Inheritance in C++

- Today’s topics:
  - inheritance
  - virtual functions

History: OOP

- The ability to define a hierarchy of types is one of the main features of object-oriented programming
- Suppose an application has a class C
  - If the application needs another class C’ that is very similar, define C’ as an extension of C
    - Reuse design of C
    - Reuse code already written and debugged for C
    - Common code easier to maintain, e.g. update only one method

Example: Memory Simulation

- Suppose we are writing an application to simulate computer memory performance
- Goal: measure time required for a series of reads and writes
- Design a class named RAM
  - attributes:
    - size, latency
  - operations:
    - load(int addr)
    - store(int addr, int val)
- Get memory.tar.Z via the lectures page

Extending the Simulator

- Next suppose we want to extend the simulator to include cache memories
- Caches have all the attributes and methods of RAMs
- Additional behavior: variable latency, as a function of past history
  - New attribute:
    - missLatency
  - New method:
    - flush()

Cache Inherits from RAM

- Instead of rewriting load(), store(), etc for the new class, define Cache as an extension of RAM
- OOP terminology:
  - Cache inherits methods and attributes from RAM, or
  - Cache is derived from RAM

- RAM: base class
- Cache: derived class

Derived Class Structure

- Instances of the derived class have all the methods and attributes of the base class plus the new methods and attributes

```
RAM mp;
Cache mc;
mp.setLatency(90);
mp.setLoad();
mc.setLatency(10);
mc.setMissLatency(100);
```
Substructures

- From the diagram on the previous page, it might seem like a Cache is a structure that has a RAM as a substructure
  ```
  class Cache {
    RAM m;
    int missLatency;
    int flush();
  };
  ```

- Base classes are not substructures
- Members of substructures are used very differently than members of base classes

Base Class in C++

- In the last lecture you saw how to define a class
  ```
  class RAM {
    public:
    RAM(); // constructor(s)
    int latency(); // method prototypes
    void setLatency();
    ...
    private:
    int _latency; // instance variables
  };
  ```

Derived Class in C++

- To define a derived class, specify the base class name following the class name
  ```
  class Cache : RAM {
    // members of RAM have same access in Cache
  };
  ```

Base Class Visibility

- On the previous slide it’s clear the new methods are public, but what about the base class methods?
- In some designs, public base class methods should not be part of the interface of the derived class
  ```
  class Cache : public RAM {
    // members of RAM have same access in Cache
  };
  ```

Protected Members

- When we start writing code for Cache, we’ll discover a problem
- Methods and variables declared as private in RAM are not visible in derived classes
- We need to change RAM
  ```
  class RAM {
    ...
    protected:
    int _latency; // instance variables
  };
  ```
- Now _latency is visible to Cache but not visible to any other code
A New load() for Cache

- Having Cache inherit the load() method from RAM is not really what we want for the memory simulation.
  - RAM loads have a consistent latency.
  - Cache loads are fast (if the access is a hit) or slow (if it's a miss).
- Cache needs its own load() method.
  - If address is in cache, use fast access time.
  - Otherwise, update the internal directory, use slow access time.
- Cache will inherit other attributes and methods.
  - The Cache method should use the RAM method to do the actual transfer -- we want to reuse working code.

Plan 1: A New Method Name

- As a first approach, the Cache load() method will have a new name.
  ```cpp
  class Cache : public RAM {
    int cacheload(int addr) {
      if (hit(addr)) {
        load(addr); // call base class load()
        return _latency;
      } else
        ...
        ...
    }
  };
  ```

Plan 2: Overload the Base Class Name

- Using a new name defeats the purpose of OOP.
- There should be one method name, one interface implemented separately by each class.
  ```cpp
  class Cache : public RAM {
    int load(int addr) {
      if (hit(addr))
        load(addr); // infinite recursion!
    }
  }
  ```

Plan 2 (cont'd)

- A reference to the base class can be qualified by using the scope operator.
  ```cpp
  class Cache : public RAM {
    int load(int addr) {
      if (hit(addr))
        RAM::load(addr); // this works...
    }
  };
  ```

A Simulation with Several Memories

- Once the classes are defined, an application can instantiate objects of each type.
  ```cpp
  RAM mp;
  Cache mc;
  ```
  ```cpp
  t += mp.load(addr);
  t += mc.load(addr);
  ```
  The compiler knows the type of each object, compiles a call to the correct version of load().

Several Memories (cont'd)

- What if you want a variable number of memories?
- Since a Cache is a RAM, you might consider this:
  ```cpp
  RAM memory[n];
  memory[0] = Cache(...);
  ```
  ```cpp
  ```
  - The problem is, RAM and Cache objects have different sizes, so this either.
    - generates an error or warning at compile time, or
    - compiles, but blows up the program at runtime, or
    - compiles, but the copy is limited to the RAM subset.
Several Memories (cont’d)

✓ The size problem can be solved with pointers:

```cpp
RAM *memory[N];
memory[0] = new Cache(...);
memory[1] = new RAM(...);
```

But the pointer version introduces a new problem. What does the compiler do with the following code?

```cpp
RAM *memory[N];
memory[0] = new Cache(...);
...
memory[0]->load(addr);
```

Since memory is an array of pointers to RAM objects, the call is to RAM::load, despite the fact the object is an instance of Cache.

Several Memories (cont’d)

✓ But the pointer version introduces a new problem
✓ What does the compiler do with the following code?

Binding of a Method Name

✓ The problem comes down to a search:
- When there are several methods to choose from, which one should be used?

✓ Two aspects of the problem
  - when is the search performed?
    - compile time (preferred by C++, for efficiency)
    - runtime (OOP philosophy)
  - in which order does the search proceed?
    - base derived
    - derived base

Virtual Methods

✓ To define a method that is selected at runtime, use the keyword virtual
✓ Specify a method as virtual in the base class

```cpp
class RAM {
PUBLIC:
  public:
    virtual int load(...);
};
class Cache : RAM {
PUBLIC:
  public:
    int load(...);
};
```

Comparing Binding Order

If load is a normal function, the binding is done at compile time, and the compiler uses a base class method

If load is a virtual function, the binding occurs at runtime, and the search proceeds from the derived class toward the base class

Example

See the programs in memory.tar
- mem.C derived class overloads load() method
- vmem.C base class defines load() as virtual

Challenge:
- modify vmem.C so the << operator also prints an object state that is selected at runtime
  ```cpp```
  ```cpp
  for (i = 0; i < N; i++)
    cout << memory[i] << endl;
  ```
  ```cpp```