Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

September 6, 2018

Review of C++ programming

- This lecture contains a collection of "useful things" and lessons.
- This document is mostly a review of material about programming in C++.
- Some new material is also included, which extends previously taught material.
- Specifically, we introduce the keyword **const.**

1 Cars

Do you own a car, or at least have you driven a car? *Why do I ask?* Look at the dashboard of a car and the pedals. It is the same design in all cars from Rolls Royce down to the cheapest cars.

That is an **interface.**

You know, even without reading the user manual, how to operate the car. The steering wheel, turn indicator lights, headlights, wipers, speedometer and fuel gauge, pedals for accelerator and brake. You know how to drive a car without being an auto mechanic.

Why do automobile companies do that?

Because it benefits everyone. If you had to learn a new user design for every car you drive, there would be mistakes in operation and terrible accidents. It increases sales for all the companies to follow the pattern.

Because of this interface, which is adopted by all, the rental car business can exist. You can go on a business trip or vacation and get a rental car. You can drive that rental car even though you have never driven that make or model before. That is an example of a successful interface. Because of that, there is a great increase in business and also the vacation industry. A good design has side effects which go far beyond the original purpose.

- It was not always that way with cars. Mercedes–Benz is named after two people Daimler and Benz (Mercedes was the daughter of Daimler).
- When Carl Benz designed his first car it was a tricycle.
  `https://www.daimler.com/company/tradition/company-history/1885-1886.html`
- In the early days of automobiles, there were designs where the driver sat in the back and the passengers sat in front to enjoy the view.
- Eventually things evolved to the design we know today.
- A good design is not obvious from the beginning.
- A successful design by itself inspires changes in technology.

2 Object oriented programming (OOP)

- The concept of a good interface and product design, etc., apply also to software.
- On many web pages (or applications) there is a pulldown menu which choices such as "Open file, new file, save, save as, print, ..." *You know what to do, without having to read a user manual.* And you also do not need to be an expert programmer.
- The concept of a pulldown is itself an interface. It is a successful software design.
- There are basic programming concepts, to achieve the goal of a good design.
- But programs have to be written in a specific language.
- **Software designs are independent of any language.**
- We shall learn C++ in this class. Java is taught in CS212.
- In both classes you will be taught something called **Object Oriented Programming (OOP).**
- You will see how the same basic concepts are expressed in the two languages.
- **OOP is a set of concepts.**
- But programs must be written in a specific language (C++ in this course).
- Learn to be a good programmer in both languages.

3 Review of basics: example = GCD greatest common divisor

We review basic C++ programming concepts. We employ the example of the calculation of the greatest common divisor (GCD) of two positive integers as an example.

For two positive integers a and b, there is a simple and efficient algorithm to compute their greatest common divisor (gcd). It dates from classical Greece and we call it "Euclid's algorithm" although Euclid himself never claimed to invent it. *But he wrote it down, and his work has survived.* It is possible that other ancient civilizations also knew it (China, India, Egypt, etc.?) but we do not know. The algorithm is as follows:

- Our first test is to check that $a \geq b$.
- **If $a < b$, return $\gcd(b, a)$.**
- Next caclulate the **remainder after integer division**

$$c = a \ \% \ b \qquad \text{(remainder after integer division)}\,.$$

- If c equals 0, then return b (because b divides a).
- If c equals 1, then return 1.
- Else if $c > 1$ then proceed recursively and compute $\gcd(b, c)$.
- ***The process must terminate because the integers are positive and always get smaller and eventually must reach 0 or 1.***

A simple implementation of the above algorithm is as follows.

```
int gcd(int a, int b)
{
  if (a < b)
    return gcd(b,a);

  int c = a % b;

  if (c == 0) {
    return b;
  }
  else if (c == 1) {
    return 1;
  }

  return gcd(b,c);
}
```

Compile and run it and test it. It works (for positive integers).

4 Recursion

- The function `gcd(...)` employs recursion.
- Let us call `gcd(...)` using the following main program and trace the recursion.

```
#include <iostream>

int gcd(int a, int b)
{
  // etc
}

int main()
{
  int x = 18;
  int y = 30;
  std::cout << "gcd(x,y) = " << gcd(x,y) << std::endl;
  return 0;
}
```

- Then `gcd(...)` is initially called with $a = 18$ and $b = 30$.

$\gcd(18, 30)$ $a = 18,\ b = 30$				level 1
	$\gcd(30, 18)$ $a = 30,\ b = 18$ $c = 30\,\%\,18 = 12$			level 2
		$\gcd(18, 12)$ $a = 18,\ b = 12$ $c = 18\,\%\,12 = 6$		level 3
			$\gcd(12, 6)$ $a = 12,\ b = 6$ $c = 12\,\%\,6 = 0$	level 4
			return 6	level 4
		return 6		level 3
	return 6			level 2
return 6				level 1

- Note that at each level of the recursion, *the variables* (a, b, c) *are local to that level only.*
- They occupy a different memory location than the (a, b, c) variables at the other levels.
- This fact is related to the concept of **call by value** which we shall discuss below.

5 Call by value, call by reference

- Consider the following functions to swap two integers.

```
#include <iostream>

void swap1(int u, int v)
{
  int t = u;
  u = v;
  v = t;
}

void swap2(int &u, int &v)
{
  int t = u;
  u = v;
  v = t;
}

int main()
{
  int a = 3;
  int b = 4;
  int c = 5;
  int d = 6;
  cout << "original:  "  << a << "   " << b  << "   " << c  << "   " << d << endl;

  swap1(a,b);
  cout << "swap1:  "  << a << "   " << b << endl;

  swap2(c,d);
  cout << "swap2:  "  << c << "   " << d << endl;
  return 0;
}
```

- **See next page.**

- The function `swap1` **does not work.**
  1. The reason is because C++ employs **call by value.**
  2. The values of u and v inside `swap1` are **copies** of a and b in the main program.
  3. Swapping the values of u and v inside `swap1` has no effect on the values of a and b in the main program.
- The situation is different for `swap2`.
  1. The reason is because the inputs to `swap2` are **references.**
  2. The variables u and v in `swap2` are **bound to the variables c and d in the main program.**
  3. Swapping the values of u and v inside `swap2` therefore swaps the values of c and d in the main program.
  4. Hence the function `swap2` **works correctly.**

6 Keyword `const`, constant call by reference

- We introduce the keyword `const`.
- Consider the following functions to print an input integer.

```
void print_call_by_val(int x)
{
  cout << x << endl;
}

void print_call_by_ref(int &y)
{
  cout << y << endl;
}
```

- Clearly, they both work.
  1. However, the input to the function `print_call_by_ref` is a *reference.*
  2. This introduces the risk that if the value of y were inadvertently changed inside the function, it would affect the value of a variable in the calling application.
  3. Although the above function is only a simple example, in a more complicated function such a risk is a significant possibility.
- There is a third way to formulate the input to the print function.
- It is known as **constant call by reference.**
  1. The function signature is as follows.

     ```
     void print_const_call_by_ref(const int &z)
     {
       cout << z << endl;
     }
     ```

  2. The input argument z has been tagged as a **const reference.**
  3. The "`const`" keyword is a promise to the compiler that **the value of z will not be changed inside the function.**
  4. Hence z is a reference, and is therefore bound to a variable in the calling application, but we have given a pronise that the value of z will not be changed inside the function.
  5. If for any reason code is written which changes the value of z inside the function, it will be flagged as a compilation error.
  6. The resulting code will not compile.
- Note that the `const` keyword does not speed up program execution or save on memory.
- The `const` keyword allows the compiler to perform extra checking of the code.

7 Reference as return value

- We know that the input to a function can be a reference.
- **The return value of a function can also be a reference.**
- Consider the following function.

```
int& returnReference1(int &x)
{
  return x;                  // return value is reference to x
}
```

- **The return type is a reference to `int`.**
- The syntax is `int&`.
- This is what the function does:
  1. The input reference x is bound to the value of a variable in the calling application.
  2. The output (function return value) is a reference which is bound to x,
  3. **Therefore the return reference is bound to the same variable as x.**
  4. Here is a main program.

```
int main()
{
  int a = 3;
  int &b = returnReference1(a);          // b is reference to a
  cout << a << "   " << b << endl;       // print 3   3

  b = 4;                                 // also changes value of a
  cout << a << "   " << b << endl;       // print 4   4
  return 0;
}
```

  5. The reference b is bound to the return value of the function, which is bound to a.
  6. Hence overall b is bound to a.
  7. Therefore setting $b = 4$ also changes the value of a to $a = 4$.

- **See next page.**

- A reference as a function return value can be tricky, though.
- Consider the following function.

```
double& returnReference2()
{
  double y = 1.2345;
  return y;                      // reference to local variable  *** bad ***
}
```

- **The return type is a reference to `double`.**
- The syntax is `double&`.
- **However, this function has a serious problem.**
- This is what the function does:
  1. The output (function return value) is a **reference to a local variable** in the function.
  2. The local variable y **goes out of scope** when the function exits.
  3. **Therefore the reference is bound to deallocated memory.**
  4. ***This is very bad.***
  5. Here is a main program.

```
int main()
{
  double d = returnReference2();
  cout << d << endl;
  return 0;
}
```

  6. The program will compile.
  7. The program may or may not crash when executed.
  8. **However, the value of d is an unpredictable undefined number.**
- Hence returning a reference is possible, but we must be careful.

8 Function overloading

- Here are two functions to print output and a main program to call them.

```
#include <iostream>

void print_const(const int &z)
{
  cout << z << endl;
}

void print_const(int n, const int a[])
{
  for (int i = 0; i < n; ++i)
    cout << a[i] << endl;
}

int main()
{
  int n = 3;
  int a[] = {7, 10, 12};

  print_const(n);
  print_const(n, a);
  return 0;
}
```

- The first function employs constant call by reference.
- The second function applies the keyword `const` to an array, and thereby gives a promise that the elements of the array a will not be changed inside the function.
- The other main point of this example is that **both functions have the same name.**
- This is called **function overloading.**
- The compiler tells the functions apart by their different input signatures.
- We shall see many examples of the use of `const` and of overloading, later in this course.

9 Forward declarations

- Suppose we have two functions `f1` and `f2`.
- Suppose also that (i) `f1` internally calls `f2`, (ii) `f2` internally calls `f1`.
- Clearly, this is a problem.
  1. If we write the code for `f1` first, then we find that `f2` is not defined yet.
  2. If we write the code for `f2` first, then we find that `f1` is not defined yet.
- To solve this problem, C++ supports **forward declarations.**
  1. A forward declaration is the function signature, written at (or near) the top of the file.
  2. **A forward declaration tells the compiler that a function with that signature exists "somewhere" in the project.**
  3. The compiler therefore knows that the function code will be supplied later.
- **After the forward declarations are written,** the function bodies of `f1` and `f2` can be **written anywhere below in the file.**
- Example of forward declarations for `f1` and `f2`.

```
string f1(int a);                    // forward declaration (top of file)
string f2(int a);                    // forward declaration (top of file)

string f1(int a)
{
    f1 function code
}
string f2(int a)
{
    f2 function code
}
```

- **See next page.**

- The following C++ program demonstrates the use of forward declarations for `f1` and `f2`.
  1. If input `a >= 0`, then `f1` returns a string, else calls `f2`.
  2. If input `a < 0`, then `f2` returns a string, else calls `f1`.

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

string f1(int a);                          // forward declaration
string f2(int a);                          // forward declaration

int main()
{
  cout << f1(3) << endl;             // prints "f1"
  cout << f1(-3) << endl;            // prints "f1 first, f2"
  cout << f2(4) << endl;             // prints "f2 first, f1"
  cout << f2(-4) << endl;            // prints "f2"
  return 0;
}

string f1(int a)                           // f1 function body
{
  if (a >= 0) {
    return "f1";
  }
  else {
    string s = string("f1 first, ");
    return s + f2(a);
  }
}

string f2(int a)                           // f2 function body
{
  if (a < 0) {
    return "f2";
  }
  else {
    string s = string("f2 first, ");
    return s + f1(a);
  }
}
```