



C++ for C Coders

Youngsup Kim
idebtor@gmail.com



Introduction

From C to C++

Objectives

In this chapter, you will learn:

- C vs. C++
- input and output
- namespace
- memory allocation
- reference
- default parameter

Objectives

In this chapter, you will learn:

- overloading
- inline
- const
- in-house coding principles
 - NMN
 - DRY
 - NSE
 - IIS - Interface and Implementation Separation

Declare variable in C and C++

```
//Declare variable in C
#include <stdio.h>

int main(int argc, char const * argv[]){
    int i;
    char c;
    double d;
    float f;

    return 0;
}
```

Colored by Color Scripter



```
//Declare variable in C++
#include <iostream>

int main(int argc, char const * argv[]){
    int i;
    char c;
    double d;
    float f;

    return 0;
}
```

Colored by Color Scripter



For in C++

```
//for in C++
#include <iostream>

int main(int argc, char const * argv[]){
    for(int i = 0; i < 10; i++){
        std::cout << i << std::endl;
    }

    return 0;
}
```

Colored by Color Scripter

For in C++

```
//for in C++  
#include <iostream>  
  
int main(int argc, char const * argv[]){  
    for(int i = 0; i < 10; i++){  
        std::cout << i << std::endl;  
    }  
  
    return 0;  
}
```

Colored by Color Scripter




if-else in C++

```
//if-else in C++
#include <iostream>

int main(int argc, char *argv[]){
    int i;
    std::cout << "Input a number! : " << std::endl;
    std::cin >> i;

    if(i == 7){
        std::cout << "You entered lucky 7!" << std::endl;
    }
    else{
        std::cout << "Your input is " << i << "!" << std::endl;
    }

    return 0;
}
```



Colored by Color Scripter

Hello world in C and C++

// Simple C program to display "Hello World"

//Header file for input output functions

`#include<stdio.h>`

//main function where the execution of program begins

`int main()`

{

// prints hello world

`printf("Hello World\n");` 

`return 0;`

}

// Simple C++ program to display "Hello World"

// Header file for input output functions

`#include<iostream>`

// main function where the execution of program begins

`int main()`

{

// prints hello world

`std::cout << "Hello World" << std::endl;` 

`return 0;`

}

Colored by Color Scripter

Hello world in C++

```
// Simple C++ program to display "Hello World"

// Header file for input output functions
#include<iostream>

using namespace std;

// main function where the execution of program begins
int main()
{
    // prints hello world
    cout << "Hello World" << endl;

    return 0;
}
```

Colored by Color Scripter

Hello world in C++

```
// Simple C++ program to display "Hello World"

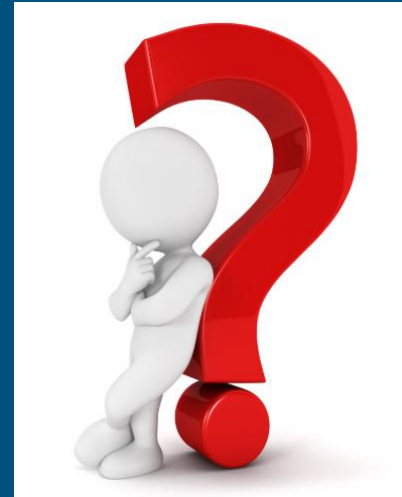
// Header file for input output functions
#include<iostream>

using namespace std;

// main function where the execution of program begins
int main()
{
    // prints hello world
    cout << "Hello World" << endl;

    return 0;
}
```

Colored by Color Scripter



Namespaces

- Global identifiers in a header file used in a program become global in the program
 - Syntax error occurs if an identifier in a program has the same name as a global identifier in the header file
- Same problem can occur with third-party libraries
 - Common solution: third-party vendors begin their global identifiers with `_` (underscore)
 - Do not begin identifiers in your program with `_`

Namespaces

- ANSI/ISO Standard C++ attempts to solve this problem with the namespace mechanism
- Syntax:

```
namespace namespace_name  
{  
    members  
}
```

where a member is usually a variable declaration, a named constant, a function, or another namespace

Namespaces

briefly, it's a fence!



Namespaces

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

namespace first { int var = 5; }
namespace second { int var = 3; }

int main (int argc, char **argv) {
    std::cout << first::var << std::endl << second::var << std::endl;
    return 0;
}
```

Output:



5
3

Namespaces

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

namespace first { int var = 5; }
namespace second { int var = 3; }

int main (int argc, char **argv) {
    std::cout << first::var << std::endl << second::var << std::endl;
    return 0;
}
```



Namespaces

In C++, **scope resolution operator ::** is used for following purposes.

- 1) To access a global variable when there is a local variable with same name.
- 2) To define a function outside a class.
- 3) To access a class's static variables.
- 4) In case of multiple Inheritance.

Input/Output

- `cin` is used with `>>` to gather input

`cin >> variable ;`

- The stream extraction operator is `>>`
- For example, if `miles` is a double variable

```
cin >> miles;
```

- Causes computer to get a value of type `double`
- Places it in the variable `miles`

Input/Output

- Using more than one variable in `cin` allows more than one value to be read at a time
- For example, if `feet` and `inches` are variables of type `int`, a statement such as:

```
cin >> feet >> inches;
```

- Inputs two integers from the keyboard
- Places them in variables `feet` and `inches` respectively

Input/Output

```
#include <iostream>
using namespace std;
int main()
{
    int feet;
    int inches;
    cout << "Enter two integers separated by spaces: ";
    cin >> feet >> inches;
    cout << endl;
    cout << "Feet: " << feet << endl;
    cout << "Inches: " << inches << endl;
    return 0;
}
```

Colored by Color Scripter

Input/Output

```
#include <iostream>
using namespace std;
int main()
{
    int feet;
    int inches;
    cout << "Enter two integers separated by spaces: ";
    cin >> feet >> inches;
    cout << endl;
    cout << "Feet: " << feet << endl;
    cout << "Inches: " << inches << endl;
    return 0;
}
```

Colored by Color Scripter

CS

Output:

```
Enter two integers separated by spaces: 23 7

Feet: 23
Inches: 7
```

Avoid using `cin >>` if you can

Using ``cin`` to get user input is convenient sometimes since we can specify a primitive data type. However, it is **notorious** at causing input issues because it doesn't remove the newline character from the stream or do type-checking. So anyone using ``cin >> var;`` and following it up with another ``cin >> stringtype;`` or ``std::getline();`` will receive empty inputs. It's the best practice not to mix the different types of input methods from ``cin``.

Another disadvantage of using ``cin >> stringvar;`` is that ``cin`` has no checks for length, and it will break on a space. So you enter something that is more than one word, only the first word is going to be loaded. Leaving the space, and following word still in the input stream.

JoyNode: A more elegant solution, much easier to use, is the ``std::getline(``.

Refer to [this note](<https://github.com/idebtor/nowic/blob/master/02GettingInput.md>) for detail.

Input/Output

- The syntax of cout and << is:

```
cout << expression or manipulator << . . . ;
```

- Called an output statement
- The stream insertion operator is <<
- Expression evaluated and its value is printed at the current cursor position on the screen

Input/Output

- A manipulator is used to format the output
 - Example: `endl` causes insertion point to move to beginning of next line

Statement	Output
1 <code>cout << 29 / 4 << endl;</code>	7
2 <code>cout << "Hello there." << endl;</code>	Hello there.
3 <code>cout << 12 << endl;</code>	12
4 <code>cout << "4 + 7" << endl;</code>	4 + 7
5 <code>cout << 4 + 7 << endl;</code>	11
6 <code>cout << 'A' << endl;</code>	A
7 <code>cout << "4 + 7 = " << 4 + 7 << endl;</code>	4 + 7 = 11
8 <code>cout << 2 + 3 * 5 << endl;</code>	17
9 <code>cout << "Hello \nthere." << endl;</code>	Hello there.

