Programming Languages

Generics,
Containers and Iterators

Spring 2006
Let's us abstract over types and other non-value entities.

Examples:

- A sorting algorithm has the same structure, regardless of the types being sorted.
- Stack primitives have the same semantics, regardless of the objects stored on the stack.

One common use:

- algorithms on containers: updating, iteration, search

Language models:

- **C**: macros (textual substitution) or unsafe casts
- **Ada**: generic units and instantiations
- **C++**, **Java**, **C#**: templates
- **ML**: functors
<table>
<thead>
<tr>
<th>Construct</th>
<th>parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>bounds, element type</td>
</tr>
<tr>
<td>subprogram</td>
<td>values (arguments)</td>
</tr>
<tr>
<td>Ada generic package</td>
<td>values, types, packages</td>
</tr>
<tr>
<td>Ada generic subprogram</td>
<td>values, types</td>
</tr>
<tr>
<td>C++ template</td>
<td>values, classes</td>
</tr>
<tr>
<td>Java generic</td>
<td>classes</td>
</tr>
<tr>
<td>ML function</td>
<td>values (including other functions)</td>
</tr>
<tr>
<td>ML type constructor</td>
<td>types</td>
</tr>
<tr>
<td>ML functor</td>
<td>values, types, structures</td>
</tr>
</tbody>
</table>
template <class T> class Vector {
public:
    Vector (size_t); // constructor
    T& operator[] (size_t); // subscript operator
    // hopefully with checks!

private:
    ... // a struct with a size and a pointer to an array
};

Vector<int> V1 (100); // instantiation

Vector<int> Vdef; // use default constructor

typedef Vector<employee> Dept; // named instance
template <class T, int i> class Buffer {
  T v[i];    // storage for buffer
  int sz;    // total capacity
  int count; // current contents
public:
  Buffer () : sz(i), count(0) { };
  T read (){
    void write (T elem);
};

Buffer <Shape *, 100> picture;
template <class T> class List {
    struct Link {  // for a list node
        Link *pre, *succ;  // doubly linked
        T val;
        Link (Link *p, Link *s, const T& v) :
            pre(p), succ(s), val(v) { }
    }
    Link *head;
public:
    void printAll () {
        for (Link *p = head; p; p = p->succ)
            // operator<< better exist for T!
            cout << p->val << "\n";
    }
};
Instantiated implicitly at point of call:

```cpp
template <class T> void sort (vector<T>&) { ... }

void testit (vector<int>& vi) {
    sort (vi); // implicit instantiation
    // can also write sort<int> (vi);
}
```
Templates and regular functions overload each other:

template <class T> class Complex {...};

template <class T> T sqrt (T); // template
template <class T> Complex<T> sqrt (Complex<T>); // different algorithm
double sqrt (double); // regular function

void testit (complex<double> cd) {
    sqrt (2); // sqrt<int>: specialization
    sqrt (2.0); // sqrt (double): regular function
    sqrt (cd); // sqrt<complex<double>> : specialization
}
Containers are data structures to manage collections of items
Typical operations: insert, delete, search, count
Typical algorithms over collections use:
- imperative languages: iterators
- functional languages: map, fold

```cpp
template <class Coll, class Elem> Iterator {
    Iterator (Coll over) { ... } // build iterator
                      // for collection
    Elem firstElem ();       // start iteration
    Elem nextElem ();        // move to next element
                      // in collection
    bool done () const;      // termination condition
};
```
**STL**: A set of useful data structures and algorithms in C++, mostly to handle collections.

- Sequential containers: `list`, `vector`, `deque`
- Associative containers: `set`, `map`

We can *iterate* over these using (what else?) *iterators*.

Iterators provided (for `vector<T>`):

```
vector<T>::iterator
vector<T>::const_iterator
vector<T>::reverse_iterator
vector<T>::const_reverse_iterator
```
For the standard collection classes, we have the member functions `begin` and `end`, that return iterators.

There is a wide variety of sequence operations that work on arbitrary “sequences”. A sequence is defined by a pair of iterators:

- the first points to the first element in the sequence
- the second points to *one past* the last element in the sequence

We can do the following with an iterator `p` (subject to restrictions):

- `*p`  "Dereference" it to get the element it points to
- `++p, p++`  Advance it to point to the next element
- `--p, p--`  Retreat it to point to the previous element
- `p+i`  Advance it `i` times
- `p-i`  Retreat it `i` times
#include <vector>
#include <string>
#include <iostream>

int main () {
    using namespace std;
    string s = "Hello";
    vector<string> ss(20);
    for (int i = 0; i < 20; i++)
        ss[i] = string(1, 'a'+i);
    vector<string>::iterator loc =
        find (ss.begin(), ss.end(), "d");
    cout << "found:␣" << *loc
    << "＠at＠position" << loc - ss.begin() << endl;
}
STL provides a wide variety of standard “algorithms” on sequences.

