

Templates

A template is like a class, except it has a “hole” in it, which has to be “filled” with a type. For example, `list<string>` defines a list that can contains String objects. A **container** is a kind of object that holds other objects. For example, a `vector<T>` is a container for objects of type T. A template is defined like a class, except we are allowed to parameterize the template with a type parameter that is *substituted into the template at compile time to generate a new class (type)*.

```
template<class T> class list { // implements a sequence as a doubly-linked list
private:
    ... // internal dynamically allocated linked list implementation
public:
    typedef bidirectional_iterator<T, distance_type> iterator;
    typedef bidirectional_iterator<T, distance_type> const_iterator;
    ...
    iterator begin()
    iterator end();
    const_iterator begin() const; const_iterator end() const;
    ...
    iterator insert(iterator position, const T& x);
    void push_front(const T& x) { insert(begin(), x); }
    void push_back(const T& x) { insert(end(), x); }
    void pop_front() { erase(begin()); }
    void pop_back() { iterator tmp = end(); erase(--tmp); }
    T& back();
};
```

I recommend the standard C++ list template (e.g., there is one available in GNU libstdc++). For source code, check out the STL code on the course web page or look at the `<list>` header file for your compiler (e.g., `/usr/gnu/include/g++/stl_list.h`). A list template also provides a special “helper” object, called an iterator, that allows you to iterate over elements of the list, similar to traversing a linked list, but without having to worry about direct pointer manipulation. The iterator is also used to implement many of the list methods. **Q: How might you implement a Stack using a list template like the one shown above?**

Sequence Containers: Lists, Vectors, Deques

Lists and vector containers have a similar interface, but are not the same kind of container. A list is implemented as a doubly linked list, so you can insert/delete any element in the list in constant time, but elements are accessed in linear-time since the list has to be traversed ($O(n)$). A vector is implemented as a dynamically growing array, so you can insert/remove an item at the end of the vector in constant time, but inserting/removing an item from any other position requires relocating up to $n-1$ elements. However, random access takes constant time since it is an array (of varying length). A deque is like a vector, except that it also provides constant time insert/delete operations at the front and the back.

```
char* s = "C++ is a better C";  
int len = strlen(s);
```

Suppose we want to find the first occurrence of a letter in a character sequence. The find algorithm requires three arguments: the start position of the sequence, the end position, and the character to search for

```
// in plain C or C++  
char* where = find(&s[0], &s[len], 'e'); // or args s and s+len  
  
// using an ANSI C++ list template and iterator  
  
list<char> lst(&s[0], &s[len]);  
list<char>::iterator where = find (lst.begin(), lst.end(), 'e');  
  
// using an ANSI C++ vector template and iterator  
vector<char> vec(&s[0], &s[len]);  
vector<char>::iterator where = find (vec.begin(), vec.end(), 'e');  
  
deque<char> deq(&s[0], &s[len]);  
deque<char>::iterator where = find(deq.begin(), deq.end(), 'e');
```

Iterators are an object-oriented abstraction for a pointer/index (i.e., an address) as the way to traverse a sequence of elements, but are “safer” in the sense that the iterator object encapsulates the details of the actual pointer/index.

Iteration Abstraction

A C++ **iterator** is an abstraction for the common operation of iterating over some kind of sequence, such as an array, vector or a list. The nice thing about iterators, is that they are sufficiently abstract such that you don't have to worry about maintaining pointers. Compare the following code examples:

```
// a doubly linked list node in C
typedef struct dll_node {
    int val;
    struct dll_node *next, *prev;    // forward and backware node pointers
} DllNode;
```

```
DllNode *head, *p;
... // dynamically create nodes and insert them into a linked list of nodes
for (p = head; p != 0; p = p->next)
    printf("%d\n", p->val);
```

```
// a doubly linked list in C++
list<int> int_list;
... // insert integers into the list
list<int>::const_iterator iter; // why use a const_iterator?
for (iter = int_list.begin(); iter != int_list.end(); ++iter)
    cout << *i << endl;
```

```
vector<string> string_vector;
for (vector<string>::iterator i = string_vector.begin(); i != string_vector.end(); i++)
    cout << *i << endl;
```

Iterators can be thought of as “index objects” for the positions in a sequence (list or vector). Note that the list template is implemented as a doubly-linked list of nodes (using forward and back pointers). The post-increment operator ++ is used to traverse a list object by following the forward node pointer. You might also traverse a list in reverse:

```
for (list<int>::iterator i = int_list.end(); i != int_list.begin(); --i) { ... }
```

