Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

September 6, 2018

C++ Classes: Part I

- This lecture contains an introduction to **C++ classes.**
- Classes are probably the single most important feature of C++.
- Classes allow us to implement the features of **object oriented programming (OOP).**
- Note that object oriented programming (OOP) is a set of *programming concepts.*
  1. Object oriented programming is independent of any language.
  2. For example Java also implements object oriented programming.
  3. In this class we shall learn how to implement object oriented programming using C++.
  4. The first step is to learn about C++ classes.

1 Introduction

- At the simplest level, a **class** in C++ is a generalization of the basic (or primitive) data types, such as `char`, `int`, `long`, `double`, etc.

  1. We can create (or **instantiate**) multiple variables of a data type, such as `int` or `double`:

     ```
     int i, j, k;
     double x, y, z;
     ```

  2. We can also make copies of variables (for example call by value in function calls).
  3. We can assign variables (e.g. $i = j$ or $x = y$).
  4. Variables are destroyed when they go out of scope. (In this context, pay attention to dynamic memory allocation and release.)

- The C++ language is designed to support the same basic operations for a user-defined class.

  1. Let us suppose we have a user-defined class called `MyClass`.
  2. The variables are called **objects** of a class.
  3. **Create:**
     We can create (or instantiate) multiple objects of a class:

     ```
     MyClass obj1, obj2, obj3;
     ```

  4. **Copy:**
     We can make copies of objects (call by value in function calls).
  5. **Assign:**
     We can assign objects: `obj1 = obj2`.
  6. **Destroy:**
     Objects are destroyed when they go out of scope. The memory allocated for them is deallocated.

- Hence we must first learn how to declare a C++ class, then learn how to (i) create objects, (ii) make copies, (iii) assign objects, (iv) destroy objects when they go out of scope.

- We shall learn how to do all of these things.

- We can also declare arrays and pointers for the primitive data types:

  ```
  int a[10];
  int *iptr;
  double d[100];
  double *pd;
  ```

- We can similarly declare arrays and pointers for user-defined classes.

  ```
  MyClass array_mc[25];
  MyClass *p_mc;
  ```

- We shall learn how to do all of these things.

2 Declaration of C++ class

- At the simplest level, a class is a **container of data.**
- **As an example, let us declare a class `Point`.**
- The class `Point` contains two coordinates x and y, both of type `double`.

```
class Point {                          // name
public:                                // keyword "public"
  double x, y;                         // data
};                                     // semicolon
```

- We **declare a class** by writing "`class Point`" (the name of the class).
- The class contains two data members x and y, which are both of type `double`.
  1. Obviously a class can contain other data members (`int or string`, etc. also arrays).
  2. ***A class can even contain objects of other classes.*** We shall see examples later.
- *What does the keyword "*`public`*" mean?.*
  1. The keyword **public** means *all the items in the class are visible to outside applications.*
  2. Any other part of the program can read/write the values of x and y.
  3. There are also two other keywords **private** and **protected**. They are more complicated. We shall study them later.
- Note the semicolon "**};**" after the closing brace at the end of the declaration.
  1. The class declaration is terminated by a **semicolon at the end**.
  2. The semicolon at the end looks (and is) peculiar but it is necessary.
  3. **The code will not compile without that final semicoloc.** (Try it.)
  4. The reason for the final semicolon is because C++ supports some obscure features, which are not important and which we shall never use. It is explained in textbooks.

3 Example program #1

- Here is a working C++ program to declare the class `Point` and a main program to use it.

