CS 12: Intro. to Computer Science II

Wagner Truppel
Lecturer, Dept. of Computer Science & Engineering
UC Riverside

wagner@cs.ucr.edu
http://www.cs.ucr.edu/~wagner

http://www.cs.ucr.edu/cs12

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 1

This Week's Topics

- Intro to Object-Oriented Programming
  - Procedural Programming
  - Object-Oriented Programming
- Object-Oriented Programming (OOP)
  - Classes
  - UML
  - Constructors
  - Static members

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 2

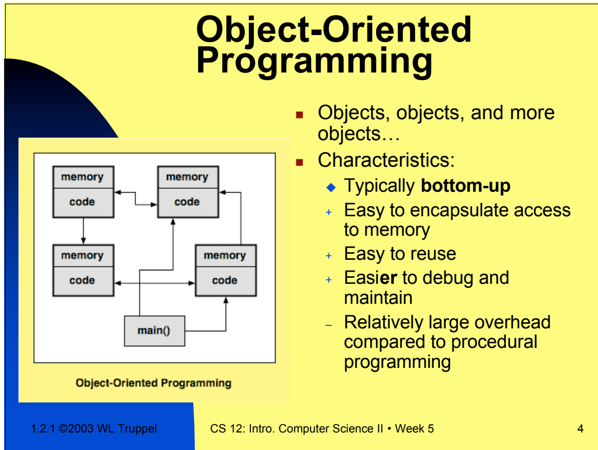
Procedural Programming

Memory
f2()
f3()
f1()
f4()
main()

Procedural Programming

- Functions, functions, functions…
- Characteristics:
  - Typically **top-down**
    - – Hard to encapsulate access to memory
    - – Hard to reuse
    - – Hard to debug and maintain
    - + Relatively low overhead

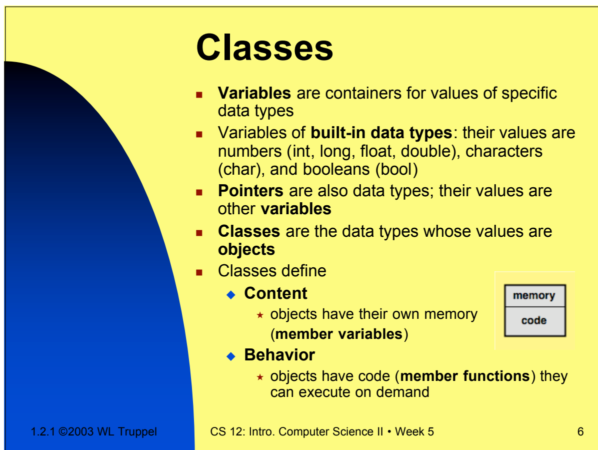
1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 3



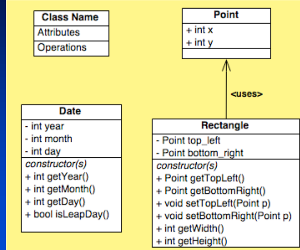
Programming Languages

- Procedural only
  - C, Pascal and other older languages
- Procedural and Object-oriented
  - C++
- Object-oriented only
  - Java
- There are many others
  - Smalltalk, Eiffel, C#, Python, Perl, etc

1.2.1 ©2003 WL Truppel   CS 12: Intro. Computer Science II • Week 5   5



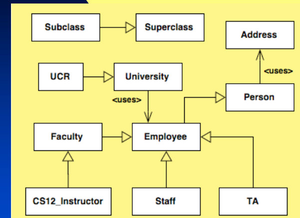
Representing Classes



- **UML** (**U**nified **M**odeling **L**anguage)
  - Standard way to describe classes and their relationships within a program
  - **Very** useful – you should learn more about it (but I won't ask you for more than the absolute basics here in CS 12)

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 7

Classes



- Classes often form **hierarchies**
- A **subclass** may **inherit** (ie, may have direct access to) the member variables and/or the member functions of its **superclass** (also called **base class**)
- We'll talk more about **inheritance** later on

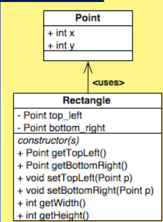
1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 8

Classes in C++

- Are very much like structures

```
class Point
{
    public:
        int x;
        int y;
};
```

```
class Rectangle
{   private:
        Point top_left;
        Point bottom_right;
    public:
        Point getTopLeft();
        Point getBottomRight();
};
```




```
Point topL = Point(2, 3); // this requires a constructor
Point botR = Point(5, 8); // this requires a constructor
Rectangle rect = Rectangle(topL, botR); // same here
// let's output the x coordinate of this
// rectangle's top-left corner
std::cout << rect.getTopLeft().x << std::endl;
```

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 9

Member functions

- We've seen how to declare them inside the class definition. How to implement them ?
- Same way as always, but must include the class name the member function belongs to

```
Point Rectangle::getTopLeft()
{ return top_left; }

Point Rectangle::getBottomRight()
{ return bottom_right; }
```

- Note that inside the function you can refer to the member variables directly by name
- The **::** is the **scope resolution operator**

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 10

The Principle of Encapsulation

- An object's inner details should not be anyone's business but its designer's
- Take a built-in data type, such as an **int**
  - Do we really know how it is implemented? No!
  - Do we care? Most of the time, no!
  - Should we care? Not really.
  - Can we still use it effectivelly? Yes!

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 11

The Principle of Encapsulation

- Built-in data types aren't objects (no class definition for them!) but they **are A**bstract **D**ata **T**ypes
- An **ADT** is an entity about which all we know and care are:
  - its **interface** (syntax)
  - its **behavior** (semantics)

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 12

Syntax x Semantics

- **Syntax** refers to what is acceptable according to the rules of a language
- **Semantics** refers to what is acceptable meaning or behavior
- "*Me not sleep yesterday.*" is **not** syntactically correct English
- "*My pet mosquito took my computer out on a date.*" **is** syntactically correct but makes no sense

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 13

The Principle of Encapsulation

- An **ADT** is an entity about which all we know and care are: its **interface** (syntax) and its **behavior** (semantics)
- All we care about an **int** is that it **represents integer numbers**, that is, we can apply to them the **operations** add, subtract, multiply, and so on (semantics!)
- The way to do so is, say, 7 + 2, not 7 2 + or some other expression (syntax!)
- The **interface** of an ADT defines its syntax
- The **axioms** of an ADT define its behavior
- We'll cover ADT axioms in more detail in CS 14

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 14

The Principle of Encapsulation

- Good object-oriented design recommends that **all member variables** and **all internally used member functions** should be **hidden** from users of the class defining them
- Sometimes there are valid reasons to break encapsulation, but until you're more experienced with OOP, you should **always** hide your classes' guts
- Other advantages of encapsulating a class' member variables:
  - Enforcement of preconditions and postconditions
  - Aids in debugging

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 15

The Principle of Encapsulation

- How to encapsulate (hide) a class' inner details ? Be **private** about them…

```
class Point
{
    private:
        int x;
        int y;
    public:
        Point(int x, int y); // this is a constructor
        int getX(); // this is an accessor function
        int getY(); // this is an accessor function
        void setX(int x); // this is a mutator function
        void setY(int y); // this is a mutator function
};
```


Structures & Classes

- Structures may have private member variables as well, not just public ones
- Structures may have private and public member functions too
- In fact, everything you can do with classes, you can also do with structures
- The only difference is that if you omit the **access modifiers** (private, public), then
  - Classes assume private
  - Structures assume public
- Why two names for the same idea?
  - C++ is C on steroids, so it requires structures for compatibility
  - C++ is an OO language, so it requires classes

Recall this example…

```
class Point
{
    private:
        int x;
        int y;
    public:
        Point(int ax, int ay); // this is a constructor
        int getX(); // this is an accessor function
        int getY(); // this is an accessor function
        void setX(int ax); // this is a mutator function
        void setY(int ay); // this is a mutator function
};
```

Constructors

Point
- int x - int y
+ Point(int ax, int ay) + int getX() + int getY() + void setX(int ax) + void setY(int ay)

- Special kind of member function
  - Used to initialize member variables
  - More generally, used to initialize the **state** of the object being created
  - Automatically invoked when declaring a class variable (an object of the class)
- Must have the same name as the class they're defined for
- No return type (**not** even *void*)
- You **can** have more than one
- **Not** invoked like other functions (more on this later)

Declaring constructors…

Point
- int x - int y
+ Point(int ax, int ay) + int getX() + int getY() + void setX(int ax) + void setY(int ay)

```
class Point
{
    private:
        int x;
        int y;
    public:
        Point(int ax, int ay); // this is a constructor
        int getX(); // this is an accessor function
        int getY(); // this is an accessor function
        void setX(int ax); // this is a mutator function
        void setY(int ay); // this is a mutator function
};
```


Declaring constructors…

Display 7.1, page 262

Display 7.1 Class with Constructors *(part 1 of 2)*

```
1  #include <iostream>
2  #include <cstdlib> //for exit
3  using namespace std;

4  class DayOfYear
5  {
6  public:
7      DayOfYear(int monthValue, int dayValue);
8      //Initializes the month and day to arguments.

9      DayOfYear(int monthValue);
10     //Initializes the date to the first of the given month.

11     DayOfYear( );
12     //Initializes the date to January 1.

13     void input( );
14     void output( );
15     int getMonthNumber( );
16     //Returns 1 for January, 2 for February, etc.

17     int getDay( );
18 private:
19     int month;
20     int day;
21     void testDate( );
22 };

23 int main( )
24 {
25     DayOfYear date1(2, 21), date2(5), date3;
26     cout << "Initialized dates:\n";
27     date1.output( ); cout << endl;
28     date2.output( ); cout << endl;
29     date3.output( ); cout << endl;

30     date1 = DayOfYear(10, 31);
31     cout << "date1 reset to the following:\n";
32     date1.output( ); cout << endl;
33     return 0;
34 }
35
36 DayOfYear::DayOfYear(int monthValue, int dayValue)
37                         : month(monthValue), day(dayValue)
38 {
39     testDate( );
40 }
```

This definition of DayOfYear is an improved version of the class DayOfYear given in Display 6.4.

default constructor (line 11)

This causes a call to the default constructor. Notice that there are no parentheses. (line 25)

an explicit call to the constructor DayOfYear::DayOfYear (line 30)

Defining constructors…

Point
- int x - int y
+ Point(int ax, int ay) + int getX() + int getY() + void setX(int ax) + void setY(int ay)

- Somewhere on the same file, after the class declaration
- 2 ways to declare a constructor:

```
Point::Point(int ax, int ay)
{
    x = ax;
    y = ay;

    // we could now verify the input
}
```

```
Point::Point(int ax, int ay) : x(ax), y(ay)
{
    // this body does not have to be empty !
    // we can use it to verify the input
}
```

- The second one is considered preferable (it's more efficient)

1.2.1 ©2003 WL Truppel   CS 12: Intro. Computer Science II • Week 5   22

Defining constructors…

Display 7.1, page 262

Display 7.1 **Class with Constructors *(part 1 of 2)***

```
1  #include <iostream>
2  #include <cstdlib> //for exit
3  using namespace std;

4  class DayOfYear
5  {
6  public:
7      DayOfYear(int monthValue, int dayValue);
8      //Initializes the month and day to arguments.

9      DayOfYear(int monthValue);
10     //Initializes the date to the first of the given month.

11     DayOfYear( );
12     //Initializes the date to January 1.

13     void input( );
14     void output( );
15     int getMonthNumber( );
16     //Returns 1 for January, 2 for February, etc.

17     int getDay( );
18 private:
19     int month;
20     int day;
21     void testDate( );
22 };

23 int main( )
24 {
25     DayOfYear date1(2, 21), date2(5), date3;
26     cout << "Initialized dates:\n";
27     date1.output( ); cout << endl;
28     date2.output( ); cout << endl;
29     date3.output( ); cout << endl;

30     date1 = DayOfYear(10, 31);
31     cout << "date1 reset to the following:\n";
32     date1.output( ); cout << endl;
33     return 0;
34 }
35
36 DayOfYear::DayOfYear(int monthValue, int dayValue)
37                         : month(monthValue), day(dayValue)
38 {
39     testDate( );
40 }
```

This definition of DayOfYear is an improved version of the class DayOfYear given in Display 6.4.

default constructor (line 11)

This causes a call to the default constructor. Notice that there are no parentheses. (line 25)

an explicit call to the constructor DayOfYear::DayOfYear (line 30)

1.2.1 ©2003 WL Truppel   CS 12: Intro. Computer Science II • Week 5   23

Invoking constructors…

Point
- int x - int y
+ Point(int ax, int ay) + int getX() + int getY() + void setX(int ax) + void setY(int ay)

- Implicitly:
  - Point corner(3, 7);
  - DayOfYear holiday(2, 17);
- Explicitly:
  - Looks like a regular call to a function returning a value. Looks are deceiving…
  - Point corner = Point(3, 7);
  - DayOfYear holiday = DayOfYear(2, 17);
- Watch out…
  - Point corner; // also calls a constructor !
  - DayOfYear holiday; // same here…

1.2.1 ©2003 WL Truppel   CS 12: Intro. Computer Science II • Week 5   24

Invoking constructors…

- Recall that *int x;* declares the variable *x*, allocates space for it, but does **not** set its value
- Object declaration is similar
  - *Point p;* declares *p* as an object of class *Point*, allocates space for it, **but also calls a constructor**
  - But which constructor? We only defined one and it takes arguments

Point
- int x - int y
+ Point(int ax, int ay) + int getX() + int getY() + void setX(int ax) + void setY(int ay)

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 25

Invoking constructors…

- Object declaration is similar
  - *Point p;* declares *p* as an object of class *Point*, allocates space for it, **but also invokes a constructor**
  - But which constructor? We only defined one and it takes arguments
- The invoked constructor is **Point()**
- But we didn't define it, so the compiler complains

Point
- int x - int y
+ Point(int ax, int ay) + int getX() + int getY() + void setX(int ax) + void setY(int ay)

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 26

Invoking constructors…

- The **no-argument** constructor is called the **default** constructor
- If you don't define any constructor for your class, the compiler automatically creates one for you - the default constructor
- But if you **do** define *any* constructor, the compiler will **not** create the default for you
- In such a case, declaring an object as in *Point p;* is an **error** (unless one of the constructors you defined is the default)

Point
- int x - int y
+ Point(int ax, int ay) + int getX() + int getY() + void setX(int ax) + void setY(int ay)

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 27

Invoking constructors…

- What if you do define the no-argument constructor? How do you invoke it?
  DayOfYear today;
  DayOfYear today = DayOfYear();
- Compare with
  DayOfYear today(2, 11);
  DayOfYear today = DayOfYear(2, 11);
- For a good example that puts it all together, look at the **BankAccount** class on **pp. 363-368** of the textbook

Passing objects to functions

- Objects are typically “heavy” (have lots of internal stuff)
- Better never to pass them by value but, instead, by reference (to avoid copying)
- But what if you don’t want to change the object’s internal state? Use **const**

```
void drawRect(const Rectangle& rect,
              const Color& color)
```


A concrete example


```
class Rectangle
{
    private:
        Point top_left;
        Point bottom_right;
    public:
        Rectangle(Point topL, Point botR);
        Point getTopLeft() const;
        Point getBottomRight() const;
        void setTopLeft(Point p);
        void setBottomRight(Point p);
        int getWidth() const;
        int getHeight() const;
};
```

A concrete example

Point
+ int x
+ int y

<uses>

Rectangle
- Point top_left
- Point bottom_right

constructor(s)
+ Point getTopLeft()
+ Point getBottomRight()
+ void setTopLeft(Point p)
+ void setBottomRight(Point p)
+ int getWidth()
+ int getHeight()

```
Rectangle:: Rectangle(Point topL, Point botR) : top_left(topL),
     bottom_right(botR) {}

Point Rectangle::getTopLeft() const
{ return top_left; }

Point Rectangle::getBottomRight() const
{ return bottom_right; }

void Rectangle::setTopLeft(Point p)
{ top_left = p; }

void Rectangle::setBottomRight(Point p)
{ bottom_right = p; }

int Rectangle::getWidth() const // assuming x grows to the right
{ return (bottom_right.x - top_left.x); }

int Rectangle::getHeight() const // assuming y grows downwards
{ return (bottom_right.y - top_left.y); }
```

1.2.1 ©2003 WL Truppel

Another use for const

- Sometimes we don’t want a member function of a class to change the state of the object on which it’s invoked

```
Rectangle r =
   Rectangle(Point(2, 3), Point(10, 17));
cout << “Width = ” << r.getWidth() << endl;
```

1.2.1 ©2003 WL Truppel

Static members

- Sometimes we want to have a member variable which is shared by all objects of a given class
- Example: a count of how many objects have been created
- Such variables are not tied to any one object in particular
- Member variables like this are called **static member variables**
- They are a sort of global variable
- They must be **initialized only once**
- They must be initialized **outside the class** they’re defined in

1.2.1 ©2003 WL Truppel

Static member variables

```
class Connection
{
      private:
            // private members
      public:
            Connection(); // default constructor
            static int connCount; // public just as an example
            // other public members
};

int Connection::connCount = 0;

Connection::Connection()
{
   if (connCount > 10)
         ; // should raise an exception (an error)
   connCount++;
}
```

1.2.1 ©2003 WL Truppel   CS 12: Intro. Computer Science II • Week 5   34

Static member variables

- How do you access them?
- Since they're not bound to any one object, you could access them from any object of its class:

```
Connection conn1; // calls the default constructor
cout << conn1.connCount;
```

- The preferred way, however, is to use the class name:

```
cout << Connection::connCount;
```

- In this example, this is possible because *connCount* was defined as **public**. What if it was a private variable?

1.2.1 ©2003 WL Truppel   CS 12: Intro. Computer Science II • Week 5   35

Static member variables

```
class Connection
{
      private:
            static int connCount;
            // other private members
      public:
            Connection(); // default constructor
            static int getNumConns();
            // other public members
};

int Connection::connCount = 0;

Connection::Connection()
{
   if (connCount > 10)
         ; // should raise an exception (an error)
   connCount++;
}

int Connection::getNumConns()
{ return connCount; }
```

1.2.1 ©2003 WL Truppel   CS 12: Intro. Computer Science II • Week 5   36

Static member variables

- How do you access a private static member variable?
- You need an **accessor** member function
- But since the variable is static, the accessor function must also be static

```
cout << Connection::getNumConns();
```

- You cannot access object-specific info from within a static function

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 37

A bit of perspective

- What do we gain by using well-defined classes?
  - Self-contained components
  - Easier to develop
  - Easier to debug
  - Easier to maintain
  - Re-usable
  - Are good representations of the world

1.2.1 ©2003 WL Truppel CS 12: Intro. Computer Science II • Week 5 38