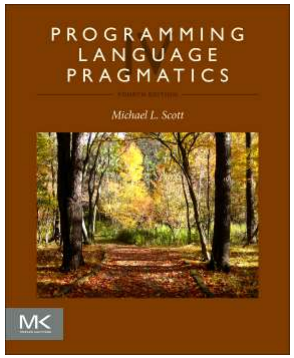
Chapter 10 :: Data Abstraction and Object Orientation

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



ELSEVIER

Object-Oriented Programming

- Control or PROCESS abstraction is a very old idea (subroutines!)
- 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
- How did we do it before OO?
  - We created and manipulated a data structures.
  - "Manager functions": pre-OO formalism

```
l = list_create()
list_append(l, o)
```

C libraries still use this paradigm: GTK, Linux Kernel, etc.

- In this chapter, object = instance of a class.

Object-Oriented Programming

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

Object-Oriented Programming

- **Elements of object-oriented programming:**
- Data items to be manipulated are ***objects***.
- Objects are members of ***classes***, that is, classes are types.
- Objects store data in ***fields*** and behaviour in ***methods*** specified by their classes.
- **Main characteristics of most object-oriented programming systems:**
- ***Encapsulation*** by hiding internals of an object from the user of the object.
- Customization of behaviour through ***inheritance***.
- Polymorphism through ***dynamic method binding***.

Object-Oriented Programming

- OOP is cross-cuts other programming paradigms:
  - –Imperative OO (C++, Java, C#, Python, etc.)
  - –Functional OO (Ocaml, CLOS, etc.)
  - –Logical OO (Flora-2)
- OO adds:
  - –Convenient syntax,
  - –Inheritance,
  - –Dynamic method binding,
  - –Encapsulation.

Benefits of Object-Oriented Programming

- It ***reduces conceptual load***:
  - –It reduces the amount of detail the programmer must think about at the same time
- It provides ***fault and change containment***:
  - – It limits the portion of a program that needs to be looked at when debugging
  - –It limits the portion of a program that needs to be changed when changing
  - –The behaviour of an object without changing its interface
- It provides ***independence of program components*** and thus ***facilitates code reuse***.
- **Note:** Most of these are consequences of encapsulation and thus apply also to programming using modules.

Object-Oriented Programming

- The language needs a way of defining a class:
  - –Name,
  - –Superclasses (incl. Interfaces),
  - –Fields,
  - –Methods.

  Fields + Methods = Members of the class
- Note: classes do not need to be defined in a single file:
  - –C++ allows a definition to be split up over multiple files,
  - –Java allows more than one class per file (one is public)
- A class needs ways to:
  - –Allocate objects (constructors and the `new` operator)
    - •Some languages allow to allocate in all 3 areas, not just heap
  - –Change fields
  - –Invoke methods

Class Example in C++

Header

```
class list_node {

  list_node *prev, *next, *head;

public:

  int val;

  list_node();
  ~list_node();

  list_node *predecessor();
  list_node *successor();
  bool singleton();
  void insert_before(list_node *new_node);
  void remove();
};
```

Private fields

Public field

Constructor

Destructor

Public methods

Implementation

```
void list_node::insert_before(list_node *new_node) {
  if (!new_node->singleton())
    throw new list_err("inserting more than a single node");
  prev->next     = new_node;
  new_node->prev = prev;
  new_node->next = this;
  prev           = new_node;
  new_node->head_node = head_node;
}
```

Method definition outside class needs to be qualified.

Reference to current object

ELSEVIER

Protection

- OO supports data hiding / protection:
  - –Keep implementation details from leaking into the larger program
- The 4 kinds of Visibility protection in Java:
  - –Public
  - –Protected
  - –Default (visible to classes in same module)
  - –Private
- Others are possible: C++ has Friend
  -

Protection

- A C++ friend class in C++ can access the "private" and "protected" members of the class in which it is declared as a friend:

```
class B {
    friend class A; // A is a friend of B
private:
    int i;
};
class A {
public:
    A(B b) { // the object has to be passed as a parameter to the function
        b.i = 0; // legal access due to friendship
    }
};
```

Inheritance

- Using inheritance we can define a new ***derived*** or ***child class*** based on an existing ***parent class*** or ***superclass***.
  - The derived class
  - Inherits all fields and methods of the superclass,
  - Can define additional fields and methods, and
  - Can override existing fields and methods.
- **Purpose:** Extend or specialize the behaviour of the superclass. This allows us to define a ***class hierarchy***.
  - If only single inheritance is allowed, the hierarchy is a tree.
  - If multiple inheritance is allowed, the hierarchy is a lattice.
- **C++**

  class push_button : public widget { ... }
- **Java**

  public class push_button extends widget { ... }
- **Ada**

  type push_button is new widget with ...