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

class Point {
public:
  double x, y;
};

int main()
{
  Point apt;
  apt.x = 2.5;
  apt.y = 2.0;

  cout << apt.x << endl;
  cout << apt.y << endl;
  return 0;
}
```

- **Although short and simple, the above code contains several items to learn.**
- **See next page(s).**

4 Default actions by compiler

- Let us examine what happens in the main program in Sec. 3.
- The statement "`Point apt`" creates or **instantiates an object of the `Point class`.**
- *However, we did not specify a rule to create objects of the class* `Point`*.*
- Nevertheless, the compiler knows how to create an object of the class `Point`.
- **If we do not specify explicit rules for a user-defined C++ class, the compiler automatically generates functions to perform the following operations by default:**
  1. Create (instantiate) an object.
  2. Make a copy of an object (recall "call by value").
  3. Assign an object (recall "$x = y$" assignment).
  4. Destroy an object when it goes out of scope.
- ***Note: the default actions by the compiler may not always be what we want.***
  1. This should be obvious.
  2. Later, we shall learn how to write functions to supply explicit rules for instantiation, copy, assignment and destruction.
- The compiler also automatically knows how to instantiate the following:
  1. Array of objects.
  2. Pointers to objects.
- The compiler also automatically generates the operators `new` and `delete` for dynamic memory allocation and deallocation.
- The compiler also automatically generates the `sizeof` operator to compute the size of an object.
  1. *The size of an object is not necessarily the sum of the sizes of the data members in the class.*
  2. Always use the `sizeof` operator and do not attempt to outsmart the compiler.
  3. We write `sizeof(Point)` in the same way that we write `sizeof(int)`, etc.

5 Dot operator: "member of" operator

- We continue our analysis of the main program in Sec. 3.
- The dot operator is the **"member of" operator** for a C++ class.

```
apt.x                    // data member "x" in object "apt"
apt.y                    // data member "y" in object "apt"
```

- The meanings of the following statements should be obvious:

```
apt.x = 2.5;       // set the value of "x" in object "apt" to the value 2.5
apt.y = 2.0;       // set the value of "y" in object "apt" to the value 2.0
```

- The "`cout`" statements obviously print the values of `apt.x` and `apt.y`.

```
cout << apt.x << endl;                // print the value of apt,x
cout << apt.y << endl;                // print the value of apt,y
```

6 Dot operator: operator precedence

- **The dot operator binds very strongly.**
  1. The dot operator has a higher precedence than $+, -, *, /$ and $++$ and $--$.
  2. Only the parentheses "()" and bracket "[]" operators have higher precedence than dot.
- Here is another working C++ program, with expressions involving the dot operator.

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

class Point {
public:
  double x, y;
};

int main()
{
  Point apt;
  apt.x = 2.5;
  apt.y = 2.0;

  cout << "point x = " << apt.x << endl;      // same as example 1
  cout << "point y = " << apt.y << endl;

  double xy_plus   = apt.x + apt.y;   // dot binds more strongly than +, -, *, /
  double xy_minus  = apt.x - apt.y;
  double xy_mult   = apt.x * apt.y;
  double xy_div    = apt.x / apt.y;

  cout << "xy_plus   = " << xy_plus  << endl;
  cout << "xy_minus  = " << xy_minus << endl;
  cout << "xy_mult   = " << xy_mult  << endl;
  cout << "xy_div    = " << xy_div   << endl;

  apt.x++;                     // dot binds more strongly than ++
  --apt.y;                     // dot binds more strongly than --

  cout << "apt.x++ = " << apt.x << endl;
  cout << "--apt.y = " << apt.y << endl;
  return 0;
}
```

6.1 Dot and operators $+, -, *, /$

- These are relatively obvious.
- Obviously the dot in "`apt.x`" and "`apt.y`" binds more strongly than the "+" operator.
- Hence the expression "`apt.x + apt.y`" returns the sum of `apt.x` and `apt.y`.
- Similarly for the expressions `apt.x - apt.y` and `apt.x * apt.y` and `apt.x / apt.y`.
- It would not make sense to do anything else.

6.2 Dot and operators $++, --$

- The $++$ and $--$ operators are more complicated.
- Obviously there are totally four expressions `++apt.x`, `--apt.x`, `apt.x++`, and `apt.x--` (similarly for `apt.y`). The two cases in the above example are sufficient to illustrate the basic ideas.
- The expression "`apt.x++`" means the following:
  1. First apply the dot operator and get `apt.x`.
  2. Then apply the $++$ operator and increment the value of x in `apt`.
  3. *Technically, there are details because the $++$ in the above expression is a "postfix" operator. We ignore such subtleties for now.*
- The expression "`--apt.y`" means the following:
  1. First apply the dot operator and get `apt.y`.
  2. Then apply the $--$ operator and decrement the value of y in `apt`.
  3. **Note that the $--$ operator does not apply to the object `apt` itself (it does not make sense to "decrement the object `apt`").**
  4. We must read "`apt.y`" first and then apply the $--$ operator apply to y in `apt`.
- **In general, expressions such as `apt.x++` and `--apt.y` are confusing to read.**
  1. **I suggest that you avoid writing such code.**
  2. One can find other ways to do the job.

