CS 242

Modularity and Object-Oriented Programming

John Mitchell

Reading: Chapter 10 and parts of Chapter 9

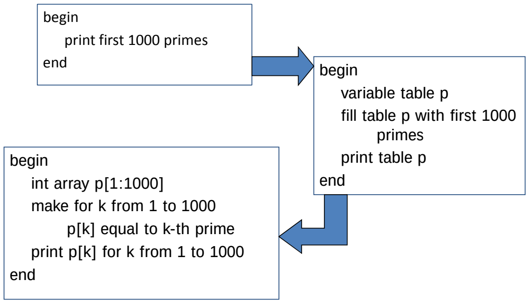
Topics

- Modular program development
  - Step-wise refinement
  - Interface, specification, and implementation
- Language support for modularity
  - Procedural abstraction
  - Abstract data types
  - Packages and modules
  - Generic abstractions
    - Functions and modules with type parameters

Stepwise Refinement

- Wirth, 1971
  - "… program … gradually developed in a sequence of refinement steps"
  - In each step, instructions … are decomposed into more detailed instructions.
- Historical reading on web (CS242 Reading page)
  - N. Wirth, Program development by stepwise refinement, *Communications of the ACM,* 1971
  - D. Parnas, On the criteria to be used in decomposing systems into modules, *Comm ACM,* 1972
  - Both *ACM Classics of the Month*

Dijkstra's Example (1969)

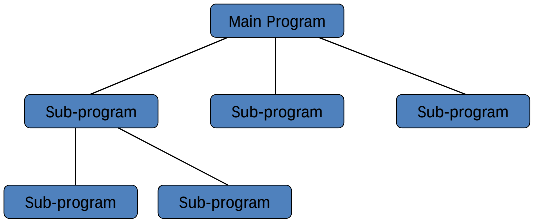


```
begin
    print first 1000 primes
end
```

```
begin
    variable table p
    fill table p with first 1000
          primes
    print table p
end
```

```
begin
    int array p[1:1000]
    make for k from 1 to 1000
          p[k] equal to k-th prime
    print p[k] for k from 1 to 1000
end
```

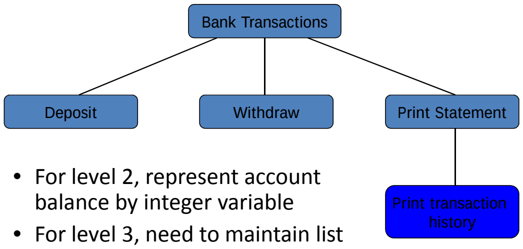
Program Structure



Data Refinement

- Wirth, 1971 again:
  - As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel

Example


Bank Transactions
Deposit
Withdraw
Print Statement
Print transaction history

- For level 2, represent account balance by integer variable
- For level 3, need to maintain list of past transactions

Modularity: Basic Concepts

- Component
  - Meaningful program unit
    - Function, data structure, module, …
- Interface
  - Types and operations defined within a component that are visible outside the component
- Specification
  - Intended behavior of component, expressed as property observable through interface
- Implementation
  - Data structures and functions inside component

Example: Function Component

- Component
  - Function to compute square root
- Interface
  - float sqroot (float x)
- Specification
  - If x>1, then sqrt(x)*sqrt(x) ≈ x.
- Implementation

```
float sqroot (float x){
  float y = x/2; float step=x/4; int i;
  for (i=0; i<20; i++){if ((y*y)<x) y=y+step; else y=y-step; step = step/2;}
  return y;
}
```

Example: Data Type

- Component
  - Priority queue: data structure that returns elements in order of decreasing priority
- Interface
  - Type pq
  - Operations empty : pq
    insert : elt * pq → pq
    deletemax : pq → elt * pq
- Specification
  - Insert add to set of stored elements
  - Deletemax returns max elt and pq of remaining elts

Heap sort using library data structure

- Priority queue: structure with three operations
  empty : pq
  insert : elt * pq → pq
  deletemax : pq → elt * pq
- Algorithm using priority queue (heap sort)

```
begin
  create empty pq s
  insert each element from array into s
  remove elements in decreasing order and place in array
end
```

This gives us an O(n log n) sorting algorithm (see HW)

Abstract Data Types

- Prominent language development of 1970's
- Main ideas:
  - Separate interface from implementation
    - Example:
      - Sets have empty, insert, union, is_member?, …
      - Sets implemented as … linked list …
  - Use type checking to enforce separation
    - Client program only has access to operations in interface
    - Implementation encapsulated inside ADT construct

Modules

- General construct for information hiding
- Two parts
  - Interface:
    - A set of names and their types
  - Implementation:
    - Declaration for every entry in the interface
    - Additional declarations that are hidden
- Examples:
  - Modula modules, Ada packages, ML structures, ...

