Programming Language & Compiler

# *Data Abstraction*

**Hwansoo Han**

# Data Abstraction

- Data abstraction's roots can be found in Simula67

- An abstract data type (ADT) is defined
  - In terms of the operations that it supports (*i.e.*, that can be performed upon it)
  - Not its structure or implementation

# Abstraction

- Why abstractions?
  - Easier to think about
    - Hide what doesn't matter
    - Reduce conceptual load
  - Fault containment (protection)
    - Prevent inappropriate usages of components
    - Prevent access to things you shouldn't see
  - Independence among components
    - Modification of internal implementation without changing external code
    - Division of labor in software projects

# Object-Oriented Programming

- Three key factors in OOP
  - Encapsulation (data hiding)
  - Inheritance
  - Dynamic method binding

# Public and Private Members

- Public members
  - Visible outside the class
- Private members
  - Only visible within the class
  - In C++, members are private by default
  - In Java, members are public by default
- Class declaration in *header* file (*.h)
- Method bodies in *implementation* file (*.cc)
  - Scope resolution operator '::'

    void list_node::insert(list_node* new_node) { … }

```
class list_node {
    list_node* next;
    list_node* head;
 public:
    int  val;
    list_node();
    void insert(list_node*);
};
```

# Tiny Subroutines

- OOP tends to make many more subroutine calls
  - Many of them tend to be short
- *Property* mechanism in C#
  - Specify accessors (*get* and *set* values)

```
// definitions of accessors
class list_node {
    int val;            // private
    public int Val {
        get { return val; }
        set { val = value; }  // optional
    }
    ...
}
```

```
// usage of accessors
list_node n;
...
int a = n.Val;  // implicitly call get
n.Val = 3;      // implicitly call set
```

# Derived Classes

- A class X is refined from an existing class Y
  - *Derived* class (child class, subclass): the refined class X
  - *Base* class (parent class, superclass): the existing class Y
  - A derived class *inherits* pre-existing fields and methods
  - A derived class can add new fields and methods
  - A derived class can hide/redefine members of base class

- Class hierarchy
  - By deriving classes from existing ones, programmers can create arbitrarily deep class hierarchies

```
class queue : public list {
    // derived from class list
    public:
    int type;

    void enqueue(list_node*);
    list_node* dequeue();
};
```

# Overloaded Constructors

- Multiple constructors can be specified
  - Depending on parameters, appropriate constructors are used to initialize the class object
  - In C++, constructors of base classes are executed before constructors of derived classes

```
class list_node : public base_list {
public:
    int  val;
    list_node() { val = 0; }
    list_node(int v)  { val = v; }
    …
};

list_node element;                          // val = 0
list_node *e_ptr = new list_node(3);        // val = 3
```

# Modifying Base Class Methods

- To redefine, simply declare a new version of method in a derived class

- To access the base class method
  - list::remove() // C++
  - super.remove() // Java
  - base.remove() // C#
  - super remove // Smalltalk
  - [super remove] // Objective-C

```
class list {
public:
    void remove() { … }
    void add() { … }
    …
};

class queue : public list {
public:
    void remove() {
        …
    }
    void add() {
        …
        list::add();
        …
    }
};
```

# Encapsulation in C++

- C++ distinguishes among
  - Public class members
    - Accessible to anybody
  - Protected class members
    - Accessible to members of this or derived classes
  - Private class members
    - Accessible just to members of this class

- C++ structure (*struct*)
  - Simply a class whose members are public by default
    (vs. C++ class members are private by default)

# Encapsulation in C++ (cont'd)

- Derived classes can restrict the visibility of the members of base classes, but cannot relax the visibility

- Example:

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

  - Public members of shape act like protected members

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

  - Public/protected members of shape act like private members
  - Selectively make them visible with "using"