7 Point class and dot operator (Part 2 for both)

- *A class can contain more than just data.*
- **A class can also contain functions.**
- Functions which belong to a class are called **class methods**.
- Here is another working C++ program.
  1. We have added a method `set(...)` to set the values of x and y.
  2. We have added a method `print()` to print the values of x and y.
  3. We also see something new.
  4. The **"`const`" keyword is applied to a function** (not a function argument).
- **The dot operator is used to access the class methods.**

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

class Point {
public:
  // methods
  void set(double a, double b)
  {
    x = a;
    y = b;
  }

  void print() const
  {
    cout << "print x,y    " << x << "    " << y << endl;
  }

  // data
  double x, y;
};

int main()
{
  Point bpt;
  bpt.set(1.23, 4.56);      // dot operator for class methods
  bpt.print();              // dot operator for class methods
  return 0;
}
```

- **See next page(s).**

7.1 Dot operator and class methods

- The dot operator applies to class methods using the same syntax as for class data.
- Hence we write `bpt.set(...)` and `bpt.print()`.
- **A class method must have a return type** (`void` in these examples).
  1. **A method is a function just like any C++ function.**
  2. The only difference is that a method belongs to a class.
- The method `set(double a, double b)` is relatively obvious. It takes two input arguments, and it assigns the values $x = a$ and $y = b$.
- **The `print()` method has been tagged `const`.**
- The use of `const` to tag a class method means **the method will not change the values of the internal data in the class.**
  1. Obviously `print()` does not change the values of x and y.
  2. Hence `print()` can be tagged as `const`.
  3. *However* `set(...)` *changes the values of x and y.*
  4. **Hence `set(...)` cannot be tagged as `const`.**

8 Accessor/mutator methods

- Let us learn about **accessor** and **mutator** methods.
- **Accessor** methods return the values of data members in a class.
  1. That is why accessor methods are also called "get" methods.
  2. We have written two accessor methods `getx` and `gety`.
  3. Accessor methods can be tagged as `const` because they do not change the values of the data members in a clsss.
- **Mutator** methods set (or "mutate") the values of data members in a class.
  1. The method "`set`" is obviously a mutator method.
  2. Mutator methods are called "set" or "put" methods.
- Here is another working C++ program, which uses accessor and mutator methods.

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

class Point {
public:
  double getx() const { return x; }              // accessor method
  double gety() const { return y; }              // accessor method

  void set(double a, double b)                   // mutator method
  {
    x = a;
    y = b;
  }

  void print() const
  { cout << "print x,y    " << x << "    " << y << endl; }

  // data
  double x, y;
};

int main()
{
  Point bpt;
  bpt.set(1.23, 4.56);
  bpt.print();
  cout << "get:   " << bpt.getx() << "     " << bpt.gety() << endl;
  return 0;
}
```

9 Pointer

- **The compiler automatically knows how to create pointers for a user-defined class.**
- **The compiler also automatically sets up the operators `new` and `delete`.**
- Here is the same C++ program as in Sec. 8, but we instantiate a pointer and employ dynamic memory allocation.
- **Notice the "arrow operator" `->`.** This will be explained below.

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

class Point {
  // etc

int main()
{
  Point * p = new Point;
  p->set(1.23, 4.56);
  p->print();
  cout << "get:   " << p->getx() << "     " << p->gety() << endl;
  delete p;
  return 0;
}
```

10 Arrow operator: "pointer member of" operator

- The "**arrow operator ->**" is the "pointer member of" operator.
- **It is the partner of the dot operator for objects.**
- The following statements are equivalent.