Modules and Data Abstraction

```
module Set
   interface
      type set
      val empty : set
      fun insert : elt * set -> set
      fun union : set * set -> set
      fun isMember : elt * set -> bool
   implementation
      type set = elt list
      val empty = nil
      fun insert(x, elts) = ...
      fun union(…) = ...
      ...
end Set
```

◆ Can define ADT
- Private type
- Public operations

◆ More general
- Several related types and operations

◆ Some languages
- Separate interface and implementation
- One interface can have multiple implementations

Generic Abstractions

- Parameterize modules by types, other modules
- Create general implementations
  - Can be instantiated in many ways
- Language examples:
  - Ada generic packages, C++ templates, ML functors, …
  - ML geometry modules in book
  - C++ Standard Template Library (STL) provides extensive examples

C++ Templates

- Type parameterization mechanism
  - template<class T> … indicates type parameter T
  - C++ has class templates and function templates
- Instantiation at link time
  - Separate copy of template generated for each type
  - Why code duplication?
    - Size of local variables in activation record
    - Link to operations on parameter type

Example (discussed in earlier lecture)

- Monomorphic swap function

```
void swap(int& x, int& y){
     int tmp = x;  x = y;  y = tmp;
}
```

- Polymorphic function template

```
template<class T>
void swap(T& x, T& y){
    T tmp = x;  x = y;  y = tmp;
}
```

- Call like ordinary function

```
float a, b;  …  ; swap(a,b); …
```

Standard Template Library for C++

- Many generic abstractions
  - Polymorphic abstract types and operations
- Useful for many purposes
  - Excellent example of *generic programming*
- Efficient running time (but not always space)
- Written in C++
  - Uses template mechanism and overloading
  - Does *not* rely on objects – No virtual functions

Architect: Alex Stepanov

Main entities in STL

- Container: Collection of typed objects
  - Examples: array, list, associative dictionary, ...
- Iterator: Generalization of pointer or address
- Algorithm
- Adapter: Convert from one form to another
  - Example: produce iterator from updatable container
- Function object: Form of closure ("by hand")
- Allocator: encapsulation of a memory pool
  - Example: GC memory, ref count memory, ...

Example of STL approach

- Function to merge two sorted lists
  - merge : range(s) × range(t) × comparison(u) → range(u)

  This is conceptually right, but not STL syntax.
- Basic concepts used
  - range(s) - ordered "list" of elements of type s, given by pointers to first and last elements
  - comparison(u) - boolean-valued function on type u
  - subtyping - s and t must be subtypes of u

How merge appears in STL

- Ranges represented by iterators
  - iterator is generalization of pointer
  - supports ++ (move to next element)
- Comparison operator is object of class Compare
- Polymorphism expressed using template

