Chapter 9 :: Data Abstraction and Object Orientation

Programming Language Pragmatics

Michael L. Scott

ELSEVIER

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

- By deriving new classes from old ones, the programmer can create arbitrarily deep *class hierarchies*, with additional functionality at every level of the tree.
- The Smalltalk class hierarchy for Smalltalk has as many as seven levels of derivation (see attached Figure 9.2)

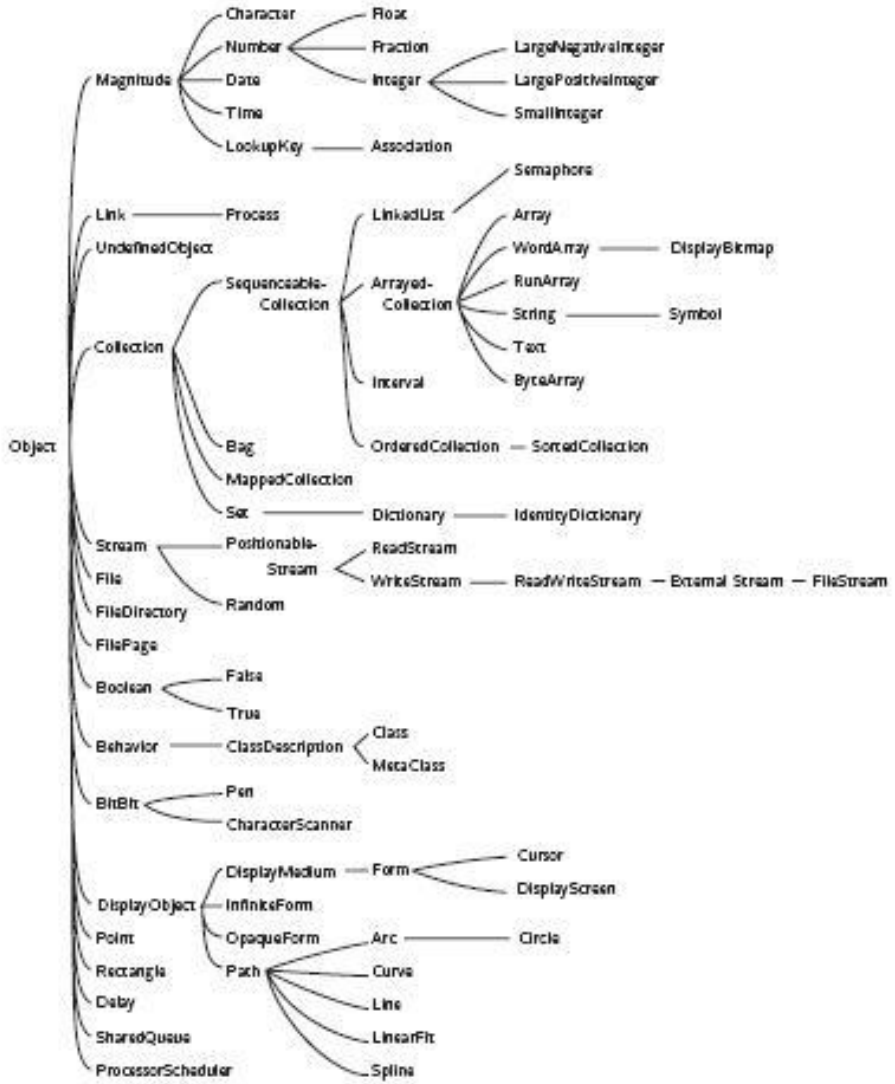

Figure 9.2: The standard class hierarchy of Smalltalk-80.

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

Encapsulation and Inheritance Classes (C++)

- C++ distinguishes among
  - public class members
    - accessible to anybody
  - protected class members
    - accessible to members of this or derived classes
  - private
    - accessible just to members of this class
- A C++ structure (*struct*) is simply a class whose members are public by default
- C++ base classes can also be public, private, or protected

Encapsulation and Inheritance Classes (C++)

- C++ access specifiers

```
1  // classes example
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      void set_values (int,int);
9      int area() {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13   width = x;
14   height = y;
15 }
16
17 int main () {
18   Rectangle rect;
19   rect.set_values (3,4);
20   cout << "area: " << rect.area();
21   return 0;
22 }
```

Encapsulation and Inheritance Classes (C++)

- Example:

```
class circle : public shape { ...
```

anybody can convert (assign) a circle* into a shape*

```
class circle : protected shape {
...
```

only members and friends of circle or its derived classes can convert (assign) a circle* into a shape*

```
class circle : private shape { ...
```

only members and friends of circle can convert (assign) a circle* into a shape*

Encapsulation and Inheritance Classes (C++)

- inheritance example
  - derived classes contain

    `width, height, set_values`
  - output

```
20
10
```

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Encapsulation and Inheritance Classes (C++)

- inheritance
  - access types and inheritance

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

  - inherited members have same access permissions as in base class

```
1 Polygon::width              // protected access
2 Rectangle::width            // protected access
3
4 Polygon::set_values()       // public access
5 Rectangle::set_values()     // public access
```

since

```
class Rectangle: public Polygon { /* ... */ }
```

Encapsulation and Inheritance Classes (C++)

- Disadvantage of the module-as-manager approach: include explicit create/initialize & destroy/finalize routines for every abstraction
  - Even w/o dynamic allocation inside module, users don't have necessary knowledge to do initialization
  - Ada 83 is a little better here: you can provide initializers for pieces of private types, but this is NOT a general approach
  - Object-oriented languages often give you constructors and maybe destructors
    - Destructors are important primarily in the absence of garbage collection

Encapsulation and Inheritance Classes (C++)

