CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Data Abstraction, Object Orientation

Language Specification

- General issues in the design and implementation of a language:
  - Syntax and semantics
  - Naming, scopes and bindings
  - Control flow
  - Data types
  - Subroutines

- Specific issues
  - Non-imperative models: functional and logic languages
  - Data abstraction and object orientation

Data Abstraction

- Abstraction – associate a name with a potentially complex program fragment
  - consider the fragment in terms of its purpose, not implementation
- Control abstraction – purpose corresponds to an operation
  - very old idea (subroutines)
- Data abstraction – purpose is to represent information
  - data structures (another old idea)
  - modules, classes (also include operations)
- Abstract data type
  - combine both information and operations
  - define data in terms of operations that it supports, rather than of its structure or implementation

Why abstractions?

- Reduced conceptual load
  - minimize the amount of detail the programmer needs to think about
  - hide what doesn't matter
- Fault containment
  - prevent using a program component in inappropriate ways
  - restrict the use of a component to a limited part of program
  - easier to find and fix bugs
- Independence among program components
  - division of labor in software projects – assign separate components to different programmers
  - modification of internal implementation of components
    - without changing (recompilation, rewriting) external code that uses them
    - libraries - code reuse

Modules

Module-as-manager for a stack (Modula-2):

```
CONST stack_size = ...
TYPE element = ...
...
MODULE stack_manager;
IMPORT element, stack_size;
EXPORT stack, init_stack, push, pop;
TYPE
    stack_index = [1..stack_size];
    STACK = RECORD
        s : ARRAY stack_index OF element;
        top : stack_index;          (* first unused slot *)
    END;

PROCEDURE init_stack (VAR stk : stack);
BEGIN
    stk.top := 1;
END init_stack;

PROCEDURE push (VAR stk : stack; elem : element);
BEGIN
    IF stk.top = stack_size THEN
        error;
    ELSE
        stk.s[stk.top] := elem;
        stk.top := stk.top + 1;
    END;
END push;
```

```
PROCEDURE pop (VAR stk : stack) : element;
BEGIN
    IF stk.top = 1 THEN
        error;
    ELSE
        stk.top := stk.top - 1;
        return stk.s[stk.top];
    END;
END pop;

END stack;
```

```
var A, B : stack;
var x, y : element;
...
init_stack (A);
init_stack (B);
...
push (A, x);
...
y := pop (B);
```

- Export the type stack
- Allow for declaring several stacks
- Must pass the stack as argument to each subroutine

Modules

Module-as-type for a stack (Euclid):

```
const stack_size := ...
type element : ...
...
type stack = module
    imports (element, stack_size)
    exports (push, pop)
type
    stack_index = 1..stack_size
var
    s   : array stack_index of element
    top : stack_index

procedure push (elem : element) = ...
function pop returns element = ...
...
initially
    top := 1
end stack
```

- Subroutines "belong" to the stack
- Do not need to pass the stack as argument explicitly
  - however, the implementation is similar (pass a hidden argument)
  - more intuitive for programmer

```
var A, B : stack
var x, y : element
...
A.push (x)
...
y := B.pop
```

Object-Oriented Programming

- Which property (reduced conceptual load, fault containment, independence) is difficult to achieve with modules?
  - independence (in the context of code reuse)
  - if additional (or different) features are needed – copy entire module code, then change it
  - has been the motivation for introducing classes (with inheritance)

Object-Oriented Programming

- Key factors in object-oriented programming:
- Encapsulation
  - data hiding
  - was also provided by modules
- Inheritance
  - enable a new abstraction (a derived class) to be defined as an extension of an existing one
  - retain key characteristics from base class
- Dynamic method binding
  - enable use of new abstraction (derived class) to exhibit new behavior
  - important when used in a context where old abstraction is expected

Visibility

- Classes - same visibility rules as modules
  - additional issue raised by inheritance:
  - How much control should the base class exercise over visibility in derived classes?
- C++:
  - Visibility controlled by labels (public, private, protected) applied to:
    - members
    - inheritance process

Visibility

- C++:
- Members can be:
  - public – accessible to anybody
  - private – accessible only to methods of this class
  - protected – accessible to methods of this class and derived classes
- How can the derived class modify the visibility of the members of base class?
  - can restrict visibility, but never increase it
- The inheritance from a base class can be:
  - public – preserves the visibility of members from base class
  - private – all members from base class become private in derived class
  - protected – private remains private, protected remains protected, public becomes protected

Visibility

- C++:
- How can we allow access to private members for some select group of classes (not derived) or subroutines?
  - declare them (in this class) as friend
  - friendship is not mutual

```
class A
{
    friend class B;     // Declare a friend class
  private:
    int topSecret;
};
```

```
class B
{
  public:
    void change (A x);
};

void B::change (A x)
{
    x.topSecret++ ;    // Can access private data
}
```