Using Iterators

The find algorithm for list, vector, and deque is written to be independent of the kind of sequence being linearly searched, since all that matters is that iterators that work with the particular sequence are provided as arguments.

```
template<class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value)
{
    while (first != last && *first != value)
        ++first;
    return first;
}
```

All sequence containers implement insert and erase methods, which take iterators as arguments. So, it is easy to update the elements of a list:

```
list<string> the_list;
...
//deletion
list.erase(the_list.front()); // same as list.pop_front();
list.erase(the_list.end()); // same as list.pop_back();
list.erase(the_list.begin(), the_list.end()); // erase all elements, like ~list does
...
//insertion
list<string> a_list;
a_list.insert(a_list.begin(), string("Hello")); // same as push_front
a_list.insert(a_list.end(), string("Goodbye")); // same as push_back
list.insert(the_list.begin(), a_list.begin(), a_list.end()); // copy a_list into list
```

By using iterators as abstractions for positions within a sequence, you can perform traversals and modifications to a sequence in a generic way. However, when doing insertions/deletions, you have to be conscious of the “cost” associated with performing the operation. Sometimes a list is better (lots of random updates and few linear traversals) or a vector is better (only add at the end) and require constant time random access.

Container Adapters

Suppose that we want to implement a Stack template. First we observe that the interface to a stack is a restriction of the operations on a general sequence to only allow insertions and deletions in LIFO order. So, we can easily create a Stack<T> template that is implemented in terms of a list of objects of type T. Since the list allows constant/time insertions deletion, we can insert/delete into the front of the list:

```
template<class T> class stack {
    list<T> _list;
public:
    ... // constructors/destructors

    typedef T value_type; // "publish" type parameter T

    bool empty() { return _list.empty(); }
    const value_type& top() const { return _list.front(); }
    value_type& top() { return _list.front(); }

    void push (const value_type& value) { _list.push_front(value); }
    value_type pop() { value_type x = top(); _list.pop_front(); return x; }
};
```

Note that a common technique when implementing a template is to define a typedef for the type parameter given to the template. This is very useful in practice, as it is often necessary for a user of a class to be able to determine what type parameter T was used to instantiate the class.

The pop_front method of the list class erases the front element and does not return a result, so in implementing the pop method, we first copy the top element, then pop_front, and return the element. We don't need to worry about implementing the full predicate function in this case because the list is dynamically extensible.

Question: Does it matter whether or not we insert into the front or back of the list when implementing a stack using a list? That is, could we just as easily replace all the "front" operations with "back" operations?

A Stack as a Container Adapter

The vector template classes has an interface that is very similar to the list interface . An exception is that the vector class does not provide an explicit `push_front` method, because the implementors wanted to discourage a user from trying to insert into the front of a vector, because such insertions cannot be done in constant time and require that all elements in the vector be shifted one position to the right in order to make room for a new front element.

So, to implement a stack using a vector, we need to push/pop into the back of the vector.

```
template<class T> class stack {
    vector<T> _vec;
public:
    typedef T value_type;
    bool empty() { return _vec.empty(); }
    const value_type& top() { return _vec.back(); }
    value_type& top() { return _vec.back(); }
    void push (const value_type& value) { _vec.push_back(value); }
    value_type pop() { value_type x = top(); _vec.pop_back(); return x; }
};
```

Question: What change would be necessary to implement the stack using a deque? How about having a generic Stack template that can be parameterized by either a `list<T>`, a `vector<T>`, or a `deque<T>`?

```
template<class Container> class stack {
    Container _c;
public:
    typedef Container::value_type value_type;
    ...
    void push(const value_type& value) { _c.push_back(value); }
    value_type pop() { value_type = top; _c.pop_back(); return top; }
};
```

```
stack< vector<string> > vstack; // stack implemented using a vector
stack< list<string> > lstack; // stack implemented using a list
```

Overloading + Parametric Polymorphism

Overloading becomes redundant after about the third class you implement that contains relational operators. You can combine ad hoc with parametric polymorphism to make life simpler. You should recognize that all you need are `operator==` and `operator>` implemented as member/friend functions in a class. With additional template function definitions for the other relational operators, you can save yourself a lot of unnecessary coding effort by defining the remaining relational operators in terms of the two relational operators defined as member/friend functions.

```
template<class T> bool operator != (const T& x, const T& y) { return !(x == y); }
template<class T> bool operator <= (const T& x, const T& y) { return !(y < x); }
template<class T> bool operator >= (const T& x, const T& y) { return !(x < y); }
template<class T> bool operator > (const T& x, const T& y) { return !(x <= y); }
```

It is also easy to write common template functions using the template relational operators. For example, the min/max template functions assume that `operator<` and `operator>` are defined for objects of type T:

```
template<class T> inline T& max(const T& x, const T& y) { return x > y ? x : y; }
template<class T> inline T& min(const T& x, const T& y) { return x < y ? x : y; }
```

