (parent) - the class being inherited from

To inherit from a class, use the `:` symbol.

In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):



```
// Base class
class Vehicle {
public:
    string brand = "Ford";
    void honk() {
        cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
public:
    string model = "Mustang";
};

int main() {
    Car myCar;
    myCar.honk();
    cout << myCar.brand + " " + myCar.model;
    return 0;
}
```

myCar is Car class's instance, and it inherits brand from Vehicle Class

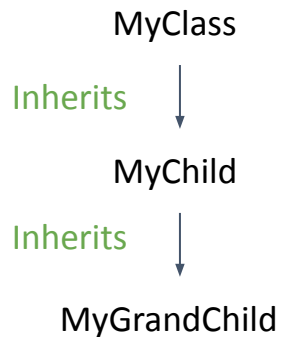


C++ Inheritance

Multilevel Inheritance

A class can also be derived from one class, which is already derived from another class.

In the following example, MyGrandChild is derived from class MyChild (which is derived from MyClass).



```
// Base class (parent)
class MyClass {
public:
    void myFunction() {
        cout << "Some content in parent class." ;
    }
};

// Derived class (child)
class MyChild: public MyClass {
};

// Derived class (grandchild)
class MyGrandChild: public MyChild {
};

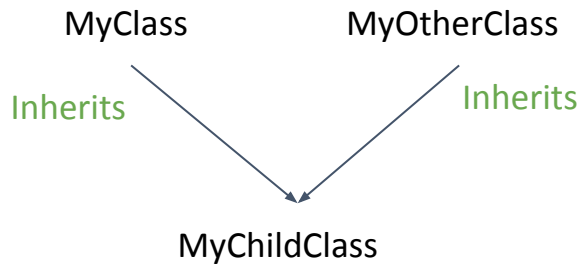
int main() {
    MyGrandChild myObj;
    myObj.myFunction();
    return 0;
}
```



C++ Inheritance

Multiple Inheritance

A class can also be derived from more than one base class, using a **comma-separated list**:



```
// Base class
class MyClass {
public:
    void myFunction() {
        cout << "Some content in parent class." ;
    }
};

// Another base class
class MyOtherClass {
public:
    void myOtherFunction() {
        cout << "Some content in another class." ;
    }
};

// Derived class
class MyChildClass: public MyClass, public MyOtherClass {
};

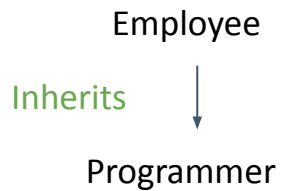
int main() {
    MyChildClass myObj;
    myObj.myFunction();
    myObj.myOtherFunction();
    return 0;
}
```



C++ Inheritance

Access Specifiers

The access specifier, `protected`, is similar to private, but it can also be accessed in the **inherited** class:



```
// Base class
class Employee {
    protected: // Protected access specifier
    int salary;
};

// Derived class
class Programmer: public Employee {
    public:
    int bonus;
    void setSalary(int s) {
        salary = s;
    }
    int getSalary() {
        return salary;
    }
};

int main() {
    Programmer myObj;
    myObj.setSalary(50000);
    myObj.bonus = 15000;
    cout << "Salary: " << myObj.getSalary() << "\n";
    cout << "Bonus: " << myObj.bonus << "\n";
    return 0;
}
```



C++ Polymorphism

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter;

Inheritance lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

```
// Base class
class Animal {
public:
    void animalSound() {
        cout << "The animal makes a sound \n";
    }
};

// Derived class
class Pig : public Animal {
public:
    void animalSound() {
        cout << "The pig says: wee wee \n";
    }
};

// Derived class
class Dog : public Animal {
public:
    void animalSound() {
        cout << "The dog says: bow wow \n";
    }
};
```



C++ Polymorphism

Try this



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

// Base class
class Animal {
public:
    void animalSound() {
        cout << "The animal makes a sound \n";
    }
};

// Derived class
class Pig : public Animal {
public:
    void animalSound() {
        cout << "The pig says: wee wee \n";
    }
};

// Derived class
class Dog : public Animal {
public:
    void animalSound() {
        cout << "The dog says: bow wow \n";
    }
};

int main() {
    Animal myAnimal;
    Pig myPig;
    Dog myDog;

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
    return 0;
}
```

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.



C++ Files

C++ Files

The `fstream` library allows us to work with files.

To use the `fstream` library, include both the standard `<iostream>` **AND** the `<fstream>` header file:

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

There are three classes included in the `fstream` library, which are used to create, write or read files:

Class	Description
<code>ofstream</code>	Creates and writes to files
<code>ifstream</code>	Reads from files
<code>fstream</code>	A combination of <code>ofstream</code> and <code>ifstream</code> : creates, reads, and writes to files



C++ Files

Create and Write To a File

To create a file, use either the `ofstream` or `fstream` class, and specify the name of the file.

To write to the file, use the insertion operator (`<<`).

```
// Create a text string, which is used to output the text file
string myText;

// Read from the text file
ifstream MyReadFile("filename.txt");

// Use a while loop together with the getline() function to read the file line by line
while (getline (MyReadFile, myText)) {
    // Output the text from the file
    cout << myText;
}

// Close the file
MyReadFile.close();
```



C++ Files

Read a File

To read from a file, use either the ifstream or fstream class, and the name of the file.

Note that we also use a while loop together with the getline() function (which belongs to the ifstream class) to read the file line by line, and to print the content of the file:



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

int main () {
    // Create a text file
    ofstream MyWriteFile("filename.txt");

    // Write to the file
    MyWriteFile << "Files can be tricky, but it is fun enough!";

    // Close the file
    MyWriteFile.close();

    // Create a text string, which is used to output the text file
    string myText;

    // Read from the text file
    ifstream MyReadFile("filename.txt");

    // Use a while loop together with the getline() function to read the file line by line
    while (getline (MyReadFile, myText)) {
        // Output the text from the file
        cout << myText;
    }

    // Close the file
    MyReadFile.close();
}
```



C++ Exceptions

C++ Exceptions

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an **exception** (throw an error).

C++ try and catch

Exception handling in C++ consist of three keywords: **try**, **throw** and **catch**:

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **throw** keyword throws an exception when a problem is detected, which lets us create a custom error.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

```
try {  
    // Block of code to try  
    throw exception; // Throw an exception when a problem arise  
}  
catch () {  
    // Block of code to handle errors  
}
```



C++ Exceptions

Example

```
try {  
    int age = 15;  
    if (age >= 18) {  
        cout << "Access granted - you are old enough.";  
    } else {  
        throw (age);  
    }  
}  
catch (int myNum) {  
    cout << "Access denied - You must be at least 18 years old.\n";  
    cout << "Age is: " << myNum;  
}
```

A red arrow originates from the `(age)` parameter in the `throw (age);` statement within the `try` block. It points downwards and then to the left, ending at the `(int myNum)` parameter in the `catch (int myNum)` block, illustrating the transfer of the exception object.

The catch statement takes a **parameter**: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.



C++ Exceptions

Handle Any Type of Exceptions (...)

If you do not know the throw **type** used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

```
try {  
    int age = 15;  
    if (age >= 18) {  
        cout << "Access granted - you are old enough.";  
    } else {  
        throw 505;  
    }  
}  
catch (...) {  
    cout << "Access denied - You must be at least 18 years old.\n";  
}
```



Assignment

Take the quiz on C++ at W3School:

<https://www.w3schools.com/quiztest/quiztest.asp?qtest=CPP>



The End