

Introduction to OOP

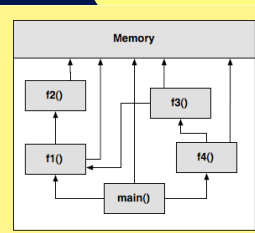
Wagner Truppel
Lecturer, Dept. of Computer Science & Engineering
UC Riverside

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

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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 1

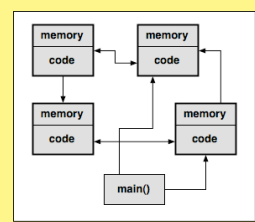
Procedural Programming



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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 2

Object-Oriented Programming



- Objects, objects, and more objects...
- Characteristics:
 - ◆ Typically **bottom-up**
 - + Easy to encapsulate access to memory
 - + Easy to reuse
 - + Easier to debug and maintain
 - Relatively large overhead compared to procedural programming

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 4

Classes

- **Variables** are containers for values of specific data types
- Variables of **built-in data types**: their values are numbers (int, long, float, double), characters (char), and booleans (bool)
- **Pointers** are also data types; their values are other **variables**
- **Classes** are the data types whose values are **objects**
- Classes define
 - ◆ **Content**
 - * objects have their own memory (**member variables**)
 - ◆ **Behavior**
 - * objects have code (**member functions**) they can execute on demand

memory
code

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

Representing Classes

Class Name
Attributes
Operations

Date
- int year
- int month
- int day
constructor(s)
+ int getYear()
+ int getMonth()
+ int getDay()
+ bool isLeapDay()

Point
+ int x
+ int y

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()

- **UML** (Unified Modeling Language)
 - ◆ 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)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 6

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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 7

Classes in C++

- Are very much like structures

```

class Point
{
+ int x
+ int y
}

class 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()
}
                    
```

```

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;
                    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 8

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**

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 9

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 effectively? Yes!

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 10

The Principle of Encapsulation

- Built-in data types aren't objects (no class definition for them!) but they **are Abstract Data Types**
- An **ADT** is an entity about which all we know and care are:
 - ◆ its **interface** (syntax)
 - ◆ its **behavior** (semantics)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 11

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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 12

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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 13

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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 14

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
};
                    
```

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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 15

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

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 16

Reminder

- **Midterm next lecture**
- **Everything** we've covered in class **and** in the labs up to and **including today's lecture** is fair game
- Bring your **UCR Student ID card** or else...
- **No** other form of ID
- Lab review sessions

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 7 17
