CS11 – Introduction to C++

Fall 2012-2013
Lecture 7
Computer Strategy Game

- Want to write a turn-based strategy game for the computer
- Need different kinds of units for the game
  - Different capabilities, strengths, weaknesses, etc.
- Examples:
  - Infantry unit
  - Transport ship
  - Trebuchet
Unit Classes

- Infantry unit data members:
  - Current location, strength, experience level
  - Who owns the unit

- Member functions:
  - Move to another location
  - Attack another unit, receive attack from another unit

- What about transport ships?
  - Probably not “experience level” or “attack a unit”
  - Need to add a list of things the ship is carrying

- Trebuchets are very similar to infantry units
  - Just a different way of attacking other units
Common Themes

- Very clear we have common themes here!
  - All units have a location, strength, owner
  - All units can move around
  - All units can be attacked

- Also have specialized capabilities
  - Infantry units and trebuchets can’t carry other stuff, and they can’t make it very far on water
  - Of course, boats can’t make it very far on land
Single-Class Implementation

- Doesn’t make sense to put all features in one class

```cpp
class GameUnit {
    UnitType type;
    Point location;
    int playerID;
    int strength;
    // For transport ships only!
    int transportCapacity;
    vector<GameUnit *> contents;

public:
    ...
    bool move(const Map &map, int direction);
    // For infantry and trebuchets only!
    bool attack(const Map &map, int direction);
    // For transport ships only!
    bool transportItem(GameUnit *u);
};
```
Single-Class Implementation (2)

- Not a clean abstraction!
  - Both data and operations for infantry units are mixed together with transport ships, etc.
  - Hard to understand what is going on
- Very likely to lead to bugs, e.g.
  - Infantry units can accidentally carry other units
  - Transport ships can attack other units
- Also wastes space
  - Infantry unit doesn’t need list of units it’s carrying
C++ Class Inheritance

- C++ provides class inheritance
  - (as do all OOP languages)
- Base class provides generalized capabilities
  - A “game unit” type with location and owner
  - Basic ability to move unit around, retrieve owner info, etc.
- Derived classes inherit the state and behavior of the base class
  - They can also move around or report the unit’s owner
- Derived classes also provide specialized capabilities
  - The “transport ship” type adds the ability to hold other units
  - The “trebuchet” type adds the capability to rain fiery death from above
Class Inheritance (2)

Class inheritance models an “is-a” relationship
- An infantry unit “is a” game unit
- A transport ship “is a” game unit
- A trebuchet “is a” game unit

Terminology:
- Base class, parent class, superclass
- Derived class, child class, subclass
A Game Unit Base-Class

```cpp
class GameUnit {
    Point location;
    int playerID;
    int strength;

    public:
        GameUnit(const Point &loc, int owner, int strength);
        Point getLocation() const;
        bool move(const Map &map, int direction);
        string getUnitType() const {
            return "generic game unit";
        }
};
```

- Basic features common to all game units
Extending the Generic Game Unit

- Transport Ship class will extend **GameUnit**
  - Add in new state and behaviors
    ```cpp
class TransportShip : public GameUnit {
    int capacity;
    vector<GameUnit *> contents;

    public:
    TransportShip( /* TODO */ );
    bool loadUnit(GameUnit *u);
    vector<GameUnit *> unloadAll();
};
```
  - Don’t need to specify location, owner, etc.
    - State and associated behaviors inherited from parent class
  - Can also provide new capabilities
    - Ability to load and unload units from the transport

- State and associated behaviors inherited from parent class
  - Can also provide new capabilities
Generic Functions

- This class can use `GameUnit` type for its functions
  ```
  class TransportShip : public GameUnit {
    int capacity;
    vector<GameUnit *> contents;

  public:
    TransportShip( /* TODO */ );
    bool loadUnit(GameUnit *u);
    vector<GameUnit *> unloadAll();
  };
  ```
- Because *any subclass* of `GameUnit` “is a” `GameUnit`, we can pass any subclass to our load/unload functions
- Don’t need to provide a version for each specific subclass!
- **Same general principle for other APIs!**
  - Write functions against the base-type where appropriate
Transport-Ship Constructor

- Need to implement the transport ship constructor
  - Problem: base class also has its own data members
  - We need to initialize these members too

- First attempt:

  ```cpp
  TransportShip(const Point &loc, int owner, int capacity) :
  location(loc), playerID(owner),
  strength(100), capacity(capacity) {
  assert(capacity > 0);
  }
  ```

- Doesn’t compile! Subclasses cannot initialize base-class data members directly.
Calling Base-Class Constructor

- Want to call the base-class constructor from the subclass constructor
  - Must call the base-class constructor in member initializer list
  - Must be the first item of the member initializer list!

- Transport ship constructor, take 2:
  ```cpp
  TransportShip(const Point &loc, int owner, int capacity) :
      GameUnit(loc, owner, 100), capacity(capacity) {
    assert(capacity > 0);
  }
  ```
  - Now we get to reuse our `GameUnit` constructor code!
Initialization of Derived Classes

- Remember the class initialization process:
  - All (non-primitive) data members of a class are initialized before the class’ constructor-body is run
  - Constructor-body can refer to these members, because they have already been initialized

- Same is true for class hierarchies
  - Base-class initialization occurs before derived-class initialization
  - Derived class’ constructor body can also refer to data members in the base class!
Initialization of Derived Classes (2)

- Initialization of base-class members normally happens automatically
  - Base class’ default constructor is called before the derived class is initialized
  - Don’t need to put this in the member initializer list; it happens automatically!
- But, not every base class has a default constructor!
  - ...or, you may simply not want the default initialization
- In these cases, need to put the non-default base-class constructor call in the member initializer list
  - ...and it needs to be first in the initializer list.
Transport Ship’s Unit-Type

- **GameUnit** class has **getUnitType()** function
  - Returns “generic game unit”
- Would like our Transport Ship class to say that it’s a “transport ship”
- **TransportShip** class can **override** the function’s implementation
  ```cpp
  string TransportShip::getUnitType() const {
      return "transport ship";
  }
  ```
- Derived classes can override a base-class function
  - Replace the implementation completely, or specialize its behavior in some way
Overriding Base-Class Functions

- When overriding a function, function signature in derived class must match base class signature
  
  ```
  string getUnitType() const
  ```

- If signatures don’t match, C++ thinks you are simply adding a new function to the derived class
Calling Our Function

Try out our new “unit-type” function

GameUnit u(Point(20, 30), 1, 100);
TransportShip s(Point(30, 40), 1, 4);

cout << "I am a " << u.getUnitType() << endl;
cout << "I am a " << s.getUnitType() << endl;

Prints out:
I am a generic game unit
I am a transport ship
Display Helper-Function

- Now we decide to write a helper function:
  ```cpp
  void display(const GameUnit &u) {
    cout << "I am a " << u.getUnitType() << endl;
  }
  
  GameUnit u(Point(20, 30), 1, 100);
  TransportShip s(Point(30, 40), 1, 4);
  
  display(u);
  display(s);
  
  Prints out:
  I am a generic game unit
  I am a generic game unit
  ```
By default, C++ uses the variable’s type to figure out which function to call
- The type specified in the source code, and known at compile-time
- Called the variable’s “static type”

Code:
```c++
void display(const GameUnit &u) {
    cout << "I am a " << u.getUnitType() << endl;
}
```
- Static type of u is `GameUnit`, so C++ always calls `GameUnit::getUnitType()`
- ...even when u refers to a subclass of `GameUnit`
Virtual Functions

- The `virtual` keyword tells C++ to call the member function based on what is referred to, not on the static type of the variable
  - Called the runtime type of the variable
- In `GameUnit` class declaration, update the `getUnitType()` declaration:
  - `virtual` string getUnitType() const { ... }
- `TransportShip` doesn’t need to re-specify `virtual` when it overrides `getUnitType()`
  - The function’s “virtual-ness” is also inherited
- If `TransportShip` added its own new functions, and wanted its subclasses to override them:
  - Have to specify `virtual` on the new function declarations
Virtual Function Mechanics

- Why not make all functions virtual?
  - Need to understand how C++ implements virtual functions
- Declaring a function virtual causes C++ to store an extra pointer to correct version of the function
  - Stored in a “virtual function pointer table”
- The object itself knows which function to call
Virtual Function Mechanics (2)

- Two-step process for calling virtual functions:
  - Look up which version of the function to call, using the object itself
  - Go ahead and call that function

- Two costs of virtual functions:
  - To support virtual functions, objects require an extra pointer’s worth of space
    - If you need many objects of a particular type in memory at the same time, need to avoid virtual functions!
  - Virtual function calls also incur a (small) time overhead
    - For extremely performance-critical applications, may also want to avoid virtual functions due to this overhead

- Only declare functions `virtual` when you expect them to be overridden
One more fun test:

```
GameUnit *u1 = new GameUnit(p1, 1, 100);
GameUnit *u2 = new TransportShip(p2, 1, 4);
...
delete u1;
delete u2;
```

Which destructors are called?

- **Both** delete operations call the `GameUnit` destructor!
Virtual Destructors

- Base classes also need a **virtual destructor**
  - Probably the most important thing to do right, when it comes to class hierarchies

- C++ standard specifies that:
  - Deleting a derived class through a base-class pointer, with a non-virtual destructor, results in **undefined** behavior
  - Normally, you just leak memory, because the object isn’t entirely cleaned up

- When you intend a class to be subclassed, **always** give it a virtual destructor
  - Conversely, if you don’t want a class to be subclassed, **don’t** give it a virtual destructor
Virtual Destructors (2)

- When specifying a virtual destructor, make sure to give the destructor a body
  - Even when the destructor doesn’t do anything!
  - Not specifying a body for the destructor will produce linker errors at compile-time

- Updating our `GameUnit` class:
  ```cpp
  class GameUnit {
    ...

