Programming Languages 2012: Object-Oriented Programming

(Based on [Sethi 1996])

Yih-Kuen Tsay

1 Introduction

Decomposition and Abstraction

- Decomposition

  Large programs are partitioned into smaller pieces that are implemented by one or more people.

- Forms of Abstraction
  - Procedures
  - Modules
  - Abstract data
  - Objects

    Object-oriented programming treats an overall system as a collection of interacting objects.

- The various forms of abstraction are supported by a technique called *information hiding*.

Modules

- The idea that data and operations go together is the basis for modules. With modules, the groupings of variables and procedures are explicit in the source text.

- A *module* is a collection of declarations, typically including both variables and procedures. *We cannot create new modules or copies of existing modules dynamically as a program runs.*

- The *interface* of a module is a subset of declarations in the module. An *implementation* of the module consists of everything else about the module.

- Programming with modules:
  1. Describe the roles of the modules (in general terms).
  2. Design the interfaces.
  3. Implement the interfaces, hiding design decisions in the private part.

Classes and Objects

- The term *class* is an abbreviation of "class of objects."

- A class corresponds (essentially) to a *type*.

- An *object* is a run-time entity with data on which operations can be performed.

- Objects can be *created* and *deleted* at run time.

Classes and Objects (cont.)

- Example (in pseudocode):

```
class Stack {
public:
        Stack();
    void push(int a);
    int  pop();
private:
    ...
};
```

  Procedures *push* and *pop* operate on private data.

- The procedure *Stack*, with the same name as the class, is a *constructor*. The constructor is called automatically when an object of the class is created, so initialization code can be put in the constructor.

- A class can also have a *destructor* procedure, which is called automatically just before the object disappears.

2 Information Hiding

Information Hiding

- An *abstract specification* tells us the behavior of an object independent of its implementation; that is, an abstract specification tells us *what* an object does independent of *how* it works.
- A *concrete representation* tells us how an object is implemented, how its data is laid out inside a machine, and how this data is manipulated by its operations.
- The *implementation hiding* principle: design a program so that the implementation of an object can be changed without affecting the rest of the program.
- Scope rules, which control the visibility of names, are the primary tool for achieving implementation hiding.

Data Invariants

- A grouping of data and operations has a local state, consisting of the values of its variables.
- A *data invariant* for an object is a property of its local state that holds whenever control is not in the object.
- Design an object around data invariants:
  - Initialization of Private Variables

    Since the private data of an object is inaccessible from outside, initialization of the data belongs with the code for the object. Initialization is needed to set up data invariants when the object is created.
  - Assignments to Public Variables

    Assignments to public variables can change the local state of an object. It is up to the user to ensure that such assignments do not disturb the desired data invariants.

3 Constructs in C++

Structures as Classes in C++

- Classes in C++ are a generalization of records, called *structures* in C and C++.
- A structure is traditionally a grouping of data; C++ allows both data and functions to be structure members. Example:

```
struct Stack {
  int top;
  char elements[101];
    char pop();
  void push(char);
  Stack() { top = 0; }
};

char Stack::pop() {
  top = top - 1;
  return elements[top+1];
}

void Stack::push(char c) {
  top = top + 1;
  elements[top] = c;
}

#include <stdio.h>
main() {
  Stack s;
  s.push('!'); s.push('@'); s.push('#');
  printf("%c %c %c\n", s.pop(), s.pop(), s.pop());
}
```

Overloaded Function Names

- The same name can be given to more than one function in a class, provided we can tell the overloaded functions apart by looking at the number and types of their parameters.
- Constructors are functions, so they too can be overloaded.
- Example:

```
struct Complex {
  float re;
  float im;
        Complex(float r)     { re = r; im = 0; }
        Complex(float r, i)  { re = r; im = i; }
};
```

Public, Private, and Protected Members

- Privacy and access control in C++ are class-based. That is, access to members is restricted through keywords in a class declaration.

- C++ has three keywords—`public`, `private`, and `protected`—for controlling the accessibility of member names in a class declaration:
  - *Public members* are accessible to outside code.
  - *Private members* are accessible to the member functions in this class declaration. They are accessible to all objects of this class.
  - *Protected members* behave like private members, except for derived classes. Protected members are visible through inheritance to derived classes but not to other code.

Dynamic Allocation in C++

- C++ objects can be created in three ways:
  1. through variable declarations,
  2. dynamically through `new`, and
  3. as static objects whose lifetime is the entire life of the program.