Visibility

- Eiffel:
  - more flexible than C++
  - a derived class can either restrict or increase visibility of the members of base class
  - for each member - can specify its export status:
    - NONE - private
    - ANY - generally available
    - specify a list of classes - selectively available to those
- Java:
  - same labels (public, private, protected) for members
  - no labels for inheritance
  - a derived class can neither restrict nor increase visibility of the members of base class
  - still, how can we restrict access?
    - redefine a method to do nothing (or to produce a run-time error if used)

Visibility

- Smalltalk:
  - no issue of visibility
  - method invocation ⇔ send a message to an object
    - if the object has that method – invoke it
    - otherwise – run-time error
  - no way to make a method available only to some parts of a program, but not to others

Initialization and Finalization

- Lifetime of an object – interval during which it occupies space and can hold data

- Special methods
  - constructor
  - destructor

- What exactly does a constructor do?
  - it does <u>not</u> allocate space for the object
  - it gives the programmer a chance to initialize space that has been already allocated

Constructors

- C++
  - requires an appropriate constructor to be called for every elaborated object

```
foo b;              // calls foo::foo ()
foo b (10, 'x');    // calls foo::foo (int, char)
```

  - another example:

```
foo a;
bar b;
...
foo c (a);          // calls foo::foo (foo&)
foo d (b);          // calls foo::foo (bar&)

         // foo::foo (foo&) is called a copy constructor
```

  - when is a copy constructor also called?
    - pass an object by value: `my_func (c);`
    - return an object by value: `return c;`

Constructors

- C++
  - Same example:

```
foo a;
bar b;
...
foo c (a);        // calls foo::foo (foo&)
foo d (b);        // calls foo::foo (bar&)
```

  - Is the following code equivalent to the previous one?

```
foo a;
bar b;
...
foo c = a;        // calls foo::foo (foo&)
foo d = b;        // calls foo::foo (bar&)
```

  - Yes, the copy constructor is called at initialization

Constructors

- C++
  - Same example:

```
foo a;
bar b;
...
foo c (a);        // calls foo::foo (foo&)
foo d (b);        // calls foo::foo (bar&)
```

  - Is the following code equivalent to the previous one?

```
foo a, c, d;
bar b;
...
c = a;            // calls foo::operator= (foo&)
d = b;            // calls foo::operator= (bar&)
```

  - No, the assignment operator is called

Execution Order

- C++
  - order of calling constructors:
    - constructor of base class
    - constructors of member objects
    - constructor of derived (this) class
  - at declaration, specify arguments only for the constructor of derived class
  - how are arguments specified for the constructor of base class?

```
class bar { ... };
class foo : public bar { ... };

foo::foo (<foo_params>) : bar (<bar_args>)
{
                    ...
}
// <bar_args> can be arbitrary expressions involving <foo_params>
```

Execution Order

- C++
  - how are arguments specified for constructors of member objects?

```
class bar { ... };
class foo : public bar
{
M1 member1;          // M1 and
M2 member2;          // M2 are classes
...
};

foo::foo (foo_params) : bar (bar_args), member1 (mem1_args),
        member2 (mem2_args)
{
...
}
```

  - All "constructor calls" specified in header are executed before the constructor of foo

Execution Order

- C++

```
foo::foo (a, b, c, d) : bar (a), member1 (b),
    member2 (c)
{
    ...
}
```

– Can also initialize them in the constructor of foo:

```
foo::foo (a, b, c, d)
{
bar::x = a;
member1 = b;
member2 = c;
...
}
```

- What is the difference?
  - in the second approach, the order is:
    - call to default (no-argument) constructor for bar
    - call to default (no-argument) constructors for M1 and M2
    - call to foo constructor
    - assignments

Dynamic Method Binding

- Consequence of inheritance
  - derived class D has all members of its base class B
  - can use an object of class D everywhere an object of class B is expected
    - a form of polymorphism
- Example (C++):

```
class person { ... };
class student : public person { ... };
class professor : public person { ... };

student s;
professor p;
...
person * x = &s;          // Both student and professor objects have all properties of
person * y = &p;          // a person object
                          // Both can be used in a person context
```

Dynamic Method Binding

- Example (C++):

```
class person { ... };
class student : public person { ... };
class professor : public person { ... };

void person::print_mailing_label () { ... }

student s;
professor p;
...
person * x = &s;
person * y = &p;

s.print_mailing_label ();        // person::print_mailing_label ()
p.print_mailing_label ();        // person::print_mailing_label ()
```

