Class Templates

A stack template:

```
template <typename T>
  // T must be DefaultConstructible (new T[size]) and
  // Assignable (v[top++] = item, return v[--top])
class Stack {
public:
    Stack(size_t size) : v(new T[size]), top(0) {}
    ~Stack() { delete[] v; }
    void push(const T& item) { v[top++] = item; }
    T pop() { return v[--top]; }
    bool empty() const { return top == 0; }
    Stack(const Stack&) = delete;
    Stack& operator=(const Stack&) = delete;
private:
    T* v;
    size_t top;
};
```

Defining Template Class Members

It is possible to write the definition of a class member function outside the class definition, but then the template information must be repeated:

```
template <typename T> // this goes in a header file
class Stack {
    ...
    void push(const T& item);
    ...
};

template <typename T> // this goes in the same header file
void Stack<T>::push(const T& item) { v[top++] = item; }
```

Using a Class Template

With function templates, the compiler can deduce template parameter types from the function calls. With class templates, the types must be explicitly supplied when an object is created.

```
Stack<char> cs(100);
Stack<Point> ps(200);

cs.push('X');
ps.push(Point(10, 20));
...
char ch = cs.pop();
```

Nontype Template Parameters

A template parameter need not be a type, it can also be a constant value. And template parameters may have default values:

```
template <typename T = int, size_t size = 100>
class Stack {
public:
    Stack() : top(0) {}
    ...
private:
    T v[size];
    size_t top;
};

void f() {
    Stack<double, 200> s1; // 200 doubles
    Stack<Point> s2;       // 100 Points
    Stack s3;              // 100 ints
    ...
}
```

Types Inside the Template Definition

Template classes often export type aliases:

```
template <typename T>
class vector {
public:
    using value_type = T;
    ...
};
```

This means that you can define variables of the vector's element type:

```
void sort(vector<int>& v) {
    ...
    vector<int>::value_type tmp = v[i];
    ...
}
```

Here, you could just as well have written `int` — but see next slide.

Using Exported Types in a Template, typename

Now suppose that the function `sort` is a template that can sort both vectors and other containers (of type `T`). The element type is `T::value_type`, but before the type specifier you must write `typename`. This is because the compiler cannot determine whether `T::value_type` is a type or a (static) member variable.

```
template <typename T>
void sort(T& v) {
    ...
    typename T::value_type tmp = v[i]; // C++11: auto tmp =
    ...
}
```

Rule: write `typename` before the type specifier when using a type that depends on a template parameter.

Template Metaprogramming

A template can instantiate itself recursively, so the "template language" is in fact a Turing-complete programming language. Suppose that you in a program need values of *n!*, where the *n*-s are constant:

```
template <int n> struct Factorial {
    static const int value = n * Factorial<n - 1>::value;
};

template <> struct Factorial<1> { // specialization for n = 1
    static const int value = 1;
};

int main() {
    cout << Factorial<6>::value << endl;
}
```

The value 720 is computed by the *compiler* through recursive instantiations of the class.

About the Book and the Slides

We return to the standard library, chapters 9–11 of Lippman. However, we will treat the material in almost the reverse order from Lippman:

Iterators 3.4, 9.2.1–9.2.3, 10.4

Function objects 10.3

Algorithms 10.1–10.2, 10.5–10.6

Containers Chapters 9 and 11

Containers, Algorithms, Iterators

The standard library (also called the standard template library, STL) provides these components:

Containers for homogenous collections of values (vectors, deques, lists, sets, maps, stacks, queues, priority queues).

Algorithms for operating on containers (searching, sorting, copying, . . .).

Iterators for iterating over containers.

One important design goal for the library was efficiency. Since everything builds on templates, there is no execution-time penalty for using the library.

The library is *not* an object-oriented library, although inheritance is used internally in the implementation. There is nothing like Java's strange `Collection` hierarchy.

Iterators

We have already used iterators to traverse `vector`'s. You can:

- set an iterator to the start of a container,
- access the value that an iterator "points to",
- increment the iterator to point to the next value,
- check if the iterator has reached the end of a container.

A pointer is an iterator for arrays. We have done things like the following:

```
int ia[] = {5, 7, 2, 3};
for (int* p = ia; p != ia + 4; ++p) {
    cout << *p << endl;
}
```

An Algorithm Using Pointers

The following function finds the first occurrence of a value in an `int` array, from the address `beg` up to, but not including, `end`:

```
int* find(const int* beg, const int* end, int value) {
    while (beg != end && *beg != value) {
        ++beg;
    }
    return beg;
}
```

This should be possible to generalize:

- The function applies only to arrays of `int`'s.
- The algorithm (linear search) is usable for any linear sequence (array, vector, linked list, ...), but as it's written the function applies only to arrays.

A Generic Algorithm

With a function template you can write a generic algorithm:

```
template <typename InputIterator, typename T>
InputIterator
find (InputIterator beg, InputIterator end, const T& value) {
    while (beg != end && *beg != value) {
        ++beg;
    }
    return beg;
}
```

This version of `find` works equally well as the previous one for `int` arrays:

```
int ia[] = ...;
int nbr = ...;
int* p = find(ia, ia + 4, nbr);
if (p != ia + 4) {
    cout << nbr << " found at pos " << p - ia << endl;
} else {
    cout << nbr << " not found" << endl;
}
```

Using the Generic Algorithm

The generic algorithm can be used for arrays of all types (that support equality checking with `!=`):

```
double id[] = ...;
double nbr = ...;
double* p = find(id, id + 4, nbr);
```

And it can be used for other containers (that have iterators):

```
vector<int> v = ...;
int nbr = ...;
vector<int>::iterator p = find(v.begin(), v.end(), nbr);
```

The `find` algorithm is one of the standard algorithms in the library.

Requirements on Iterators

The find algorithm places the following requirements on an `InputIterator`:

- must be `Assignable` (argument passed by value).
- must be `EqualityComparable` (`!=`).
- must support dereferencing for reading (`*`).
- must support prefix increment (`++`).

Iterators for other containers must meet these requirements. See the following slides for an example.

Given a container, it must also be possible to find the begin and end of the container (like the `begin()` and `end()` functions in `vector`).

A Linked List (Sketch Only)

In a singly-linked list of `int`'s, a list and a list node could look like this (many details are omitted):

```
class List {
public:
    using iterator = ListIterator;
    iterator begin() { return iterator(first); }
    iterator end() { return iterator(nullptr); }
    ...
private:
    Node* first;
};

struct Node {
    int data;
    Node* next;
};
```

An Iterator Class (Sketch Only)

The `ListIterator` class can look like this:

```
struct ListIterator {
    Node* current;

    explicit ListIterator(Node* c) : current(c) {}

    bool operator!=(const ListIterator& rhs) const {
        return current != rhs.current;
    }

    int& operator*() { return current->data; }

    ListIterator& operator++() {
        current = current->next;
        return *this;
    }
};
```

Getting Iterators From Containers

Each library container has two functions `begin()` and `end()`. `begin()` returns an iterator to the first element of the container, `end()` returns an iterator one past the last element.

```
template <typename T>
class vector {
public:
    using iterator = T*;
    iterator begin() { return values; }
    iterator end() { return values + size; }
    ...
private:
    T* values;    // dynamically allocated array of T's
    size_t size; // number of elements
};
```

This is an example only — it is not guaranteed that a `vector` iterator is a pointer.

Versions of `begin()` and `end()`

Each library container class defines two iterator types:

`iterator` Allows reading and writing (`x = *it` and `*it = x`)