```
p->print();
(*p).print();              // equivalent
```

- However it is simpler and more convenient to use the arrow operator `->` for pointers.
- **The arrow operator "->" and dot operators "." have the same level of operator precedence.**

11 Keyword "private"

- An obvious weak feature of the `Point` class is that everything in the class is accessible to outside calling applications (keyword "public").
- Here is a revised class design.
- It has been named `Point1` to distinguish it from the previous version.
- It is the same as `Point` except that the **keyword "`private`"** has been introduced.
  1. **The data members x and y are now "private" data members of the class.**
  2. The keyword "`private`" means that **outside applications cannot directly access the values of x and y.**
  3. To access the class data, outside applications must go through a controlled interface (the accessor and mutator methods).
  4. *We can therefore control outside access to the internal class data.*
- **Anything which is tagged "public" (data or methods) is accessible to outside applications.**
- **Anything which is tagged "private" (data or methods) is accessible inside the class only.**
- It is also possible to write private class methods.

```
class Point1 {
public:                                                 // keyword public

  double getx() const { return x; }
  double gety() const { return y; }

  void set(const double &a, const double &b)
  {
    x = a;
    y = b;
  }

  void print() const
  {
    cout << "print x,y    " << x << "    " << y << endl;
  }

private:                                                // keyword private
  double x, y;
};
```

12 Example program for keyword "private"

- The function "`distance`" calculates the distance between two points a and b:

$$d(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \,.$$

- **However, because x and y are private data members, they cannot be accessed directly in the function `distance`.**
  1. Their values must be obtained using the accessor methods `getx` and `gety`.
  2. The methods `getx` and `gety` are public (also `set` and `print` used in the main program).
- **Note also the inputs to the function `distance` are tagged as "`const`" references.**
  1. The `const` keyword gives a promise to the compiler that the code inside the function "`distance`" will not change the data values in a and b.
  2. This promise is satisfied because "`getx`" and "`gety`" are `const` class methods, which do not change the data values in a and b.
- Outside applications must employ the public methods to manipulate the class data.
- Hence we provide an interface to control how outside applications can use the class data.

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

class Point1 {
  // etc
};

double distance(const Point1 &a, const Point1 &b)
{
  double dx = a.getx() - b.getx();
  double dy = a.gety() - b.gety();
  return sqrt(dx*dx + dy*dy);
}

int main()
{
  Point1 p1, p2;
  p1.set(1.5, 2.5);
  p2.set(4.5, -1.5);

  double d = distance(p1, p2);
  cout << "distance = " << d << endl;
  return 0;
}
```

13 Inline and non-inline functions

- The class definitions in the above examples are **inline definitions.**
- An **inline method** is one where the body of the function (class method) is written **inside the declaration of the class itself.**
- This is not necessarily a good idea for a class where the methods contain a lot of code. The declaration of the class becomes very long and is difficult to read and maintain.
- It is possible to write the class declaration in a header file, and to write the definitions of the function bodies (class methods) in a separate cpp file (or multiple cpp files).
- There is also the nice feature that if even the code in one cpp file is edited, the rest of the project does not need to be recompiled.
- If the code for an inline method is changed, all the code for the entire class must be recompiled.

13.1 Class `Point2`: non-inline functions

- Here is the C++ code for a class `Point2`.
- It is the same as `Point1` but `set(...)` and `print()` have been written as non-inline methods.
- The class declaration can be placed in a header file.

```
class Point2 {
public:
  const double& getx() const { return x; }       // inline
  const double& gety() const { return y; }       // inline

  void set(const double &a, const double &b);    // not inline
  void print() const;                            // not inline

private:
  // data
  double x, y;
};
```

- **This is how non-inline code for the function definitions is written.**

```
void Point2::set(const double &a, const double &b)
{
  x = a;
  y = b;
}

void Point2::print() const
{
  cout << "print x,y    " << x << "    " << y << endl;
}
```

- **The prefix "`Point2::`" identifies the function as a method of the class `Point2`.**
- The non-inline code can be placed in a cpp file (or multiple cpp files).