Creative Software Design

Introduction to C++

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Today's Topics

- Introduction to C++

- Difference between C and C++
  - Namespace
  - Input/Output
  - String
  - Boolean
  - Function Overloading
  - Default Arguments
  - Brief Intro to Class, Reference, Template, Exception Handling, STL(Standard Template Library)

- Introduction to C++ Standard Versions

Introduction to C++

Introduction to C++

- Developed by Bjarne Stroustrup at Bell Labs since 1979, as an extension of the C language

- Provides both low-level functionality & efficient abstraction
  - Low-level hardware access, performance & memory efficiency
  - High-level abstraction using object-oriented, generic programming paradigm

- But, high complexity!

C++ Structure of Program

```
// Preprocessor processes #-directives.
#include <iostream>

using namespace std;  /* Use std namespace */

int main() {
  cout << "hello_world\n";  // Print hello_world.
  return 0;
}
```

- Overall structure:
  - Comments.
  - Preprocessor-related parts : #-directives.
  - C/C++ part : statements, declarations or definitions of functions and classes.
- A few notes:
  - A statement ends with a semicolon (;).
  - Blanks (spaces, tabs, newlines) do not affect the meaning, at least in C/C++ parts.

C++ Variables and Data Types

- Fundamental data types
  - Integer: `int`, `char`, `short`, `long`, `long long`, (+`unsigned`).
  - Boolean : **`bool`**.
  - Floating point numbers : `float`, `double`, `long double`.
- Variables
  - Variables : specific memory locations
  - Declaration : `int a; double b = 1.0; char c, d = 'a';`
  - Scope : whether the variable is visible (= usable).

```
void MyFunc() {
  int a = 0, b = 1;
  { int a = 2, c = 3;
    cout << "a = " << a << ", b = " << b << ", c = " << c <<
endl;}
  cout << "a = " << a << ", b = " << b << endl;
}
```

C++ Constants

- Integer : `123` (123), `0123` (83), `0x123` (291) / `123u`, `123l`, `123ul`.
- Floating-points : `0.1` (d), `0.1f` (f). / `1e3`, `0.3e-9`.
- Character and string literal : `'c'`, `"a string\n"`.
- **Boolean : `true`, `false`.**

- Defined constants vs. declared constants.
  - Defined constant : `#define MY_NUMBER 1.234`
  - Declared constant : `const double MY_NUMBER = 1.234;`

C++ Operators

- C++ operators
  - Increment/decrement : `++a`, `a++`, `--a`, `a--`.
  - Arithmetic : `a + b`, `a - b`, `a * b`, `a / b`, `a % b`, `+a`, `-a`.
  - Relational : `a == b`, `a != b`, `a < b`, `a <= b`, `a > b`, `a >= b`.
  - Bitwise : `a & b`, `a | b`, `a ^ b`, `~a`, `a >> b`, `a << b`.
  - Logical : `a && b`, `a || b`, `!a`.
  - Conditional : `a ? b : c`
  - (Compound) assignment : `a = b`, `a += b`, `a &= b`, …
  - Comma : `a, b` (e.g. `a = (b = 3, b + 2);`)
  - Other : type casting, `sizeof()`, …
- Operator precedence.
  - Enclose with () when not sure.

C++ Operators Precedence (1/2)

- Operator precedence.
  - Enclose with () when not sure.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++` `a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left
	`+a` `-a`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new` `new[]`	Dynamic memory allocation	
	`delete` `delete[]`	Dynamic memory deallocation	
4	`.*` `->*`	Pointer-to-member	Left-to-right
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	

C++ Operators Precedence (2/2)

- Operator precedence. (cont.)
  - Enclose with () when not sure.

8	<=>	Three-way comparison operator (since C++20)	
9	< <= > >=	For relational operators < and ≤ and > and ≥ respectively	
10	== !=	For equality operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR	
17	,	Comma	Left-to-right

Difference between C and C++

Namespace

lib1.h

```
void func() {

}
```

lib2.h

```
void func() {

}
```

```
#include <lib1.h>
#include <lib2.h>
int main(void){
    func();
    return 0;
}
```

???