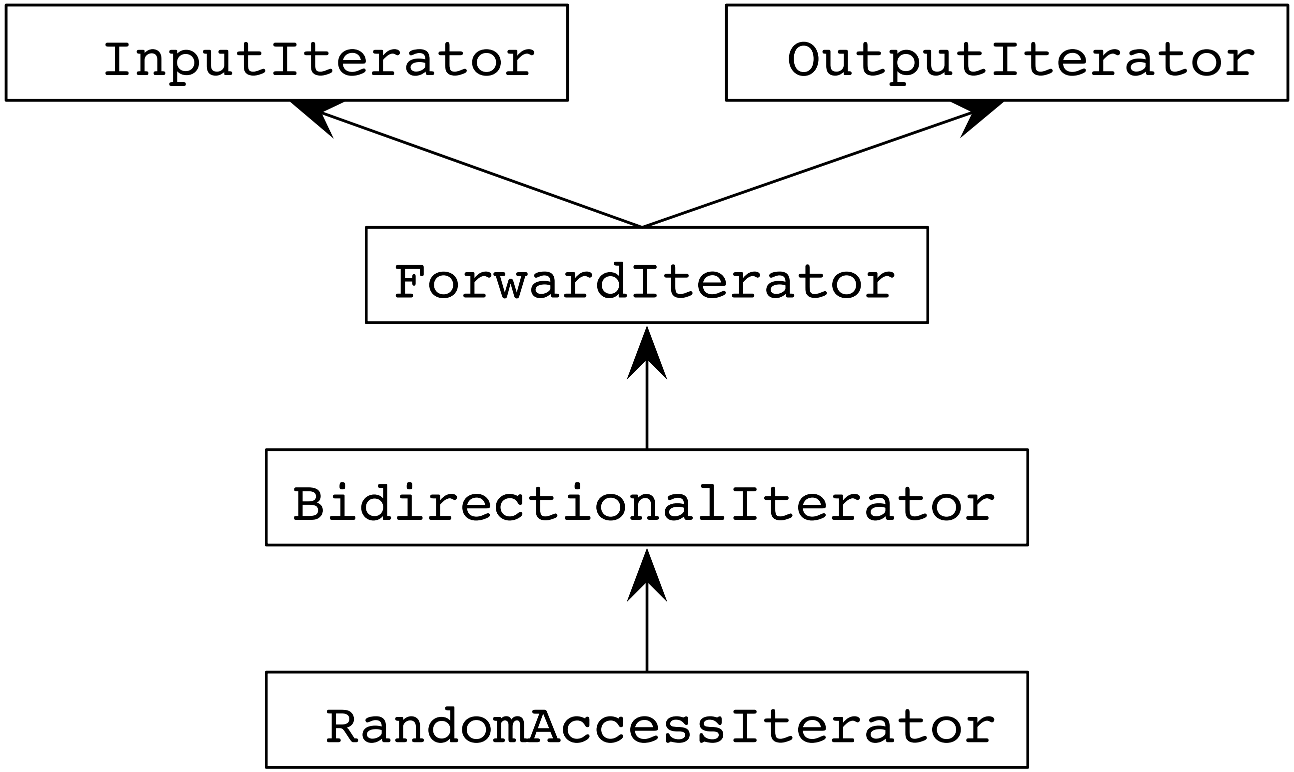
`const_iterator` Only allows reading (`x = *cit`)

The `begin()` and `end()` functions are overloaded on `const`. They return const iterators for const objects, non-const iterators for non-const objects. The new library introduced `cbegin()` and `cend()` members that always return const iterators `C++11`. Examples:

```
void f(const vector<int>& v1) {
    vector<int> v2;
    auto it1 = v1.begin();  // vector<int>::const_iterator
    auto it2 = v2.begin();  // vector<int>::iterator
    auto it3 = v1.cbegin(); // vector<int>::const_iterator
    auto it4 = v2.cbegin(); // vector<int>::const_iterator
}
```

Iterator Concepts

The requirements on an input iterator are weak; often more "powerful" iterators are needed. There is a hierarchy among iterator concepts:



Iterator Concepts, Details

An iterator "points to" a value. All iterators are `DefaultConstructible` and `Assignable` and support `++it` and `it++`. Additional concepts:

`InputIterator` Can be dereferenced to read a value, is `EqualityComparable`.

`OutputIterator` Can be dereferenced to (over)write a value.

`ForwardIterator` Can both read and write.

`BidirectionalIterator` Can move both forwards and backwards.

`RandomAccessIterator` Allows pointer arithmetic and subscripting.

A built-in pointer is a model of `RandomAccessIterator`.

Input and Output Iterators, Example

The following algorithm copies the range `[beg, end)` (from `beg` up to but not including `end`) to the range starting at `dest`.

```
template <typename InIt, typename OutIt>
  // InIt is a model of InputIterator
  // OutIt is a model of OutputIterator
OutIt copy(InIt beg, InIt end, OutIt dest) {
    for (; beg != end; ++beg, ++dest) {
        *dest = *beg;
    }
    return dest;
}
```

- The implementation relies on the fact that the iterators are passed by value. This is normally the case — iterators are built-in pointers or "small" objects, so there is no efficiency penalty.
- If the iterators had been passed by constant reference you would have needed temporary variables in the algorithm.