Example: finding an element that matches a given condition

```cpp
// Find first 7 in the sequence
list<int>::iterator p = find (c.begin(), c.end(), 7);

// Find first number less than 7 in the sequence
bool less_than_7 (int v) {
    return v < 7;
}

list<int>::iterator p = find_if (c.begin(), c.end(),
                                less_than_7);
```
Example: doing something for each element of a sequence

It is often useful to pass a function or something that acts like a function:

template <class T> class Sum {
    T res;
public:
    Sum (T i = 0) : res (i) { } // initialize
    void operator() (T x) { res += x; } // accumulate
    T result () const { return res; } // return sum
};

void f (list<double>& ld) {
    Sum<double> s;
    s = for_each (ld.begin(), ld.end(), s);
    cout << "the sum is" << s.result() << "\n";
}
template <class Arg, class Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

struct R {
    string name; ...
};

class R_name_eq : public unary_function<R, bool> {
    string s;
public:
    explicit R_name_eq (const string& ss) : s(ss) { }
    bool operator() (const R& r) const { return r.name == s; }
};

void f (list<R>& lr) {
    list<R>::iterator p = find_if (lr.begin(), lr.end(),
                                   R_name_eq ("Joe"));
    ...
}
template <class Arg, class Arg2, class Res>
struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};

template <class T>
struct less : public binary_function<T,T,bool> {
    bool operator() (const T& x, const T& y) const {
        return x < y;
    }
};
template <class BinOp>
class binder2nd
  : public unary_function<
      typename BinOp::first_argument_type,
      typename BinOp::result_type>
  { }

protected:
  BinOp op;
  typename BinOp::second_argument_type arg2;

public:
  binder2nd (const BinOp& x,
     const typename BinOp::second_argument_type& v)
    : op(x), arg2(v) { }
  return_type operator () (const argument_type& x) const {
    return op(x, arg2);
  }
};

template <class BinOp, class T>
binder2nd<BinOp> bind2nd (const BinOp& op, const T& v) {
  return binder2nd<BinOp> (op, v);
}
void f (list<int>& c) {
    list<int>::const_iterator p =
        find_if (c.begin(), c.end(),
                 bind2nd (less<int>(), 7));
    ...
}

From Strousstrup, p 520:

    “Is this readable? ... The notation is logical, but it takes some
    getting used to.”

Equivalent to the following in ML:

fun f c = let val p = List.find (fn x => x < 7) c
          in ... end
C++ templates are Turing complete

Templates in C++ allow for arbitrary computation to be done at compile time!

```cpp
template <int N> struct Factorial {
    enum { V = N * Factorial<N-1>::V };
};

template <> struct Factorial<1> {
    enum { V = 1 };
};

void f () {
    const int fact12 = Factorial<12>::V;
    cout << fact12 << endl;  // 479001600
}
```
Only class parameters

Implementation by *type erasure*: all instances share the same code

```
interface Collection <E> {
    public void add (E x);
    public Iterator <E> iterator ();
}
```

`Collection <Thing>` is a parametrized type

`Collection (by itself)` is a raw type!
class Collection <A extends Comparable<A>> {
    public A max () {
        Iterator<A> xi = this.iterator();
        A biggest = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (biggest.compareTo(x) < 0) {
                biggest = x;
            }
        }
        return biggest;
    }
    ...
}
Why functors, when we have parametric polymorphic functions and type constructors (e.g. containers)?

- Functors can take \textit{structures} as arguments. This is not possible with functions or type constructors.
- Sometimes a type needs to be parameterized on a \textit{value}. This is not possible with type constructors.
signature SET =
  sig
    type elem
    type set

    val empty : set
    val singleton : elem -> set
    val member : elem * set -> bool
    val union : set * set -> set
    ...
  end
functor SetFn (type elem
  val compare : elem * elem -> order) : SET =
structure
  type elem = elem
  datatype set = EMPTY
  | SINGLE of elem
  | PAIR of set * set
val empty = EMPTY
val singleton = SINGLE

fun member (e, EMPTY) = false
  | member (e, SINGLE e’) = compare (e, e’) = EQUAL
  | member (e, PAIR (s1,s2)) = member (e, s1) orelse
    member (e, s2)
...
end
Example functor: the instantiation

structure IntSet =
  SetFn (type elem = int
         compare = Int.compare)

structure StringSet =
  SetFn (type elem = string
         compare = String.compare)

fun cmp (is1, is2) = ...

structure IntSetSet = SetFn (type elem = IntSet.set
                             compare = cmp)

Compare functor implementation with a polymorphic type: how are element comparisons done?
I/O for integer types.

