Queens College, CUNY, Department of Computer Science
Object Oriented Programming in C++
CSCI 211 / 611
Summer 2018
Instructor: Dr. Sateesh Mane

September 7, 2018

C++ Classes: Part II

- In this lecture we shall learn how to create (or *construct*) objects of a class.
- We already know that the compiler will do it for us automatically.
- However, the default action by the compiler may not always do what we want.
- In this lecture we shall learn how to write explicit rules to construct an object.
- This is accomplished by writing **constructors** for a user-defined C++ class.

1 Class `Point1`: review

- Recall the class `Point1`:

```
class Point1 {
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:
  double x, y;
};
```

- Consider the following main program to use the class `Point1`:

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

class Point1 {
  // etc
};

int main()
{
  Point1 dpt;
  dpt.print();
  cout << "get:   " << dpt.getx() << "     " << dpt.gety() << endl;
  return 0;
}
```

- The compiler constructed the object `dpt`, **but the values of x and y are uninitialized.**
- The compiler does not automatically initialize the values of the data members.
- Hence this is weak point in the design of the class `Point1`.

2 Constructors: introduction

- To specify explicit rules to construct an object, we write a **constructor**.
- There are actually many different types of constructors for a C++ class.
- We begin with the simplest, which is the **default constructor.**
- The class `Point3` below is the same as `Point1`, but we add a default constructor.
- Here is a working C++ program to declare the class `Point3` and a main program to use it.

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

class Point3 {
public:
  Point3() {                                  // default constructor
    x = 1.1;
    y = 2.4;
  }

  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:
  double x, y;
};

int main()
{
  Point3 dpt;
  dpt.print();
  cout << "get:   " << dpt.getx() << "     " << dpt.gety() << endl;
  return 0;
}
```

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

3 Default constructor

- The default constructor looks like a class method, but it has some special features.
- **The constructor of a C++ class has two distinguishing features.**
  1. The name of a constructor is the **same as the name of the class.**
  2. A constructor has **no return type.**
- *The compiler recognizes the function as a constructor because its name is the same as the name of the class and it has no return type.*
- In addition to the above two features, **the default constructor has no input arguments.**
  1. Obviously, therefore, a non-default constructor has one or more input arguments.
  2. Because the default constructor has no input arguments, *it is unique.*
  3. **There is only one default constructor for a class.**
  4. However, there can be many non-default constructors, all with different input arguments.
  5. We shall study non-default constructors later.
- Inside the function body of the constructor, we initialize the values of the data members. (This is just a simple demo example. In real life applications it would be more usual to initialize x and y to zero.)

```
Point3() {
  x = 1.1;
  y = 2.4;
}
```

- **The constructor is called (or "invoked") by the compiler when an object of the class is instantiated.**
  1. Because we supplied a constructor, the values of the data members of the class are initialized when the object is created.
  2. In the example above, the compiler invokes the default constructor when the object `dpt` is instantiated in the main program.
  3. The values of x and y are initialized, hence `getx()` and `gety()` return valid numbers.
  4. The `print` method also prints meaningful output.

4 Non-default constructors

- Writing non-default constructors is easy. They simply have one or more input arguments, which can be used to set the values of the data members.
- **It should be obvious that the compiler does not automatically generate non-default constructors.**
- *How would the compiler know what input arguments to generate?*
- Here is a class `Point4`, with one default and multiple non-default constructors.

```
class Point4 {
public:

  Point4()                    { x = 0; y = 0; }   // default constructor
  Point4(double a)            { x = a; y = a; }   // non-default constructor #1
  Point4(double a, double b) { x = a; y = b; }   // non-default constructor #2

  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:
  double x, y;
};
```

- The actions of the non-default constructors should be obvious.
  1. If an object is instantiated with no input arguments, the default constructor is invoked.
  2. If an object is instantiated with one input argument, non-default constructor #1 is invoked.
  3. If an object is instantiated with two input arguments, non-default constructor #2 is invoked.
- We shall see some examples below.
- **See next page(s).**

5 Example program: default and non-default constructors

- Here is an example program to instantiate objects of the class `Point4`.

