C++: Inheritance III

Advanced Topics

Logistics

• Project
  – Part 2 (water) due Sunday, Oct 16th
• Questions?

Logistics

• Final exam
  – Good news…bad news
  – Good news
    • Last day of finals, November 18th
  – Bad news
    • 8am-10am
  – Room
    • 01-3338

Plan for this week

• Today: Intro to Memory Management
• Tomorrow: Memory Problems
• Thursday: Standard Libraries
  – Advanced Inheritance

Before we begin

• Any questions?

Standard Libraries

• Standard “C” libraries
• Standard C++ libraries
  – Standard C libraries
  – istream
  – STL
  – http://www.cplusplus.com/ref/
Plan for today

- Inheritance
  - What goes on behind the scenes
  - What’s up with the syntax:

```cpp
virtual void foo() = 0;
```

Subclassing and Inheritance

- When you define a class as a subclass:
  - The subclass \textit{inherits} all of the data members and methods of the superclass.
  - In addition, a subclass can have data/methods that are its own.
  - Inheritance is transitive:
    - I.e. If B is a subclass of A and C is a subclass of B, then C inherits the data/methods from both B and A.

Inheritance

- Behind the scenes
  - The memory allocated for an object of a derived class consists of:
    - An area for the base class’s data
    - An area for the derived class’s data

```cpp
class Performer
{
private:
  float basePay;
  char *name;
  char *talent;
  
public:
  Performer(char *name, char *talent);
  virtual float calculatePay();
}
```

```cpp
class Musician : public Performer
{
private:
  Instrument *axe;
  bool isRecorded;
  
public:
  Musician(char *name);
  virtual float calculatePay();
  virtual void setInstrument(Instrument *I);
}
```

Subclassing

- Define a more general class “Performer”.
- Both Actors and Musicians are specializations of Performer

```
  \text{superclass}
  \text{Performers}
  \text{subclasses}
  \text{Actors}
  \text{Musicians}
```

Inheritance

- What the memory for Musician looks like
Inheritance

- Let’s add a Drummer
  ```cpp
  class Drummer : public Musician {
  private:
    Drum kit[];
  public:
    Drummer (char *name);
    virtual void setInstrument (Instrument *I);
  }
  ```

Inheritance

- What the memory for drummer looks like

Virtual Functions

- What about virtual functions?
  - C/C++ allows one to define pointers to functions.
  - For all classes with virtual functions (defined or redefined), the compiler will create an array of pointers to virtual functions (VTABLE)
  - Objects of such classes have a hidden data member (vpt) which will point to the correct VTABLE array.
- Note: suggested, not required implementation
  - Your mileage may vary.

Virtual Functions

- Vtable created at compile time
  - Initialized with values of Base class unless overridden by derived class.

- Vptr set at run time.
  - Each object (derived or otherwise) will have a SINGLE vptr.
  - During construction
  - Based on actual derived class.
Virtual Functions

• Calling virtual functions
  – In the original C++ interpreter:

    `Instrument *I = new Instrument();
    I->play();`

  Was converted to

    `((I->__vptr)[0]) (I);`

Pure virtual functions

• What happens if you have a pure virtual function?
  – Virtual functions are nothing but entries in an array of function pointers.
  – Pointers are pointers
    • I.e. All pointers can be set to NULL.
  – Thus, the syntax:
    `virtual void foo() = 0;`

Virtual Functions and Constructors

• Why you should not call virtual functions in a constructor
  – Recall:
    • When a member of a derived class is constructed, the constructor of it’s base class is called first
      • This fills in the memory area containing members of the base class
      • This includes the setting of the vptr
      • The base constructor will conclude before the derived constructor begins.

Pure virtual functions

• If a Base class defines a pure virtual function not overridden by the derived class:
  – That entry in the derived class’s VTABLE will be 0.
  – Of course, the compiler will catch this.

Virtual Functions and Constructors

```cpp
class A {
    public:
        A();
        ~A();
        virtual void func1();
        virtual void func2();
        virtual void func3() = 0;
    }

class AA : public A {
    public:
        AA();
        ~AA();
        void func3();
    }
```
Virtual Functions and Constructors

• It’s worse if A’s constructor calls func3().
  – What happens after statement
    • AA*myAA = new AA();

1. AA::AA() calls A::A()
2. A::A() sets vptr to A’s VTABLE
3. A::A() executes (calls A’s func3())
4. core dumped and compiler will NOT catch this.

Slicing

• Recall that polymorphism can only be achieved using pointers rather than objects themselves.
• Attempts to copy a base class object with a derived class object will cause slicing.
  – Only the base class section of the derived object will be copied.

Slicing

basePay
name
talent
P

Musician M ("Ringo");
Performer P (M);
Space allocated for P
Copy constructor called

Correct Polymorphism

M
P

Musician *M (new Musician ("Ringo");)
Performer *P (M);
M allocated on heap
Pointer copy.
M still has Musician’s vptr since no copy constructor was called.

Virtual Functions and Slicing

• During copy construction,
  – The vptr will get set to the VTABLE for the class of object being copied
    Musician M ("Ringo");
    Performer P (M);
  – Performer’s copy constructor called
    • P is a Performer
    • vptr points to Performer’s VTABLE
    • virtual functions are Performer’s.

Virtual Functions and Destructors

• Why define destructors to be virtual
  – The destructor of a derived class will always call the destructor of it’s Base class.
  – However, even with correct polymorphism, there’s no guarantee that the derived class’s destructor will be called.
Virtual Functions and Destructors

Base *B = new Derived();
...
delete B; // Base::~Base() called
      // Derived part of B never gets cleaned up

Virtual Functions and Destructors

• If Base’s destructor was declared as virtual.

class Base {
  public:
    Base();
    virtual ~Base();
  }

• A pointer to the destructor will be placed in the VTABLE.
  – This pointer will get overridden by derived class’s destructor

Virtual Functions and Destructors

Base *B = new Derived();
...
// Destructor of B is found in the VTABLE pointed to
// by B’s vptr (namely ~Derived)
// Derived destructor is called
// which in turn will call Base’s destructor.

Questions?

Virtual Functions and Destructors

• Tomorrow: Open Source / GNU
• Thursday: Final Exam Review.

Questions?