Namespace

- A method for preventing name conflicts (of variables, functions, ...) in large projects

- All identifiers (variable names, function names, ...) declared in *code* belong to namespace *ns*

```
namespace ns {
        code
}
```

```
#include <iostream>

// first name space
namespace first_space {
  void func() {
     std::cout << "Inside first_space" << std::endl;
  }
}

// second name space
namespace second_space {
void func() {
   std::cout << "Inside second_space" <<
std::endl;
  }
}

int main () {
  // Calls function from first name space.
  first_space::func();

  // Calls function from second name space.
  second_space::func();
  return 0;
}
```

scope resolution operator

Namespace std

- All the classes, objects, and functions of the *C++ standard library* are defined within "standard" namespace named **std**

- For example, `std::cout, std::cin, std::endl` for input/output

using namespace

- **using namespace *ns*;**
  - **"Import"** the namespace *ns* into the current scope
  - Subsequent code will use identifiers in the namespace *ns* **as if they were in current namespace**
  - This effect applies only **within the scope "using namespace" used**

```
#include <iostream>
using namespace std; // import std into global scope

namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}

int main () {
   // import first_space into the current scope (main())
   // at this moment, std and first_space are imported
   using namespace first_space;

   func(); // first_space::func();
   return 0;
}
```

using namespace

```
#include <iostream>
using namespace std; // import std into global
scope

namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}
int main () {
   using namespace first_space; // import
first_space into the current scope (main())

   func(); // first_space::func();

   using namespace second_space; // import
second_space into the current scope (main())

   // at this moment, std, first_space, and
second_space are imported
   func(); // so, generates an error

   return 0;
}
```

```
#include <iostream>
using namespace std; // import std into global
scope

namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}
int main () {
   {
      using namespace first_space; // import
first_space into the current scope
      func(); // first_space::func();
   }
   {
      using namespace second_space; // import
second_space into the current scope
      func(); // second_space::func();
   }
   return 0;
}
```

Quiz #1

- What is the expected output?

```
#include <cstdio>
namespace lfc1
{
    int x = 10;
}
namespace lfc2
{
    int x = 20;
}
int main ()
{
    int x = 30;
    lfc1::x;
    lfc2::x;
    printf("%d\n", x);
    return 0;
}
```

Input / Output

- C: printf(), scanf()
  - `#include <stdio.h>`
  - `scanf("%d", &num);`
  - `printf("hello %d\n", num);`

- C++: std::cout, std::cin, stream operators (>>, <<)
  - `#include <iostream>`
  - `std::cin >> anum;`
  - `std::cout << "hello " << num << std::endl;`
  - This is the C++ way of input / output, but you can still use C- style input / output in your C++ code.

C++ Stream IO

- Stream: a sequence of bytes flowing in and out of the programs
- >> - stream extraction operator. [stream] >> [variable]
- << - stream insertion operator. [stream] << [variable or value]
- std::cout - standard output stream, normally the screen
- std::cin - standard input stream, normally the keyboard
- std::endl - inserts a newline character (‘\n’)

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

int main()
{
    string s2; int i; double d;
    cin >> s2 >> i >> d; // text input for s2, i, d should be separated by
a space, tab, or enter.
// the program execution is blocked until all variables values are finally
// entered by pressing Enter.
    cout << "s2:" << s2 << ", i:" << i << ", d:" << d << endl;
    return 0;
}
```

(FYI) std::endl

- std::endl and '\n'
  - The same meaning: newline
  - The only difference is that std::endl **flushes** the output buffer, and '\n' doesn't.
    - flushing: transferring the data from the buffer to the stdout or file (and clear the buffer).

```
std::cout << std::endl;

// is equivalent to

std::cout << '\n' << std::flush;
```

String

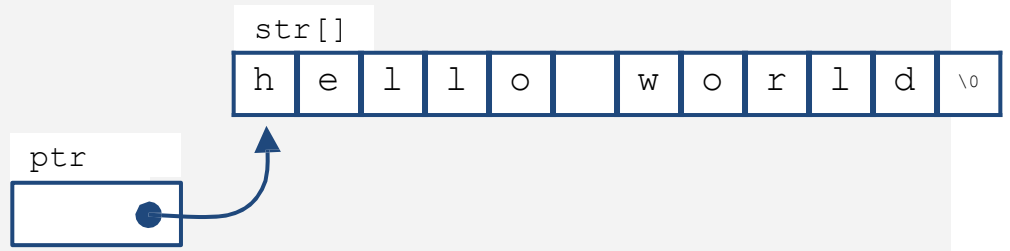
- C: C-style null-terminated string (using C-style array)
  - `char str1[] = "My String";`
  - `const char* str2 = "Your String";`
  - Just an array of characters terminated with a null character ('\0')

- C++: std::string
  - `#include <string>`
  - `std::string str1 = "abc";`
  - `std::string str2("def");`
  - Many convenient operations are available such as:
    - `str1 += "123" + str2.substr(0, 2);`
  - Much more powerful and convenient.
  - Use this way in C++. But you still need to understand C-style string because of the legacy code.

C-Style String

- A string is basically an array of characters (char []).
- C standard requires a string must be terminated with 0 ('\0').

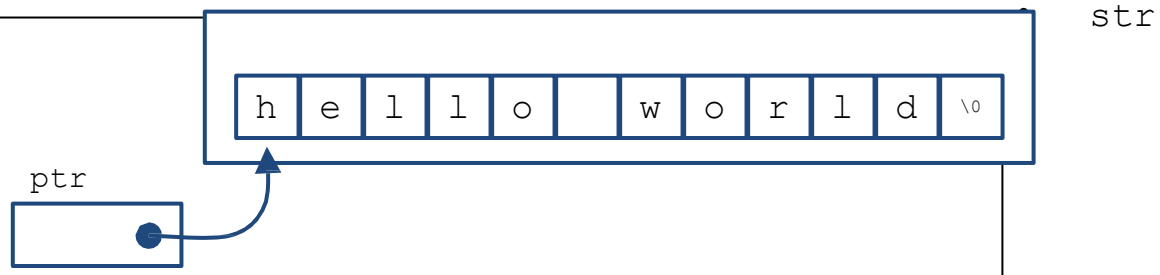
```c
#include <stdio.h>