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

class Point4 {
  // etc
};

int main()
{
  double a = -3.6;
  double b =  5.4;
  int i =  4;
  string str("abcd");
  Point4 pt40;           //   no input arguments (default constructor)
  Point4 pt41(a);        //  one input argument  (non-default constructor #1)
  Point4 pt42(i);        //  one input argument  (non-default constructor #1)
  Point4 pt43(a,b);      //  two input arguments (non-default constructor #2)
  //Point4 pt44(str);    //  *** NO MATCH, WILL NOT COMPILE ***

  pt40.print();
  pt41.print();
  pt42.print();
  pt43.print();
  return 0;
}
```

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

- For each object, the compiler knows which constructor to invoke based on the input signature.
    1. Object `pt40` is instantiated with no input arguments, hence the default constructor is invoked.
    2. Object `pt41` is instantiated with one input argument, hence non-default constructor #1 is invoked.
    3. Object `pt43` is instantiated with two input arguments, hence non-default constructor #2 is invoked.
- The object `pt42` is a special case.
    1. There is no constructor which takes one input argument of type `int`.
    2. The compiler searches for an acceptable match and invokes non-default constructor #1.
- The object `pt44` (commented out) is another special case.
    1. The compiler will generate an error if we attempt to instantiate `pt44`.
    2. There is no constructor which takes one input argument of type `string`.
    3. There is also no acceptable conversion of `string` to `double`.
    4. Hence for `pt44` the compiler will generate an error to say there is no acceptable match.
- Hence the compiler does not give up just because an exact match is not found.
    1. The compiler searches for an acceptable match and uses it, if available.
    2. However, if there is no acceptable match, then the compiler generates an error.

6 Class without default constructor

- It is not necessary to write a default constructor for a class.
- All the class constructors can be non-default constructors.
- The absence of a default constructor simply means that every object must be instantiated with input arguments.
- This is a common programming practice and there is nothing wrong with it.
- **Note that if a constructor is written for a class, even if it is a non-default constructor, the compiler will not automatically generate a default constructor.**
- The compiler automatically generates a default constructor *only if we do not write any constructors for the class.*

7 Array of objects

- The program in Sec. 5 instantiated only individual objects.
- *Can we also instantiate an array of objects for a user-defined class?*
- The obvious answer should be yes, but the correct answer is **not always.**
- We can declare an array of objects **only if the class has a default constructor.**
- Let us revisit the main program in Sec. 5, and declare arrays of objects using the default and non-default constructors.

```
// headers
// class declaration

int main()
{
  double a = -3.6;
  double b =  5.4;
  Point4 array_pt40[10];       // YES: array using default constructor
  Point4 array_pt43[10](a,b);  // NO:  non-default constructor *** WILL NOT COMPILE ***

  for (int i = 0; i < 10; ++i) {
    array_pt40[i].print();        // ok
    array_pt43[i].print();        // fail, array does not exist
  }
  return 0;
}
```

- The array `array_pt40` is an array of 10 objects of the class `Point4`.
  1. All the objects in the array are created using the default constructor.
  2. This will compile successfully.
- The next line is an attempt to declare an array `array_pt43` of 10 objects of the class `Point4`, but using a non-default constructor.
  1. **The declaration for `array_pt43` generates a compiler error.**
  2. Input arguments are not permitted, when declaring arrays.
- Therefore if a class does not have a default constructor, *we cannot declare arrays of objects for that class.* This is not necessarily a bad thing. It is a design decision. It all depends on what we wish to do with the class (or what we allow others to do).
- *Admittedly this is peculiar.* If the compiler can create an array of objects with default arguments, why can't the language be designed to create an array of objects with non-default arguments? But that's the way C++ is.

8 Pointers and dynamic memory

- Let us revisit the main program in Sec. 5, but employ pointers and dynamically memory allocation.
- We also dynamically allocate an array of objects using the default constructor.

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

class Point4 {
  // etc
};

int main()
{
  int n = 10;
  double a = -3.6;
  double b =  5.4;
  Point4 *ptr40 = new Point4;          // dynamic allocation, default constructor
  Point4 *ptr43 = new Point4(a,b);     // dynamic allocation, non-default constructor
  Point4 *arr40 = new Point4[n];       // dynamic allocation, array, default constructor

  ptr40->print();
  ptr43->print();
  for (int i = 0; i < n; ++i) {
    arr40[i].print();
  }

  delete ptr40;                        // deallocate single object
  delete ptr43;
  delete [] arr40;                     // deallocate array
  return 0;
}
```