Dynamic Allocation in C++ (cont.)

```
class Cell {
  int info;
  Cell *next;

  Cell(int i) { info = i; next = this;}
  Cell(int i, Cell *n) { info = i; next = n;}
friend class List;
};

class List {
  Cell *rear;
public:
  void put(int);
  void push(int);
  int pop();
  int empty() { return rear==rear->next; }
  List() { rear = new Cell(0); }
  ~List() { while (!empty()) pop(); }
};
```

Dynamic Allocation in C++ (cont.)

```
void List::push(int x) {
  rear->next = new Cell(x, rear->next);
}

void List::put(int x) {
  rear->info = x;
  rear = rear->next = new Cell(0, rear->next);
}
```

```
int List::pop() {
  if (empty()) return 0;
  Cell *front = rear->next;
  rear->next = front->next;
  int x = front->info;
  delete front;
  return x;
}
```

Templates: Parameterized Types

```
template<class T> class Stack {
  int top;
  int size;
  T *elements;
public:
  Stack(int n) {
    size = n; elements = new T[size]; top = 0;
  }
  ~Stack() { delete elements;}
  void push(T a) { top++; elements[top] = a;}
  T pop() { top--; return elements[top+1]; }
};
```

Usage:

```
Stack<int> s(99);
Stack<char> t(80);
```

Implementation of a C++ Program in C

```
struct Stack {
  int top;
  char elements[101];

  char pop();
  void push(char);
  Stack() { top = 0; }
};

char Stack::pop() {
  top = top - 1;
  return elements[top+1];
}
```

```
struct Stacklay {
  int top;
  char elements[101];
};

void StackStack(struct Stacklay *p) {
  p->top = 0;
}

char Stackpop(struct Stacklay *p) {
  char c;
  c = p->elements[p->top];
  p->top = p->top - 1;
  return c;
}
```

Implementation of a C++ Program in C (cont.)

```
void Stack::push(char c) {
  top = top + 1;
  elements[top] = c;
}

#include <stdio.h>
main() {
  Stack s;
  s.push('!');
  s.push('@');
  s.push('#');
  printf("%c %c %c\n",
    s.pop(),
    s.pop(),
    s.pop());
}
```

```
void Stackpush(struct Stacklay *p, char c) {
  p->top = p->top + 1;
  p->elements[p->top] = c;
}

#include <stdio.h>
main() {
  struct Stacklay s;
  StackStack(&s);
  Stackpush(&s, '!');
  Stackpush(&s, '@');
  Stackpush(&s, '#');
  printf("%c %c %c\n",
    Stackpop(&s),
    Stackpop(&s),
    Stackpop(&s));
}
```

In-Line Expansion

- Implementation hiding can result in lots of little functions that manipulate the data in an object.
- C++ implements such functions efficiently by using *in-line* expansion, which replaces a call by the function body. *In-line expansion in C++ preserves the semantics of call-by-value parameter passing.*
- Suppose a public function `isempty` is added to class `stack`:

  ```
  int isempty() { return top == 0;}
  ```

  With in-line expansion, the following conditional statement

  ```
  if ( s.isempty() )
  ```

  expands to

  ```
  if ( (s.top == 0) )
  ```

- In-line expansion eliminates the overhead of function calls at run time, so it encourages free use of functions. It also encourages data hiding.

4 Derived Classes

Base and Derived Classes

- The extension of a base class is called a *derived class*. Example:

  ```
  class B {            // declaration of class B
  public:
    int x;             // the full name is B::x
    char f();          // public member function
    B();
  };

  class D : public B { // D derived from B
    int x;             // D::x is added,
                       // B::x is inherited
    int g();           // added member function
  };
  ```

- A member added by a derived class `D` can have the same name as a member of its base class `B`.

  `B::m` and `D::m` refer to the member `m` in `B` and `D`, respectively.

Public Base Classes

- A distinguishing feature of object-oriented programming in any language is that an object of a derived class can appear where an object of a base class is expected. That is, a derived object can behave like a base object.
- In C++, members of a public base class retain their visibility in the derived class. That is, a public member of the base class is a public member of the derived class, and similarly for protected and private members. Therefore,

  > an object of a derived class can appear wherever an object of a *public base class* is expected.

Virtual Functions

- Virtual functions in C++ allow a derived class to supply the function body.

  ```
  class B {
  public:
  virtual char f() { return 'B';}
    char g() { return 'B';}
    char testF() { return f(); }
    char textG() { return g(); }
  };

  class D : public B {
  public:
    char f() { return 'D'; }
    char g() { return 'D'; }
  };

  main() {
    D d;
    print d.textF(), d.testG();
  }
  ```

Private Base Classes

- C++ also supports private base classes. The purpose of a private base class is quite different from that of a public base class.
- A derived class simply shares the code of the private base class. Such code sharing is sometimes called *implementation inheritance.*
- All members of a private base class become private in the derived class. Nonprivate inherited members can be made visible by writing their full names in the derived class.

The Privacy Principle

- Functions in a derived class cannot access the private members of its base class.
- Otherwise, the following principle would be violated:

> *Privacy principle*: The private members of a class are accessible only to member functions of the class.