int main() {
  char str[] = "hello world";

  char* ptr = str;
  while (*ptr != '\0') {
    printf("%c", *ptr++);
  }
  return 0;
}
```



C++ std::string

- C++ provides a powerful string class.

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

int main()
{
    string str = "hello world";
    cout << str << endl;    // C++ way of printing to stdout

    string str1 = str + " - bye world";
    cout << str1 << endl;            // hello world - bye world
    cout << str1.length() << endl;   // 23
    cout << str1[0] << endl;         // h

    str[0] = 'j';
    str.resize(5);
    cout << str << endl;  // jello

    const char* ptr = str.c_str();
    printf("%s\n", ptr); // use c_str() for printf(), c++ string -> const char*

    return 0;
}
// check out http://www.cplusplus.com/reference/string/string/
// resize(), substr(), find(), etc.
```



More C++ Input Functions

```
std::string str;
std::cin >> str; // read a word (separated by a space, tab, enter)
```

```
#include <iostream>

using namespace std;

int main(){
  string line;
  cout <<"write a line " << endl;
  while (cin >> line && line != "q")
        cout << line << "---" << endl;
  return 0;
}
```

```
write a line
I like HY⏎
I---
like---
HY---
I love my son⏎
I---
love---
my---
son---
q⏎
```

More C++ Input Functions

```
std::string str;
std::cin >> str; // read a word (separated by a space, tab, enter)

std::getline(cin, str); // read characters until the default
                        // delimiter '\n' is found
```

```
#include <iostream>

using namespace std;

int main(){
  string line;
  cout <<"write a line " << endl;
  while (getline(cin, line)){
    cout << line << "---" << endl;
  }
  return 0;
}
```

```
write a line
I like HY ⏎
I like HY---
I love my son ⏎
I love my son---
```

More C++ Input Functions

```
std::string str;
std::cin >> str; // read a word (separated by a space, tab, enter)

std::getline(cin, str, ':'); // read characters until the delimiter
                             // ':' is found
```

```
#include <iostream>

using namespace std;

int main(){
  string line;
  cout <<"write a line " << endl;
  while (getline(cin, line, ':')){
    cout << line << "---" << endl;
  }
  return 0;
}
```

```
write a line
I:like:HY ⏎
I---
like---
I:love:my:son ⏎
HY
I---
love---
my---
: ⏎
son
---
```

More C++ Input Functions

- Note that `std::string` automatically resize to the length of target string.

```
char fname[10];
string lname;
cin >> fname;       // could be a problem if input size > 9 characters
cin >> lname;       // can read a very, very long word
cin.getline(fname, 10);  // may truncate input
getline(cin, lname);     // no truncation
```

Quiz #2

- What is the expected output?

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
    string name ("Jobs");
    string family ("Steve");
    name += " Apple ";
    name += family;
    name += '\n';
    cout << name;
    return 0;
}
```

Boolean

- To express Boolean values (true or false),

- C:
  - `int var1 = 1; // true`
  - `int var2 = 0; // false`
  - Non-zero values are regarded as ‘true’
  - (C99 standard support ‘bool’ type with <stdbool.h> header)

- C++:
  - `bool var1 = true;`
  - `bool var2 = false;`
  - More intuitive. Use this way in C++.

Function Overloading

- Using multiple functions sharing the same name
  - A family of functions that do the same thing but using different argument lists

```
void print(const char * str, int width);  // #1
void print(double d, int width);          // #2
void print(long l, int width);            // #3
void print(int i, int width);             // #4
void print(const char *str);              // #5


print("Pancakes", 15);          // use #1
print("Syrup");                 // use #5
print(1999.0, 10);              // use #2
print(1999, 12);                // use #4
print(1999L, 15);               // use #3
```

Function Overloading

- The *function signature*, not the function type, enables function overloading
  - A function signature consists of function name, parameter order & types, but **does not include return type**