Dynamic Method Binding

- Example (C++):
  - Suppose that we redefine print_mailing_label in both derived classes

```
void student::print_mailing_label () { ... }
void professor::print_mailing_label () { ... }

student s;
professor p;
...
person * x = &s;
person * y = &p;

s.print_mailing_label ();        // student ::print_mailing_label ()
p.print_mailing_label ();        // professor ::print_mailing_label ()
```

  - But what about:

```
x->print_mailing_label ();    // ??
y->print_mailing_label ();    // ??
```

Dynamic Method Binding

- Example (C++):

```
student s;
professor p;
...
person * x = &s;
person * y = &p;

x->print_mailing_label ();      // ??
y->print_mailing_label ();      // ??
```

  - Two alternatives for choosing the method to call:
    - according to the types of variables (references) x and y – static method binding (will call the method of person in both cases)
    - according to the types of objects s and p to which x and y refer – dynamic method binding (will call the methods of student / professor)
  - Example – list of persons that have overdue library books
    - list may contain both students and professors
    - traverse the list and print a mailing label - call the appropriate subroutine

Dynamic Method Binding

- Disadvantage of dynamic method binding
  - run-time overhead
- Smalltalk, Modula-3
  - dynamic method binding
- Java, Eiffel
  - dynamic method binding by default
  - individual methods can be labeled final (Java) or frozen (Eiffel)
    - cannot be overriden by derived classes
    - use static method binding
- Simula, C++, Ada 95
  - static method binding by default
  - how do we specify dynamic binding in C++?
    - label individual methods as virtual

Virtual and Non-Virtual Methods

- Terminology in C++:
  - redefine a method that uses static binding
  - override a method that uses dynamic binding

- C++

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

  - if print_mailing_label is overridden in classes student and professor
    - at run time, the appropriate one is chosen – dynamic binding

Abstract Classes

- C++

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

  - Abstract method – virtual method with no body
    - also called pure virtual method in C++
  - Abstract class – it has at least one abstract method
    - cannot declare objects of an abstract class, just pointers
  - Purpose of an abstract class:
    - serve as base to derive concrete classes
    - a concrete class must provide a definition for every abstract method it inherits
  - Interface (Java) – class with no other members than abstract methods

Member Lookup

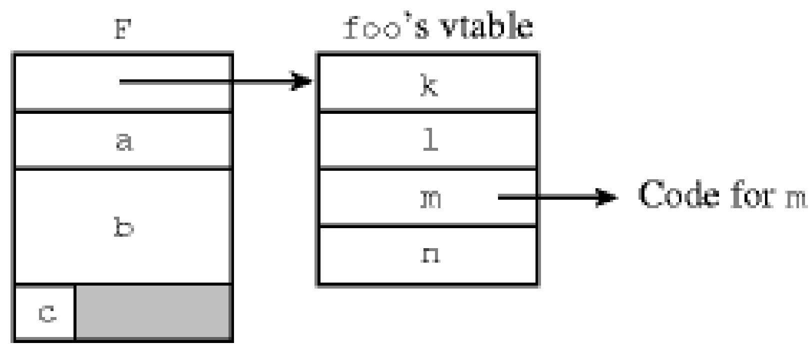
- Static method binding:
  - easy to find the method to call, based on the type of the variable
  - performed at compile time

- Dynamic method binding
  - appropriate method is identified at run-time
  - objects must contain information to allow for finding the appropriate method
  - each object contains a pointer to a virtual method table (vtable)
  - all objects of a given class have the same vtable

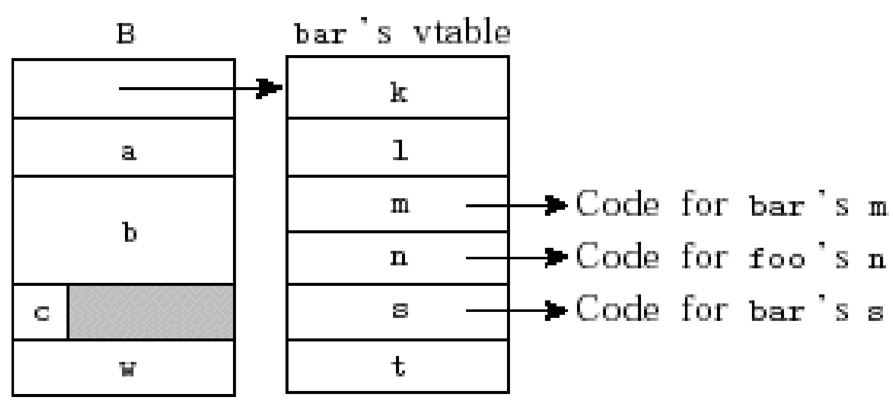
Member Lookup

- Implementation of a vtable:

```
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;
```