You should always be able to determine which template functions are instantiated and called based on an examination of the contextual type information:

```
int x = 9;
int y = 10;
int z = min(x,y); // uses builtin overloaded operator < for int
```

```
String a = "hello";
String b = "world";
String c = max(a, b); // uses programmer-defined overloaded operator > for String
```

Overloading + Parametric Polymorphism

There are cases where you might need to parameterize a procedure with function type information so you can pass a pointer to a function. For example, a sort function might need to be parameterized by an array of type T and a function that can be used to compare two objects of type T.

```
template<class T> void sort(T array[], int n, bool (*lt)(const T&, const T&));
```

The second argument to sort is a **pointer to a function** that returns a bool, and takes a two objects of type T as reference arguments. In C++, pointers to functions are usually introduced using a typedef:

```
typedef bool (*FuncPtr)(const int&, const int&);
```

This typedef defines a type alias FuncPtr that is a pointer type to a procedure that takes two integer arguments and returns a boolean result. Using a typedef name is cleaner syntax which is more “human parseable”. We would then write the sort function as:

We then need to have a less than function that works for each particular type that we would want to sort. E.g.,

```
bool int_lt(const int& a, const int& b) { return a < b; }

int a[10] = { 9, 4, 1, 3, 5, 0, 2, 6, 7, 8 };
sort(a, 10, int_lt) // call sort with int array, size of array, and pointer to lt function
```

A better approach might be to have a sort procedure that took an Array<T> argument, where there is an explicitly overloaded operator< method defined for objects of type T, and two endpoints of the array to sort.

```
template<class T> void sort(Array<T>& a, int first, int last);
```

It is often useful to implement an overloaded inline procedure that just sorts the entire array.

```
template<class T> inline void sort(Array<T>& a, int n) { qsort(a, 0, n-1); }
```

Overloaded Quick Sort Template Procedure

Using an `Array<T>` template, we can write a template `qsort` procedure that sorts any array of objects of type `T`. The algorithm is independent of the type of the elements in the array. However, it is necessary for the type parameter `T` to have certain characteristics. For example, it must have an assignment operator defined, so that the swap function works correctly, and it must also have an `operator<` defined so that the comparisons can be made.

```
template<class T> void qsort(Array<T>& a, int first, int last) {
    if (first+5 > last) // insertion sort is faster for "small" arrays
        return insertion_sort(a, first, last);
    int mid = (first + last) / 2; // compute the midpoint & sort low, mid, high
    if (a[mid] < a[first])
        swap(a[first], a[mid]);
    if (a[last] < a[first])
        swap(a[first], a[last]);
    if (a[last] < a[mid])
        swap(a[mid], a[last]);

    T pivot = a[mid]; // select a pivot
    swap(a[mid], a[last-1]); // and swap with the 2nd-to-last element
    int i = first, j = last-1;
    while(true) {
        while(a[++i] < pivot) { } // scan left to right
        while(pivot < a[--j]) { } // scan right to left
        if (i < j)
            swap(a[i], a[j]);
        else
            break;
    }
    swap (a[i], a[last-1]); // restore pivot
    qsort(a, first, i-1); // recursively sort left side of pivot
    qsort(a, i+1, last); // recursively sort right side of pivot
}
```

Compiler Generated Template Functions

In order for the `qsort` template procedure to be useful, we need to have a `swap` template function defined:

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

The `swap` function requires that `T::operator=(const T&)` be defined, otherwise it will not be possible to swap two objects of type `T`.

Note that if `T` is a builtin type, like `int`, the assignment operator is predefined. For user-defined types (i.e., classes), the programmer has to make sure that assignment for user-defined class types is defined. This is why it is always a good idea to overload the assignment operator when implementing a class. That way, assignment is always a well-defined operation for a user-defined type `T`.

This is another example of why you should almost always define an overloaded assignment operator to do a deep copy instead of allowing the compiler to automatically generate a default assignment operator, which will do a shallow copy. The result of a shallow copy in C++ is very often not the right way to copy an object because of the possibility that you will end up with a dangling pointer when an object is destructed.

The `qsort` function also requires that there be a suitable overloading of `operator<` for an object of type `T`. This is why when using a type `T` in a container like `list`, `vector`, or `deque`, the compiler requires that `operator<` be defined. Otherwise, the generic algorithms like `sort` would not work, as they expect a container to contain objects of a type `T` that can be compared using the less-than operator.

When a particular instance of the `qsort` procedure is used, then the compiler will take care of generating instances of all the template procedures that are needed to implement the sorting procedure.