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator1 last2,
                       OutputIterator result, Compare comp)
```

Comparing STL with other libraries

- C:

  qsort((void*)v, N, sizeof(v[0]), compare_int);
- C++, using raw C arrays:

  int v[N];

  sort(v, v+N);
- C++, using a vector class:

  vector v(N);

  sort(v.begin(), v.end());

Efficiency of STL

- Running time for sort

	N = 50000	N = 500000
C	1.4215	18.166
C++ (raw arrays)	0.2895	3.844
C++ (vector class)	0.2735	3.802

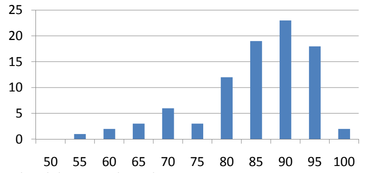
- Main point
  - Generic abstractions can be convenient and efficient !
  - But watch out for code size if using C++ templates…

Announcements (10/28/07)

- Midterm exam

  Mean: 83.2

  Std.Dev: 9.69



- Overall scores
  - We will send email with homework totals
  - We'll let you know if we are missing homework scores

Object-oriented programming

- Primary object-oriented language concepts
  - dynamic lookup
  - encapsulation
  - inheritance
  - subtyping
- Program organization
  - Work queue, geometry program, design patterns
- Comparison
  - Objects as closures?

Objects

- An object consists of
  - hidden data
    - instance variables, also called member data
    - hidden functions also possible
  - public operations
    - methods or member functions
    - can also have public variables in some languages
- Object-oriented program:
  - Send messages to objects

hidden data	
msg$_1$	method$_1$
. . .	. . .
msg$_n$	method$_n$

What's interesting about this?

- Universal encapsulation construct
  - Data structure
  - File system
  - Database
  - Window
  - Integer
- Metaphor usefully ambiguous
  - sequential or concurrent computation
  - distributed, sync. or async. communication

Object-Orientation

- Programming methodology
  - organize concepts into objects and classes
  - build extensible systems
- Language concepts
  - dynamic lookup
  - encapsulation
  - subtyping allows extensions of concepts
  - inheritance allows reuse of implementation

Dynamic Lookup

- In object-oriented programming,
  object → message (arguments)
  code depends on object and message

- In conventional programming,
  operation (operands)
  meaning of operation is always the same

Fundamental difference between abstract data types and objects

Example

- Add two numbers x → add (y)
  different add if x is integer, complex

- Conventional programming add (x, y)
  function add has fixed meaning

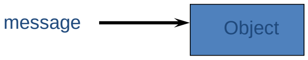
Very important distinction:
Overloading is resolved at compile time,
Dynamic lookup at run time

Language concepts

- “dynamic lookup”
  - different code for different object
  - integer “+” different from real “+”
- encapsulation
- subtyping
- inheritance

Encapsulation

- Builder of a concept has detailed view
- User of a concept has “abstract” view
- Encapsulation separates these two views
  - Implementation code: operate on representation
  - Client code: operate by applying fixed set of operations provided by implementer of abstraction



Language concepts

- “Dynamic lookup”
  - different code for different object
  - integer “+” different from real “+”
- Encapsulation
  - Implementer of a concept has detailed view
  - User has “abstract” view
  - Encapsulation separates these two views
- Subtyping
- Inheritance

Subtyping and Inheritance

- Interface
  - The external view of an object
- Subtyping
  - Relation between interfaces
- Implementation
  - The internal representation of an object
- Inheritance
  - Relation between implementations

Object Interfaces

- Interface
  - The messages understood by an object
- Example: point
  - x-coord : returns x-coordinate of a point
  - y-coord : returns y-coordinate of a point
  - move : method for changing location
- The interface of an object is its *type*.

Subtyping

- If interface A contains all of interface B, then A objects can also be used B objects.

Point
x-coord
y-coord
move

Colored_point
x-coord
y-coord
color
move
change_color

◆ Colored_point interface contains Point
  - Colored_point is a *subtype* of Point

Inheritance

- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

Example

