Review

Last week

- Constructors, Destructors, and Assignment Operators
- Classes and Functions: Design and Declaration
- Classes and Functions: Implementation
- Inheritance and Object-Oriented Design
Outline

- Generic Programming

Sources for today’s lecture:

PLP, 8.4
Generic programming

Subroutines provide a way to abstract over *values*.

Generic programming lets us abstract over *types*.

**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 software components

<table>
<thead>
<tr>
<th>Construct</th>
<th>Parameter(s):</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++ class template</td>
<td>values, types</td>
</tr>
<tr>
<td>C++ function template</td>
<td>values, types</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>
Templates in C++

template <typename T>
class Array {
public:
    explicit Array (size_t); // constructor
    T& operator[] (size_t);   // subscript operator
    ... // other operations
private:
    ... // a size and a pointer to an array
};

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

typedef Array<employee> Dept; // named instance
Type and value parameters

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;
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)
            // will fail if operator<< does not exist for T
            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);
}
```
Templates and regular functions overload each other:

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

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

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

The *Standard Template Library (STL)* is 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>`):

```cpp
vector<T>::iterator
vector<T>::const_iterator
vector<T>::reverse_iterator
vector<T>::const_reverse_iterator
```

Note: Almost no inheritance used in STL.
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 \( p \) (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

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

There are a variety of operations that work on sequences.
# Iterator example 1

```cpp
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> v;
    for (int i = 0; i < 10; ++i) v.push_back(i);
    // Print list
    vector<int>::iterator it;
    for (it = v.begin(); it != v.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl << endl;
    // Use reverse iterator to print in reverse order
    vector<int>::reverse_iterator rit;
    for (rit = v.rbegin(); rit != v.rend(); ++rit) {
        cout << *rit << " ";
    }
    cout << endl;
}
```
# Iterator example 2

```cpp
#include <vector>
#include <string>
#include <iostream>

using namespace std;

int main () {
    vector<string> ss(20); // initialize to 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: " << *loc
        << " at position " << 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
#include <algorithm>

// 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);
```
STL algorithms, part 2

**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
#include <iostream>
#include <algorithm>

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

```cpp
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");
    ...
}
```
template<typename Arg, typename Arg2, typename Res>
struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};

template<typename T>
struct less : public binary_function<T,T,bool> {
    bool operator() (const T& x, const T& y) const {
        return x < y;
    }
};
Currying with function objects

template <typename 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 <typename BinOp, typename T>
binder2nd<BinOp> bind2nd (const BinOp& op, const T& v) {
  return binder2nd<BinOp> (op, v);
}
Partial application with function objects

```cpp
void f (const list<int>& xs, int limit) {
    list<int>::const_iterator it =
        find_if(xs.begin(), xs.end(),
            bind2nd(less<int>(), limit));
    int num = *it;
    ...
}
```

“Is this readable? ... The notation is logical, but it takes some getting used to.” – Stroustrup, p. 520

Equivalent to the following in ML:

```ml
fun f xs limit =
    let val optNum = List.find (fn x => x < limit) xs
    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
}
```
Generics in JAVA

Only class parameters

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

```java
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!*
Generic methods in JAVA

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

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: the signature

```ml
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
```
Example functor: the implementation

```
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)
```