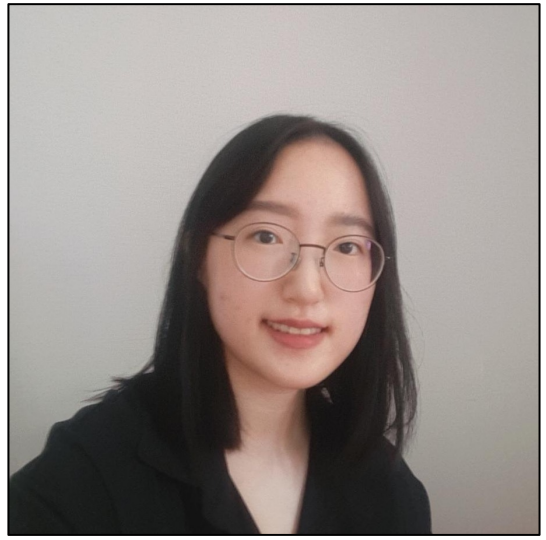
[CSED233-01] Data Structure

Basic C++ Programming (1)

Jaesik Park

POSTECH

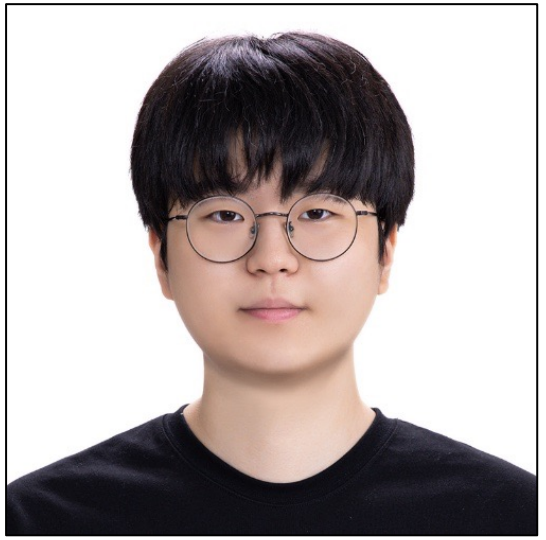
Teaching Assistants (Newer Ver.)



Suhyeon Jeong
(정수현)
posjsh@postech.ac.kr
Computer Vision Lab.

Sangjun Lee
(이상준)
leesangjun@postech.ac.kr
Computing Systems Lab.

Minjae Jeong
(정민재)
minjaetidtid@postech.ac.kr
Medical Imaging & Vision Lab.



Seunguk Do
(도승욱)
hu556727@postech.ac.kr
Computer Vision Lab.

What is C++?

- Extension to the C language
- Used to create computer programs
- Cross-platform language that can be used to create high-performance applications
- High level of control over system resources and memory
- The language was updated 4 major times in 2011, 2014, 2017, and 2020 to C++11, C++14, C++17, C++20

Why use C++?