- A few C++ features you may not have learned:
  - classes as members

    ```
    foo::foo (args0) : member1 (args1),
    member2 (args2) { ...
    ```

    args1 and args2 need to be specified in terms of args0
    - The reason these things end up in the header of foo is that they get executed *before* foo's constructor does, and the designers consider it good style to make that clear in the header of foo::foo

Encapsulation and Inheritance Classes (C++)

- A few C++ features (2):
  - initialization v. assignment

```
foo::operator=(&foo) v.
foo::foo(&foo)
        foo b;
        foo f = b;
              // calls constructor
        foo b, f;
              // calls no-argument constructor
        f = b;
              // calls operator=
```

Encapsulation and Inheritance Classes (C++)

– example

```
1  // example: class constructor
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      Rectangle (int,int);
9      int area () {return (width*height);}
10 };
11
12 Rectangle::Rectangle (int a, int b) {
13   width = a;
14   height = b;
15 }
16
17 int main () {
18   Rectangle rect (3,4);
19   Rectangle rectb (5,6);
20   cout << "rect area: " << rect.area() << endl;
21   cout << "rectb area: " << rectb.area() << endl;
22   return 0;
23 }
```

notes:

results same as before

set_values omitted

values passed to constructor

output:

```
rect area: 12
rectb area: 30
```

Encapsulation and Inheritance Classes (C++)

- A few C++ features (3):
  - virtual functions (see the next dynamic method binding section for details):
    Key question: if child is derived from parent and I have a parent* p (or a parent& p) that points (refers) to an object that's actually a child, what member function do I get when I call p->f (p.f)?
    - Normally I get p's f, because p's type is parent*.
    - But if f is a virtual function, I get c's f.

Encapsulation and Inheritance Classes (C++)

- A few C++ features (4):
  - virtual functions (continued)
    - If a virtual function has a "0" body in the parent class, then the function is said to be a *pure* virtual function and the parent class is said to be *abstract*
    - You can't declare objects of an abstract class; you have to declare them to be of derived classes
    - Moreover any derived class ***must*** provide a body for the pure virtual function(s)
    - multiple inheritance in Standard C++ (see next)
  - friends
    - functions
    - classes

Initialization and Finalization

- In Section 3.2, we defined the lifetime of an object to be the interval during which it occupies space and can hold data
  - Most object-oriented languages provide some sort of special mechanism to *initialize* an object automatically at the beginning of its lifetime
    - When written in the form of a subroutine, this mechanism is known as a *constructor*
    - A constructor does not allocate space
  - A few languages provide a similar *destructor* mechanism to *finalize* an object automatically at the end of its lifetime

Initialization and Finalization

Issues

- choosing a constructor
- references and values
  - If variables are references, then every object must be created explicitly - appropriate constructor is called
  - If variables are values, then object creation can happen implicitly as a result of elaboration
- execution order
  - When an object of a derived class is created in C++, the constructors for any base classes will be executed before the constructor for the derived class
- garbage collection

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 9.4)
  - 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 (Figure 10.5)
    - 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

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k ( ...
    virtual int l ( ...
    virtual void m ();
    virtual double n( ...
    ...
} F;
```

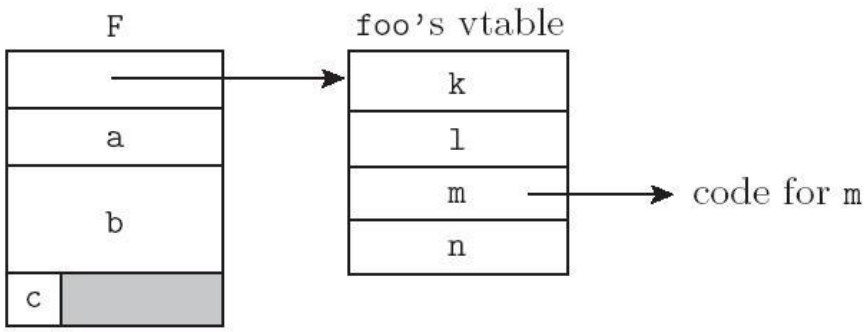

Figure 9.4: **Implementation of virtual methods.** The representation of object `F` begins with the address of the vtable for class `foo`. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. 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;
```

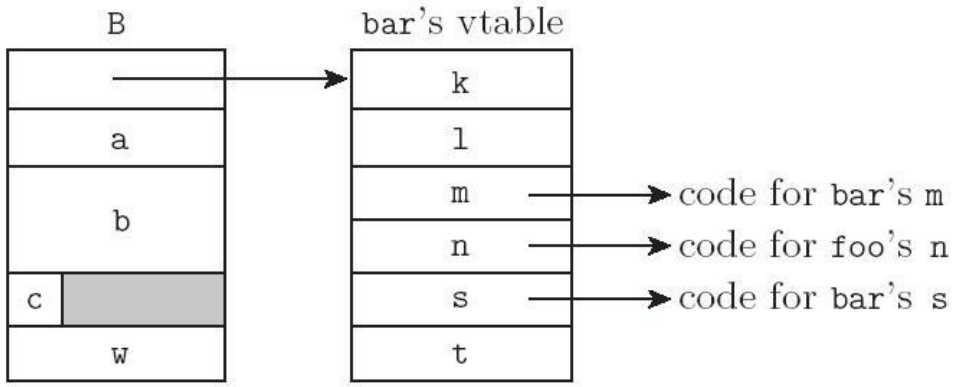

Figure 9.5: **Implementation of single inheritance.** As in Figure 9.4, the representation of object `B` begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for `foo`, except that one—`m`—has been overridden and now contains the address 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 `bar`.

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

```
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

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

Or you could get both:

```
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

Object-Oriented Programming

- 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

Object-Oriented Programming

- In the same category of questions:
  - Is Prolog a logic language?
  - Is Common Lisp functional?
- 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