CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 12
October 11, 2011

Interacting Classes and UML (continued)

Review for Exam

Interacting Classes and UML (continued)

Association Relationship

The **association** relationship can be diverse. Some developers label text describing the relationship on the edge connecting two associated classes.
Later in the course we will explore and see more examples.

Accessing `B` in `A`'s methods

Access patterns:

- parameter, local variable, or return has type `B`/`B&`/`B*`
- a method in `A` accesses `B`'s data members: `B::var` or `b.var`
- a method in `A` invokes `B`'s methods: `B::func()` or `b.func()`
- indirect: `c.b.func()`

If `A` knows `B` only through parameter or local variables, we also say that `A` **uses** `B`. The **use** relationship is generally considered to be a weak relationship.

"Law" of Consistency/Encapsulation

Relation of `B::var` or `b.var` in `A` is typically not recommended because it violates encapsulation and may lead to inconsistent state.

Why is the design below not desirable?

```
class SpeedDataCollection{
  ...
  // add a new data value
  public void addValue(int speed);

  // return average speed
  public double averageSoFar;
  ...
};
```

Limiting coupling between classes

Chaining such as `c.b.func()` is typically not recommended as it increases coupling.

For example, assume class `A` has a data member `Dog* dog`. One way to ask the `dog` object to move is `dog->leg()->walk()`. But this is less desirable than calling `dog->walk()`.

In OO design, this is called the **"Law" of Demeter**, also called **"Law" of Least Knowledge**:

- "the method of a class should not depend on any way on the structure of any class, except the immediate (top-level) structure of its own class."

This principle has other names such as **Delegation** and **Do not talk to Strangers**.

"Law" of Demeter

Formally, the "Law" of Demeter for functions requires that a method `M` of an object `A` may only invoke the methods of the following kinds of objects:

- `A` itself
- `M`'s parameters
- any objects created/instantiated within `M`
- `A`'s direct component objects
- a global variable, accessible by `A`, in the scope of `M`

One can consider layered architecture of many systems (e.g., the layered network architecture) as following this design guideline.

Review for Exam

Goals of OO Programming

- Efficient reusable code
- Modularity and code isolation
- Modeling tool
- Large-scale software construction
- Safe and reliable code

Insertion sort example

- Illustrates use of `class` to build a sortable collection of data (`DataPack`).
- Example of multifile program: `main.cpp`, `datapack.hpp`, `datapack.cpp`.
- Shows use of dynamic memory management paradigm, where constructor allocates and destructor deletes.
- Illustrates use of inline and out-of-line functions.
- Example of how to manage persistent data—realize an internal data structure from a file.
- Poor man's example of a generic collection (using `typedef` instead of templates).
- Visibility, data hiding, and `const`.

Compiling and linking

Stages of compilation:

- Each `.cpp` file is run through the preprocessor, which processes the #-directives.
- The result is compiled to a `.o` **object file**.
- The `.o` files are linked together with the libraries to form an `executable` file.

Compiler errors

Errors produced at each stage:

- Preprocessor errors: Missing `#include` files, mismatched `#ifndef`...`#endif` pairs, etc.
- Compilation errors: Syntax, semantics, missing/erroneous declarations. Sometimes the file produced by the preprocessor and seen by the compiler is not what the programmer intended.
- Linker errors are generally methods that were declared but never defined, often because of mismatched signatures. Duplicate definitions from different modules also detected here, e.g., multiple definitions of `main()`.

Tool set

- `valgrind`
- `make`
- `eclipse`
- `g++`

What does each of these tools do?
When should they be used?

C++ goodies

- Operator extensions.
- Adding new methods to a function.
- Member functions and implicit argument; `this`.
- Supplying comparison function for sort using operator extension instead of functional parameter.

Stream I/O

- Opening and closing streams.
- Testing for successful open.
- Reading data from streams.
- Writing data to streams.
- Manipulators.
- End-of-file and error handling.

Functions and methods

- Passing parameters to functions
- Passing results back from functions
- Parameter types and calling mechanisms
- When to use which parameter type
- The implicit argument

Variables and data

- Parts of a variable: name, type, storage register.
- Simple variables: declaration, initialization, assignment.
- L-values and R-values.
- Reference values ("pointers") and pointer variables.
- Creating and following pointers.
- References: types, uses, comparison with pointers.

BarGraph demo

- Illustrates tightly-coupled classes (`Row` and `Cell`).
- Example of aggregated data member (`Row::head`, `Cell::Item`).
- Illustrates use of class-type data items (type `Item`) in an array (`Graph::bar`).
- Illustrates use of delegation (e.g., `Graph::print()`).
- Illustrates private nested class definition (in file `rowNest.hpp`).

More on variables

Properties

- `sizeof` operator.
- Visibility of variables.
- Lifetime of variables.
- Anonymous variables.
- Storage classes: `auto`, `static`, dynamic.
- How to use static variables.

Assignment and copying

- Shallow and deep copies.
- Aliasing.

Five kinds of memory errors

1. **Memory leak**—Dynamic storage that is no longer accessible but has not been deallocated.
2. **Amnesia**—Storage values that mysteriously disappear.
3. **Bus error**—Program crashes because of an attempt to access non-existent memory.
4. **Segmentation fault**—Program crashes because of an attempt to access memory not allocated to your process.
5. **Waiting for eternity**—Program is in a permanent wait state or an infinite loop.

C++ bells and whistles

- Optional parameters and overloading
- `const` variables
- `const` parameters
- `const` implicit variables
- Operator extensions

Derivation

Derived classes

- Base class, derived class, syntax.
- Purpose.
- Overriding data and function members.
- Construction and initialization.
- Ctors.
- Destruction.
- Structure of a derived class instance.
- Copy constructors.
- Referencing data and function members in the base class and in the derived class.

Polymorphic derivation

- Polymorphic pointers.
- Virtual functions.
- Runtime type tags and vtables.
- Virtual destructors.
- Pure virtual functions.
- Abstract classes.
- Visibility keywords: `public`, `protected`, `private`.
- Kinds of derivation: `public`, `protected`, `private`.
- Visibility rules.
- Declaration context and reference context: inside and outside references.
- Inaccessible base class.