Input/Output

- The new line character is '\n'
 - May appear anywhere in the string

```
cout << "Hello there.";
cout << "My name is James.";
```

- Output:

```
Hello there.My name is James.
```

```
cout << "Hello there.\n";
cout << "My name is James.";
```

- Output:

```
Hello there.
```

```
My name is James.
```

Input/Output

TABLE 2-4 Commonly Used Escape Sequences

	Escape Sequence	Description
<code>\n</code>	Newline	Cursor moves to the beginning of the next line
<code>\t</code>	Tab	Cursor moves to the next tab stop
<code>\b</code>	Backspace	Cursor moves one space to the left
<code>\r</code>	Return	Cursor moves to the beginning of the current line (not the next line)
<code>\\</code>	Backslash	Backslash is printed
<code>\'</code>	Single quotation	Single quotation mark is printed
<code>\"</code>	Double quotation	Double quotation mark is printed

Input/Output

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

int main()
{
    int i;
    cout << "Please enter an integer value: ";
    cin >> i;
    cout << "The value you enter is " << i << endl;
    return 0;
}
```

Colored by Color Scripter

Input/Output

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

int main()
{
    int i;
    cout << "Please enter an integer value: ";
    cin >> i;
    cout << "The value you enter is " << i << endl;
    return 0;
}
```

Colored by Color Scripter

CS

Output:

```
Please enter an integer value: 11
The value you entered is 11
```

New/Delete

Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack**

The new and delete keywords are used to allocate and free memory. They are "object-aware" so you'd better use them instead of malloc and free.

New/Delete

The **new** operator denotes a request for memory allocation on the Heap.

If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided **delete** operator by C++ language.

delete does two things: it calls the destructor and it deallocates the memory.

New/Delete

```
// Combine declaration of pointer and their assignment
int *p = new int;           // one int memory allocated
int *q = new int(7);        // initialized with 7

delete p;                   // deallocate memory
delete q;

int *r = new int[10];       // allocated for array of 10 integers
int *s = (int *)malloc(sizeof(int) * 10); // using malloc()

delete[] r;                 // deallocate array memory
free(s);                   // deallocate 'malloc'ed memory
```

References

A reference allows to declare an ***alias*** to another variable.

As long as the aliased variable lives, you can use indifferently the variable or the alias.

References

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

int main()
{
    int x;    x 와 foo가 똑같은 메모리를 가르킴
    int& foo = x;

    foo = 42;
    cout << x << endl;

    return 0;
}
```

References

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

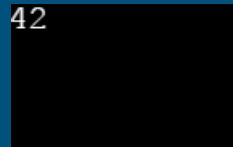
int main()
{
    int x;
    int& foo = x;

    foo = 42;
    cout << x << endl;

    return 0;
}
```

CS

Output: 42



← What can be done more?

References

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

int main()
{
    int x;
    int& foo = x;

    foo = 42;
    cout << x << endl;

    return 0;
}
```

CS

Output: 42

- foo acts as alias of x.
- x must be a lvalue.
- The code below is invalid, why?
`int& joe = 10;`
(interpret the error message.)

lvalue = 왼쪽에 올 수 있는 값. 실제로 메모리값을 사용하는 값
rvalue = 오른쪽에 올 수 있는 값. 임시로 사용하는 값

address가 있는지, 없는지 확인하면 됨

References

References are *extremely useful* **when used with function arguments** since it saves the cost of copying parameters into the stack when calling the function.

References

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

void swap(int& x, int& y){ //swap(int& x, int& y)
    int temp;
    temp = x;    //save the value at address x
    x = y;       // put y into x
    y = temp;    // x and y are really swapped.
    return;
} 함수에서 받을 때 reference로 받기 때문에 메모리 공유. 주소
  값을 전달 할 필요가 없음. 왕왕편하다.

int main()
{
    int a = 1, b = 2;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

Colored by Color Scripter

CS

Output:

References

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

void swap(int& x, int& y){ //swap(int& x, int& y)
    int temp;
    temp = x;    //save the value at address x
    x = y;       // put y into x
    y = temp;    // x and y are really swapped.
    return;
}

int main()
{
    int a = 1, b = 2;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

Colored by Color Scripter



Output:

a = 2, b = 1

Default Parameters

You can specify default values for function parameters. When the function is called with fewer parameters, default values are used. This is very useful, but not flexible as Python because of the following reasons.

- When an argument is left out of a function call (because it has default value), all the arguments that come after it must be left out too.
- When a function has a mixture of parameters both with and without default arguments, the parameters with default arguments must be declared last.
- A function's default arguments should be assigned in the earliest occurrence of the function name. This will usually be the function prototype.

Default Parameters

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

float foo(float a = 0, float b = 1, float c = 2);

int main()
{
    cout << foo(1) << endl;
    cout << foo(1, 2) << endl;
    cout << foo(1, 2, 3) << endl;

    return 0;
}

float foo(float a, float b, float c)
{return a + b + c;}
```

Output:

Default Parameters

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

float foo(float a = 0, float b = 1, float c = 2);

int main()
{
    cout << foo(1) << endl;
    cout << foo(1, 2) << endl;
    cout << foo(1, 2, 3) << endl;

    return 0;
}

float foo(float a, float b, float c)
{return a + b + c;}
```

Output:

4
5
6

Overloading

- In a C++ program, several functions can have the same name
 - This is called function overloading

Overloading

Function overloading refers to the possibility of creating **multiple functions with the same name** as long as they have different parameters (type and/or number) which is called a *signature* of function.

파라미터

Overloading

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

void print(int i){
    cout << " Here is int " << i << endl;
}
void print(float f){
    cout << " Here is float" << f << endl;
}
void print(char const *c){
    cout << " Here is char* " << c << endl;
}
} 이름이 같은 함수를 사용해도 파라미터 타입에 맞는 함수를
알아서 사용함 >> function overloading
int main()
{
    print(10);
    print(10.10);
    print("ten");

    return 0;
}
```

Colored by Color Scripter

Output:

Overloading

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

void print(int i){
    cout << " Here is int " << i << endl;
}
void print(float f){
    cout << " Here is float " << f << endl;
}
void print(char const *c){
    cout << " Here is char* " << c << endl;
}

int main()
{
    print(10);
    print(10.10);
    print("ten");

    return 0;
}
```

Colored by Color Scripter

CS

Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

inline

```
#define SQUARE(X) X*X
#include <iostream>
using namespace std;

int main(){
    int result = SQUARE(2+3);

    cout << "result: " << result << endl;

    return 0;
}
```

Colored by Color Scripter

CS



what is the value of result?

inline

```
#define SQUARE(X) ((X)*(X))
#include <iostream>
using namespace std;

int main(){
    int result = SQUARE(2+3);

    cout << "result: " << result << endl;

    return 0;
}
```

Colored by Color Scripter

CS

It's not 25, but 11.

Why?

preprocessor changes
int result = SQUARE(2+3); to
int result = 2+3*2+3;

#define SQUARE(X) ((X)*(X))
The above is the right
expression.

inline

```
#define SQUARE(X) X*X
#include <iostream>
using namespace std;

int main(){
    int result = SQUARE(2+3);

    cout << "result: " << result << endl;

    return 0;
}
```

Colored by Color Scripter

CS

In C, defines and macros are efficient.

However, coder should be aware when using it.

Also, it is not essentially a function, so it can cause several logical errors.

Plus, it cannot format parameters.

inline

In C++, you don't have to worry.

For macros, prefer the `inline` notation:

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

int inline square(int x){
    return x*x;
}

int main(){
    int result = square(2+3);

    cout << "result: " << result << endl;

    return 0;
}
```

Colored by Color Scripter

const

For constants, prefer the `const` notation:

```
const int two = 2;
```

‘two’ becomes **read only**, so its value cannot be modified.

const

Why constants?



As the safety device of pistol prevents the trigger from being pulled when it is not to be shot; **const** prevents the value from being changed (overwritten).

const - pointer to const

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

int main(){
    int value = 5;
    int *ptr = &value;
    *ptr = 6; // change value to 6

    cout << value << endl;

    return 0;
}
```

CS

Output:

const - pointer to const

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

int main(){
    int value = 5;
    int *ptr = &value;
    *ptr = 6; // change value to 6

    cout << value << endl;

    return 0;
}
```

However, what happens if value is const?

Output:

6

const - pointer to const

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

int main(){
    const int value = 5; // value is const
    int *ptr = &value;
    *ptr = 6; // change value to 6

    cout << value << endl;

    return 0;
}
```

Colored by Color Scripter



Output:

const - pointer to const

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

int main(){
    const int value = 5; // value is const
    int *ptr = &value;
    *ptr = 6; // change value to 6

    cout << value << endl;

    return 0;
}
```

Colored by Color Scripter



The error is checked during compilation time, not run-time.