```
int test1(int n, float m);          // different signatures,
double test1(float n, float m);     // hence allowed

int test2(int n, float m);        // the same signatures
double test2(int n, float m);     // compile error
```

Function Overloading

```
void dribble(char * bits);          // overloaded
void dribble (const char *cbits);   // overloaded
void dabble(char * bits);           // not overloaded
void drivel(const char * bits);     // not overloaded
```

```
const char p1[20] = "How's the weather?";
char p2[20] = "How's business?";
dribble(p1);       // dribble(const char *);
dribble(p2);       // dribble(char *);
dabble(p1);        // no match
dabble(p2);        // dabble(char *);
drivel(p1);        // drivel(const char *);
drivel(p2);        // drivel(const char *);
```

Default Arguments

- A *default argument* is a default value provided for a function parameter.
  - a parameter with a default value provided is often called an *optional parameter*.

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

int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}
int main()
{
    cout << sum(10, 15) << endl;       // sum(10, 15, 0, 0)
    cout << sum(10, 15, 25) << endl;   // sum(10, 15, 25, 0)
    cout << sum(10, 15, 25, 30) << endl; // sum(10, 15, 25, 30)
    return 0;
}
```

Default Arguments

- If a default argument is used, all subsequent parameters must have default arguments as well.

```
int sum(int x, int y, int z=0, int w) // compile error
```

- You cannot skip a default argument.

```
int sum(int x, int y, int z=0, int w=0) {...}
void main() {
    cout << sum(10, 15, 30) << endl;// 30 is copied to z

    // There is no way z can take the default argument
and specify w as 30.
    cout << sum(10, 15, , 30) << endl; // compile error
}
```

Default Arguments

- Default arguments can only be declared once.

```
void printValues(int x, int y=10);

void printValues(int x, int y=10) // compile error
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

- Best practice is to declare the default argument in the **function declaration** and not in the function definition,
  - because the declaration is more likely to be seen by other files.

```
void printValues(int x, int y=10);

void printValues(int x, int y)
 {
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

Default Arguments

- Functions with default arguments may be overloaded.

```
void print(std::string string) {...}
void print(char ch=' ') {...}

void main(){
    print();    // calls print(' ')
}
```

- But optional parameters do NOT count towards the parameters that make the function unique.

```
void printValues(int x) {...}
void printValues(int x, int y=10) {...}

int main(){
    printValues(5); // error: call of overloaded
'printValues(int)' is ambiguous
}
```

Quiz #3

- What is the expected output?

```
#include <iostream>
using namespace std;
void Values(int n1, int n2 = 10)
{
    using namespace std;
    cout << "1st value: " << n1 << "\n";
    cout << "2nd value: " << n2 << "\n";
}
int main()
{
    Values(1);
    Values(3, 4);
    return 0;
}
```

References

- References can be used similar to pointers (Think of it as a "referenced pointer")
  - Less powerful but safer than the pointer type.

```
int b = 10;
int& rb = b; // rb can be regarded as an "alias" of b
rb = 20;
cout << b << " " << rb << endl;    // 20 20
```

- **Will be covered in the next lecture**

Template

- Generalizes function or class by delaying type specification until compile-time.

```
// We also want to sort a double array.
void SelectionSort(double* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    double tmp = array[i];  array[i] =
    array[min_idx]; array[min_idx] =
    tmp;
  }
}
// And also a string array.
void SelectionSort(string* array, int size) {
   for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    string tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
// Suppose we want to sort an array of type T.
template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

- **Will be covered in later lectures**

STL (Standard Template Library)

- Powerful, template-based, reusable components
- STL extensively uses templates

- Divided into three components:
  - Containers: data structures that store objects of any type
  - Iterators: used to manipulate container elements
  - Algorithms: searching, sorting and many others

- **Will be covered in later lectures**

Exception Handling

- Examples of exceptions:
  - Memory allocation error - out of memory space.
  - Divide by zero.
  - File IO error.
- – …

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

- C++ provides a systematic way of handling exceptions

- **Will be covered in later lectures**

Evolution of C++

- C++98 (the first standard) / C++03 (its minor revision)
  - Called “traditional C++”
- C++11 / C++14 / C++17 / C++20...
  - Many cool & useful features such as smart pointer, auto keyword, lambda function, etc
  - Called “Modern C++”
- This class is mainly based on C++98 / C++03
  - The large majority of C++ is still same to C++98 / C++03
  - A large number of codebases are written in C++98 / C++03
- References to modern C++:
  - https://github.com/AnthonyCalandra/modern-cpp-features
  - https://en.cppreference.com/w/cpp/compiler_support