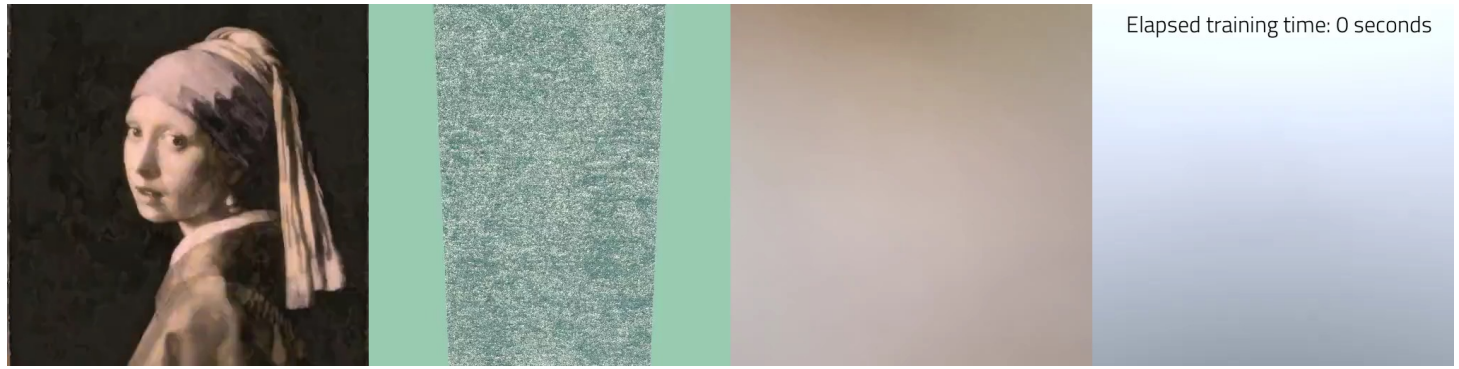
- Can be found in today's operating systems, Graphical User Interfaces, and embedded systems.
- C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- Can be adapted to multiple platforms.
- Fun and easy to learn!
- It makes it easy for programmers to learn other languages

High-performance computing

Examples in My Field

- My research interests lie in visual computing
  - Graphics + vision + imaging + display + learning +…
  - High-performance computing is desired

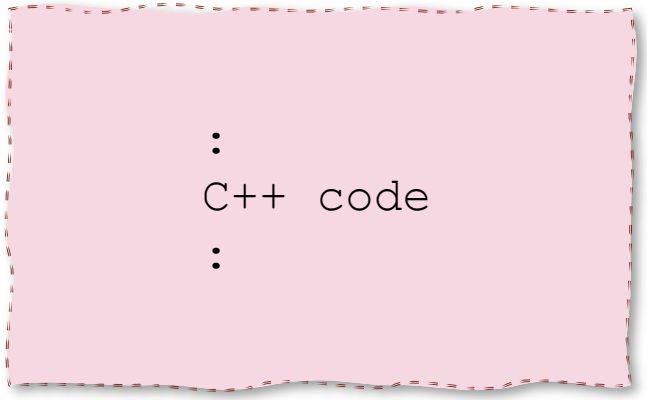
Jakob et al., Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering, ACM TOG 2022

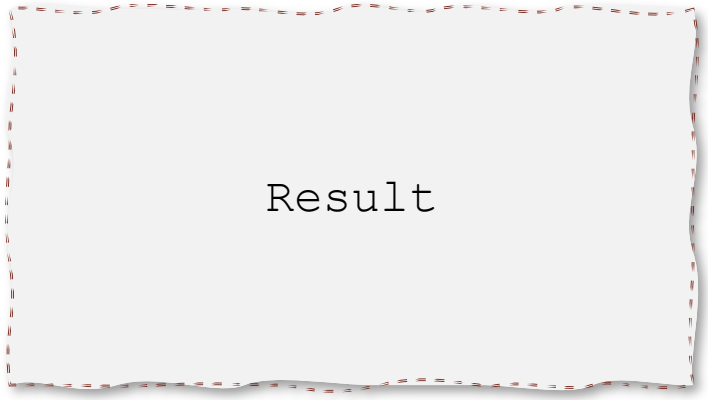


Muller et al., Instant Neural Graphics Primitives with a Multiresolution Hash Encoding, ACM TOG 2022

