Programming Languages

Lecturer: William W.Y. Hsu
Object-Oriented Programming

- Control or PROCESS abstraction is a very old idea (subroutines!), though few languages provide it in a truly general form (Scheme comes close).
- Data abstraction is somewhat newer, though its roots can be found in Simula67.
  - An Abstract Data Type is one that is defined in terms of the operations that it supports (i.e., that can be performed upon it) rather than in terms of its structure or implementation.
Object-Oriented Programming

- Why abstractions?
  - Easier to think about - hide what doesn't matter
  - Protection - prevent access to things you shouldn't see.
  - Plug compatibility
    - Replacement of pieces, often without recompilation, definitely without rewriting libraries.
    - Division of labor in software projects.
Object-Oriented Programming

- We talked about data abstraction some back in the unit on naming and scoping.
- Recall that we traced the historical development of abstraction mechanisms.
  - Static set of var Basic
  - Locals Fortran
  - Statics Fortran, Algol 60, C
  - Modules Modula-2, Ada 83
  - Module types Euclid
  - Objects Smalltalk, C++, Eiffel, Java
    Oberon, Modula-3, Ada 95
Object-Oriented Programming

- **Statics** allow a subroutine to retain values from one invocation to the next, while hiding the name in-between.
- **Modules** allow a collection of subroutines to share some statics, still with hiding.
  - If you want to build an abstract data type, though, you have to make the module a manager.
Object-Oriented Programming

- **Module** types allow the module to be the abstract data type - you can declare a bunch of them:
  - This is generally more intuitive.
  - It avoids explicit object parameters to many operations.
  - One minor drawback: If you have an operation that needs to look at the innards of two different types, you'd define both types in the same manager module in Modula-2.
  - In C++ you need to make one of the classes (or some of its members) "friends" of the other class.
Object-Oriented Programming

- Objects add inheritance and dynamic method binding.
- Simula 67 introduced these, but didn't have data hiding.
- The 3 key factors in OO programming:
  - Encapsulation (data hiding).
  - Inheritance.
  - Dynamic method binding.
Encapsulation and Inheritance

- Visibility rules:
  - Public and Private parts of an object declaration/definition
  - 2 reasons to put things in the declaration:
    - So programmers can get at them.
    - So the compiler can understand them.
  - At the very least the compiler needs to know the size of an object, even though the programmer isn't allowed to get at many or most of the fields (members) that contribute to that size.
    - That's why private fields have to be in declaration.
Object Oriented Programming Benefits

- Reduces conceptual load.
- Fault containment.
- Independent between components.
Class-as-Type

- A class can be viewed as a “class-as-type,” when it contains multiple variables and/or functions to manipulate these variables.
Public Variables

• Generally, people consider it bad form to put variables in the public space.
• Why?
Class-as-Manager

A class can be viewed as a “class-as-manager” when a is not directly used to contain variables but rather to “manage” other objects.
Public, Private, and Protected (in C++)

- **Public** fields are available to everyone.
- **Private** fields are available only to the class.
- **Protected** are available only to decedents.
- Furthermore, when inherited classes are declared as protected this makes all public functions of the base act like Protected.
- A C++ structure (**struct**) is simply a class whose members are public by default.
- C++ base classes can also be public, private, or protected.
Inheritance

class foo{
    public int x;
    private int y;
    protected int z;
}

class bar : foo {
    public int xy() {return x+y;} //invalid
    public int xz() {return x+z;} //valid
}

... 
bar bbb;
bbb.x = 1; //valid
bbb.y = 1; //invalid
bbb.z = 1; //invalid
Scoping

- In order to refer to a function for a particular class, then one must use the scope operator ::

```cpp
class foo{
    public int bar();
}
int foo::bar(){ return 0;}
```
General Purpose Base Class

• Its possible to define a general purpose base class as the highest level of abstraction.
  ▫ For example, Object in Java.
Modifying Base Class Methods

- It's possible to modify base methods for each inherited function.

```java
class foo{
    public int bar();
}
...
class qud : foo{
    public int bar() { return 1; }
}
```
Modifying Base Class Methods

• Usually better just to use base if possible and try to return errors.

class foo{
    public int bar();
}
...

class qud : foo{
    public int bar() { try {return foo::bar();} ... };
}
Protection Rules (in C)

- Any class can limit visibility of its member.
- A derived class can restrict the visibility of a base class, but never increase it.
- A derived class that limit visibility of members of a base class as protected or private can restore the visibility of individual members by inserting a “using” declaration in protected or public portion of the derived class.
Java vs C#

- Under Java derived classes cannot change the visibility from the base class.
  - BUT! It can be redefined.
- Protected has a slightly different meaning.
  - Under Java, protected means that it is visible by everything in the package and derived objects.
  - Without protected, it is visible by everything in the package, but not derived packages.
Dynamic Method Binding

- Virtual functions in C++ are an example of **dynamic method binding**.
  - You don't know at compile time what type the object referred to by a variable will be at run time.
- Simula also had virtual functions (all of which are abstract).
- In Smalltalk, Eiffel, Modula-3, and Java all member functions are virtual.
**Dynamic Method Binding**

- Note that inheritance does not obviate the need for generics.
  - You might think: Hey, I can define an abstract list class and then derive `int_list, person_list, etc.` from it, but the problem is you won't be able to talk about the elements because you won't know their types.
  - That's what generics are for: abstracting over types.
- Java doesn't have generics, but it does have (checked) dynamic casts.
Dynamic Method Binding