Where is an error? At (1) or (2)?

Output:

const - pointer to const

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

int main(){
    const int value = 5; // value is const
    int *ptr = &value;    ← (1)
    *ptr = 6; // change value to 6 ← (2)

    cout << value << endl;

    return 0;
}
```

Colored by Color Scripter



The error is checked during compilation time, not run-time.

Where is an error? At (1) or (2)?

Output:

```
main.cpp: In function 'int main()':
main.cpp:5:14: error: invalid conversion from 'const int*' to 'int*' [-fpermissive]
    int *ptr = &value;
               ^
```


const - pointer to const

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

int main(){
    const int value = 5; // value is const
    const int *ptr = &value;           ← (1)
    *ptr = 6; // change value to 6     ← (2)

    cout << value << endl;

    return 0;
}
```

Colored by Color Scripter



The error is checked during compilation time, not run-time.

Where is an error? At(1) or (2)?

Output:

const - pointer to const

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

int main(){
    const int value = 5; // value is const
    const int *ptr = &value;
    *ptr = 6; // change value to 6

    cout << value << endl;

    return 0;
}
```

Colored by Color Scripter



The error is checked during compilation time, not run-time.

Where is an error? At (1) or (2)?

a pointer to a const int

Output: t2.cpp:9:10: error: assignment of read-only location '* ptr'

```
*ptr = 6; // change value to 6
      ^
```

const - pointer to const

A **pointer to a const value** is a (non-const) pointer that points to a constant value.

To declare a pointer to a const value, use the **const** keyword before the data type

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

int main(){
    const int value = 5;
    const int *ptr = &value;    // this is okay, ptr is a non-const pointer that is pointing to a "const int"

    cout << "value: " << value << endl;
    cout << "*ptr: " << *ptr << endl;

    return 0;
}
```

Colored by Color Scripter

const - const pointer

You can use the `const` to define a constant pointer. For example,

```
int value = 5;  
int *const ptr = &value;
```

CS

Notice in the definition of `ptr` the word `const` appears after the asterisk.

This means that `ptr` is a `const` pointer.

const - const pointer

A compiler error will result if we write code that makes `ptr` point to anything else.

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

int main(){

    int value1 = 5;
    int value2 = 6;

    int * const ptr = &value1; // okay, the const pointer is initialized to the address of value1
    ptr = &value2;

    return 0;
}
```

const - const pointer

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

int main(){

    int value1 = 5;
    int value2 = 6;

    int * const ptr = &value1; // okay, the const pointer is initialized to the address of value1
    ptr = &value2;

    return 0;
}
```

Colored by Color Scripter

Output:

```
main.cpp: In function 'int main()':
main.cpp:8:6: error: assignment of read-only variable 'ptr'
    ptr = &value2;
    ^
```

const - const pointer

Although the parameter is `const` pointer, we can call the function multiple times with different arguments.

```
// The ptr itself cannot be changed,  
// but the item the ptr points to can be changed.  
#include <iostream>  
using namespace std;  
  
void set_to_zero(int *const ptr) { *ptr = 0; }  
  
int main() {  
    int x, y, z;  
  
    set_to_zero(&x);  
    set_to_zero(&y);  
    set_to_zero(&z);  
  
    cout << "x: " << x << endl;  
    cout << "y: " << y << endl;  
    cout << "z: " << z << endl;  
  
    return 0;  
}
```

Colored by Color Scripter

const - const pointer

```
// The ptr itself cannot be changed,  
// but the item the ptr points to can be changed.
```

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

```
void set_to_zero(int *const ptr) { *ptr = 0; }
```

```
int main() {  
    int x, y, z;
```

```
    set_to_zero(&x);  
    set_to_zero(&y);  
    set_to_zero(&z);
```

```
    cout << "x: " << x << endl;  
    cout << "y: " << y << endl;  
    cout << "z: " << z << endl;
```

```
    return 0;
```

```
}
```

Colored by Color Scriptor

Output:

```
x: 0
```

```
y: 0
```

```
z: 0
```


const - const pointer to const

You can also have constant pointers to constants. For example,

```
int value = 7;  
const int *const ptr = &value;
```

In this example, `ptr` is a `const int`.

Notice the word `const` appears before `int`, indicating that `ptr` points to a `const int`, and it appears after asterisk, indicating that `ptr` is a constant pointer.

```
int * ptr;
```

ptr is a **pointer** to **int**

```
const int * const ptr;
```

ptr is a **constant pointer** to **const int**

```
const int * ptr;
```

ptr is a **pointer** to **int constant** (i.e. **const int**)

```
int const * ptr;
```

ptr is a **pointer** to **const int**

```
int * const ptr;
```

ptr is a **const pointer** to **int**

const



Confused?

Below is the good description!

<https://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const>

Interface and Implementation Separation(IIS)

```
#pragma once           MyString.h

class CMyString
{
public:
    CMyString();
    ~CMyString();

private:
    // 문자열을 저장하는 동적메모리 포인터
    char* m_data;

    // 저장된 문자열의 길이
    int m_len;

public:
    int SetString(const char* p_data);
    const char* GetString();
    void Release();
};
```

Colored by Color Scripter

CS

We can manage the project efficiently by separating interface and implementation.

← Interface

Interface and Implementation Separation(IIS)

```
#include "MyString.h"
```

MyString.cpp

```
CMyString::CMyString():  
m_data(NULL), m_len(0)  
{  
}
```

```
CMyString::~CMyString()  
{  
    // 객체가 소멸하기 전에 메모리를 해제한다.  
    Release();  
}
```

```
int CMyString::SetString(const char* p_data)  
{  
    // 새로운 문자열 할당에 앞서 기존 정보를 해제한다.  
    Release();  
}
```

Colored by

```
// NULL을 인수로 함수를 호출했다는 것은 메모리를 해제하고  
// NULL로 초기화하는 것으로 볼 수 있다.
```

```
if (p_data == NULL) return 0;
```

```
// 길이가 0인 문자열도 초기화로 인식하고 처리한다.
```

```
int nLength = strlen(p_data);
```

```
if (nLength == 0) return 0;
```

```
// 문자열의 끝인 NULL 문자를 고려해 메모리를 할당한다.
```

```
m_data = new char[nLength + 1];
```

```
// 새로 할당한 메모리에 문자열을 저장한다.
```

```
strcpy_s(m_data, sizeof(char)* (nLength + 1), p_data);
```

```
m_len = nLength;
```

```
// 문자열의 길이를 반환한다.
```

```
return nLength;
```

```
}
```

Colored by Color Scripter

Interface and Implementation Separation(IIS)

```
const char* CMyString::GetString()
{
    return m_data;
}
void CMyString::Release()
{
    // 이 함수가 여러번 호출될 경우를 고려해 주요 멤버를 초기화한다.
    if (m_data != NULL)
        delete[] m_data;
    m_data = NULL;
    m_len = 0;
}
```

MyString.cpp

Colored by Color Scripter

CS

← Implementation

Interface and Implementation Separation(IIS)

```
#include "MyString.h"
```

MyStringDriver.cpp

```
int main(int argc, char *argv[]) {  
{  
    CMyString strData;  
    strData.SetString("Hello");  
    cout << strData.GetString() << endl;  
  
    return 0;  
}
```

Colored by Color Scripter

CS

← Development

Interface and Implementation Separation(IIS)

```
#include "MyString.h"
```

MyStringDriver.cpp

```
int main(int argc, char *argv[]) {  
{  
    CMyString strData;  
    strData.SetString("Hello");  
    cout << strData.GetString() << endl;  
  
    return 0;  
}
```

Colored by Color Scripter

CS

Output: Hello

Mixing C and C++

When transitioning from C to C++, you are not likely to refresh your entire code base.

Instead, you will need to maintain a mix of C and C++ code, hopefully getting the two sets of code to work together.

One common situation is that you have a C++ library with C-style interfaces.

While it seems like you should be able to `#include` the headers and rely on the linker, you will find that your program fails to compile and link.

Mixing C and C++

The magic bullet here is ***extern "C"***.

C and C++ use different function name mangling techniques.

C++ allows function overloading, which means the linker needs to mangle the function name to indicate which specific prototype it needs to call.