# Initialization and Finalization

- Most OOLs provide a special mechanism to *initialize* an object automatically at the beginning of its lifetime
  - *Constructor* – written in the form of a subroutine
    - Not allocate space, but initializes the allocated object
- A few languages provide a similar mechanism to *finalize* an object automatically at the end of its lifetime
  - *Destructor* – written in the form of a subroutine
    - Not deallocate object's space, but usually does deallocate the unnecessary space pointed by its members

# Issues in Initialization and Finalization

- Choosing a constructor among multiple ones
  - Differ in names (e.g., Eiffel, Smalltalk) *or*
  - Differ in number of arguments and/or types of arguments (e.g., C++, Java, C#)
- References and values
  - For reference variables, objects must be created explicitly
    - E.g., Java, Python, Ruby, Simula, Smalltalk,
  - For value variables, object creation happens implicitly as a result of elaboration
    - E.g., C++, Modula-3, Ada95, Oberon
- Execution order of constructors
  - Execute base class's constructor before derived class's constructor in C++
  - Destructors will be executed in a reverse order in C++
    - Garbage collection – automatic storage reclamation reduces the need of destructor

# Dynamic Method Binding

- Virtual functions are an example of *dynamic method binding*
  - You don't know at compile time what type the object will be referred to by a variable at run time
  - In C++, you can selectively specify member functions as virtual functions
  - In Java, Smalltalk, Eiffel, and Modula-3, all member functions are virtual
- Virtual function *vs.* subtype polymorphism
  - Virtual functions often require two different implementations for base class and derived class
  - A function in subtype polymorphism has one implementation, but acts differently due to the usage of virtual methods

# Virtual Functions

- Virtual functions in C++

```
class parent {
public:
      int foo() { … }
      virtual int voo() { … }
};

class child : public parent {
public:
      int foo() { … }
      virtual int voo() { … }
};
```

```
parent& p;
parent*  ptr;
child      c;

p = c;
p.foo();            // call parent::foo() or child::foo()

ptr = &c;
ptr->voo();         // call parent::voo() or child::voo()
```

# Abstract Classes and Abstract Methods

- Abstract method
  - If the body of virtual function is omitted, we call it abstract method. (in C++, pure virtual function)

```
public abstract int foo();    // Java and C#

public:
        virtual int foo() = 0;          // C++
```

- Abstract class
  - If at least one of virtual methods is abstract, we call the class abstract class
  - Abstract class cannot have an instance
  - Abstract class serves as a base class for other, *concrete classes*

# Member Lookup for Virtual Functions

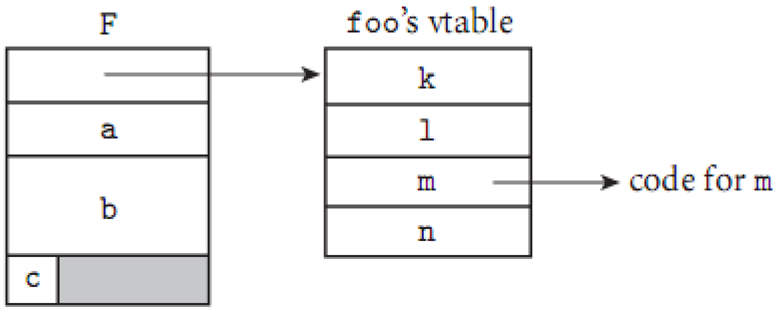
- Need a mechanism to call functions
  based on the object type, not the variable type
  - Creates a dispatch table (*vtable*) for the class
  - Puts a pointer to that table in the data of the object
  - Objects of a derived class have a different dispatch table

- Dispatch table
  - Virtual functions defined in the parent come first
  - Some of the pointers point to overridden versions of functions

  - You could put the whole dispatch table in the object itself to reduce the access time, but lots of space will be wasted for the same table

