Modularity

• 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
Step-wise Refinement

• Wirth, 1971
  – “... program ... gradually developed in a sequence of refinement steps”
  – “In each step, instructions ... are decomposed into more detailed instructions.”

• ACM Classics of the Month
  – D. Parnas, “On the criteria to be used in decomposing systems into modules,” Communications of the ACM, 1972
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]
    for k from 1 to 1000
        make p[k] equal to k-th prime
    for k from 1 to 1000 print p[k]
end
```
Program Structure

Main program

Sub-program

Sub-program

Sub-program

Sub-program

Sub-program
Data Refinement

• Wirth, 1971 again:
  – As tasks are refined, so the data may have to be refined, decomposed, or structured
  – Natural to refine the program and data specifications in parallel

◊ For level 2, represent account balance by integer variable
◊ For level 3, 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
  – empty – priority queue containing no elements
  – insert – add element to the priority queue
  – deletemax – returns max element and the Pq of remaining elts
Example: Heap sort using priority queue (Pq)

- Priority queue: data structure with three operations
  - empty : pq
  - insert : elt * pq -> pq
  - deletemax : pq -> elt * pq

- Algorithm to sort an array using Pq (heap sort)
  begin
    create empty Pq s
    insert each element from array into s
    use deletemax to delete elts from s in decreasing order and place each in array
  end
Abstract Data Type (ADT)

• Prominent language development of 1970’s
  – Separate interface from implementation

  ◊ Example:
    - `Set` data type provides
      - `empty_set, insert, union, intersect, is_member?`, ...
    - `Set` implemented as ... lots of possibilities
      - array, linked list; sorted, unsorted; no duplicate elts, repeated elts; etc.
  – Use type checking to enforce separation

  ◊ Client program only has access to operations in interface
  ◊ Implementation encapsulated inside ADT construct

• What are advantages of ADT’s?
Module

- General construct for information hiding

- Two parts
  - Interface:
    - A set of names and their types, which are *exported* (*public*)
  - Implementation:
    - Declaration for every name in the interface
    - Declarations for additional data objects, which are *hidden* (*private*)

- Examples:
  - Modula modules, Ada packages, ML structures, C++ class, ...
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: generic C++ function

• 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); ...
    int n, m; ... ; swap(n,m); ...
C++ Standard Template Library (STL)

• 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 STL entities

• Container: Collection of typed objects
  – Examples: array, list, associative dictionary, ...

• Iterator: Generalization of pointer or address

• Algorithm: Useful functions for operating on ranges
  – Examples: for_each, count, find, remove, ...

• Adapter: Convert from one form to another
  – Example: produce iterator from updatable container

• Function object (functor): Generalization of function (a form of closure)

• Allocator: encapsulation of a memory pool
Example of STL approach

• Function to merge two sorted lists

• Conceptually (not STL syntax)

  \texttt{merge: range(s)*range(t)*comparison(u) \rightarrow range(u)}

where

- \texttt{range(s)/range(t)} – ordered “list” of elements of type \texttt{s/t}, given by pointers to first and last elements
- \texttt{comparison(u)} – boolean-valued function on type \texttt{u}
- \texttt{s} and \texttt{t} are subtypes of \texttt{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

```c++
#include <algorithm>

template < class InIterator1, class InIterator2,
           class OutIterator, class Compare >
OutIterator merge(InIterator1 first1, InIterator1 last1,
                 InIterator2 first2, InIterator2 last2,
                 OutIterator result, Compare comp)
```
Comparing STL with other libraries

- C:
  
  ```c
  qsort( (void*)v, N, sizeof(v[0]), compare_int );
  ```

- C++, using raw C arrays:
  
  ```c
  int v[N];
  sort( v, v+N );
  ```

- C++, using a vector class:
  
  ```c
  vector v(N);
  sort( v.begin(), v.end() );
  ```

- What trade-offs should you think about in choosing one over the other?
Efficiency of STL

• Running time for sort

<table>
<thead>
<tr>
<th></th>
<th>N = 50000</th>
<th>N = 500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1.4215</td>
<td>18.166</td>
</tr>
<tr>
<td>C++ (raw arrays)</td>
<td>0.2895</td>
<td>3.844</td>
</tr>
<tr>
<td>C++ (vector class)</td>
<td>0.2735</td>
<td>3.802</td>
</tr>
</tbody>
</table>

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