Mixing C and C++

By declaring a function with `extern "C"`, it changes the linkage requirements so that the C++ compiler does not add the extra mangling information to the symbol.

```
extern "C" void foo(int bar);
```



Mixing C and C++

If you have a library that can be shared between C and C++, you will need to make the functions visible in the C namespace.

The easiest way to accomplish this is with the following pattern:

<pre>#ifdef __cplusplus extern "C" { #endif //C code goes here #ifdef __cplusplus } // extern "C" #endif</pre>	<pre></pre>
--	-------------

This pattern relies on the presence of the `__cplusplus` definition when using the C++ compiler. If you are using the C compiler, `extern "C"` is not used.

Make and makefile

Since most of our builds are simple enough, I don't think that it is worthy to learn first and use `make` and `makefile` in our class.

If you want to learn it now, consider learning `cmake`.

- [Is it worth?](#)
- [How is CMake used?](#)
- [Introduction to CMake](#)

Classes

Class

A class specifies the attributes and member functions that a particular type of object may have.

Think of a class as a blueprint that object may be created from.

The blueprint itself is not a house, but is a detailed description of a house.

Class

We could say we are building an instance of the house described by the blueprint.

If we so desire, we can build several identical house from the same blueprint.

Each house is a separate instance of the house described by the blueprint.

Each instance is called an **object**

Class

A class might be considered as an extended concept of a data structure: instead of holding only data, it can hold both data and functions.

An object is an instantiation of a class.

By default, all attributes and functions of a class are private.

If you want a public default behavior, you can use keyword `struct` instead of keyword `class` in the declaration.

Class

```
class Foo {  
    int attribute;  
    int function( void ) { };  
};
```

```
struct Bar {  
    int attribute;  
    int function( void ) { };  
};
```

```
Foo foo;  
foo.attribute = 1; // WRONG
```

```
Bar bar;  
bar.attribute = 1; // OK
```

Constructors

It is possible to specify zero, one or more constructors for the class.

```
#include <iostream>

class Foo {
public:
    Foo( void )
    { std::cout << "Foo constructor 1 called" << std::endl; }
    Foo( int value )
    { std::cout << "Foo constructor 2 called" << std::endl; }
};

int main( int argc, char **argv ) {
    Foo foo_1, foo_2(2);
    return 0;
}
```

Constructors

```
#include <iostream>

class Foo {
public:
    Foo( void )
    { std::cout << "Foo constructor 1 called" << std::endl; }
    Foo( int value )
    { std::cout << "Foo constructor 2 called" << std::endl; }
};

int main( int argc, char **argv ) {
    Foo foo_1, foo_2(2);
    return 0;
}
```

Output:

```
Foo constructor 1 called
Foo constructor 2 called
```

Destructor

There can be only one destructor per class.

It takes no argument and returns nothing.

```
#include <iostream>

class Foo {
public:
    ~Foo( void )
    { std::cout << "Foo destructor called" << std::endl; }
}

int main( int argc, char **argv ) {
    Foo foo();
    return 0;
}
```

Destructor

There can be only one destructor per class.

It takes no argument and returns nothing.

```
#include <iostream>

class Foo {
public:
    ~Foo( void )
    { std::cout << "Foo destructor called" << std::endl; }
}

int main( int argc, char **argv ) {
    Foo foo();
    return 0;
}
```

Output:

```
Foo destructor called
```

Destructor

There can be only one destructor per class.

It takes no argument and returns nothing.

```
#include <iostream>

class Foo {
public:
    ~Foo( void )
    { std::cout << "Foo destructor called" << std::endl; }
}

int main( int argc, char **argv ) {
    Foo foo();
    return 0;
}
```

Note that you generally never need to explicitly call a destructor.

Access control by access specifiers

You can have fine control over who is granted access to a class function or attribute by specifying an explicit access policy

- `public` : Anyone is granted access
- `protected`: Only derived classes are granted access
- `private`: No one but friends are granted access

Access control by access specifiers

- A class's `private` member variables are allowed only through the `public` member functions. The `private` member variables cannot be reference by name any place except within the definitions of the member functions of its class.
- All the items that follow the word `public` can be referenced by name anyplace. There are no restrictions on the use of public members.

JoyNote: A good programming practices require that all member variables be `private` and that typically most member functions be `public`.

Initialization list

Object's member should be initialized using initialization lists

```
class Foo {  
    int _value;  
public:  
    Foo(int value=0) : _value(value) { };  
};
```

It's cheaper, better and faster.

Operator overloading

```
class Foo {  
private:  
    int _value;  
  
public:  
    Foo( int value ) : _value(value) { };  
  
    Foo operator+ ( const Foo & other ) {  
        return Foo( _value+ other._value );  
    }  
  
    Foo operator* ( const Foo & other ); {  
        return Foo( _value * other._value );  
    }  
}
```

Friends

Friends are either functions or other classes that are granted privileged access to a class.

```
class Foo {
private:
    double _value;
public:
    friend std::ostream& operator<< ( std::ostream& output,
        Foo const & that ) {
        return output << that._value;
    }
};

int main( int argc, char **argv ) {
    Foo foo;
    std::cout << "Foo object: " << foo << std::endl;
    return 0
}
```

Friends

Friends are either functions or other classes that are granted privileged access to a class.

```
class Foo {
private:
    double _value;
public:
    friend std::ostream& operator<< ( std::ostream& output,
                                     Foo const & that ) {
        return output << that._value;
    }
};

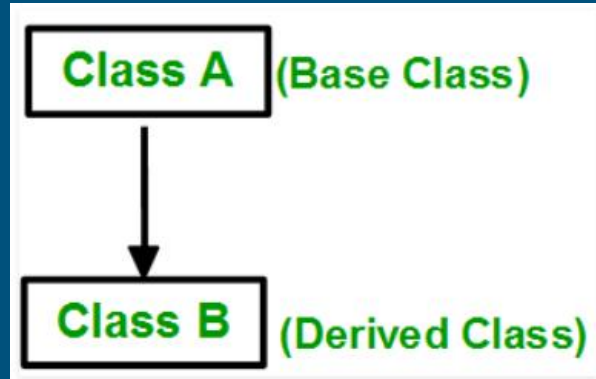
int main( int argc, char **argv ) {
    Foo foo;
    std::cout << "Foo object: " << foo << std::endl;
    return 0
}
```

Output:

```
Foo object: 0
```

Inheritance

Basics



The capability of a class to derive properties and characteristics from another class is called ***Inheritance***.

Inheritance is one of the most important feature of Object Oriented Programming.

Basics

Inheritance is done at the class definition level by specifying the base class and the type of inheritance.

```
class Foo                { /* ... */ };  
class Bar_public : public Foo    { /* ... */ };  
class Bar_private : private Foo  { /* ... */ };  
class Bar_protected : protected Foo { /* ... */ };
```


Basics

// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object

```
class Foo
{
public:
    int x;
protected:
    int y;
private:
    int z;
};
```

```
class Bar_public : public Foo
{
    // x is public
    // y is protected
    // z is not accessible from B
};
```

```
class Bar_protected : protected Foo
{
    // x is protected
    // y is protected
    // z is not accessible from C
};
```

```
class Bar_private : private Foo // 'private' is default for
classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Colored by Color Scripter

Virtual methods

A virtual function allows derived classes to replace the implementation provided by the base class

Non virtual methods are resolved statically (at compile time) while virtual methods are resolved dynamically (at run time).

Virtual methods

Guess what the output is!

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

class Parent{cout << "x: " << x << endl;
public:
    void printAge(){
        cout << age << endl;
    }
private:
    int age = 50;
};

class Child : public Parent{
public:
    void printAge(){
        cout << age << endl;
    }
private:
    int age = 20;
};

int main(){
    Child a;
    a.printAge();

    Parent &b = a;
    b.printAge();
    return 0;
}
```

Virtual methods

Output:

20
50

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

class Parent{cout << "x: " << x << endl;
public:
    void printAge(){
        cout << age << endl;
    }
private:
    int age = 50;
};

class Child : public Parent{
public:
    void printAge(){
        cout << age << endl;
    }
private:
    int age = 20;
};

int main(){
    Child a;
    a.printAge();