# Dispatch Table (vtable)

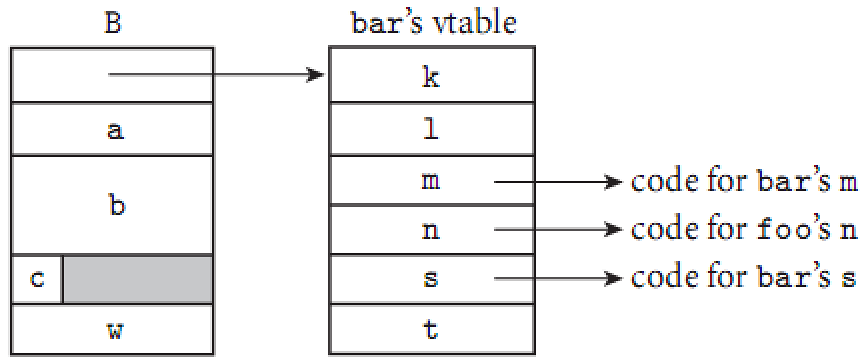
```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```




- The representation of object F begins with the address to the *virtual method table (vtable)*
- All objects of this class will point to the same *vtable*
    - Each content of *vtable* is the address of function code

# Inheriting Dispatch Table

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

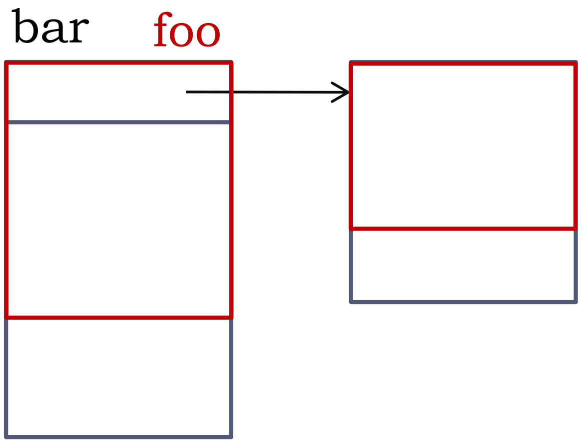



- First four entries in *vtable* represent the same method as the base class, except the method, m
  - The address of the overridden method, m, is replaced with the method of the derived class
- Entries for two methods, s and t, are added at the end

# Types of Objects in Virtual Functions

- You need to get the run-time type info of an object
  - The standard implementation technique is to put a pointer to the type info at the beginning of the *vtable*
  - In C++, you have a *vtable* only for the object that has virtual functions in its class

# Type Check




```
class foo { … }
class bar : public foo { … }

foo  F;
bar  B;
foo* fp;
bar *bp;

fp = &B;        // OK, fp will use prefixes of B's data space and vtable
bp = &F;        // Static type error, F lacks the additional data and vtable entries

bp = dynamic_cast<bar *>(fp);     // perform run-time type check

bp = (bar *) fp;          // permitted, but risky
```

# Multiple Inheritance

- Allow multiple parent classes (C++, Python)

  ```
  class student : public cs_student, public ee_student { ...
  ```

  - Get all the members of cs_student and ee_student
  - What if a member of the same name and arg types in both?
    - Ambiguous member – causes a compile-time error

- Single inheritance only
  - Smalltalk, Objective-C, Modula-3, Ada 95
- Limited *mix-in* form of multiple inheritance
  - *Interface* in Java, C#, Ruby
  - Inherit from one parent class and only methods from the others

# Object-Oriented Programming in Java

- 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 in C++

- C++
  - Multiple inheritance and generics (templates)
  - Allows creation of user-defined classes that look just like built-in ones
  - Has friends
  - Static type checking

- Is C++ object-oriented languages?
  - Uses all the right buzzwords
  - Has all the low-level C stuff to escape the paradigm
  - C++ can be used in an object-oriented style

# Summary

- Abstract data type (ADT)
  - Detailed internals are hidden, but interfaces are public

- Object-oriented programming
  - Encapsulation – data hiding (private attributes)
  - Inheritance – overriding
  - Dynamic method binding – virtual method invocation

- Implementation of OOP
  - Dispatch table – allows dynamic method binding
  - Data layout