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

- 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 ();
}
```

Encapsulation and Inheritance

- 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

- 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*

Constructors and Destructors

- Objects often need to be initialized before they are used.
  - –This is because Objects represent things with semantics.
- Constructors are used to initialize objects.
- Languages often support multiple constructors.
  - –Overloading on # of arguments.
  - –Named constructors.
    - •Example: a Coordinate class.
      - –double x, double y OR
      - –double angle, double radius
- Important constructors:
  - –Zero Argument,
  - –Copying: must take a reference.

In C++ (objects use value semantics):
class C { ... }
C a; <- calls 0-argument constructor.
C b = a; <- calls copying constructor.
Differs from: C b;
b = a;

- Destructors:
  - –Called when an object goes away, to free up resourced used by it.
  - –Interact poorly with GC, especially in the presence of cycles.

Named Constructor

```
class Point {
public:
  Point(float x, float y);      // Rectangular coordinates
  Point(float r, float a);      // Polar coordinates (radius and angle)
  // ERROR: Overload is Ambiguous: Point::Point(float,float)
};

int main()
{
  Point p = Point(5.7, 1.2);    // Ambiguous: Which coordinate system?
  ...
}
```

One way to solve this ambiguity is to use the Named Constructor Idiom:

```
#include <cmath>                // To get std::sin() and std::cos()

class Point {
public:
  static Point rectangular(float x, float y);      // Rectangular coord's
  static Point polar(float radius, float angle);   // Polar coordinates
  // These static methods are the so-called "named constructors"
  ...
private:
  Point(float x, float y);      // Rectangular coordinates
  float x_, y_;
};

inline Point::Point(float x, float y)
  : x_(x), y_(y) { }

inline Point Point::rectangular(float x, float y)
{ return Point(x, y); }

inline Point Point::polar(float radius, float angle)
{ return Point(radius*std::cos(angle), radius*std::sin(angle)); }
```

Now the users of `Point` have a clear and unambiguous syntax for creating `Points` in either coordinate system:

```
int main()
{
  Point p1 = Point::rectangular(5.7, 1.2);    // Obviously rectangular
  Point p2 = Point::polar(5.7, 1.2);          // Obviously polar
  ...
}
```

Copy Constructor

```
class A {
  int x;
public:
  A()            : x(0)            { cout << "C1"; }
  A(const A& a) : x(a.x)           { cout << "C2"; }
  const A& operator =(const A& a) { x = a.x; cout << "A"; }
};

int main() {
  A u;      // Prints "C1"
  A v(u);   // Prints "C2"
  A w = u; // Prints "C2"
  A x;      // Prints "C1"
  x = u;    // Prints "A"
}
```

Nested Inner Class

- Inner Class = class defined inside another class:
  - –Need to decide which fields such a class sees
    - •Nothing - no special relationship to outer class (Python)
    - •Statics Only (C++/C#).
    - •Associated with every instance (Java)
      - –Needs link to instance of enclosing class.

Nested Inner Class Example

```
class CPU {
	double price;
	class Processor{
		double cores;
		String manufacturer;
		double getCache(){
			return 4.3;
		}
	}
	protected class RAM{
		double memory;
		String manufacturer;
		double getClockSpeed(){
			return 5.5;
		}
	}
}
public class Main {
	public static void main(String[] args) {
		CPU cpu = new CPU();
		CPU.Processor processor = cpu.new Processor();
		CPU.RAM ram = cpu.new RAM();
		System.out.println("Processor Cache = " + processor.getCache());
		System.out.println("Ram Clock speed = " + ram.getClockSpeed());
	}
}
```

When you run above program, the output will be:

```
Processor Cache = 4.3
Ram Clock speed = 5.5
```

Metaclasses

- In some languages, a class is an object (Python, Flora-2, not Java)
  - –Classes are instances of metaclass
    - •Constructing a metaclass

The type of the built-in classes you are familiar with is also `type`:

Python >>>

```
>>> for t in int, float, dict, list, tuple:
...     print(type(t))
...
<class 'type'>
<class 'type'>
<class 'type'>
<class 'type'>
<class 'type'>
```

For that matter, the type of `type` is `type` as well (yes, really):

Python >>>

```
>>> type(type)
<class 'type'>
```

`type` is a metaclass, of which classes are instances. Just as an ordinary object is an instance of a class, any new-style class in Python, and thus any class in Python 3, is an instance of the `type` metaclass.

Static vs. Dynamic Method Binding

- In languages with a reference model of variables or when using pointers in C++, we can use an object of a derived class where an object of the base class is expected.
- Assume the derived class overrides a method of the base class.
- When accessing an object of the derived class through a variable whose type is the base class, which method should we call?