- The above program compiles and runs successfully.
- The compiler automatically defines pointers for a user-defined class.
- The compiler automatically generates `operator new` and `delete` for a user-defined class.
- The same rules apply for dynamically allocating single objects and arrays.
- **Dynamic allocation of individual objects: default or non-default constructor.**
- **Dynamic allocation of array: default constructor only.**

9 Private constructor

- ***Can a constructor be private?***
- *Of course a constructor can be private.*
- Anything can be private.
- *What does a private constructor mean?*
- Just as the private keyword can be used to restrict access to the data members of a class, we can employ the private keyword to restrict the ability of outside applications to instantiate objects of a class.
- We may wish to control not only the access to the class data, but also to how objects of the class are instantiated.
- We can exercise a lot of control for a user-defined class.

10 Memberwise initialization (advanced topic)

- For every constructor we have written above, the construction proceeds in two steps:

  1. Memory is allocated for the data members first.
  2. The values of the data members are then set inside the function body of the constructor.

- It is possible to write a constructor so that the value of a data member is set at the same time the memory for it is allocated.

- This is known as **memberwise initialization.**

- We rewrite the class `Point4` as `Point4a`. The class constructors are rewritten to employ memberwise initialization, otherwise there is no change.

```
class Point4a {
public:
  Point4a()                   : x(0), y(0) {}   // memberwise initialization
  Point4a(double a)           : x(a), y(a) {}
  Point4a(double a, double b) : x(a), y(b) {}

  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:
  double x, y;
};
```

- **Note the new syntax of the constructors.**

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

- The syntax of memberwise initialization is as follows:

```
Point4a(...) : x(value for x), y(value for y)
{
  // function body
}
```

- **The colon ":" signifies the start of the member initialization list.**

  1. In the member initialization list, we specify the initializations of the data members.
  2. The initialization value is set at the same time that the memory for the data member is allocated.

- After the member initialization list ends, then the braces enclose the function body of the constructor.

- In the class `Point4a`, all the initializations are set in the member initialization list, hence the function body of the constructor is empty.

- To outside applications, the class `Point4a` behaves exactly the same as the class `Point4`.

- Memberwise initialization has some subtleties, which are not important in this course.

  1. The data members are initialized **in the order they appear in the class declaration.**
  2. Therefore x is initialized first and y is initialized second.
  3. Consider non-default constructor #2 with y before x in the member initialization list:

     ```
     Point4a(double a, double b) : y(b), x(a) {}   // y before x in list
     ```

  4. ***It does not matter:*** x is still initialized first and y second.

- Memberwise initialization is significant in some circumstances. We shall require it when we discuss the subject of inheritance for C++ classes.

- Before we jump to conclusions, it is not necessary to include every member of the class in the list. The following constructor also works.

```
Point4a(double a, double b) : x(a)
{
  y = b;
}
```

- There are some caveats to bear in mind about memberwise initialization.

  1. It can be dangerous to rely on the order of initialization of the data members.
  2. In large projects, where multiple software developers edit the same classes, other programmers may add extra data members into a class and this will change the order of the data initializations.
  3. This can have dangerous side effects.