- Data members of classes are implemented just like structures (records).
  - With (single) inheritance, derived classes have extra fields at the end.
  - A pointer to the parent and a pointer to the child contain the same address - the child just knows that the `struct` goes farther than the parent does.
Dynamic Method Binding

- Non-virtual functions require no space at run time; the compiler just calls the appropriate version, based on type of variable.
  - Member functions are passed an extra, hidden, initial parameter: this (called current in Eiffel and self in Smalltalk).
- C++ philosophy is to avoid run-time overhead whenever possible (Sort of the legacy from C).
  - Languages like Smalltalk have (much) more run-time support.
Dynamic Method Binding

- Virtual functions are the only thing that requires any trickiness (Figure 9.3).
  - They are implemented by creating a dispatch table (vtable) for the class and putting a pointer to that table in the data of the object.
  - Objects of a derived class have a different dispatch table.
    - In the dispatch table, functions defined in the parent come first, though some of the pointers point to overridden versions.
    - You could put the whole dispatch table in the object itself.
    - That would save a little time, but potentially waste a LOT of space.
Dynamic Method Binding

Figure 9.3 Implementation of virtual methods. The representation of object F begins with the address of the vtable for class foo. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of F consists of the representations of its fields.
Dynamic Method Binding

class bar : public foo {
    int w;
public:
    void m(); //override
    virtual double s( ... 
    virtual char *t( ... 
    ...
} B;

Figure 9.4 Implementation of single inheritance. As in Figure 9.3, the representation of object B begins with the address of its class’s vtable. The first four entries in the table represent the same members as they do for foo, except that one —m— has been overridden and now contains the address of the code for a different subroutine. Additional fields of bar follow the ones inherited from foo in the representation of B; additional virtual methods follow the ones inherited from foo in the vtable of class.
Dynamic Method Binding

- Note that if you can query the type of an object, then you need to be able to get from the object to run-time type info.
  - The standard implementation technique is to put a pointer to the type info at the beginning of the vtable.
  - Of course you only have a vtable in C++ if your class has virtual functions.
  - That's why you can't do a dynamic_cast on a pointer whose static type doesn't have virtual functions.
Multiple Inheritance

- In C++, you can say
  ```cpp
  class professor : public teacher, public researcher {
      ...
  }
  ```
  Here you get all the members of teacher and all the members of researcher
- If there's anything that's in both (same name and argument types), then calls to the member are ambiguous; the compiler disallows them.
Multiple Inheritance

You can of course create your own member in the merged class:

```cpp
professor::print () {
    teacher::print ();
    researcher::print (); ...
}
```

Or you could get both:

```cpp
professor::tprint () {
    teacher::print ();
}
professor::rprint () {
    researcher::print ();
}
```
Multiple Inheritance

- Virtual base classes: In the usual case if you inherit from two classes that are both derived from some other class B, your implementation includes two copies of B's data members.
- That's often fine, but other times you want a *single* copy of B.
  - For that you make B a virtual base class.
Object-Oriented Programming

- Anthropomorphism is central to the OO paradigm - you think in terms of *real-world* objects that interact to get things done.
- Many OO languages are strictly sequential, but the model adapts well to parallelism as well.
- Strict interpretation of the term:
  - Uniform data abstraction - everything is an object.
  - Inheritance.
  - Dynamic method binding.
Object-Oriented Programming

- Lots of conflicting uses of the term out there object-oriented style available in many languages:
  - Data abstraction crucial.
  - Inheritance required by most users of the term O-O.
  - Centrality of dynamic method binding a matter of dispute.
Object-Oriented Programming

- **SMALLTALK** is the canonical object-oriented language.
  - It has all three of the characteristics listed above.
  - It's based on the thesis work of Alan Kay at Utah in the late 1960’s.
  - It went through 5 generations at Xerox PARC, where Kay worked after graduating.
  - Smalltalk-80 is the current standard.

- Other languages are described in what follows:
  - **Modula-3**
    - Single inheritance.
    - All methods virtual.
    - No constructors or destructors.
Object-Oriented Programming

- Ada 95
  - *Tagged* types.
  - Single inheritance.
  - No constructors or destructors.
  - *Class-wide* parameters:
    - Methods static by default.
    - Can define a parameter or pointer that grabs the object-specific version of all methods.
    - Base class doesn't have to decide what will be virtual.
  - Notion of child packages as an alternative to friends.
Object-Oriented Programming

- Java
  - Interfaces, mix-in inheritance.
  - Alternative to multiple inheritance.
  - Basically you inherit from one real parent and one or more interfaces, each of which contains only virtual functions and no data.
  - This avoids the contiguity issues in multiple inheritance above, allowing a very simple implementation.
  - All methods virtual.
Object-Oriented Programming

• Is C++ object-oriented?
  ▫ Uses all the right buzzwords.
  ▫ Has (multiple) inheritance and generics (templates).
  ▫ Allows creation of user-defined classes that look just like built-in ones.
  ▫ Has all the low-level C stuff to escape the paradigm.
  ▫ Has friends.
  ▫ Has static type checking.

• In the same category of questions:
  ▫ Is Prolog a logic language?
  ▫ Is Common Lisp functional?
Object-Oriented Programming

• However, to be more precise:
  ▫ Smalltalk is really pretty purely object-oriented.
  ▫ Prolog is primarily logic-based.
  ▫ Common Lisp is largely functional.
  ▫ C++ can be used in an object-oriented style.