

CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 9
September 30, 2010

Remarks on Problem Set 2

Classes

Derivation

Construction, Initialization, and Destruction

Remarks on Problem Set 2

Seth Hamman

Coding conventions for this course

- ▶ Use `#pragma once` at top of include files.
- ▶ The functions `say()` and `fatal()` in `tools.cpp` are to help you with code development and debugging while you are learning C++. They should not be used in a finished product once you have learned the standard C++ ways for handling console output and exception handling.
- ▶ Don't use STL until it has been covered in class.
- ▶ Don't define global variables.
- ▶ Use file name suffix `.cpp` for C++ code files and `.hpp` for C++ header files.
- ▶ Use convention that `.cpp` file only includes corresponding `.hpp` file. The `.hpp` file contains all necessary includes.

Coding conventions (cont.)

- ▶ Use short identifiers in local contexts; longer more descriptive names in larger contexts.
- ▶ Use consistent naming style, e.g., data member ends in ''.
- ▶ Typical structure of a class definition:

```
class Name {  
private:  
    vars  
    methods  
public:  
    constructors  
    destructor  
    methods  
};
```

Error checking and error handling

What should your program do with “bad” inputs? **Options:**

1. Detect and report/handle all bad inputs.
2. Detect and report bad inputs that would cause unexpected behavior such as a crash, divide by 0, and so forth, but don't check for inputs that violate the input specs if they are handled reasonably by your program. (E.g., a 6-digit number where the specs said to expect a 5-digit number.)
3. Don't bother with error checking.

(1) is most desirable.

(2) is acceptable in this course but not in industrial strength code where the specs and code should agree.

(3) is **not acceptable**.

Design choices

Programming involves design choices.

There is not always one “right” choice; rather, there are tradeoffs, and the choice requires judgement.

Use [report.txt](#) to discuss and justify the design choices you make in your own programs.

PS2 issues

- ▶ `atoi()` is deprecated; use `strtol()` instead.
- ▶ Refer to man pages for documentation of unfamiliar functions, e.g., `man atoi` or `man 3 atoi`.
- ▶ `class Random` should not be instantiated more than once, since all instances share `rand()`, and a subsequent call on `srand()` will interfere with previous instances. Thus, it should be a *singleton class* (later).
- ▶ Put game logic into the `Craps` class, not into `Dice`.
- ▶ Use `switch` for clean coding of game logic.

Classes

What is a class?

- ▶ A collection of things that **belong together**.
- ▶ A **struct with associated functions**.
- ▶ A way to **encapsulate behavior**: public interface, private implementation.
- ▶ A way to **protect data integrity**, providing world with functions that provide a read-only view of the data.
- ▶ A **data type** from which objects (instances) can be formed. We say the instances **belong** to the class.
- ▶ A way to **organize and automate** allocation, initialization, and deallocation.
- ▶ A way to **break** a complex problem **into manageable, semi-independent pieces**, each with a defined interface.
- ▶ A **reusable module**.

Class relationships

Classes relate to and collaborate with other classes.

Many ways in which one class relates to other.

We first explore *derivation*, where one class modifies and extends another.

Derivation

What is derivation?

One class can be derived from another.

Syntax:

```
class A {  
    public:  
        int x;  
        ...  
};  
class B : public A {  
        int y;  
        ...  
};
```

A is the **base class**; B is the **derived class**.

B **inherits** data members from A.

Instances

A base class instance is contained in each derived class instance.

Similar to composition, except for inheritance.

Function members are also inherited.

Data and function members can be **overridden** in the derived class.

Derivation is a powerful tool for allowing variations to a design.

Some uses of derivation

Derivation has several uses.

- ▶ To allow a family of related classes to share common parts.
- ▶ To describe abstract interfaces à la Java.
- ▶ To allow generic methods with run-time dispatching.
- ▶ To provide a clean interface between existing, non-modifiable code and added user code.

Construction, Initialization, and Destruction

Structure of an object

A simple object is like a `struct` in C.

It consists of a block of storage large enough to contain all of its data members.

An object of a derived class contains an instance of the base class followed by the data members of the derived class.

Example:

```
class B : A { ... };  
B b;
```

Then “inside” of `b` is an `A`-instance!

Referencing a composed object

Constrast the previous example to

```
class B { A a; ...};  
B b;
```

Here **B** composes **A**.

The embedded **A** object can be referenced using data member name **a**.

Referencing a base object

How do we reference the base object embedded in a derived class?

Example:

```
class A { public: int x; int y; ...};  
class B : A { int y; ...};  
B b;
```

- ▶ The data members of **A** can be referenced directly by name.
x refers to data member **x** in class **A**.
y refers to data member **y** in class **B**.
A::y refers to data member **y** in class **A**.
- ▶ **this** points to the whole object.
Its type is **B***.
It can be coerced to type **A***.

Initializing an object

Whenever a class object is created, one of its constructors is called.

If not specified otherwise, the **default constructor** is called.
This is the one that takes no arguments.

If you do not define the default constructor, then the **null constructor** (which does nothing) is used.

This applies not only to the “outer” object but also to all of its embedded objects.

Construction rules

The rule for an object of a simple class is:

1. Call the constructor/initializer for each data member object in sequence.
2. Call the constructor for the class.

The rule for an object of a derived class is:

1. Call the constructor for the base class recursively.
2. Call the constructor/initializer for each data member object of the derived class in sequence.
3. Call the constructor for the derived class.

Destruction rules

When an object is deleted, the destructors are called in the opposite order.

The rule for an object of a derived class is:

1. Call the destructor for the derived class.
2. Call the destructor for each data member object of the derived class in reverse sequence.
3. Call the destructor for the base class.

Constructor ctors

Ctors (short for constructor/initializers) allow one to supply parameters to implicitly-called constructors.

Example:

```
class B : A {  
    B( int n ) : A(n) {};  
    // Calls A constructor with argument n  
};
```

Initialization ctors

Ctors also can be used to initialize primitive (non-class) variables.

Example:

```
class B {  
    int x;  
    const int y;  
    B( int n ) : x(n), y(n+1) {}; // Initializes x and y  
};
```

Multiple ctors are separated by commas.

Ctors present must be in the same order as the construction takes place – base class ctor first, then data member ctors in the same order as their declarations in the class.

Initialization not same as assignment

Previous example using ctors is not the same as writing

```
B( int n ) { y=n+1; x=n; };
```

- ▶ The order of initialization differs.
- ▶ `const` variables can be initialized but not assigned to.
- ▶ Initialization uses the constructor (for class objects).
- ▶ Initialization from another instance of the same type uses the copy constructor.

Copy constructors

- ▶ The **copy constructor** is automatically defined for each new class `A` and has prototype `A(A&)` or `A(const A&)`. It initializes a newly created `A` object by making a copy of its argument.
- ▶ Copy constructors are also used for call-by-value.
- ▶ Assignment uses `operator=()`, which by default copies the data members but does not call the copy constructor.