Multiple Inheritance

- Useful for a derived class to inherit features for more than one base class
- Multiple inheritance
  - allowed in C++, Eiffel, CLOS
  - not in Simula, Smalltalk, Modula-3, Ada 95, Oberon
  - Java – only a limited ("mix-in") form of multiple inheritance
- Example – keep all students in a list:

```
class student : public person, public gp_list_node
{
    ...
};
```

Multiple Inheritance

- Semantic ambiguities – suppose that:
  - professors also take courses

```
class professor : public person, public gp_list_node { ... };
class student : public person, public gp_list_node { ... };
class student_prof : public student, public professor { ... };
```

  - student_prof inherits twice from person and gp_list_node
  - Do we want to have two instances of their members in student_prof?
    - one instance of person (address, etc)
    - two instances of gp_list_node (appear in the list of students, and in the list of professors)

```
gp_list_node    person    gp-list_node
          \       /   \         /
          student   professor
                \       /
              student_prof
```

Multiple Inheritance

```
gp_list_node    person    gp-list_node
           \        /   \         /
           student   professor
                 \       /
              student_prof
```

- Repeated inheritance – multiple inheritance where there are multiple paths to an ancestor
- Types of repeated inheritance
  - replicated inheritance – separate copies (gp_list_node)
  - shared inheritance – single copy (person)
- Eiffel – by default shared inheritance
  - can get replicated inheritance of individual members by renaming them
- C++ – by default replicated inheritance
  - can get shared inheritance by labeling the inheritance as virtual:

```
class professor : public virtual person, public gp_list_node { ... };
class student : public virtual person, public gp_list_node { ... };
```

Object-Oriented Languages

- Are all the languages mentioned truly object-oriented?
  - they differ in the extent to which they require an object-oriented style of programming

- Ideally
  - the language should make it impossible to write non-OO programs
  - uniform object model of computing
    - every data type is a class
    - every variable is a reference to an object
    - every subroutine is an object method

Object-Oriented Languages

- How close (or far) is C++?
  - simple types (int, char) are not classes
  - subroutines outside of classes
  - static method binding by default
  - in general – retains all low-level mechanisms of C

- Closest to the object-oriented ideal – Smalltalk

Smalltalk

- Designed by Alan Kay at the University of Utah (1960s)
- Adopted and revised by Xerox Palo Alto Research Center (PARC)
- Considered the canonical object-oriented language
- Is integrated into its programming environment
  - programs are meant to be viewed within the browser of Smalltalk implementation
- Untyped reference model for variables
  - every variable refers to an object
  - the class of the object need not be statically known
- Common ancestor for all classes - the standard class Object
- All data is contained in objects - example:
  - true (of class Boolean)
  - 3 (of class Integer)

Smalltalk

- Consistently follows a message-based model
- All operations are considered as messages sent to objects:

3 + 4 "send a + message to the object 3,
"with a reference to object 4 as argument"
"in response, object 3 creates and returns a reference to object 7"

- Multi-argument messages have multi-word names:

myBox displayOn: myScreen at: location

"send a displayOn: at: message to object myBox, "
"with objects myScreen and location as arguments"

- Even control flow is represented with messages

Smalltalk

- Selection – the ifTrue: ifFalse: message

```
n < 0
    ifTrue: [abs <- n negated]
    ifFalse: [abs <- n]
```

- expression evaluation – from left to right
- send a < message (with 0 as argument) to n
- in response, n returns either the object true or the object false
  - now n < 0 has been evaluated to true or false
- send an ifTrue: ifFalse: message to this true or false object
- the arguments of ifTrue: ifFalse: message are two blocks [...]
  - a block is similar to a lambda expression in Scheme
  - to execute a block - need to send it a value message
- when receiving the ifTrue: ifFalse: message, object true sends a value message to the 1st argument (block) of the message
- when receiving the ifTrue: ifFalse: message, object false sends a value message to the 2nd argument (block) of the message

Smalltalk

- Iteration (enumeration-controlled loops) – the timesRepeat: message
  - Compute n^{10}

```
pow <- 1.
10 timesRepeat:
    [pow <- pow * n]
```

  - send message timesRepeat: to object 10 with a block as argument
  - in response, object 10 sends a value message to the block object 10 times

Smalltalk

- Iteration (enumeration-controlled loops) - the to: by: do: message
  - Compute the sum of odd-indexed elements of array a

```
sum <- 0.
1 to: 100: by: 2 do:
    [:i | sum <- sum + (a at: i)]
```

  - send message to: by: do: to object 1 with 3 arguments
  - in response, object 1 sends (50 times) a value message to the block, with its own value as a parameter, and increments itself with 2 after each iteration
  - the block has a formal parameter :i to get the value
  - message at: i sent to array a returns the i^{th} element from array a

Announcements

- Readings
  - Chapter 10