Programming Languages

Generics,
Containers and Iterators

CSCI-GA.2110-001
Fall 2012
Generic programming

Allows for type-independent data structures and functions.

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**: parametric polymorphism, functors
### Parameterizing components

<table>
<thead>
<tr>
<th>Construct</th>
<th>generic parameter(s) are:</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>bounds, element type</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++ class template</td>
<td>values, types</td>
</tr>
<tr>
<td>C++ function template</td>
<td>values, types</td>
</tr>
<tr>
<td>Java generics (all)</td>
<td>classes, interfaces</td>
</tr>
<tr>
<td>ML function</td>
<td>implicit</td>
</tr>
<tr>
<td>ML type constructor</td>
<td>types</td>
</tr>
<tr>
<td>ML functor</td>
<td>structures (containing types, values)</td>
</tr>
</tbody>
</table>
template <typename T>
class Vector {
public:
    explicit Vector (size_t);    // constructor
    T& operator[] (size_t);       // subscript operator
    ...                        // other operations
private:
    ...                        // a size and a pointer to an array
};

Vector<int> V1(100);          // instantiation
Vector<int> V2;                // use default constructor

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

Buffer<Shape *, 100> picture;
Type operations: duck typing?

template<typename 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 print (std::ostream& os) {
        for (Link *p = head; p; p = p->succ)
            // operator<< must exist for T
            // if print will be used.
            os << p->val << "\n";
    }
};
Function templates

Instantiated implicitly at point of call:

template <typename T>
void sort (vector<T>&) { ... }

void testit (vector<int>& vi) {
    sort(vi); // implicit instantiation
    // can also write sort<int>(vi);
}
Implementation of C++ templates

- Template types are not initially not known.
- Uninstantiated templates are not & cannot be compiled.
- Generic definitions must be written completely in header files.
- Once fully instantiated, all types become known.
- Compiler generates classes, functions from the template.
- Compilation proceeds in the usual manner after this.
- Compiler may optimize by reusing multiple occurrences of a fully instantiated template.
C++: inheritance vs. generics

- OOP: “is-a” vs. generics: “has-a”
- In some respects, one can be used in place of the other.
- Example: rather than inherit, pass a “base class” as a generic parameter.
- There are fundamental differences:
  - Generics are static only; no concept of runtime binding.
  - Generics: reuse the same structure/algorithm with different data.
  - Inheritance: reuse the same data and add new data.
- Alex Stepanov: “inheritance doesn’t work.”
- i.e., cannot establish type-distinct variations of base class methods because the data members are fixed.
Partial and Explicit Specialization

Templates and regular functions overload each other:

```cpp
template <typename T> class Complex {...};

template <typename T> T sqrt (T);  // most general
template <typename T> Complex<T> sqrt (Complex<T>);  // partial specialization

double sqrt (double);  // explicit specialization

void testit (Complex<double> cd) {
  sqrt(2);  // sqrt<int>
  sqrt(2.0);  // sqrt (double): regular function
  sqrt(cd);  // sqrt<complex<double> >
}
```

Partial specialization narrows the set of acceptable template parameters. Compiler will select the most specialized (specific) type.

Note: `double sqrt (double) ⇔ template <> double sqrt (double).`
Default template params

Taken from the C++ header file `vector`:

```cpp
template<typename _Tp, typename _Alloc = allocator<_Tp> >
class vector : protected _Vector_base<_Tp, _Alloc>
{ ... }
```

Template parameter `_Tp` is used to instantiate the base class.

Note: protected inheritance makes public & protected parts of `_Vector_base` protected in `vector`.
Iterators and containers

- 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

```java
interface Iterator<E> {
    boolean hasNext (); // returns true if there are
                        // more elements
    E next ();         // returns the next element
    void remove ();   // removes the current element
                      // from the collection
}
```
The Standard Template Library

**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`

Iterator concepts: trivial, input, output, forward, bidirectional, and random access.
Iterators in C++

For standard collection classes, we have member functions `begin` and `end` that return iterators.

We can do the following with an iterator \( p \):
- \(*p\) “Dereference” it to get the element it points to (trivial)
- \(++p, p++\) Advance it to point to the next element (forward)
- \(--p, p--\) Retreat it to point to the previous element (bidirectional)
- \(p+i, p-i\) Advance/retreat it \( i \) times (random access)
- \(p[i]\) Access index \( i \) (random access)

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. Cannot dereference, but address must still be valid (for pointer arithmetic).

```cpp
for (auto i = v.begin(); i != v.end(); i++)
    { ... }```

There are a wide variety of operations that work on sequences.
#include <vector>
#include <string>
#include <iostream>

int main () {
    using namespace std;

    vector<string> ss(20); // initialize 20 empty strings

    for (int i = 0; i < 20; i++)
        ss[i] = string(1, 'a'+i); // assign "a", "b", etc.

    vector<string>::iterator loc =
        find(ss.begin(), ss.end(), "d"); // find first "d"

    cout << "found:\n" << *loc
         << " at position\n" << loc - ss.begin() << endl;
}
STL algorithms, part 1

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);
```

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

```cpp
list<int>::iterator p = find_if(c.begin(), c.end(), less_than_7);
```

```cpp
// C++11:
auto 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:

```cpp
template <typename 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>& ds) {
    Sum<double> sum;
    sum = for_each(ds.begin(), ds.end(), sum);
    cout << "the sum is " << sum.result() << "\n";
}
```
Function objects

template <typename Arg, typename 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");
    ...
}
C++ templates: 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
}
```
Generics in Java

Only class parameters (no value)
Implementation by *type erasure*: all instances share the same code
Unlike C++, generics are fully compilable (uninstantiated).

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

After type erasure becomes...

```java
interface Collection {
    public void add (Object x);
    public Iterator iterator ();
}
```

Collection <Thing> is a parametrized type
Collection (by itself) is a raw type!
Generic methods in Java

We can do better than duck typing. Make the generic type requirements *explicit* and restrict the allowable set of parameter types.

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;
    }
    ...
}
Functors in ML

Functors yield *structures*, similar to the way C++ templates yield concrete classes.

Why functors, when we have parametric polymorphic functions and type constructors (e.g., containers)?

- Functors can take structures as arguments. This is not possible with functions or type constructors.
- Sometimes a type needs to be parameterized on a *value*. This is not possible with type constructors.
Example functor: signature

Similar to an interface (Java) or forward declaration (C++).

```plaintext
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

```ml
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?
Generics in Ada95

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;
A generic Package

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;
Instantiations

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;
**Type parameter restrictions**

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 basic operations (e.g., assignment, equality)</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;)</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
begin
  for J in A’range loop -- array: ’range available
    Result := Result + A(J); -- integer: "+" available
  end loop;
  return Result;
end;
Instantiating a generic function

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);
Generic private types

The only available operations are basic operations, which include 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;
Subprogram parameters

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:

```haskell
generic
type T is -- parameter
  private;
with function "<" (X, Y: T) -- parameter
  return Boolean;
type Arr is -- parameter
  array (Integer range <>) of T;
procedure Sort (A: in out Arr);
```
Supplying subprogram parameters

The actual must have a matching signature, not necessarily the same name:

```plaintext
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;
Packages as parameters

generic
    type Real is digits <>; -- any floating type
package Generic_Complex_Types is
    -- complex is a record with two real components
    -- package declares all complex operations:
    --    +, -, Re, Im...
...
end Generic_Complex_Types;

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.
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:

package Long_Complex is
    new Generic_Complex_Types (\texttt{long\_float});

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

package Long_Complex_Functions is
    new Generic_Complex_Functions (\texttt{long\_complex});