In This Class

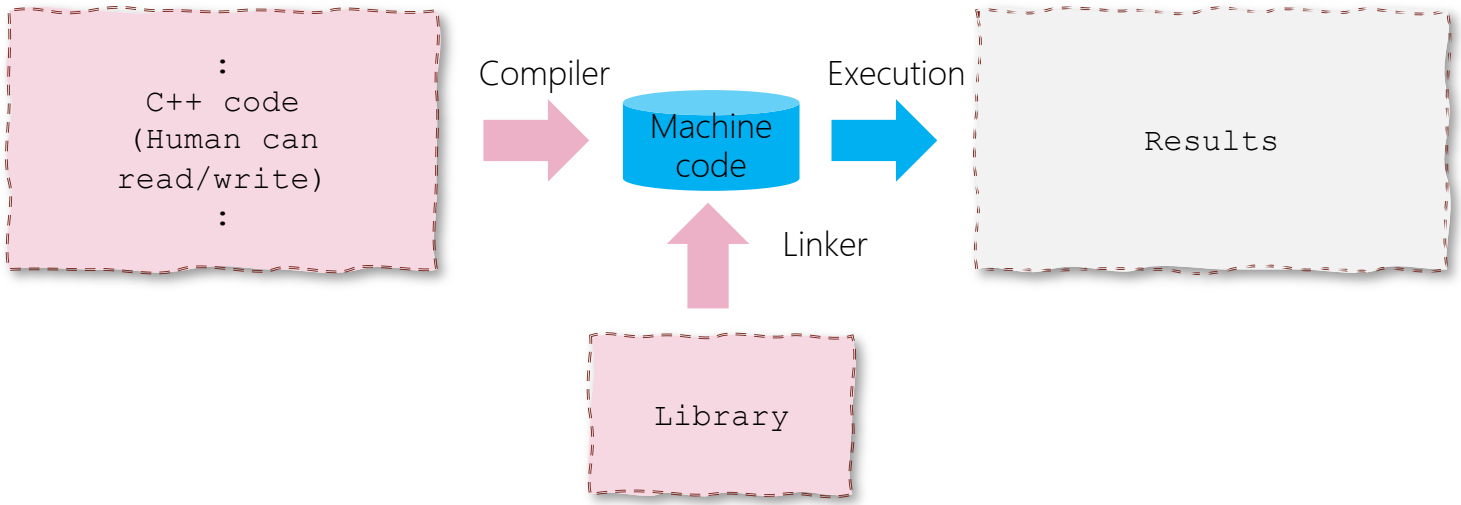
- Basic elements in C++ (Chapter 1 in the textbook)
- Writing simple codes and see how it works
- We will recap C++ in this class
  - Please familiarize yourself with C++
  - Useful reference: https://www.w3schools.com/cpp/default.asp