```
class person {
public:
  void print_mailing_label();
};

class student : public person {
public:
  void print_mailing_label();
};

class professor : public person {
public:
  void print_mailing_label();
};

int main() {
  student   s;
  professor p;
  person   *x = &s, *y = &p;

  // professor::print_mailing_label
  p.print_mailing_label();
  // student::print_mailing_label
  s.print_mailing_label();
  // ???
  x->print_mailing_label();
  y->print_mailing_label();
}
```

Static vs. Dynamic Method Binding

- In static method binding static method binding, method selection depends on the type of the variable x and y
  - Method print_mailing_label() of class person is executed in both cases in both cases
  - Resolved at compile time
- In dynamic method binding method selection depends on the class of the objects s and p
  - Method print_mailing_label() of class student is executed in the first case, while the corresponding methods for class professor is executed in the second case
  - Resolved at run time

```
class person {
public:
  void print_mailing_label();
};

class student : public person {
public:
  void print_mailing_label();
};

class professor : public person {
public:
  void print_mailing_label();
};

int main() {
  student   s;
  professor p;
  person   *x = &s, *y = &p;

  // professor::print_mailing_label
  p.print_mailing_label();
  // student::print_mailing_label
  s.print_mailing_label();
  // ???
  x->print_mailing_label();
  y->print_mailing_label();
}
```

Static vs. Dynamic Method Binding

- Given C++'s focus on efficiency, its default is static method binding.
- Dynamic method binding is available by declaring the method to be ***virtual***.

```cpp
// CPP program to illustrate
// concept of Virtual Functions
#include<iostream>
using namespace std;

class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

```
print derived class
show base class
```

Static vs. Dynamic Method Binding

- Static Method Binding:
  Square Rectangle Shape
- Dynamic Method Binding:
  Square Square Square

```
class Shape:
  def name(): print "Shape"
  def color(): print "Blue"
class Rectangle(Shape):
  def name(): print "Rectangle"
  def color(): print "Red"
class Square(Shape)
  def name(): print  "Square"
Square s = new Square()
Rectangle r = s
Shape sh = s
s.name()
r.name()
sh.name()
```

Abstract Classes

- An ***abstract method*** is a method that is required to be defined only in derived classes.
- **C++**

```
class person {
...
virtual void print_mailing_label() = 0;
...
};
```

- **Java**

```
class person {
...
abstract void print_mailing_label();
...
};
```

- An ***abstract*** class has at least one abstract method and thus cannot be instantiated.
- If all methods are abstract, then all the class does is define an interface.

Implementation of Virtual Methods

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

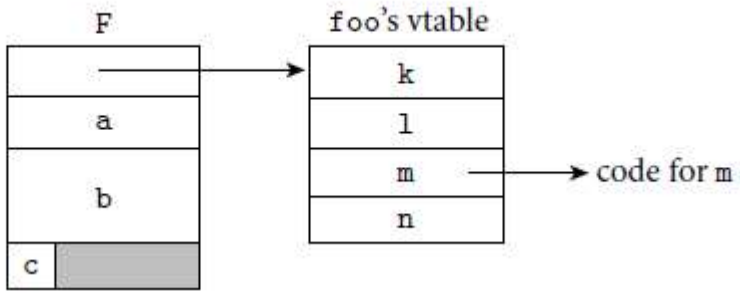
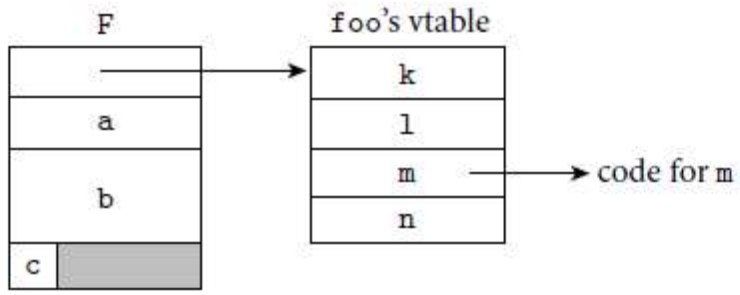


Figure 9.3 **Implementation of virtual methods.** The representation of object `F` begins with the address of the vtable for class `foo`. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class.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

- 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.

