Object-Oriented Programming

Lecture 1: Introduction to C++

01286131 Object-Oriented Programming

Software Engineering Programme
International College, KMITL

Contents

Introduction to C++

Programming Languages

- Many languages have been created for specific purpose
  - Database management, AI, graphics processing, etc.
- General purpose languages can be applied to a wide range of applications
  - Operating systems, video games, embedded systems, robotics, etc.

The Evolution of C++

- C++ is one of the most important general purpose language
- C++ inherits and enhances C in many ways:
  - Still capable of generating machine code as fast as C
  - The most important enhancement to C is the addition of **"Object-Oriented"** features to the language
  - There are many more features outside **"Object-Oriented"** that C++ adopted into the language
  - At present, C++ is still evolving (with C++20 as the latest edition)

Why C++?

- Fast and flexible
- Support different styles of programming including:
  - Structured programming
  - Object-Oriented programming
  - Generic programming
- Rich and powerful libraries
- Suitable for high performance computing and embedded systems

Getting Started

A Small C++ Program

```cpp
// a small C++ program
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Program Anatomy

Comments

```
// a small C++ program
```

#include

```
#include <iostream>
```

The **main** function

```
int main()
```

Curly braces

```
int main()
{           // left brace
            // the statements go here
}           // right brace
```

Program Anatomy (Cont')

Output statement

```
std::cout << "Hello, world!" << std::endl;
```

The **return** statement

```
return 0;
```

Equivalent Output statements

```
(std::cout << "Hello, world!") << std::endl;
```

```
std::cout << "Hello, world!";
std::cout << std::endl;
```

Expressions

- An **expression** expresses what to compute
- The computation yields a **result** and may have **side effects**
- Examples:
  - `3 + 4` yields `7` and has no side effects
  - `std::cout << "Hello"` yields the reference to `std::cout` as a result of an expression (more on this later in the course)

Syntax (or Compile-Time) Errors

- Typos

```
cot << "Hello, world!\n";
cout << "Hello, world!\";
```

- Violating the language rules
- Compiler often catches errors, and reports them:

```
hello.cpp:7: error: 'cot' was not declared in this scope
hello.cpp:7: error: missing terminating " character
```

- Common strategy for fixing syntax errors is to start fixing them from the very first error at the top down to the bottom of the source code

Misspelling Words

- Misspelled words are often not obvious
  e.g. `cot` vs `cout`, `std:cout` vs `std::cout`
- C++ is case-sensitive:
  - All keywords and most library functions and definitions are lower-case

Logic or Run-Time Errors

- Program compiles fine (the code is legal)
- Program does not do what it is supposed to do
- Much harder to find
- Need a test run to find the error

```
cout << "Hell, world\n";
```

Overflow and Round-Off Errors

- Computation result is outside of the numeric range
  - either the value is too big or too small
- Example:
  - (1.0 / 3) are **truncated** when assigned to an **`int`**
  - Rounding errors

```
int n = 4.35 * 100; // n stores 434 (as integer)
```

Fixing Errors

Testing

- Validating program correctness
- Is very important for ensuring software quality

Debugging

- Finding the source of an error
- A **debugger** is a handy tool for the task

Defensive Programming

- When possible, minimize errors even before compiling the program
- Strategies include crafting programs to limit, minimize, localize errors if they do occur

Abstractions

- C++ offers a lot of features that helps you hides the low-level details of the program
- The ability to hides the details enables us to write code in a way that:
  - Focuses on the essentials for the main part
  - Implements the details on the other part
- **User-defined types** are one of the tools C++ offers that can be used for creating useful abstractions

User-Defined Types

Example: using the random number generator

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine re;
    std::uniform_int_distribution<> roller6{1, 6};

    int x = roller6(re);
    std::cout << "You got \"" << x << "\"!\n";
    return 0;
}
```

Question: Does this code work as expected?

See "A Tour of C++, 2nd ed. section 14.5" for a reference

Initialize RNG with Random Device

```cpp
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
    std::default_random_engine re{rd()};
    std::uniform_int_distribution<> roller6{1, 6};
    // simplifying the generator
    auto roll = [&]() { return roller6(re); };

    int x = roll();
    std::cout << "You got \"" << x << "\"!\n";
    return 0;
}
```

Note: You've just seen a lot of C++ features that you might not know of at this point. We'll cover them later on.

Using C++ `class` to Create an Abstraction

```cpp
  // random.hpp
#ifndef MY_RANDOM_HPP
#define MY_RANDOM_HPP

#include <random>

class Rand_int {
public:
    Rand_int(int low, int high): dist{low,high} {}

    // draw an integer number
    int operator()() { return dist(re); }

    // choose new random engine seed
    void seed(int s) { re.seed(s); }
private:
    std::default_random_engine re;
    std::uniform_int_distribution<> dist;
};

#endif /* MY_RANDOM_HPP */
```

```cpp
  // random_test.cpp
#include "random.hpp"

#include <iostream>
#include <random>
#include <vector>

int main()
{
    constexpr int max = 9;
    // make a uniform random number generator
    Rand_int rnd{0, max};

    std::random_device rd;
    rnd.seed(rd());

    // make a vector of appropriate size
    std::vector<int> histogram(max + 1);
    for (int i = 0; i != 200; ++i)
        // fill histogram with the frequencies
        // of numbers [0:max]
        ++histogram[rnd()];

    // write out a bar graph
    for (int i = 0; i != histogram.size(); ++i) {
        std::cout << i << '\t';
        for (int j = 0; j != histogram[i]; ++j)
            std::cout << '*';
        std::cout << std::endl;
    }
}
```

CLI Input

Standard Input (CLI)

- The input stream defines **>>** (**stream-extraction** operator) for expressing the data extraction operation
- Use **`std::cin`** to read an input from the console (standard input)

```
int pennies;
std::cin >> pennies;
```

- Each extraction will try to convert the input into the appropriate value of the target variable (or object)
- The data conversion process varies depending on the type of the target object

Standard Input (CLI)

- Similar to output stream the **>>** operator can be chained to read multiple values sequentially

```
cin >> pennies >> nickels >> dimes >> quarters;
```

- Inputs are separated by one or more white space characters (including new-lines)

```
81  0 4 35
```

```
81
 0
  4
35
```

Input Extraction (1)

The processing of the extraction begins with the first **>>** operator

- The program will wait for user input at the first operator as the interal buffer inside **`cin`** will be initially "empty"
- Demonstration:

```
(1) Wait for user input
```

```
(cin >> pennies) >> nickels >> dimes >> quarters;
```

```
Program State:
  [pennies: ?]
  [nickels: ?]

  [std::cin:
     buffer: <empty>]
```

Input Extraction (2)

- The first **>>** will finished only when user hit the enter key

```
(2) User input: [8][1][<space>][<space>]
```

```
(cin >> pennies) >> nickels >> dimes >> quarters;
```

```
[pennies: ?]
[nickels: ?]

[std::cin:
   buffer: [
     '8', '1', ' ', ' ']]
```

```
(3) User input: [0][<space>][4][<space>][3][5][<enter>]
```

```
(cin >> pennies) >> nickels >> dimes >> quarters;
```

```
[pennies: ?]
[nickels: ?]

[std::cin:
   buffer: [
     '8', '1', ' ', ' ',
     '0', ' ', '4', ' ', '3', '5']]
```

- After the enter key is hit, the internal input buffer will be filled with characters

Input Extraction (3)

- After completing the conversion, value will be stored to the variable as a side-effect
- The operation will yield **`cin`** which is used at the left side of the next **`<<`** operation

```
(4) 81 is stored to `pennies` with trailing spaces discarded
```

```
(cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
```

```
[pennies: 81]
[nickels:  ?]

[std::cin:
   buffer: [
     '0', ' ', '4', ' ', '3', '5']]
```

Input Extraction (4)

- The next **<<** will not wait for more input as the internal buffer is still having some data to extract

```
(5) After completing (cin >> nickels)
```

```
(cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
=> ((cin) >> dimes) >> quarters;
```

```
[pennies: 81]
[nickels:  0]

[std::cin:
   buffer: [
      '4', ' ', '3', '5']]
```

- After completing the whole statement

```
(cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
=> ((cin) >> dimes) >> quarters;
=> (cin >> quarters);
```

```
[pennies:  81]
[nickels:   0]
[dimes:     4]
[quarters: 35]

[std::cin:
   <state: good>
   buffer: []]
```

Failed Input

- If user input “10.75” instead, the extraction will fail to extract the second value and stops at reading the `'.'` character

```
User input: [1][0][.][7][5][<enter>]
```

```
(cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
```

```
[pennies: 10]
[nickels:  ?]

[std::cin:
   <state: fail>
   buffer: [
     '.', '7', '5']]
```

- The execution continues up to the end of statement without storing other values and the stream is now in a "fail" state

```
(cin >> pennies) >> nickels >> dimes >> quarters;
=> ((cin) >> nickels) >> dimes >> quarters;
=> ((cin) >> dimes) >> quarters;
=> (cin >> quarters);
```

```
[pennies:  10]
[nickels:   ?]
[dimes:     ?]
[quarters:  ?]

[std::cin:
   <state: fail>
   buffer: [
     '.', '7', '5']]
```

Working with Strings

Simple Interaction with CLI Input/Output

```cpp
// ask for a person's name, and greet the person
#include <iostream>
#include <string>

int main()
{
    // ask for the person's name
    std::cout << "Please enter your first name: ";

    // read the name
    std::string name; // define name
    std::cin >> name; // read into

    // write a greeting
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

Framing a Name

```
// ask for a person's name, and generate a framed greeting
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";
    std::string name;
    std::cin >> name;
    // build the message that we intend to write
    const std::string greeting = "Hello, " + name + "!";

    // build the second and fourth lines of the output
    const std::string spaces(greeting.size(), ' ');
    const std::string second = "* " + spaces + " *";

    // build the first and fifth lines of the output
    const std::string first(second.size(), '*');

    // write it all
    std::cout << std::endl;
    std::cout << first << std::endl;
    std::cout << second << std::endl;
    std::cout << "* " << greeting << " *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;

    return 0;
}
```

Constants

Constants

- Descriptive identifiers make a program easier to read
- The same is true for constants
- What is **0.355** in the following statement?

```
double total = bottles * 2 + cans * 0.355;
```

- **0.355** is the number of liters in a **12 oz.** can
- Defining constants can make the code more readable

```
const double BOTTLE_VOLUME = 2.0;
const double CAN_VOLUME = 0.355;
double total = bottles * BOTTLE_VOLUME + cans * CAN_VOLUME;
```

Constants

- Declared with the keyword `const`
- A constant can never be changed
- It must be initialized at definition:

```
const double CAN_VOLUME = 0.355;
```

- Often written all in capital letters

Constants — volume.cpp

```cpp
#include <iostream>

using namespace std;

int main()
{
    double bottles;
    cout << "How many bottles do you have? ";
    cin >> bottles;

    double cans;
    cout << "How many cans do you have? ";
    cin >> cans;

    const double BOTTLE_VOLUME = 2.0;
    const double CAN_VOLUME = 0.355;

    double total = bottles * BOTTLE_VOLUME + cans * CAN_VOLUME;
    cout << "The total volume is " << total << " liter.\n";
    return 0;
}
```

Constants

- Code is easier to read and less prone to errors
- Easier to modify/maintain code
- Consider changing bottles from 2 liters to 1/2 gallons
  - Without constants:

    Search for every 2, replace them with 1.893?
  - With constants:

    Just update your constant once
    (and its associated comment)

`const` vs `#define`

- In C, **`#define`** is often used to define a constant

```
#define CAN_VOLUME 0.355
```

- Or used to define a preprocessor macro

```
#define ADD(i, j) i + j
```

- **`#define`** is processed by the preprocessor before the compilation process so there is no compile-time checking and **should be avoided in C++**

```
#define CAN_VOLUME 0.355
#define ADD(i, j) i + j
double volume = 2 * CAN_VOLUME;
double volume2 = ADD(2, 5) * CAN_VOLUME;

// this is seen by the compiler
double volume = 2 * 0.355;
double volume2 = 2 + 5 * 0.355;
```

`const` vs `#define`

- In C++, we prefer **`const`** to `#define`
  - Smaller code
  - Feel natural with the rest of the language
  - Well supported by the C++ type system
- As an alternative to the preprocessor macro, **`inline` function** is almost always a better substitute in C++
  (We will explore **`inline` function** later)

Input/Output Manipulators

Input/Output Manipulators

- A helper functions that make it possible to control input/output streams (i.e. `std::cin` and `std::cout`)
- Example output produced using output stream + manipulators

```
     Income      Outcome      Balance
    1000.00                  21000.00
    1000.00      3000.00     18000.00
    1000.00       100.00     17900.00
    1000.00     10000.00      7900.00
```

- See https://en.cppreference.com/w/cpp/io/manip for a reference

`setw()` — <iomanip>

- **`setw(w)`** allows you to set the width **`w`** of a field for the next output (and **ONLY the next output**)

Example:

```
cout << setw(5) << 123 << 456 << endl;
cout << setw(5) << 123 << setw(5) << 456 << endl;
```

Output:

```
  123456
  123  456
```

`left` and `right` — <ios>

- **`left`** and **`right`** are used to arrange the position for the output

Example:

```
cout << setfill('*');
cout << left << setw(5) << 123 << 456 << endl;
cout << right << setw(5) << 123 << 456 << endl;
cout << left << setw(5) << 123 << setw(8) << 456 << endl;
cout << right << setw(5) << 123 << setw(8) << 456 << endl;
```

Output:

```
123**456
**123456
123**456*****
**123*****456
```

`setprecision()` — <iomanip>

- **`setprecision(n)`** sets the precision of an output stream to **`n`** (the number of digits after decimal) for the floating-point number output

Example:

```
for (int x = 1; x < 11; ++x) {
    cout << "precision " << x << ":\t"
         << setprecision(x) << 12.3456 << endl;
}
```

Output:

```
precision 1:    1e+01
precision 2:    12
precision 3:    12.3
precision 4:    12.35
precision 5:    12.346
precision 6:    12.3456
precision 7:    12.3456
precision 8:    12.3456
precision 9:    12.3456
precision 10:   12.3456
```

`fixed` and `scientific` — <ios>

- **`fixed`** displays trailing zeroes up to the current precision
- **`scientific`** changes the number format to **scientific format**

Example:

```
const double c = 3.1416;
cout << "Normal:\t\t" << c << "\n";
cout << "Scientific:\t" << scientific << c << "\n\n";

cout << "Fixed:\t\t\t\t\t" << fixed << c << '\n';
cout << setprecision(5);
cout << "Fixed with precision == 5:\t" << c << '\n';
cout << setprecision(9);
cout << "Fixed wiith precision == 9:\t" << c << endl;
```

Output:

```
Normal:     3.1416
Scientific: 3.141600e+00

Fixed:                      3.141600
Fixed with precision == 5:  3.14160
Fixed wiith precision == 9: 3.141600000
```

Output Stream Number Format

	Normal mode	Fixed mode	Scientific mode
Effect	Normally display value (0 are not added)	Control the number of digits after decimal point (0 might be added)	Display value in the scientific format
Precision	All digits	Digits after decimal point	Digits after decimal point
`123.45` with precision of 4	`123.5`	`123.4500`	`1.2345e+002`
`123.45` with precision of 6	`123.5`	`123.4500`	`1.2345e+002`

Note: use `cout.unsetf(flags)` to clear the number format

```
// to remove fixed format
std::cout.unsetf(std::ios_base::fixed);

// to remove scientific format
std::cout.unsetf(std::ios_base::scientific);
```

dec, hex, oct and showbase — <ios>

```
cout << hex << 31 << endl;
cout << showbase << hex << 31 << endl;
```

Output:

```
1f
0x1f
```

```
cout << oct << 31 << endl;
cout << showbase << oct << 31 << endl;
```

Output:

```
37
037
```

```
cout << dec << 0x1F << endl;
cout << dec << 037 << endl;
```

Output:

```
31
31
```

`dec`, `hex`, `oct` and `showbase` — <ios>

```
cout << hex << 31 << endl;
cout << showbase << hex << 31 << endl;
cout << noshowbase << hex << 31 << endl;
```

Output:

```
1f
0x1f
1f
```

Note: use `noshowbase` to cancel the effect of `showbase`

`showpos` and `noshowpos` — <ios>

- **`showpos`** displays the **+** sign for the non-negative number

```
double c = 3.1416;
double d = -3.1416;
cout << showpos << c << '\t' << d << endl;
cout << noshowpos << c << '\t' << d << endl;
```

Output:

```
+3.1416 -3.1416
3.1416  -3.1416
```

`showpoint` and `noshowpoint` — <ios>

- **`showpoint`** displays the decimal point in a floating-point value
- **`noshowpoint`** displays the decimal point when it is necessary

```
double f = 10.0;
cout << f << endl;
cout << showpoint << f << endl;
cout << noshowpoint << f << endl;
```

Output:

```
10
10.0000
10
```

`boolalpha` and `noboolalpha` — <ios>

```
bool a = true;
cout << a << endl;
cout << boolalpha << a << endl;

a = false;
cout << a << endl;
cout << noboolalpha << a << endl;
```

Output:

```
1
true
false
0
```

List of Manipulators

Manipulator	Purpose
`setw(w)`	Set the width **`w`** of a field for the next output
`left`	Set left position arrangement for the output
`right`	Set right position arrangement for the output
`setprecision(n)`	sets the precision of an output stream to **`n`** for the floating-point number output
`fixed`	Insert float-point values in fixed format
`scientific`	Insert float-point values in scientific format
`uppercase`	Use uppercase letters on insertion
`nouppercase`	Don’t use uppercase letters on insertion

List of Manipulators (Cont')

Manipulator	Purpose
`dec`	Insert values in decimal (base 10) format
`hex`	Insert values in hexadecimal (base 16) format
`oct`	Insert values in octal (base 8) format
`showbase`	Show the base of value
`noshowbase`	Do not prefix a value with its base
`showpos`	Insert the **+** sign before non-negative number
`noshowpos`	Do not insert the **+** sign before non-negative number
`showpoint`	Show the decimal point in a floating-point value
`noshowpoint`	The decimal point is only shown when necessary
`boolalpha`	Insert boolean value as text (`true` or `false`)
`noboolalpha`	Insert boolean value as number (`1` or `0`)