```
class Point
      private
          float x, y
      public
           point move (float dx, float dy);
class Colored_point
      private
          float x, y; color c
      public
           point move(float dx, float dy);
         point change_color(color newc);
```

◆ Subtyping
  - Colored points can be used in place of points
  - Property used by client program

◆ Inheritance
  - Colored points can be implemented by reusing point implementation
  - Technique used by implementer of classes

OO Program Structure

- Group data and functions
- Class
  - Defines behavior of all objects that are instances of the class
- Subtyping
  - Place similar data in related classes
- Inheritance
  - Avoid reimplementing functions that are already defined

Example: Geometry Library

- Define general concept shape
- Implement two shapes: circle, rectangle
- Functions on implemented shapes

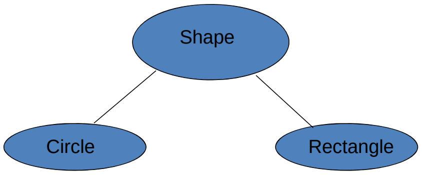
  center, move, rotate, print
- Anticipate additions to library

Shapes

- Interface of every shape must include

  center, move, rotate, print
- Different kinds of shapes are implemented differently
  - Square: four points, representing corners
  - Circle: center point and radius

Subtype hierarchy



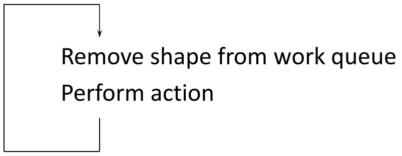
- General interface defined in the shape class
- Implementations defined in circle, rectangle
- Extend hierarchy with additional shapes

Code placed in classes

	center	move	rotate	print
Circle	c_center	c_move	c_rotate	c_print
Rectangle	r_center	r_move	r_rotate	r_print

- Dynamic lookup
  - circle → move(x,y) calls function c_move
- Conventional organization
  - Place c_move, r_move in move function

Example use: Processing Loop

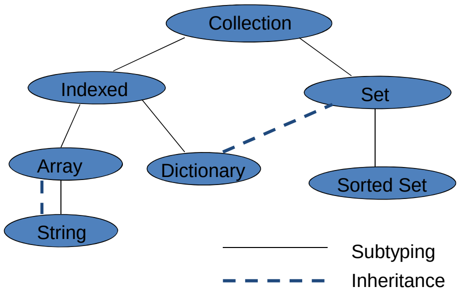


Remove shape from work queue

Perform action

Control loop does not know the type of each shape

Subtyping differs from inheritance



Design Patterns

- Classes and objects are useful organizing concepts
- Culture of *design patterns* has developed around object-oriented programming
  - Shows value of OOP for program organization and problem solving

What is a design pattern?

- General solution that has developed from repeatedly addressing similar problems.
- Example: singleton
  - Restrict programs so that only one instance of a class can be created
  - Singleton design pattern provides standard solution
- Not a class template
  - Using most patterns will require some thought
  - Pattern is meant to capture experience in useful form

Standard reference: Gamma, Helm, Johnson, Vlissides

Example Design Patterns

- Singleton pattern
  - There should only be one object of the given class
- Visitor design pattern
  - Apply an operation to all parts of structure
  - Generalization of maplist, related functions
  - Standard programming solution:
    - Each element classes has *accept* method that takes a visitor object as an argument
    - *Visitor* is interface with *visit()* method for each element class
    - The *accept()* method of an element class calls the *visit()* method for its class

Sample code

```
class Visitor {
public:  virtual void VisitElementA(ElementA*);
         virtual void VisitElementB(ElementB*);
        // and so on for other concrete elements
protected:  Visitor();
};
class Element {
public: virtual ~Element();
         virtual void Accept(Visitor&) = 0;
protected: Element();
};
class ElementA : public Element {
public: ElementA();
        virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};
```

http://www.swe.uni-linz.ac.at/research/deco/designPatterns/Visitor/visitor.defaultroles.c++.html

Sample code (cont'd)

```
class CompositeElement : public Element {
public:
     virtual void Accept(Visitor&);
private:
     List<Element*>* _children;
};

 void CompositeElement::Accept (Visitor& v) {
     ListIterator<Element*> i(_children);
     for (i.First(); !i.IsDone(); i.Next()) {
       i.CurrentItem()->Accept(v);
     }
     v.VisitCompositeElement(this);
}
```

http://www.swe.uni-linz.ac.at/research/deco/designPatterns/Visitor/visitor.defaultroles.c++.html

History of class-based languages

- Simula 1960's
  - Object concept used in simulation
- Smalltalk 1970's
  - Object-oriented design, systems
- C++ 1980's
  - Adapted Simula ideas to C
- Java 1990's
  - Distributed programming, internet

Varieties of OO languages

- class-based languages (C++, Java, …)
  - behavior of object determined by its class
- object-based (Self, JavaScript)
  - objects defined directly
- multi-methods (CLOS)
  - operation depends on all operands

Summary

- Object-oriented design
- Primary object-oriented language concepts
  - dynamic lookup
  - encapsulation
  - inheritance
  - subtyping
- Program organization
  - Work queue, geometry program, design patterns
- Comparison
  - Objects as closures?