Data Abstraction and Object Orientation

204315 OPL

Outline

- Object-Oriented (OO) Programming
- Key concepts in OO programming

Procedural versus Object-Oriented Programming

- Procedural programming
  - focuses on the process/actions that occur in a program
  - the program starts, does something, and ends.
- Object-Oriented programming
  - is based on the data and the functions that operate on it
  - objects are instances of abstract data types that represent the data and its functions.
- Key Point
  - An object or class contains the data and the functions that operate on that data. Objects are similar to structs but contain functions, as well.

Limitations of Procedural Programming

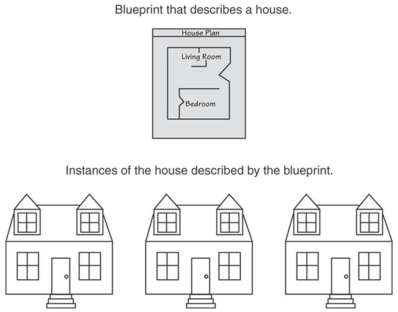
- If the data structures change, many functions must also be changed

- Programs that are based on complex function hierarchies are:
  - difficult to understand and maintain
  - difficult to modify and extend
  - easy to break

Object-Oriented Programming Terminology

- **class:** like a `struct` (allows bundling of related variables), but variables and functions in the class can have different properties than in a `struct`
- object: an **instance of a `class`**, in the same way that a variable can be an instance of a `struct`

A **Class is like a blueprint** and **objects are like houses built from the blueprint**



Object-Oriented Programming Terminology

- attributes: members of a class
- methods or behaviors: member functions of a class

- data hiding: restricting access to certain members of an object
- public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

Object-Oriented Concepts

- The 3 key concepts in OO programming
  - Encapsulation (information hiding)
  - Inheritance
  - Dynamic method binding

Key concepts: Encapsulation

- **Encapsulation**
  - **Related to abstract data types and information hiding** in programming languages
  - Divide structure of object into visible and hidden attributes
- **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.

Encapsulation

- **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
    - external users only aware of interface of the operations
    - division of labor in software projects

Encapsulation of Operations

- Object constructor
  - Used to create a new object
- Destructor operation
  - Used to destroy (delete) an object
- Modifier operations
  - Modify the states (values) of various attributes of an object

Initialization and Finalization

- **Life time of an object:** 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; it initializes space that has already been allocated.
  - 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

Key concepts: Inheritance

- Inheritance
  - All attributes inherited
  - Definition of new types based on other predefined types
  - Leads to type (or class) hierarchy
  - Instance of a subclass can be used in every context in which a superclass instance used
  - Subclass can redefine any function defined in superclass
  - **NOT FINAL:** subclasses are allowed to be defined
- Multiple inheritance
  - Subclass inherits attributes and methods of more than one superclass

Inheritance and Overloading

- Inheritance and Overloading
  - When a function is called, best match selected based on types of all arguments
  - For dynamic linking, runtime types of parameters is considered
- Operator overloading
  - Operation's ability to be applied to different types of objects
  - Operation name may refer to several distinct implementations

Encapsulation and Inheritance Classes

- 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

- 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*

Subclass

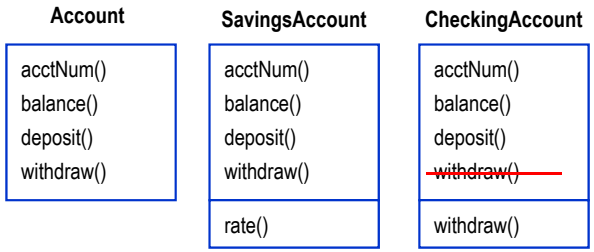
Suppose we define ***SavingsAccount*** and ***CheckingAccount*** as two new subclasses of the *Account* class.

```
class Account {
  method acctNum()
{…}
  method balance() {…}
  method deposit() {…}
  method withdraw()
{…}
}
```

```
class SavingsAccount
  extends Account {
    method rate() {…}
}
```

```
class CheckingAccount
  extends Account {
    method withdraw() {…}
}
```

Subclass



No new code has to be written for *deposit()* and other methods, they are inherited from the superclass.

Polymorphism

Polymorphism

- The same message sent to different types of objects results in:
  - execution of behavior that is specific to the object and,
  - possibly different behavior than that of other objects receiving the same message.
- **Example:** the message **draw()** sent to an object of type *Square* and an object of type *Circle* will result in different behaviors for each object.

- **Polymorphism of operations**
  - Also known as operator overloading
  - Allows same operator name or symbol to be bound to two or more different implementations
  - Depending on type of objects to which operator is applied

Polymorphism

- There are **many forms of Polymorphism in object-oriented languages:**
  - **True Polymorphism:** Same method signature defined for different classes with different behaviors (i.e. draw() for the Classes *Circle* and *Square*)
  - **Parametric Polymorphism:** This is the use of the same method name within a class, but with a different signature (different parameters).
  - **Overloading:** This usually refers to operators (such as +,-,/,*, etc) when they can be applied to several types such as int, floats, strings, etc.
  - **Overriding:** This refers to the feature of subclasses that replace the behavior of a parent class with new or modified behavior.

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
- 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 or Python)
- C++ philosophy is to avoid run-time overhead whenever possible – this is sort of the legacy from C
  - Languages like Smalltalk have (much) more run-time support

Dynamic Method Binding

- **Virtual functions**
  - 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

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

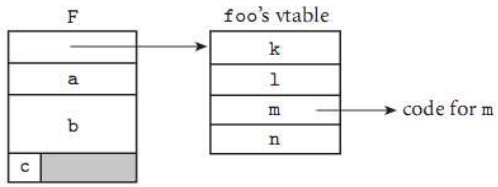


Figure 9.3 Implementation of virtual methods. The representation of object F begins with the address of the vtable for class foo. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder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

Dynamic Method Binding

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

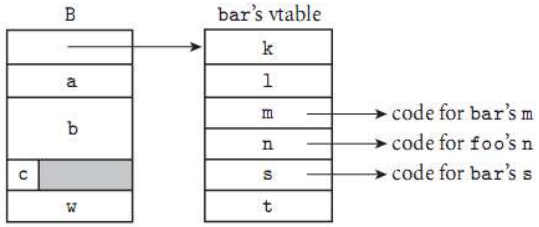


Figure 9.4 Implementation of single inheritance. As in Figure 9.3, the representation of object B begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for foo, except that one —m— has been overridden and now contains the address of the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

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

References

- Michael L. Scott: Programming Language Pragmatics (Chapter 3)
- Mitchel G. Fry: CSTA Spring Conference, Introduction to Object-Oriented Analysis and Object-Oriented Design
- CS1 Lesson 13 -- Introduction to Classes
- Ramez E. and Shamkant N., "Fundamentals of Database Systems -- 7th Edition", Addison-Wesley, 2016.