    Parent &b = a;
    b.printAge();
    return 0;
}
```

The diagram illustrates virtual method resolution. A red arrow points from the `a.printAge();` call in the `main` function to the `printAge()` method in the `Child` class. Another red arrow points from the `b.printAge();` call to the `printAge()` method in the `Parent` class. A third red arrow points from the `return 0;` statement to the `main` function's closing brace.

Virtual methods

Guess what the output is!

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

class Parent{cout << "x: " << x << endl;
public:
    virtual void printAge(){
        cout << age << endl;
    }
private:
    int age = 50;
};

class Child : public Parent{
public:
    void printAge(){
        cout << age << endl;
    }
private:
    int age = 20;
};

int main(){
    Child a;
    a.printAge();

    Parent &b = a;
    b.printAge();
    return 0;
}
```

Virtual methods

Output:

20
20

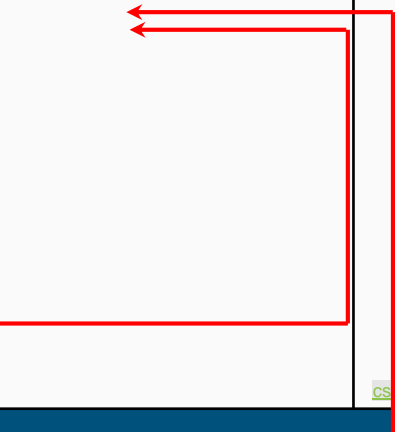
```
#include <iostream>
using namespace std;

class Parent{cout << "x: " << x << endl;
public:
    virtual void printAge(){
        cout << age << endl;
    }
private:
    int age = 50;
};

class Child : public Parent{
public:
    void printAge(){
        cout << age << endl;
    }
private:
    int age = 20;
};

int main(){
    Child a;
    a.printAge();

    Parent &b = a;
    b.printAge();
    return 0;
}
```



Virtual methods

Make sure your destructor is virtual when you have derived class.

Virtual methods

Make sure your destructor is **virtual** when you have derived class.



Because the destructors of the subclass are not called, ***severe memory leak errors can occur*** when a parent class references a child class.

Virtual methods

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

class Characters{
public:
    Characters(){p_char = new char[32];}
    ~Characters(){
        cout << "~Characters()" << endl;
        delete p_char;
    }
private:
    char *p_char;
};
```

Colored by Color S

```
class Integer : public Characters{
public:
    Integer(){p_int = new int;}
    ~Integer(){
        cout << "~Integer()" << endl;
        delete p_int;
    }
private:
    int *p_int;
};

int main(){
    Characters *ch = new Integer;
    delete ch;
    return 0;
}
```

Colored by Color Scripter

CS

Output:

Virtual methods

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

class Characters{
public:
    Characters(){p_char = new char[32];}
    ~Characters(){
        cout << "~Characters()" << endl;
        delete p_char;
    }
private:
    char *p_char;
};
```

Colored by Color S

```
class Integer : public Characters{
public:
    Integer(){p_int = new int;}
    ~Integer(){
        cout << "~Integer()" << endl;
        delete p_int;
    }
private:
    int *p_int;
};

int main(){
    Characters *ch = new Integer;
    delete ch;
    return 0;
}
```

It wasn't called.
Memory leak error occurs!

Colored by Color Scripter

CS

Output:

```
~Characters()
```

Virtual methods

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

class Characters{
public:
    Characters(){p_char = new char[32];}
    virtual ~Characters(){
        cout << "~Characters()" << endl;
        delete p_char;
    }
private:
    char *p_char;
};
```

Colored by Color Scrip

```
class Integer : public Characters{
public:
    Integer(){p_int = new int;}
    ~Integer(){
        cout << "~Integer()" << endl;
        delete p_int;
    }
private:
    int *p_int;
};

int main(){
    Characters *ch = new Integer;
    delete ch;
    return 0;
}
```

Colored by Color Scrip



Output:

Virtual methods

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

class Characters{
public:
    Characters(){p_char = new char[32];}
    virtual ~Characters(){
        cout << "~Characters()" << endl;
        delete p_char;
    }
private:
    char *p_char;
};
```

Colored by Color Scrip

```
class Integer : public Characters{
public:
    Integer(){p_int = new int;}
    ~Integer(){
        cout << "~Integer()" << endl;
        delete p_int;
    }
private:
    int *p_int;
};

int main(){
    Characters *ch = new Integer;
    delete ch;
    return 0;
}
```

Colored by Color Scrip

CS

Output:

```
~Integer()
~Characters()
```

Abstract Classes

You can define pure virtual method that prohibits the base object to be instantiated.

Derived classes need then to implement the virtual method.

Abstract Classes

Output:

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

class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Colored by Color Scripter



Abstract Classes

Output:

In Derived

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

class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Colored by Color Scripter



Abstract Classes

Pure virtual functions make a class abstract.

Output:

```
#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;
}
```


Abstract Classes

Pure virtual functions make a class abstract.

Output:

```
main.cpp: In function 'int main()':
main.cpp:15:10: error: cannot declare variable 't' to be of abstract type 'Test'
    Test t;
    ^
main.cpp:5:7: note: because the following virtual functions are pure within 'Test':
    class Test
    ^
main.cpp:9:18: note:     virtual void Test::show()
    virtual void show() = 0;
                   ^
```

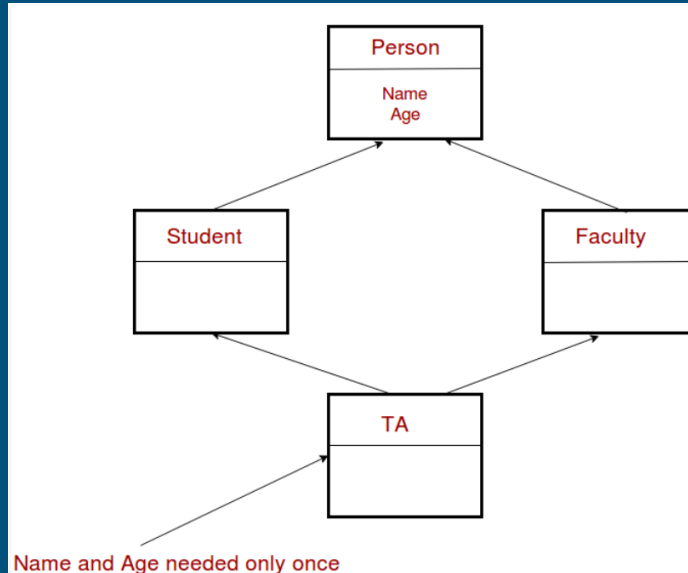
```
#include<iostream>
using namespace std;

class Test
{
    int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};

int main(void)
{
    Test t;
    return 0;
}
```

Multiple inheritance

A class may inherit from multiple base classes



Multiple inheritance

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

class MyImage{
public:
    MyImage(int height, int width)
        : m_height(height), m_width(width){}
    int getHeight() const{return m_height;}
    int getWidth() const{return m_width;}

private:
    int m_height;
    int m_width;
};
```

```
class MyShape{
public:
    MyShape(int type): m_type(type){}
    int getType() const{return m_type;}
private:
    int m_type;
};
```

```
class MyPicture: public MyImage, public MyShape{
public:
    MyPicture(): MyImage(300, 400), MyShape(1){}
};

int main(){
    MyPicture mp;
    cout << "Width: " << mp.getWidth() << endl;
    cout << "Height: " << mp.getHeight() << endl;
    cout << "Type: " << mp.getType() << endl;
    return 0;
}
```

Output:

Multiple inheritance

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

```
class MyImage{
public:
    MyImage(int height, int width)
        : m_height(height), m_width(width){}
    int getHeight() const{return m_height;}
    int getWidth() const{return m_width;}

private:
    int m_height;
    int m_width;
};
```

```
class MyShape{
public:
    MyShape(int type): m_type(type){}
    int getType() const{return m_type;}
private:
    int m_type;
};
```

```
class MyPicture: public MyImage, public MyShape{
public:
    MyPicture(): MyImage(300, 400), MyShape(1){}
};

int main(){
    MyPicture mp;
    cout << "Width: " << mp.getWidth() << endl;
    cout << "Height: " << mp.getHeight() << endl;
    cout << "Type: " << mp.getType() << endl;
    return 0;
}
```

Output:

```
Width: 400
Height: 300
Type: 1
```

Multiple inheritance

However, ***you have to be careful***, because the data reference can be ambiguous.

This problem is referred as the ***diamond problem***.

Multiple inheritance

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

class A
{
    int x;
public:
    A(){
        cout << "A()" << endl;
    }
    void setX(int i) {x = i;}
    void print() { cout << x; }
};

class B: public A
{
public:
    B() { setX(10);
        cout << "B()" << endl;}
};
```

```
class C: public A
{
public:
    C() { setX(20);
        cout << "C()" << endl; }
};

class D: public B, public C {
public:
};

int main()
{
    D d;
    d.print();
    return 0;
}
```

Multiple inheritance

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

class A
{
    int x;
public:
    A(){
        cout << "A()" << endl;
    }
    void setX(int i) {x = i;}
    void print() { cout << x; }
};
```

```
class B: public A
{
public:
    B() { setX(10);
        cout << "B()" << endl;}
};
```

```
class C: public A
{
public:
    C() { setX(20);
        cout << "C()" << endl; }
};
class D: public B, public C {
public:
};
int main()
{
    D d;
    d.print();
    return 0;
}
```

Output:

```
multiInheritance.cpp: In function 'int main()':
multiInheritance.cpp:36:7: error: request for member 'print' is ambiguous
    d.print();
    ~~~~~
multiInheritance.cpp:12:8: note: candidates are: void A::print()
    void print() { cout << x; }
    ~~~~~
multiInheritance.cpp:12:8: note: void A::print()
    void A::print()
```

Multiple inheritance

You can eliminate the problem by ***explicitly specifying the data origin*** or by ***using virtual inheritance in B and C.***

Multiple inheritance

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

class A
{
    int x;
public:
    A(){
        cout << "A()" << endl;
    }
    void setX(int i) {x = i;}
    void print() { cout << x; }
};

class B: virtual public A
{
public:
    B() { setX(10);
        cout << "B()" << endl;}
};
```

```
class C: virtual public A
{
public:
    C() { setX(20);
        cout << "C()" << endl; }
};

class D: public B, public C {
public:
};

int main()
{
    D d;
    d.print();
    return 0;
}
```

Multiple inheritance

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

class A
{
    int x;
public:
    A(){
        cout << "A()" << endl;
    }
    void setX(int i) {x = i;}
    void print() { cout << x; }
};

class B: virtual public A
{
public:
    B() { setX(10);
        cout << "B()" << endl;}
};
```

```
class C: virtual public A
{
public:
    C() { setX(20);
        cout << "C()" << endl; }
};

class D: public B, public C {
public:
};

int main()
{
    D d;
    d.print();
    return 0;
}
```

Output:

```
A()
B()
C()
20
```

Multiple inheritance

For your information, multiple inheritance is ***deprecated*** in practice.

However, it is worth knowing the multiple inheritance concept!

Exceptions



The Zen of Python (by Tim Peters)

JoyQuiz: Read *The Zen of Python* and find some verses explaining why it is quoted here.

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than right now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

Catch me if you can

You can catch any exception using the following structure

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

int main()
{
    try {
        throw 10;
    }
    catch (char *excp) {
        cout << "Caught " << excp;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Colored by Color Scripter

Catch me if you can

You can catch any exception using the following structure

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

int main()
{
    try {
        throw 10;
    }
    catch (char *excp) {
        cout << "Caught " << excp;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Colored by Color Scripter

Output:

Default Exception

Catch me if you can

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

class MyString
{
public:
    MyString(int nSize){
        str = new char[nSize];
    }

    ~MyString(){
        delete[] str;
        cout << "Instances have destructed successfully." << endl;
    }

private:
    char *str;
};
```

will throw exception when
memory allocation failed.

Colored by Color Scri

```
int main()
{
    try{
        int nSize;
        cout << "Input size: ";

        cin >> nSize;
        MyString a(nSize);
    }
    catch (bad_alloc &e){
        cerr << "/* error message */" << endl;
        cerr << e.what() << endl;
    }

    return 0;
}
```

Colored by Color Scripter

```
PS C:\Users\SeokjinKim\Documents\LuvCpp\admin> ./a
Input size: 100
```


Catch me if you can

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

class MyString
{
public:
    MyString(int nSize){
        str = new char[nSize];
    }

    ~MyString(){
        delete[] str;
        cout << "Instances have destructed successfully." << endl;
    }

private:
    char *str;
};
```

will throw exception when
memory allocation failed.

```
int main()
{
    try{
        int nSize;
        cout << "Input size: ";

        cin >> nSize;
        MyString a(nSize);
    }
    catch (bad_alloc &e){
        cerr << "/* error message */" << endl;
        cerr << e.what() << endl;
    }

    return 0;
}
```

Colored by Color Scripter

Output: Input size: 100
Instances have destructed successfully.


Catch me if you can

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

class MyString
{
public:
    MyString(int nSize){
        str = new char[nSize];
    }

    ~MyString(){
        delete[] str;
        cout << "Instances have destructed successfully." << endl;
    }

private:
    char *str;
};
```

 will throw exception when
memory allocation failed.

```
int main()
{
    try{
        int nSize;
        cout << "Input size: ";

        cin >> nSize;
        MyString a(nSize);
    }
    catch (bad_alloc &e){
        cerr << "/* error message */" << endl;
        cerr << e.what() << endl;
    }

    return 0;
}
```

Colored by Color Scripter

```
PS C:\Users\SeokjinKim\Documents\LuvCpp\admin> ./a
Input size: -1
```

Catch me if you can

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

class MyString
{
public:
    MyString(int nSize){
        str = new char[nSize];
    }

    ~MyString(){
        delete[] str;
        cout << "Instances have destructed successfully." << endl;
    }

private:
    char *str;
};
```

will throw exception when
memory allocation failed.

Output:

```
Input size: -1
/* error message */
std::bad_alloc
```

```
int main()
{
    try{
        int nSize;
        cout << "Input size: ";

        cin >> nSize;
        MyString a(nSize);
    }
    catch (bad_alloc &e){
        cerr << "/* error message */" << endl;
        cerr << e.what() << endl;
    }

    return 0;
}
```

Colored by Color Scripter

Creating your own exception

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

class Exception : public runtime_error {
public:
    Exception()
        : runtime_error("Math error: Attempted to divide by Zero\n") {}
};

float Division(float num, float den)
{
    if (den == 0)
        throw Exception();

    return (num / den);
}
```

Colored by Color S

```
int main()
{
    float numerator, denominator, result;
    cout << "numerator: ";
    cin >> numerator;
    cout << "denominator: ";
    cin >> denominator;

    try {
        result = Division(numerator, denominator);

        cout << "The quotient is " << result << endl;
    }
    catch (Exception& e) {
        cout << "Exception occurred" << endl
            << e.what();
    }

    return 0;
}
```

Colored by Color Scripter

```
numerator: 25.4
denominator: 13.2
```

Creating your own exception

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

class Exception : public runtime_error {
public:
    Exception()
        : runtime_error("Math error: Attempted to divide by Zero\n") {}
};

float Division(float num, float den)
{
    if (den == 0)
        throw Exception();

    return (num / den);
}
```

Colored by Color Scripter

Output:

```
numerator: 25.4
denominator: 13.2
The quotient is 1.92424
```

```
int main()
{
    float numerator, denominator, result;
    cout << "numerator: ";
    cin >> numerator;
    cout << "denominator: ";
    cin >> denominator;

    try {
        result = Division(numerator, denominator);

        cout << "The quotient is " << result << endl;
    }
    catch (Exception& e) {
        cout << "Exception occurred" << endl
            << e.what();
    }
    return 0;
}
```

Colored by Color Scripter

Creating your own exception

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

class Exception : public runtime_error {
public:
    Exception()
        : runtime_error("Math error: Attempted to divide by Zero\n") {}
};

float Division(float num, float den)
{
    if (den == 0)
        throw Exception();

    return (num / den);
}
```

Colored by Color S

```
numerator: 25.4
denominator: 0
```

```
int main()
{
    float numerator, denominator, result;
    cout << "numerator: ";
    cin >> numerator;
    cout << "denominator: ";
    cin >> denominator;

    try {
        result = Division(numerator, denominator);

        cout << "The quotient is " << result << endl;
    }
    catch (Exception& e) {
        cout << "Exception occurred" << endl
            << e.what();
    }
    return 0;
}
```

Colored by Color S

Creating your own exception

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

class Exception : public runtime_error {
public:
    Exception()
        : runtime_error("Math error: Attempted to divide by Zero\n") {}
};

float Division(float num, float den)
{
    if (den == 0)
        throw Exception();

    return (num / den);
}
```

Output:

```
numerator: 25.4
denominator: 0
Exception occurred
Math error: Attempted to divide by Zero
```

```
int main()
{
    float numerator, denominator, result;
    cout << "numerator: ";
    cin >> numerator;
    cout << "denominator: ";
    cin >> denominator;

    try {
        result = Division(numerator, denominator);

        cout << "The quotient is " << result << endl;
    }
    catch (Exception& e) {
        cout << "Exception occurred" << endl
            << e.what();
    }

    return 0;
}
```

Colored by Color Scripter

Standard exceptions

There exist some standard exceptions that can be raised in some circumstances

```
#include <stdexcept>
```

A small logo with the letters 'CS' in green, likely representing a course or institution.

- bad_alloc
- bad_cast
- bad_exception
- bad_typeid
- logic_error
 - domain_error
 - invalid_argument
 - length_error
 - out_of_range
- runtime_error
 - range_error
 - overflow_error
 - underflow_error

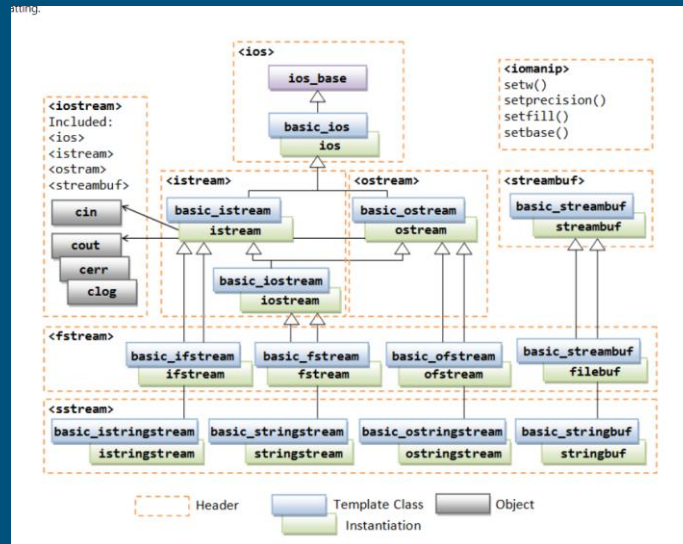
Streams

Basics

C++ provides input/output capability throughout the iostream classes that provide the stream concept (iXXXstream for input and oXXXstream for output).

Below is more description

https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp10_IO.html



iostream and ios

Screen outputs and keyboard inputs may be handled using the iostream header file

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

int main(int argc, char **argv){
    unsigned char age = 65;
    cout << static_cast<unsigned>(age) << endl;
    cout << static_cast<void const*>(&age) << endl;

    double f = 3.14159;
    cout.unsetf(ios::floatfield);
    cout.precision(5);
    cout << f << endl;
    cout.precision(10);
    cout << f << endl;
    cout.setf(ios::fixed, ios::floatfield);
    cout << f << endl;
```

```
cout << "Enter a number, or -1 to quit: ";
int i = 0;
while(cin >> i)
{
    if (i == -1) break;
    cout << "You entered " << i << '\n';
}
return 0;
}
```

Colored by Color Scripter

iostream and ios

Screen outputs and keyboard inputs may be handled using the iostream header file

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

int main(int argc, char **argv){
    unsigned char age = 65;
    cout << static_cast<unsigned>(age) << endl;
    cout << static_cast<void const*>(&age) << endl;

    double f = 3.14159;
    cout.unsetf(ios::floatfield);
    cout.precision(5);
    cout << f << endl;
    cout.precision(10);
    cout << f << endl;
    cout.setf(ios::fixed, ios::floatfield);
    cout << f << endl;
```

```
cout << "Enter a number, or -1 to quit: ";
int i = 0;
while(cin >> i)
{
    if (i == -1) break;
    cout << "You entered " << i << '\n';
}
return 0;
}
```

Colored by Color Scripter

Output:

```
65
0x61ff17
3.1416
3.14159
3.1415900000
Enter a number, or -1 to quit: 1
You entered 1
-1
```

Class input/output

You can implement a class input and output using friends functions.

```
#include <iostream>

class Foo {
public:
    friend std::ostream& operator<< ( std::ostream & output, Foo const & that )
    { return output << that._value; }
    friend std::istream& operator>> ( std::istream & input, Foo& foo )
    { return input >> foo._value; }

private:
    double _value;
};
```

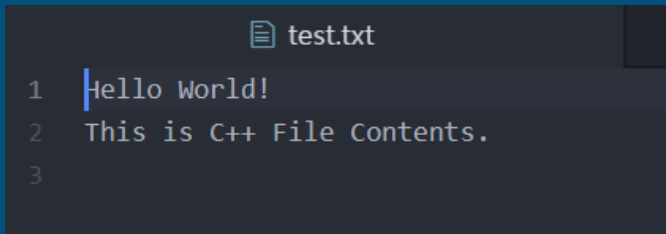
Colored by Color Scripter

Working with files

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    string filePath = "test.txt";
    // write File
    ofstream writeFile(filePath);
    if( writeFile.is_open() ){
        writeFile << "Hello World!\n";
        writeFile << "This is C++ File Contents.\n";
        writeFile.close();
    }
    // read File
    ifstream openFile(filePath);
    if( openFile.is_open() ){
        string line;
        while(getline(openFile, line)){
            cout << line << endl;
        }
        openFile.close();
    }
    return 0;
}
```

Working with files

Output:



A screenshot of a text editor window titled "test.txt". The editor shows two lines of text: "Hello World!" on the first line and "This is C++ File Contents." on the second line. The first line is highlighted with a blue selection bar.

```
1 Hello World!  
2 This is C++ File Contents.  
3
```

```
#include <fstream>  
#include <iostream>  
using namespace std;  
int main()  
{  
    string filePath = "test.txt";  
    // write File  
    ofstream writeFile(filePath);  
    if( writeFile.is_open() ){  
        writeFile << "Hello World!\n";  
        writeFile << "This is C++ File Contents.\n";  
        writeFile.close();  
    }  
    // read File  
    ifstream openFile(filePath);  
    if( openFile.is_open() ){  
        string line;  
        while(getline(openFile, line)){  
            cout << line << endl;  
        }  
        openFile.close();  
    }  
    return 0;  
}
```

Working with strings

```
#include <sstream>
#include <iostream>

int main(int argc, char **argv) {
    const char *svalue = "42.0";
    int ivalue;
    std::istringstream istream;
    std::ostringstream ostream;

    istream.str(svalue);
    istream >> ivalue;
    std::cout << svalue << " = " << ivalue << std::endl;

    ostream.clear();
    ostream << ivalue;
    std::cout << ivalue << " = " << ostream.str() << std::endl;

    return 0;
}
```


Working with strings

```
#include <sstream>
#include <iostream>

int main(int argc, char **argv) {
    const char *svalue = "42.0";
    int ivalue;
    std::istringstream istream;
    std::ostringstream ostream;

    istream.str(svalue);
    istream >> ivalue;
    std::cout << svalue << " = " << ivalue << std::endl;

    ostream.clear();
    ostream << ivalue;
    std::cout << ivalue << " = " << ostream.str() << std::endl;

    return 0;
}
```

Colored by Color Scripter

Output:

```
42.0 = 42
42 = 42
```

Templates



Templates

Templates are special operators that specify that a class or a function is written for one or several generic types that are not yet known.

The format for declaring function templates is :

```
- template <typename identifier> function_declaration;  
- template <typename identifier> class_declaration;
```

Templates

You may use `class` in place of `typename` interchangeably.

