OOP and Dynamic Method Binding

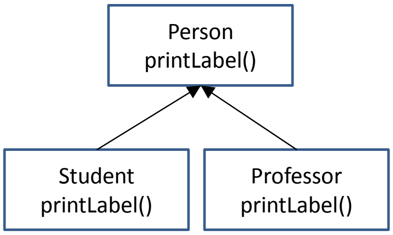
Chapter 9

Object Oriented Programming

- Skipping most of this chapter
- Focus on 9.4, Dynamic method binding
  - Polymorphism or Subtype Polymorphism
- One of three key factors in OOP
  - Encapsulation of data and methods (data hiding)
  - Inheritance
  - Dynamic method binding

Dynamic Method Binding

- Ability to use a derived class in a context that expects its base class



```
Student s = new Student()
Professor p = new Professor()

Person x = s;
Person y = p;

s.printLabel();
p.printLabel();

x.printLabel();   // Which one?
y.printLabel();
```

Dynamic Method Binding

```
x.printLabel();
```

- If we use Person’s printLabel() then this is using **static binding**
  - We say that Student’s printLabel() **redefines** Person’s printLabel()
- If we use Student’s printLabel() this this is using **dynamic binding**
  - We say that Student’s printLabel() **overrides** Person’s printLabel()
  - This is what always happens in Java
- C++ and C# let you do both

Virtual Methods

- Methods that can be overridden are called **virtual** methods
  - You might never have seen this term before since it’s not used in Java because all methods are virtual
- In C++:

```
class person
{
  public:
          virtual void printLabel();
```

Normally virtual methods are used when the object doesn’t know what implementation is to be used at compile time

Abstract Classes

- In most OOP languages we can omit the body of a virtual method in a base class
- Java and C# use the keyword abstract
  - A class defined as abstract must have at least one abstract method on it

```
abstract class person {
          public abstract void printLabel();
```

- C++ uses assignment to 0

```
class person {
          public:
                    virtual void printLabel() = 0;
```

An **interface** is identical to an abstract class with all abstract methods

Dynamic Method Binding

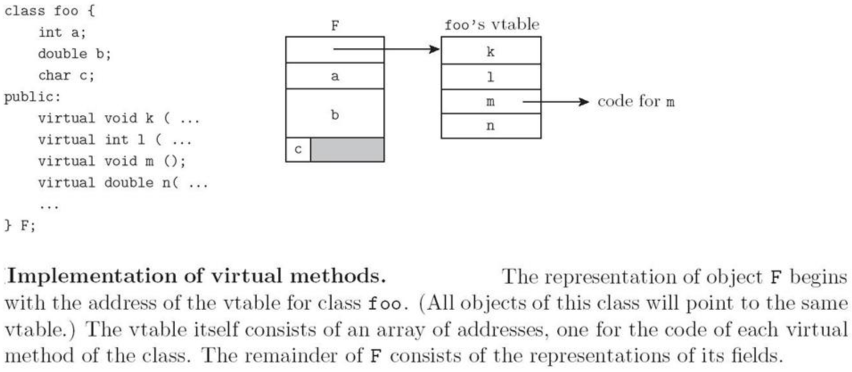
- Non-virtual methods require no space at run time; the compiler just calls the appropriate version, based on type of variable
  - Member functions are passed an extra, hidden, initial parameter: *this* (called *Me* in VB and *self* in Smalltalk)
- C++ philosophy is to avoid run-time overhead whenever 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
  - 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

Implementation of Virtual Methods

vtable = virtual method table

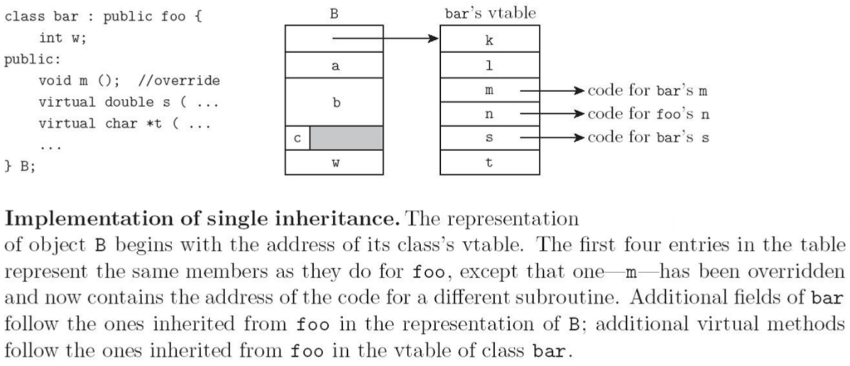


```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k ( ...
    virtual int l ( ...
    virtual void m ();
    virtual double n( ...
    ...
} F;
```

F
a
b
c
foo's vtable
k
l
m
n
code for m

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.

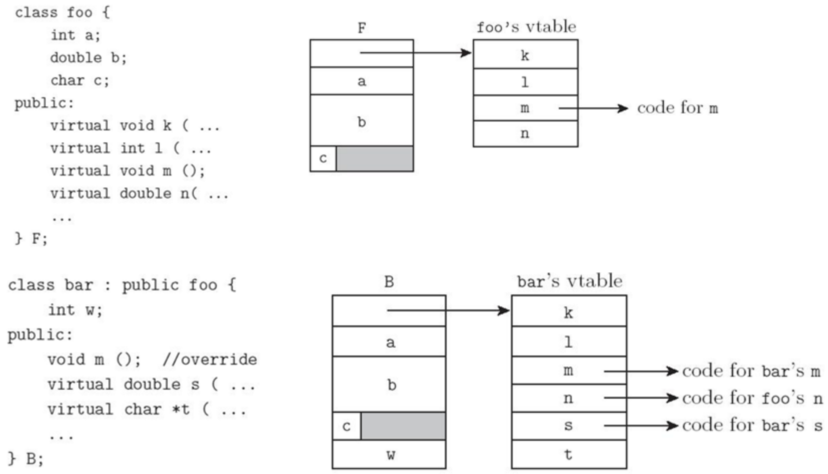
Implementation of Virtual Methods



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

B
a
b
c
w
bar's vtable
k
l
m
n
s
t
code for bar's m
code for foo's n
code for bar's s

Implementation of single inheritance. 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.



```
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;
```

```
Foo *f = new Foo();
Bar *b = new Bar();
Foo *q = b;
Bar *s = f;                    // static semantic error
```

Different on Stack:

```
Foo f;
Bar b;
Foo q = b;
Bar s = f;  // Error
```

Dynamic Type 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