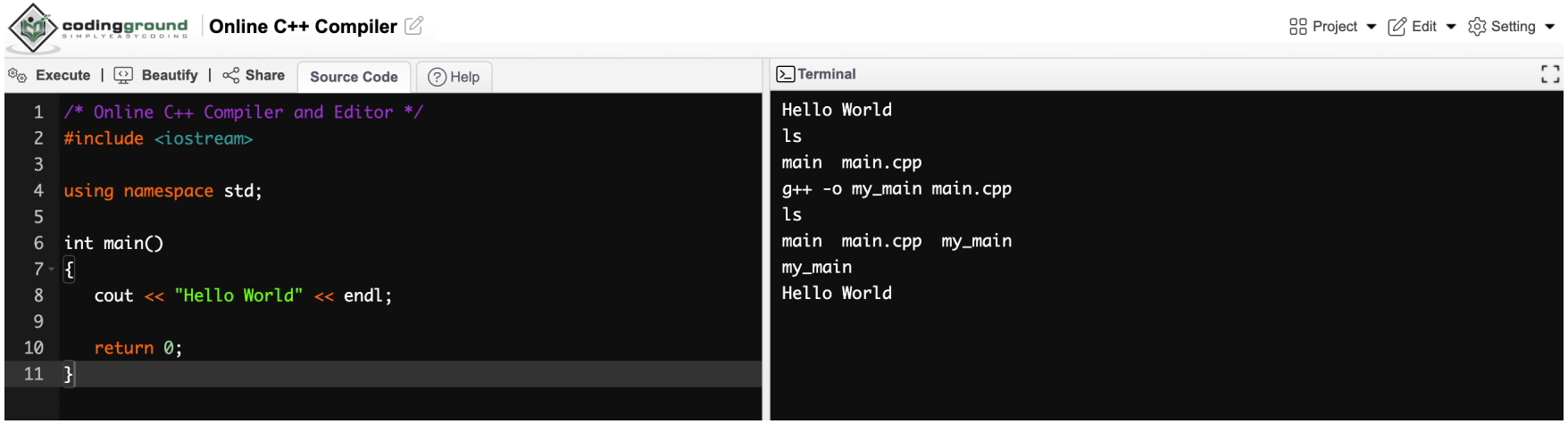
How C++ Works

1. **Create** a C++ source file and **save** it
2. **Compile**,
   - Using a *compiler*
     - creates a machine-code interpretation of your program
   - Using a *linker*
     - includes any required library functions needed
3. **Execute** program and see results



[Try yourself] Basic C++ example

- https://www.tutorialspoint.com/compile_cpp_online.php
- Click *Execute*
  - You will see Hello World on the terminal
- Type *ls* on the terminal to see file list you have
  - You will see *main* and *main.cpp*
- Type g++ -o my_main main.cpp
  - This converts *main.cpp* into a binary file (compiler + linker+ …)
  - Once you type ls again, you will see your binary file *my_main*
- Run your binary file by typing *my_main*
  - You will see Hello World



Simple C++ Program

```
>> Please enter two numbers: 7 35
Their sum is 42
```

Simple C++ Program

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y
   and outputs their sum */
int main( ) {
        int x, y;
        std::cout << "Please enter two numbers: ";
        std::cin >> x >> y;          // input x and y
        int sum = x + y;             // compute their sum
        std::cout << "Their sum is " << sum << std::endl;
        return 0;                    // terminate successfully
}
```

Header file

- #include <XX> is a header file library. Header files add functionality to C++ programs. You can make your own header files as well

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y
   and outputs their sum */
int main( ) {
        int x, y;
        std::cout << "Please enter two numbers: ";
        std::cin >> x >> y;          // input x and y
        int sum = x + y;             // compute their sum
        std::cout << "Their sum is " << sum << std::endl;
        return 0;                    // terminate successfully
}
```

Comments

- Comments can be used to explain C++ code, and to make it more readable. Comments can be singled-lined or multi-lined.

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y
   and outputs their sum */
int main( ) {
        int x, y;
        std::cout << "Please enter two numbers: ";
        std::cin >> x >> y;             // input x and y
        int sum = x + y;                // compute their sum
        std::cout << "Their sum is " << sum << std::endl;
        return 0;                       // terminate successfully
}
```

Main function

- Always appear in a C++ program: int main(). This is called a function. Any code inside its curly brackets {} will be executed.

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y
   and outputs their sum */
int main( ) {
        int x, y;
        std::cout << "Please enter two numbers: ";
        std::cin >> x >> y;          // input x and y
        int sum = x + y;             // compute their sum
        std::cout << "Their sum is " << sum << std::endl;
        return 0;                    // terminate successfully
}
```

Variable

- Variables are containers for storing data values.