You can have several templates and to actually use a class or function template, you have to specify all known types:

Templates

```
template<typename T1>           // may use class instead of typename
T1 foo1( void ) { /* ... */ };

template<typename T1, typename T2>
T1 foo2( void ) { /* ... */ };

template<typename T1>
class Foo3 { /* ... */ };

int a = foo1<int>();
float b = foo2<int, float>();
Foo<int> c;
```

Template parameter

There are three possible template types:

- Type
- Non-type
- Template

```
template<typename T> T foo( void ) { /*.....*/ }
```

```
template<int N> T foo( void ) { /*.....*/ }
```

```
template< template <typename T> > foo( void ) { /*.....*/ }
```

Template function

```
template <class T>
T max( T a, T b)
{
    return( a > b ? a : b );
}

#include <sstream>
#include <iostream>

int main( int argc, char **argv )
{
    std::cout << max<int>( 2.2, 2.5 ) << std::endl;
    std::cout << max<float>( 2.2, 2.5 ) << std::endl;
}
```

Template function

```
template <class T>
T max( T a, T b)
{
    return( a > b ? a : b );
}

#include <sstream>
#include <iostream>

int main( int argc, char **argv )
{
    std::cout << max<int>( 2.2, 2.5 ) << std::endl;
    std::cout << max<float>( 2.2, 2.5 ) << std::endl;
}
```

Output:

```
2
2.5
```


Template class

```
template <class T>
class Foo {
    T _value;

public:
    Foo( T value ) : _value(value) { };
}

int main( int argc, char **argv ) {
    Foo<int> foo_int;
    Foo<float> foo_float;
}
```

Template specialization

```
template <class T>
class Foo {
    T _value;
public:
    Foo( T value ) : _value(value)
    {
        std::cout << "Generic constructor called" << std::endl;
    };
}

template <>
class Foo<float> {
    float _value;
public:
    Foo( float value ) : _value(value)
    {
        std::cout << "Specialized constructor called" << std::endl;
    };
}

int main( int argc, char **argv ) {
    Foo<int> foo_int;
    Foo<float> foo_float;
}
```

STL

(Standard Template Library)

Containers

STL containers are template classes that implement various ways of storing elements and accessing them.

- Sequence containers
 - vector
 - deque
 - list
- Container adapter
 - stack
 - queue
 - priority_queue
- Associative containers
 - set
 - multiset
 - map
 - multimap
 - bitset

Containers and Iterator - vector

```
#include <iostream>
#include <vector>

using namespace std;
```

```
int main(){

    vector<int> v;

    // push_back
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);

    // pop_back
    v.pop_back();
```

```
// back and front
cout << "vector front value : " << v.front() << '\n';
cout << "vector end value : " << v.back() << '\n';

// [i] and at(i)
cout << "vector operator[] : " << v[3] << '\n';
cout << "vector at : " << v.at(3) << '\n';

// size
cout << "vector size : " << v.size() << '\n';

// empty
cout << "Is it empty? : " << (v.empty() ? "Yes" : "No") << '\n';

// iterator
vector<int>::iterator begin_iter = v.begin(); // auto begin_iter =
v.begin() possible
vector<int>::iterator end_iter = v.end(); // auto end_iter = v.end()
possible

cout << "vector begin value : " << *begin_iter << '\n';
cout << "vector end value: " << *end_iter << '\n';
```

Colored by Color

```
//for
for(vector<int>::iterator iter =
v.begin(); iter != v.end(); iter++){
    cout << *iter << '\t';
}
cout << endl;

//for-each
for(auto& num: v)
    cout << num << '\t';

return 0;

}
```

Colored by Color Scripter

Containers and Iterator - vector

```
#include <iostream>
#include <vector>

using namespace std;
```

```
int main(){

    vector<int> v;

    // push_back
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);
```

```
// pop_back
v.pop_back();
```

```
// back and front
cout << "vector front value : " << v.front() << '\n';
cout << "vector end value : " << v.back() << '\n';

// [i] and at(i)
cout << "vector opeartor[] : " << v[3] << '\n';
cout << "vector at : " << v.at(3) << '\n';

// size
cout << "vector size : " << v.size() << '\n';

// empty
cout << "Is it empty? : " << (v.empty() ? "Yes" : "No") << '\n';

// iterator
vector<int>::iterator begin_iter = v.begin(); // auto begin_iter =
v.begin() possible
vector<int>::iterator end_iter = v.end(); // auto end_iter = v.end()
possible

cout << "vector begin value : " << *begin_iter << '\n';
cout << "vector end value: " << *end_iter << '\n';
```

Colored by Color

```
//for
for(vector<int>::iterator iter =
v.begin(); iter != v.end(); iter++){
    cout << *iter << '\t';
}
cout << endl;

//for-each
for(auto& num: v)
    cout << num << '\t';

return 0;
}
```

Output:

```
vector front value : 1
vector end value : 4
vector opeartor[] : 4
vector at : 4
vector size : 4
Is it empty? : No
vector begin value : 1
vector end value: 5
1      2      3      4
1      2      3      4
```

Containers and Iterator - map

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main(){
    map< string, int > m;

    m.insert(make_pair("a", 1));
    m.insert(make_pair("b", 2));
    m.insert(make_pair("c", 3));
    m.insert(make_pair("d", 4));
    m.insert(make_pair("e", 5));
    m["f"] = 6; // also possible

    m.erase("d");
    m.erase("e");
    m.erase(m.find("f")); // also possible
```

```
if(!m.empty()) cout << "m size : " << m.size() << '\n';

cout << "a : " << m.find("a")->second << '\n';
cout << "b : " << m.find("b")->second << '\n';

cout << "a count : " << m.count("a") << '\n';
cout << "b count : " << m.count("b") << '\n';

cout << "traverse" << '\n';

// map< string, int >::iterator it; also possible
for(auto it = m.begin(); it != m.end(); it++){
    cout << "key : " << it->first << " " << "value : " << it->second << '\n';
}

return 0;
}
```

Colored by Color Scripter

Containers and Iterator - map

Output:

```
#include <iostream>
#include <map>
#include <string>
```

```
using namespace std;
```

```
int main(){
    map< string, int > m;
```

```
    m.insert(make_pair("a", 1));
    m.insert(make_pair("b", 2));
    m.insert(make_pair("c", 3));
    m.insert(make_pair("d", 4));
    m.insert(make_pair("e", 5));
    m["f"] = 6; // also possible
```

```
    m.erase("d");
    m.erase("e");
    m.erase(m.find("f")); // also possible
```

```
    if(!m.empty()) cout << "m size : " << m.size() << '\n';
```

```
    cout << "a : " << m.find("a")->second << '\n';
    cout << "b : " << m.find("b")->second << '\n';
```

```
    cout << "a count : " << m.count("a") << '\n';
    cout << "b count : " << m.count("b") << '\n';
```

```
    cout << "traverse" << '\n';
```

```
// map< string, int >::iterator it; also possible
```

```
for(auto it = m.begin(); it != m.end(); it++){
    cout << "key : " << it->first << " " << "value : " << it->second << '\n';
}
```

```
return 0;
```

```
}
```

```
m size : 3
a : 1
b : 2
a count : 1
b count : 1
traverse
key : a value : 1
key : b value : 2
key : c value : 3
```

Colored by Color Scripter

Algorithms

Algorithms from the STL offer fast, robust, tested and maintained code for a lot of standard operations on ranged elements. Don't reinvent the wheel !

Have a look at cplusplus.com/reference - algorithm for an overview.

Algorithms

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

bool compare( const int & first, const int & second ) {
    return (first < second);
}

int main( int argc, char **argv ) {
    vector<int> v;

    v.push_back(1);
    v.push_back(5);
    v.push_back(2);
    v.push_back(14);
    v.push_back(3);
```

Colored by Color

```
//Before sorting
for(auto& num: v)
    cout << num << '\t';

sort(v.begin(), v.end(), &compare);

//After sorting
cout << endl;
for(auto& num: v)
    cout << num << '\t';
return 0;
}
```

Colored by Color Scripter

Algorithms

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

bool compare( const int & first, const int & second ) {
    return (first < second);
}

int main( int argc, char **argv ) {
    vector<int> v;

    v.push_back(1);
    v.push_back(5);
    v.push_back(2);
    v.push_back(14);
    v.push_back(3);
```

```
//Before sorting
for(auto& num: v)
    cout << num << '\t';

sort(v.begin(), v.end(), &compare);

//After sorting
cout << endl;
for(auto& num: v)
    cout << num << '\t';
return 0;
}
```

Output:

1	5	2	14	3
1	2	3	5	14

Colored by Color Scripter

Colored by Color

Reference

<https://www.geeksforgeeks.org/c-plus-plus/>

<https://docs.microsoft.com/ko-kr/cpp/cpp/c-cpp-language-and-standard-libraries?view=vs-2017>

<https://sites.google.com/site/cs101atjust/slides>

이것이 C++이다, 최호성 저