Using copy

copy is one of the standard algorithms. It can be used like this:

```
void f() {
    int x[] = {1, 2, 3, 4};
    int y[10];
    int* last = copy(x, x + 4, y);

    vector<int> v = {5, 6, 7, 8, 9, 10};
    copy(v.begin(), v.end(), last);
}
```

- Note that there must be enough memory allocated for the destination range — the copy algorithm cannot allocate any memory (it doesn't know how or where). Examples see next slide.

Iterators Don't Allocate Memory

The following uses of `copy` are wrong:

```
vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int y[5];
copy(v.begin(), v.end(), y); // y is too short

vector<int> destV;
copy(v.begin(), v.end(), destV.begin()); // destV has no elements
```

- The first error must be fixed by creating a longer array.
- In the second call to `copy` all would be ok if the destination iterator did `destV.push_back` on the values instead of just writing. For this, the iterator must know which vector it's supposed to `push_back` on.

Insert Iterators

Iterators that know about their container and know how to insert a value are provided by the library in the form of iterator adapters called *insert iterators* or inserters. There are three kinds:

Back inserters `*it = value`: `operator=` calls `push_back(value)`, `operator*` and `operator++` do nothing.

Front inserters same, but calls `push_front(value)` (insert at front).

General inserters same, but calls `insert(pos, value)` (insert at position `pos`).

The copying problem can be solved with a back inserter:

```
vector<int> destV;
copy(v.begin(), v.end(), back_inserter(destV));
```

Stream Iterators, Input

The iterator adapter `istream_iterator` functions as an input iterator but the values are read from a stream. It takes a stream as an argument to the constructor and the value type as a template argument. The default constructor creates an iterator that represents the end of a stream. Example:

```
istream_iterator<string> input(cin); // read strings from cin
istream_iterator<string> end;        // end iterator
vector<string> v;
while (input != end) {
    v.push_back(*input++);
}
```

Better, using the copy algorithm:

```
copy(input, end, back_inserter(v));
```

Counting Words in a String

This was an exam question: count the number of words in a string `s`. Words are separated by whitespace.

- Standard solution:

```
istringstream iss(s);
string temp;
int words = 0;
while (iss >> temp) {
    ++words;
}
```

- More elegant solution — the library function `distance` computes the distance between two iterators:

```
istringstream iss(s);
int words = distance(istream_iterator<string>(iss),
                     istream_iterator<string>());
```

Stream Iterators, Output

The iterator adapter `ostream_iterator` functions as an output iterator but the values are written to a stream. It takes a stream as an argument to the constructor and the value type as a template argument. Optionally, a C-style string that will be written between the values can be specified as a second argument to the constructor.

```
vector<string> v;
...
copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
```

To write a newline after each value you must use the delimiter `"\n"`, the linefeed character. You cannot use `endl` (`endl` isn't a character or a string, but an I/O manipulator).

Reverse Iterators

A reverse iterator is an iterator that traverses a container backward. `++` on a reverse iterator accesses the previous element, `--` accesses the next element. The containers have types `reverse_iterator` and `const_reverse_iterator` and functions `rbegin()` and `rend()` (and crbegin() and crend() `C++11`).

Example (read words, print backwards):

```
vector<string> v;
copy(istream_iterator<string>(cin), istream_iterator<string>(),
     back_inserter(v));
copy(v.rbegin(), v.rend(),
     ostream_iterator<string>(cout, "\n"));
```

Iterator Traits

Sometimes, you must know more about an iterator than just the fact that it is an iterator. Suppose that you wish to write a function to sort the values in an iterator range, and that you need to define a temporary variable to swap two values:

```
template <typename It>
void sort(It beg, It end) { ... elem_type temp = *beg; ... }
```

The type of the value to which an iterator points is available from the class `iterator_traits`:

```
using elem_type = typename iterator_traits<It>::value_type;
```

Use `auto` in C++11.

Iterator Notes

- For efficiency reasons, use `++x` instead of `x++` when incrementing an iterator (`--x` when decrementing a bidirectional or random-access iterator).
- You must know what kind of iterator you are dealing with. For instance, this is allowed only for random-access iterators:

  ```
  Iterator first = ...;
  Iterator next = first + 1;
  ```

  But `Iterator next = first; ++next;` is legal for all iterators.
- The first expression below isn't legal for any iterator, but the equivalent second expression is legal for random-access iterators:

  ```
  Iterator mid = (beg + end) / 2;
  Iterator mid = beg + (end - beg) / 2;
  ```