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y
   and outputs their sum */
int main( ) {
        int x, y;
        std::cout << "Please enter two numbers: ";
        std::cin >> x >> y;          // input x and y
        int sum = x + y;             // compute their sum
        std::cout << "Their sum is " << sum << std::endl;
        return 0;                    // terminate successfully
}
```

Type

- In C++, there are different types of variables (defined with different keywords), for example:

- `bool` Boolean value, either true or false
- `char` character
- `short` short integer
- `int` integer
- `long long` long integer
- `float` single-precision floating-point number
- `double` double-precision floating-point number

Fundamental Types

Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed		**-128** to **127**
		unsigned		**0** to **255**
	16	unsigned		**0** to **65535**
	32	unsigned		**0** to **1114111** (**0x10ffff**)
integer	16	signed	**± 3.27 · 10^4**	**-32768** to **32767**
		unsigned	**0** to **6.55 · 10^4**	**0** to **65535**
	32	signed	**± 2.14 · 10^9**	**-2,147,483,648** to **2,147,483,647**
		unsigned	**0** to **4.29 · 10^9**	**0** to **4,294,967,295**
	64	signed	**± 9.22 · 10^{18}**	**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**
		unsigned	**0** to **1.84 · 10^{19}**	**0** to **18,446,744,073,709,551,615**
floating point	32	IEEE-754	• min subnormal: **± 1.401,298,4 · 10^{-45}** • min normal: **± 1.175,494,3 · 10^{-38}** • max: **± 3.402,823,4 · 10^{38}**	• min subnormal: **±0x1p-149** • min normal: **±0x1p-126** • max: **±0x1.fffffep+127**
	64	IEEE-754	• min subnormal: **± 4.940,656,458,412 · 10^{-324}** • min normal: **± 2.225,073,858,507,201,4 · 10^{-308}** • max: **± 1.797,693,134,862,315,7 · 10^{308}**	• min subnormal: **±0x1p-1074** • min normal: **±0x1p-1022** • max: **±0x1.fffffffffffffp+1023**

From https://en.cppreference.com/w/cpp/language/types

Standard Functions

- The prefix "std::" indicates that these objects are from the system's standard library. It is under "std" namespace. Those objects were included by the include <>

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y
   and outputs their sum */
int main( ) {
        int x, y;
        std::cout << "Please enter two numbers: ";
        std::cin >> x >> y;          // input x and y
        int sum = x + y;             // compute their sum
        std::cout << "Their sum is " << sum << std::endl;
        return 0;                    // terminate successfully
}
```

Inputs and Outputs

- The `cout` object, together with the << operator, is used to output values/print text
- `cin` is a predefined function that reads data from the keyboard with the extraction operator (>>).

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y
   and outputs their sum */
int main( ) {
        int x, y;
        std::cout << "Please enter two numbers: ";
        std::cin >> x >> y;          // input x and y
        int sum = x + y;             // compute their sum
        std::cout << "Their sum is " << sum << std::endl;
        return 0;                    // terminate successfully
}
```

Enumerations

- A user-defined type
  - that can hold any of a set of discrete values
  - Once defined, enumerations behave much like an integer type

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
enum Mood { HAPPY = 3, SAD = 1, ANXIOUS = 4, SLEEPY = 2 };

Day today = THU;              // today may be any of MON ... SAT
Mood myMood = SLEEPY;         // myMood may be HAPPY, ..., SLEEPY
```

Pointers

- Each variable is stored in the computer's memory at some location
  - It is called *address*
  - A *pointer* is a variable that holds the value of such an address.
    - Given a type `T`, the type `T*` denotes a pointer to a variable of type `T`.
- Essential operators are used to manipulate pointers.
  - *Address-of* operator, `&`.
    - For example, if x is an integer variable in your program &x is the address of x in memory.
  - *Dereferencing operator* *****.
    - For example,
      - if we were to declare q to be a pointer to an integer (that is, int*)
      - and then set q = &x, we could access x's value with *q.
      - Assigning an integer value to *q effectively changes the value of x.
- Very important to get familiar with C++

Pointer Example

```
char ch = 'Q';
char* p = &ch;          // p holds the address of ch
cout << *p;             // outputs the character 'Q'
ch = 'Z';               // ch now holds 'Z'
cout << *p;             // outputs the character 'Z'
*p = 'X';               // ch now holds 'X'
cout << ch;             // outputs the character 'X'
```

Arrays

- An *array* is a collection of elements of the same type
  - Given any type T and a constant *N*, a variable of type T[N] holds an array of *N* elements, each of type T
  - Each element of the array is referenced by its *index*,
    - that is, a number from 0 to *N* – 1.

```
double f[5];          // array of 5 doubles: f[0], . . ., f[4]
int m[10];            // array of 10 ints: m[0], ..., m[9]
f[4] = 2.5;
m[2] = 4;
cout << f[m[2]];      // outputs f[4], which is 2.5
```