Implementation of Single Inheritance

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




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

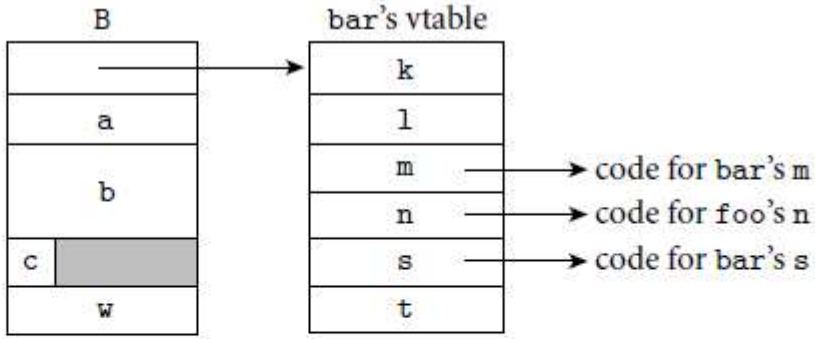
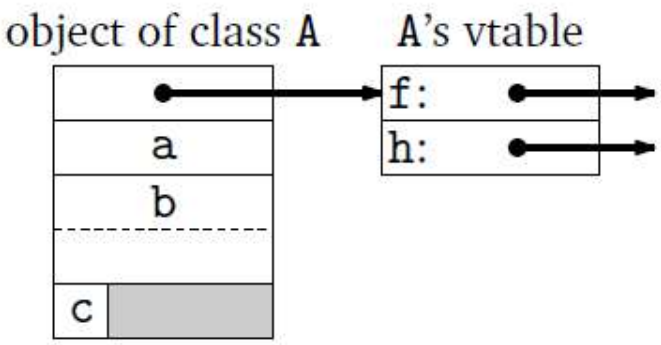


Figure 9.4 Implementation of single inheritance. As in Figure 9.3, the representation of object `B` begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for `foo`, except that one—`m`—has been overridden and now contains the address 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 `bar`.

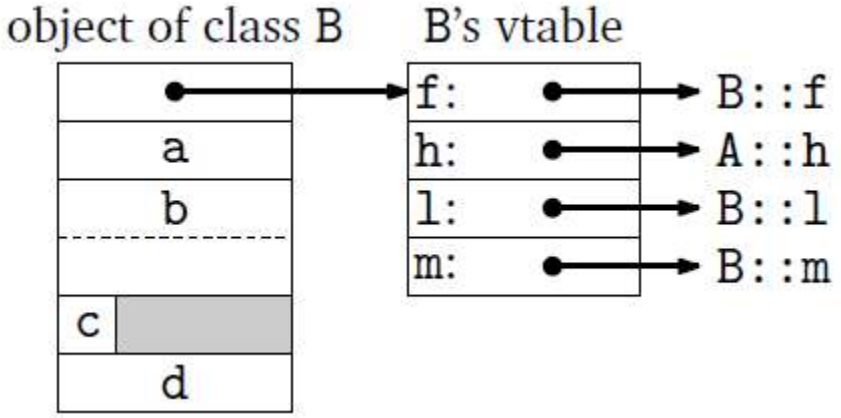
Implementation of Single Inheritance

```
class A {
  int    a;
  double b;
  char   c;
public:
  virtual void f();
  int g();
  virtual int h();
  double k();
};
```

```
class B : public A {
  int d;
public:
  void f();
  virtual double l();
  virtual double *m();
};
```



- **Vtable of derived class**
  - Copy vtable of base class.
  - Replace entries of overridden
  - virtual methods.
  - Append entries of virtual methods
  - declared in derived class.



Garbage Collection

- Automatic heap memory management.
  - –Alternative is to require explicit deletes.
  - –What is garbage? What is not garbage?
    - •Set of roots: - Registers - Stack - Statics - etc.
    - •Objects reachable from roots are live. - Recursively search objects for other objects. - Some objects don't need to be searched. (Strings, Arrays of basic types, etc.)
    - •Unreachable objects are dead, can be freed.
  - –Reference counts: Each object contains a field giving the number of objects (incl. stack frames) referring to it.
    - •When a reference is taken to object, inc refcount.
    - •When a reference is removed, dec refcount.
    - •When drops to 0, deallocate.
  - –Problem with circular references.

Garbage Collection

- Mark and Sweep:
  - Walk references, flag useful objects.
    - Requires space in header for flag.
  - Walk objects, find unflagged, dealloc.
    - Requires ability to walk objects.
  - Interesting technique: Pointer reversal.
    - Stores path to parent var in place of pointer to child obj.
- Stop and Copy (Compacting)
- Generational Collection:
  - Objects live short or long time.
  - Old objects rarely refer to much newer objects.
  - The partition of objects into different generations (time intervals) based on time of allocation, and giving them different GC policies depending on age.
- Incremental GC