Generic iterators

Dealing with unknown data structures

Example from last lecture

```
template<class X>
bool find(InSeq<X>& s, X x)
{
    while (s.avail()) {
        if (s.next() == x)
            return true;
    }
    return false;
}
```

- Where does this class rely on its argument being derived from `InSeq`?

The example generalized

- Instead of requiring an InSeq, we can use any class with the right properties

```
template<class S, class X>
bool find(S& s, X x)
{
    while (s.avail()) {
        if (s.next() == x)
            return true;
    }
    return false;
}
```

Implications

- When we write a template function with a general type parameter, we can call it with an argument that has *any* type with the appropriate properties
- The compiler checks those properties during template instantiation, before the program runs
- No overhead at execution time

More implications

- When we design a template function, we can start by thinking about how we would like to be able to use the arguments
- Afterwards, we can see if there is an available type that does what we want
- If not, we can create one or revise our template function definition

An example

- We defined `find` to work with any class that has `avail()` and `next()` members
- Another approach might be to define `find` to work with built-in arrays, or types, such as pointers, that act in conjunction with built-in arrays

Making `find` work with built-in arrays

```
template <class T>
bool find(T* tp, int n, const T& x)
{
    while (n > 0) {
        if (*tp++ == x)
            return true;
        --n;
    }
    return false;
}
```

- Where do we assume `tp` is a pointer?

Generalizing `find` to other types

```
template <class P, class T>
bool find(P p, int n, const T& t)
{
    while (n > 0) {
        if (*p++ == t)
            return true;
        --n;
    }
    return false;
}
```

What's going on here?

- Saying that p has type P (where P is a template parameter) says little about p
- Our particular version of `find` requires us to be able to evaluate `*p++` and compare the result to `t`
- The requirements on P are implicit, and come out only during instantiation

How might we make this `find` more useful?

- We should not have to know the number of elements in advance
  - we might be searching in a file …
  - … or in a linked list or similar structure
- Instead of returning `true` or `false`, it should tell us where the value was found, or return failure somehow

Defining ranges without counting

- We need a way to represent a sequence even if we don’t know how big it is when we start
- A pair of pointers turns out easiest
- Following long-standing C conventions, we use a pointers to
  - the first element of the sequence
  - one past the last element of the sequence

Advantages of off-the-end pointers

- They make it much easier to represent and detect an empty sequence
  - Otherwise, the end pointer might be less than the begin pointer
  - If there are no elements, we'll surely need a pointer that doesn't point to an element
- They provide a convenient error return
- They let us use == to test for the end

Why is == important?

- When we use an order relation (<, <=, >, >=) to check for the end of a sequence, we are making an unnecessary assumption
- For example, the elements of a linked list do not necessarily occupy ordered memory locations

The next revision of `find`

```
template<class P, class T>
P find(P begin, P end, const T& t)
{
    while (begin != end && *begin != t)
        ++begin;
    return begin;
}
```

- If you try it with symmetric bounds, you’ll see what’s wrong with them
- This version of `find` is part of the standard C++ library

Assumptions about type P

- Well behaved with regard to copying
- Prefix ++ and * defined
- Binary != defined

We can fill out the assumptions

- If we are going to assume prefix ++, we should probably assume postfix ++ also
- Ditto for `!=` and `==`
- It also makes sense to assume that `p->mem` means the same as `(*p).mem`
- The standard library calls any type for which these assumptions hold an *input iterator*

Why no null iterators?

- Suppose we want to use `find` on a singly linked list
  - Define a class that contains a pointer
  - Define ++ on that class to move the pointer to the next list element
- A null iterator would then be the logical way to mark the end of a list
- Why not return null on failure?

The answer is subtle

- Suppose we have an algorithm
  - whose input is a begin–end pair
  - whose output is either in the [begin, end] range (including one past the end) or null
- There's trouble if null can end a list, because we can't tell null from end+1
- If null can't end a list, it restricts our data structures

Another reason to return end

- If `find` returns end on failure, that tells us where to insert a new element in the sequence
- Similar arguments turn out to work well for other algorithms too

Input iterator requirements

- If
  - p is well behaved when we copy and assign it, and
  - `*p`, `p==q`, `p!=q`, `++p`, and `p++` are all sensibly defined, and
  - `p->mem` means `(*p).mem`
- Then we call p an input iterator

Output iterator requirements

- Like input iterator requirements, except that
  - we need not be able to read the value of `*p`
  - we must be able to evaluate `*p = q`
  - each distinct value of p must have `*p` assigned to exactly once

Using input and output iterators

```
template<class In, class Out>
Out copy(In begin, In end, Out out)
{
    while (begin != end)
        *out++ = *begin++;
    return out;
}
```

- This function is part of the C++ standard library

A copy example

```
char msg1[] = “Hello ”;
char msg2[] = “world”;
char message[100];
char* p = copy(msg1,
               msg1 + sizeof(msg1)-1,
               message);
copy(msg2, msg2 + sizeof(msg2), p);
cout << message << endl;
```

Why is this abstraction useful?

- It is not much harder to implement than the `avail/next` abstraction
- Built-in pointers meet the requirements as long as you use them to point to contiguous memory
  - algorithms work smoothly on built-in arrays
  - code is about as efficient as it can be

Other categories of iterators

- Forward iterators combine the properties of input and output iterators
  - you don't have to assign through each iterator exactly once
  - you can come back to a place later
- Bidirectional iterators also handle --
- Random-access iterators also allow arithmetic with integers, relational ops

Using bidirectional iterators

```
template<class T>
void reverse(T begin, T end)
{
    while (begin != end) {
        --end;
        if (begin != end) {
            swap(*begin, *end);
            ++begin;
        }
    }
}
```

A more compact version

```
template<class T>
void reverse(T begin, T end)
{
    while (begin != end)
        if (begin != --end)
            swap(*begin++, *end);
}
```

- This is another standard library function

The swap function

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

Using random-access iterators

```
template<class T, class X>
bool binary_search(T begin, T end, const X& x)
{
    while (begin <= end) {
        T mid = begin + (end-begin)/2;
        if (x == *mid)
            return true;
        if (x < *mid)
            end = mid;
        else
            begin = mid + 1;
    }
    return false;
}
```

Iterator category summary

- Input iterators can represent input files
- Output iterators can represent output files
- Forward iterators can represent singly linked lists
- Bidirectional iterators can represent doubly linked lists
- Random-access iterators can represent arrays

How the library uses these abstractions

- Every library data structure has a corresponding iterator type
- Most algorithms work on iterators, rather than directly on data structures
- Library containers have `begin()` and `end()` members that yield iterators

Discussion

- Suppose we have a function to sum the elements of a container
- In the inner loop of that function, the container will always be the same type
- Therefore, if we can push the decision about the type out of the loop, we will decide once instead of many times
- Ideally, we would like to avoid deciding during execution time altogether

Example

- If we define a function that takes an `InSeq<T>&` argument, each use of that argument requires a type decision (because `InSeq` uses virtual functions each time it accesses the corresponding data structure)
- If we make it take a `T` directly, the decision can be at compile time

Should type decisions be at compile time?

- When they can be, because of
  - faster programs
  - errors that you see instead of your users
- But it requires planning
  - heterogeneous containers
  - objects and communication
- Such problems are hard, and language sensitive

Generic applications

- Imagine an abstraction for the operations you care about in
  - a database system
  - a window system
- Then (in principle) you could write an application as a template (or local equivalent) that takes a database and a window system as parameters

Summary

- Programs don’t always need to know the exact types they use
  - they might know the types’ properties
  - they might know specific operations on those types
- Some languages let you fix the types during compilation, some during execution

Shorter summary

Sometimes it helps if you don’t know too much about what you’re doing

Projects

- Each team will be expected to demonstrate its project
  - be prepared to answer design and process related questions
- Each team has to find appropriate computing facilities for the demonstration and schedule a mutually agreeable time
- All demonstrations during exam week

Project dates

- 4 slots/day
  - 10:30 - 11:30
  - 1:30 - 2:30
  - 3:00 - 4:00
  - 4:30 - 5:30 (*)
- Reviews will be held Monday - Thursday, 5/17 - 5/20

Project scheduling

- Email us 3 choices by 4/15
- Final schedule will be distributed in class on 4/21