Arrays

- When declaring an array,
  - We can initialize its values by enclosing the elements in curly braces ({…})
  - When doing so, we do not have to specify the size of the array
    - the **compiler** can figure this out

```
int a[] = {10, 11, 12, 13};          // declares and initializes a[4]
bool b[] = {false, true};            // declares and initializes b[2]
char c[] = {'c', 'a', 't'};          // declares and initializes c[3]
```

Pointers and Arrays

- Interesting connection between arrays and pointers
  - the name of an array is equivalent to a pointer to the array's initial element and vice versa
  - In the example below, c is an array of characters, and p and q are pointers to the first element of c. They all behave essentially the same

```
char c[] = {'c', 'a', 't'};
char* p = c;                        // p points to c[0]
char* q = &c[0];                    // q also points to c[0]
cout << c[2] << p[2] << q[2];       // outputs "ttt"
```

String

- C++ provides a string type in Standard Template Library (STL)
  - In order to use STL strings it is necessary to include the header file <string>
  - Since STL strings are part of the standard namespace
  - By adding the statement "using std::string," we inform the compiler that we want to access this definition directly, so we can omit the "std::" prefix
  - Lexicographical comparison based on the ASCII code
    - From the left to the right

```
#include <string>
using std::string;
// ...
string s = "to be";
string t ="not" + s;          // t = "not to be"
string u = s + "or" + t;      // u = "to be or not to be"
if (s > t)                    // true: "to be" > "not to be"
       cout << u;             // outputs "to be or not to be"
```

String

- We can append one string to another using the += operator
- Also, strings may be indexed like arrays
- The number of characters in a string s is given by s.size()

```
string s = "John";          // s = “John”
int i = s.size();           // i = 4
char c = s[3];              // c = ’n’
s += " Smith";              // now s = “John Smith”v
```

Structures

- A *structure*
  - useful for storing an aggregation of elements
  - Unlike an array, the elements of a structure may be of different types
  - Each *member*, or *field*, of a structure is referred to by a given name.

```
enum MealType { NO PREF, REGULAR, LOW FAT, VEGETARIAN };

struct Passenger {
    string    name;          // passenger name
    MealType  mealPref;      // meal preference
    Bool      isFreqFlyer;   // in the frequent flyer program?
    string    freqFlyerNo;   // the passenger's freq. flyer number
};

// Declare and initialize a variable named "pass" of this type.
Passenger pass = { "John Smith", VEGETARIAN, true, "293145" };
```

Dynamic memory

- Data structures to create objects dynamically as the need arises
  - The C++ run-time system reserves a large block of memory called the *free store* or *heap memory*
  - The operator *new* dynamically allocates the correct amount of storage
    - For an object of a given type from the free store and returns a pointer to this object
    - That is, the value of this pointer is the address where this object resides in memory

```
Passenger *p;
// ...
p = new Passenger;          // p points to the new Passenger
p->name = "Pocahontas";     // set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";
```

Dynamic memory

- C++ does not provide automatic *garbage collection*.
  - This means that C++ programmers have the responsibility of explicitly deleting all dynamically allocated objects.
- Failure to delete dynamically allocated objects can cause problems
  - If we were to change the (address) value of p without first deleting the structure to which it points, there would be no way for us to access this object
  - Having such inaccessible objects in dynamic memory is called a *memory leak*.

```
delete p;       // destroy the object p points to
```

Dynamic memory

- Arrays can also be allocated with new.
  - The system allocator returns a pointer to the first element of the array
  - A dynamically allocated array with elements of type T would be declared being of type *T
  - Cannot be deallocated using the standard delete operator.
    - Instead, the operator **delete[ ]** is used.

```
char* buffer = new char[500];      // allocate a buffer of 500 chars
buffer[3] = 'a';                   // elements are still accessed using []
delete [] buffer;                  // delete the buffer
```

