Discussion slides for week of October 1, 2007
The University of Michigan
Agenda

- Inheritance / Polymorphism review
- Access: public, private, protected, friends
- Circular dependencies and forward declarations
- Tree, Binary Trees, Binary Search Trees
Inheritance and Polymorphism

- Allow us to specify relationships between types
  - Abstraction, generalization, specification
  - The “is a” relationship
  - Examples?

- Why is this useful in programming?
  - Allows for code reuse
  - More intuitive/expressive code
Code Reuse

- General functionality can be written once and applied to *any* derived class

- Derived classes can have selective specialization by adding members or implementing virtual functions
Interfaces and Implementation

- There are two things that can be inherited:
  - A particular *interface*
  - A particular *implementation*
Interfaces and Implementation

- There are two things that can be inherited:
  - A particular *interface*
  - A particular *implementation*

- The **virtual** keyword means that the interface is inherited, but not (necessarily) the implementation
Pure Virtual and Abstract

- Adding `=0` makes a function **pure** virtual
  - No implementation is given
  - *Must* be implemented by (concrete) derived classes

- Any class with at least 1 pure virtual member is **abstract**, and cannot be instantiated

- Often used to define pure “interface” classes with no functionality
Example: Virtual vs. Non-Virtual

class Base {
public:
    void print() {
        cout << "Base";
    }
};

class Derived: public Base {
public:
    void print() {
        cout << "Derived";
    }
};
What is the output?

Derived d;

Base *pB = &d;

Derived *pD = &d;

pB->print();
pD->print();
Virtual vs Non-virtual

- Virtual functions are interfaces
  - All derived types will have the functionality, so the function can be called on any instance of the base
  - Implementations are only “suggestions”
  - “Pure” virtual functions are only interfaces

- Non-virtual functions should be considered *mandatory* implementations for derived classes
  - Don’t override these, or you get unintuitive behavior
Base Class

class TodoList {
public:
    virtual int isempty() { return(1); };
    virtual void add(T elt) { cerr << "TodoList::add: not defined\n"; };
    virtual T next() { cerr << "TodoList::next: not defined\n"; };
    virtual int type() { cerr << "TodoList::type: not defined\n"; };
};
Derived Classes

class TodoQueue : public TodoList<T> {
private:
    int mytype;
    Queue q;
public:
    TodoQueue(): mytype(TODOQ) {}
    virtual int isempty() { return(!q.size()); };
    virtual void add(T elt) { q.enqueue(elt); }
    virtual T next() { if (q.size()) { return(q.dequeue()); } return (0); }
    virtual int type() { return (mytype); }
};
When to use Inheritance/Polymorphism

- Do you have types/classes that share functionality?
- Are you tempted to implement identical/similar functionality in several places?
- Can you define a natural hierarchy using “isa” relationships?

- Using templates vs. virtual functions
- Does the type T change the behavior of the class?
  - If no, use a template
  - If yes, use virtual functions
Multiple Inheritance

- Use sparingly and cautiously
- The “deadly diamond”:

```
class A;
class B;
class C;
class D;
```
Multiple Inheritance

- Using “virtual” inheritance resolves ambiguity
- Ensures that a single instance of the base class is inherited

```cpp
class B: public virtual A {...};
class C: public virtual A {...};
class D: public A, public B {...};
```
Access Control

- Private
  - Access only within the class
- Public
  - Access from anywhere
- Protected
  - Access within this class and any derived classes

Why do we make anything private?
Friends Functions and Classes

- Allows access to private members on a case-by-case basis

```cpp
class myClass {
    int myPrivateVar;
    friend void myFriendFunction();
    friend class myFriendClass;
};
```
Classes and Structs

- What are the differences between classes and structs in C++?
Classes and Structs

- What are the differences between classes and structs in C++?
  - Classes default to private access
  - Structs default to public access

- What about default type of inheritance?
Private Constructors/Destructors

- Can you have private constructors and destructors?
Private Constructors/Destructors

- Can you have private constructors and destructors?
  - Yes, but why?

- Gives you very explicit control over creation/deletion
  - Use friends to declare explicit ownership
  - “Singleton” design pattern
  - Reference counted objects
Public vs. Private Inheritance

- **Public inheritance**
  - public members stay public in the derived class
  - “is a” relationship
  - \textit{Class Derived: public Base \{\ldots\};}

- **Private inheritance**
  - public members become private in the derived class
  - “implemented as a”
  - \textit{Class Derived: private Base \{\ldots\};}
Circular Dependencies

- How do you resolve circular dependencies?
  - First, try to redesign without the dependency
Forward Declarations

- Let the compiler know that the definition is coming later.
  - Must use pointers (size isn’t known yet)
Binary Trees

- Root
- Height
- Size
- Full Binary Tree
- Complete Binary Tree
Binary Search Trees

- What's different?
- Searching
- Insertion
- Removal