    // Give GameUnit a virtual destructor,
    // since we intend it to be subclassed.
    virtual ~GameUnit() { }
  };
  
  // GameUnit class doesn’t manage any dynamic resources
  ```
Calling Base-Class Functions

- When overriding a base-class function, a subclass may simply want to alter the base-class behavior
- Example: transport ship also needs to move
  - But, it needs to stay in the water...
- Override `GameUnit::move()`
  - Add in extra test to ensure the ship stays in the water
  - Then, call `GameUnit`'s version of `move()`
    ```cpp
    bool TransportShip::move(const Map &map, int direction) {
        Point nextLoc = location.neighbor(direction);
        if (map.cellType(nextLoc) == WATER)
            // Reuse base-class implementation.
            return GameUnit::move(map, direction);
        else
            return false;
    }
    ```
Private Member-Variables

- **GameUnit** is declared like this:
  ```cpp
class GameUnit {
    Point location;
    int playerID;
    ...
  }
```

- What access-level are these data members?
- *private* is default access-level for classes.

- Can **TransportShip** actually do this?
  ```cpp
  bool TransportShip::move(const Map &map, 
                           int direction) {
    Point nextLoc = location.neighbor(direction);
    ...
  }
  ```

- **No!** Only the class itself can access its private members.
- Produces a compiler-error.
The protected Access-Modifier

- To make base-class members accessible to subclasses, use protected access-modifier.

- Change `GameUnit` declaration to:
  ```
  class GameUnit {
    protected: // Make accessible to subclasses!
    Point location;
    int playerID;
    ...
  }
  ```

- Now `TransportShip` can access `GameUnit` variables without compiler errors:
  ```
  bool TransportShip::move(const Map &map, int direction) {
    Point nextLoc = location.neighbor(direction);
    ...
  }
  ```
When to use `private`? When `protected`?
No hard-and-fast rule. Opinions vary!

Some suggestions:
- In general, make parent-class members private, until you discover the *need* to manipulate them from the child class. **Then** make them protected.
- If parent class manages complicated structures, **definitely** make those things private. Instead, make protected functions for child classes to use.
- Similarly, if changes to variables need to be tightly controlled (e.g. for auditing or correctness reasons), make them private.
Abstract Functions

- Some base classes just declare behavior, but don’t define it
  - Concept represented by base class is too general to be able to define any reasonable behavior
  - Base class is *intended* to be extended

- The base class defines **abstract functions**
  - Base class can’t be instantiated or used directly
    - Some of the class’ behavior hasn’t been defined!
  - Subclasses provide the implementation
  - Subclasses are instantiated and used
Pure-Virtual Functions

- Functions are declared abstract by making them **pure virtual**
- Example:
  ```cpp
  // Generic build task.
  class Task {
  ...
   virtual bool run() = 0; // pure virtual
  };
  // Task that compiles source code.
  class CompileTask : public Task {
  ...
   virtual bool run() { ...
  }::* compile */
  };
  ```
- **run()** is not defined in **Task**, only declared.
- Subclasses can provide their own definitions of **run()**
Pure-Virtual Functions

- Trying to instantiate Task gives a compiler error
  
  Task *pt = new Task();  
  COMPILÆ ERROR

- Can still have variables of type Task& or Task*
  
  Task *pt = new CompileTask(conf);  OK!

  CompileTask has all Task members...

- Especially useful when providing generic processing capabilities for a variety of specialized objects
  
  void runAllTasks(vector<Task*> tasks);

  runAllTasks can use the generic Task interface

  Each task-object provides its own implementation of Task::run()
Using Pure-Virtual Functions

- Some base-classes have *some* pure-virtual functions
  - The base-class provides *some* implementation…
  - The subclass provides the missing pieces

- An “interface” is a base class with *all* pure-virtual functions
  - The base class is only declaring behavior, but provides no implementation at all
  - This concept of an “interface” is more explicit in other languages
  - **Note:** must still provide virtual (not pure-virtual) destructor implementation, to correctly support object deletion!
The OOP Concepts (again)

- So far, we have seen:
  - **Encapsulation**: hiding and guarding internal state
  - **Abstraction**: presenting a simple, high-level interface

- Class hierarchies introduce two more:
  - **Inheritance**: a child class gets its parent class’ members/behavior
    - …and a child-class can customize parent-class behavior
  - **Polymorphism**: calling a function on an object can exhibit different behaviors, based on the object’s type
    - In C++, this requires the use of `virtual` functions
This Week’s Lab

- Homework should be pretty straightforward
  - *Lots* of class-inheritance stuff!
- Complete the implementation of a simple algebra program
  > a = 5
  > b = 2 * (a + 6)
  = 22
- Represent expressions with an Abstract Syntax Tree (AST)
- Implement functionality to evaluate expressions