Identical implementations, but need separate procedures for strong-typing reasons.

generic
  type Elem is range <>;  -- any integer type
package Integer_IO is
  procedure Put (Item: Elem);
  ...
end Integer_IO;
generic
    type Elem is private; -- parameter
package Stacks is
    type Stack is private;
    procedure Push (X: Elem; On: in out Stack);
    ...
private
    type Cell; -- linked list
    type Stack is access Cell; -- representation
    type Cell is record
        Val: Elem;
        Next: Ptr;
    end record;
end Stacks;
with Stacks;
procedure Test_Stacks is
  package Int_Stack
    is new Stacks (Integer);  -- list of integers
  package Float_Stack
    is new Stacks (Float);    -- list of floats
  S1: Int_Stack.Stack;       -- stack objects
  S2: Float_Stack.Stack;
  use Int_Stack, Float_Stack; -- OK, regular packages
begin
  Push (15, S1);
  Push (3.5 * Pi, S2);
  ...
end Test_Stacks;
The syntax is: `type T is ...;`

<table>
<thead>
<tr>
<th>Restriction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>private</code></td>
<td>any type with assignment (non-limited)</td>
</tr>
<tr>
<td><code>limited private</code></td>
<td>any type (no required operations)</td>
</tr>
<tr>
<td><code>range &lt;&gt;</code></td>
<td>any integer type (arithmetic operations)</td>
</tr>
<tr>
<td><code>&lt;&gt;(&lt;&gt;)</code></td>
<td>any discrete type (enumeration or integer)</td>
</tr>
<tr>
<td><code>digits &lt;&gt;</code></td>
<td>any floating-point type</td>
</tr>
<tr>
<td><code>delta &lt;&gt;</code></td>
<td>any fixed-point type</td>
</tr>
</tbody>
</table>

Within the generic, the operations that apply to any type of the class can be used.

The instantiation must use a specific type of the class.
A generic function

generic
  type T is range <>; -- parameter of some integer type
  type Arr is array (Integer range <>) of T;
      -- parameter is array of those
function Sum_Array (A: Arr) return T;

-- Body identical to non-generic version
function Sum_Array (A: Arr) return T is
  Result: T := 0; -- some integer type, so 0 is legal
begin
  for J in A’range loop -- array: 'range available
    Result := Result + A(J); -- integer: "+" available
  end loop;
  return Result;
end;
type Apple is range 1..2**15 - 1;
type Production is array (1..12) of Apple;

type Sick_Days is range 1..5;
type Absences is array (1..52) of Sick_Days;

function Get_Crop is new Sum_Array (Apple, Production);
function Lost_Work is new Sum_Array (Sick_Days, Absences);
The only available operations are assignment and equality.

generic
  type T is private;
procedure Swap (X, Y: in out T);

procedure Swap (X, Y: in out T) is
  Temp: constant T := X;
begin
  X := Y;
  Y := Temp;
end Swap;
A generic sorting routine should apply to any array whose components are comparable, i.e. for which an ordering predicate exists. This class includes more than the numeric types:

```
generic
  type T is
    private;
  with function "<" (X, Y: T) return Boolean;
  type Arr is
    array (Integer range <>) of T;
procedure Sort (A: in out Arr);
```
The actual must have a matching signature, not necessarily the same name:

```haskell
procedure Sort_Up is
  new Sort (Integer, "<", ...);

procedure Sort_Down is
  new Sort (Integer, ">", ...);

type Employee is record ... end record;
function Senior (E1, E2: Employee) return Boolean;
function Rank is new Sort (Employee, Senior, ...);
```
Useful to parameterize containers by size:

```
generic
  type Elem is private; -- type parameter
  Size: Positive; -- value parameter
package Queues is
  type Queue is private;
  procedure Enqueue (X: Elem; On: in out Queue);
  procedure Dequeue (X: out Elem; From: in out Queue);
  function Full (Q: Queue) return Boolean;
  function Empty (Q: Queue) return Boolean;
private
  type Contents is array (Natural range <>) of Elem;
  type Queue is record
    Front, Back: Natural;
    C: Contents (0 .. Size);
  end record;
end Queues;
```
We also want to define a package for elementary functions (\texttt{sin}, \texttt{cos}, etc.) on complex numbers. This needs the complex operations, which are parameterized by the corresponding real value.
The instantiation requires an instance of the package parameter

```pascal
with Generic_Complex_Types;
generic
  with package Compl is
    new Generic_Complex_Types (<>);
package Generic_Complex_Functions is
  -- trigonometric, exponential,
  -- hyperbolic functions.
... end Generic_Complex_Functions;

■ Instantiate complex types with \texttt{long\_float} components:

  ```pascal
  package Long_Complex is
    new Generic_Complex_Type (long_float);
  ```

■ Instantiate complex functions for \texttt{long\_complex} types:

  ```pascal
  package Long_Complex_Functions is
    new Generic_Complex_Functions (long_complex);
  ```