References

- Pointers provide one way to refer indirectly to an object
  - A *reference* is simply an alternative name for an object
    - Given a type T, the notation T& indicates a reference to an object of type T.
    - Unlike pointers, which can be NULL, a reference in C++ must refer to an actual variable.
    - When a reference is declared, its value must be initialized.
    - Afterwards, any access to the reference is treated exactly as if it is an access to the underlying object.

```
string author = "Samuel Clemens";
string& penName = author;          // penName is an alias for author
penName = "Mark Twain";            // now author = "Mark Twain"
cout << author;                    // outputs "Mark Twain"
```

Operator Precedences

Type	*Operators*
scope resolution	namespace_name :: member
selection/subscripting function call postfix operators	class_name.member pointer->member array[exp] function(args) var++ var--
prefix operators dereference/address	++var --var +exp -exp ~exp !exp *pointer &var
multiplication/division	* / %
addition/subtraction	+ -
shift	<< >>
comparison	< <= > >=
equality	== !=
bitwise and	&
bitwise exclusive-or	^
bitwise or	|
logical and	&&
logical or	||
conditional	bool_exp ? true_exp : false_exp
assignment	= += -= *= /= %= >>= <<= &= ^= |=

Expressions

- Member Selection and Indexing

```
class name.Member           // class/structure member selection
pointer->member array       // class/structure member selection
array[ exp ]                // array subscripting
```

Arithmetic Operators

```
int exp = 3;
int rst = 0;
rst = exp + exp      // addition
rst = exp - exp      // subtraction
rst = exp * exp      // multiplication
rst = exp / exp      // division
rst = exp % exp      // modulo (remainder)
```

Increment

- The *post-increment* operator
  - returns a variable's value and then increments it by 1.
- The *pre-increment* operator
  - first increments the variables and then returns the value.

```
var++          // post increment
var--          // post decrement
++var          // pre increment
--var          // pre decrement

int a[] = {0, 1, 2, 3};
int i = 2;
int j = i++;           // j = 2 and now i = 3
int k = --i;           // now i = 2 and k = 2
cout << a[k++];        // a[2] (= 2) is output; now k = 3
```

Relational and Logical Operators

- C++ provides the usual comparison operators.
- These return a Boolean result—either true or false. Comparisons can be made between numbers, characters, and STL strings

```
exp < exp              // less than
exp > exp              // greater than
exp <= exp             // less than or equal
exp >= exp             // greater than or equal
exp == exp             // equal to
exp != exp             // not equal to

!exp                   // logical not
exp && exp             // logical and
exp || exp             // logical or
```

Casting

- Casting is an operation that allows us to change the type of a variable.
- Casting to a type of higher precision or size is often needed in forming expressions.
  - For example, division between integers always produces an integer result by truncating the fractional part.

```
int cat = 14;
double dog = (double) cat;          // traditional C-style cast
double pig = double(cat);           // C++ functional cast

int i1 = 18;
int i2 = 16;
double dv1 = i1 / i2;                       // dv1 has value 1.0
double dv2 = double(i1) / double(i2);       // dv2 has value 1.125
double dv3 = double( i1 / i2 );             // dv3 has value 1.0
```

Implicit casting

- There are many instances where the programmer has not requested an *explicit cast*, but a change of types is required.
  - For example, when numbers of different types are involved in an operation, the compiler automatically casts to the stronger type.
  - C++ allows an assignment that implicitly loses information, but the compiler usually issues a warning message.

```
int i = 3;
double d = 4.8;
double d3 = i / d;    // d3 = 0.625 = double(i)/d
int i3 = d3;          // i3 = 0 = int(d3)
// Warning! Assignment may lose information
```

References

- Further reading list and references
  - https://www.w3schools.com/cpp/
  - Type conversion
    - https://www.geeksforgeeks.org/implicit-type-conversion-in-c-with-examples/

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee