Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

May 3, 2018

1 Lecture 1

- This lecture contains a collection of "useful things" and lessons.
- The goal is to provide a foundation to organize our ideas about programming in C++.
- **Note: Although this document is called "Lecture 1" the material will be taught in class in more than one lecture.**
- In other words, ***do not panic.***

This file is still under development.

1.1 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.

1.2 Object oriented programming (OOP)

- So it is with 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 lean 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.
- And learn the OOP concepts.

1.3 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**

$$c = a \ \% \ b \qquad \text{(remainder after integer division)}\,. \tag{1.3.1}$$

- 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).

There are several points to note (learn?) in the above code:

- **Conditional test "if"** `if (a < b) return gcd(b,a);`
  1. If the **scope** of an "if" statement consists of only one line, we do not need braces to enclose that line.
  2. The concept of **scope**.
  3. The term "scope" means a set of executable statements (lines of program code) which all belong in one block, and are distinct from other blocks.
  4. It is simplest to explain with some examples (later in this lecture).
- **Operator** $a \,\%\, b$
  1. The "%" operator acts on two integers a and b and returns the remainder after integer division of a by b.
  2. So for example $5 \,\%\, 3$ equals 2 (remainder is 2) and $6 \,\%\, 3$ equals 0 (because 3 divides 6 so the remainder is zero).
  3. By definition, the value of c is an integer in the interval $0 \leq c \leq b - 1$.
- **Conditional test "if/else"** `if (c == 0) ...else if (c==1)`
  1. Here we see an example of multiple branches.
     (a) *If the first test passes,* **the second test "else if" will not be executed.**
     (b) *If the first test fails,* then the second test "else if" will be evaluated.
  2. Scope again: note that I employed braces but they are not necessary in the above code. The following also works and achieves the same result:

```
if (c == 0)
  return b;
else if (c == 1)
  return 1;
```

- **Recursion** `return gcd(b,c);`
  1. The last line is an example of **recursion.**
  2. **The function calls itself (with different input arguments).**
  3. **The first line is also an example of recursion:** `if (a < b) return gcd(b,a);`
  4. Recursion is allowed in C++ and many other programming languages.
  5. Recursion is a useful tool, *but it can lead to an infinite loop.* Be careful.

1.4 Recursion

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

```
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.

1.5 Scope

- **Homework/lab — not necessary?**
- The following is a working C++ program, which you should compile and run:
- The function `scope()` contains six scopes, some of which are nested.
- Scope 3a is nested inside scope 3 which is itself inside scope 1.
- We say **the variable k in scope 1 is visible inside scopes 2, 3, 3a, 5 and 6.**
    1. The variable k in scopes 2, 3, 3a, 5 and 6 is the same variable k in scope 1.
    2. Changing the value of k in scopes 2, 3, 5 and 6 changes the value of k in scope 1.
- The variable k in scope 4 is a **local variable which hides k in scope 1.**
    1. The variable k in scope 4 **can be of a different type from k in scope 1.**
    2. *A variable in a lower level scope (more local) hides a variable of the same name from a high level scope (more global).*
    3. Changing the value of k in scope 4 **does not change** the value of k in scope 1.
- The local variables x and k in scope 4 are not visible in the program outside scope 4.
- A variable is only visible within its scope and in lower level nested scopes. It is not visible outside of its scope.

```
#include <iostream>

void scope()
{
  // scope #1
  int k = 3;
  int n = 4;

  if (k > 2) {
    // scope #2
    // braces necessary, scope contains multiple lines
    k = k + 1;
    std::cout << "scope 2: k = " << k << std::endl;
  }

  // scope 1 again
  // QUESTION 1 (part 1): what is the value of k in scope 1?

  for (int i = 0; i < n; ++i) {
    // scope #3
    // QUESTION 3: what is the value of k in scope 3?

    double x = k + i*0.5;         // k from scope #1 is visible in scope #3
    std::cout << "scope 3: x = " << x << std::endl;

    if (x > 4.7) {
      // scope #3a nested inside scope 3
      // QUESTION 3a: what is the value of k in scope 3a?

      std::cout << "scope 3a: x = " << x << std::endl;
    }
  }

  for (int j = 0; j < n; ++j) {
    // scope #4
    double k = 2.2 + 3.3*j*j; // local variable "k" in scope #3 overrides "k" from scope #1
    std::cout << "scope 4: k = " << k << std::endl;
  }

  // scope #1 again
  // QUESTION 1 (part 2): what is the value of k in scope 1?

  // scope #1 again
  if (k > 7) {
    // scope #5
```

```
    k = n + 2;
    std::cout << "scope 5: k = " << k << std::endl;
  }
  else {
    // scope #6
    k = n - 3;
    std::cout << "scope 6: k = " << k << std::endl;
  }

  // scope 1 again
  // QUESTION 1 (part 3): what is the value of k in scope 1?
}

int main()
{
  scope();
  return 0;
}
```

1.6 Call by value, call by reference

- **Homework/lab ?**
- Write a function to swap two integers.

```
void swap1(int a, int b) {
  int c = a;
  a = b;
  b = c;
}

void swap2(int &a, int &b) {
  int c = a;
  a = b;
  b = c;
}

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;
}
```

- **Explain the difference between `swap1` and `swap2`.**
- **Explain what the program will print.**

1.7 Conversion decimal to binary or hex, etc.

Homework/lab ? Use of array and/or string to hold converted digits?

Computers store numbers internally as binary digits (or hexadecimal). But they display the numbers on screen in decimal format. We also input numbers (in online forms, etc.) in decimal format. Hence conversion of digits from decimal to/from binary or hex is a frequently performed operation. Hence it is essential to do it efficiently. For simplicity we consider only positive numbers.

1.7.1 Decimal integer to binary

We begin with with a decimal integer. Proceed recursively:

- Divide the number by 2 (integer division) and store the remainder.
- Divide the quotient by 2 (integer division) and store the remainder.
- Repeat until the quotient reaches zero. This must happen and the algorithm will terminate.
- Read the remainders in reverse order, that is the binary representation of the decimal number.
- Example: $a = 19$

19	
9	1
4	1
2	0
1	0
0	1

- Hence $19_{\text{dec}} = 10011_{\text{bin}}$.

1.7.2 Decimal integer to hex

The fundamental algorithm is exactly the same, but we perform integer division by 16 (not 2). Proceed recursively:

- Example: $a = 30$

30	
1	14
0	1

- The new twist here is that the remainder takes values from 0 through 15. Hence we need a lookup array for hex digits `hex_digits` $= \{$`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F`$\}$. (Hence the output is alphanumeric.) In the above example, the lookup array says 14 maps to 'E' so

$$30_{\text{dec}} = 1E_{\text{hex}}\,.$$

1.7.3 Decimal integer to base b

Obviously the same algorithm works for any base $b \geq 2$. If $b > 10$ then we require a lookup table to output alphanumeric digits, but that is not a problem.