

## Using a Template to Define an Interface

The `IntStack` from the previous lecture is restricted to holding integer values. The behavior of a stack is independent of the type of object stored in the stack. We call this type of data structure, a *container*. Containers are the most common form of data structure. The most commonly used containers are:

1. Lists - indexed access data structure.
2. Stacks - Last-in, First-Out (LIFO) accessed data structure.
3. Queues - First-in, First-Out (FIFO) data structure.

The interface for a list is independent of the kind of thing that is stored in the list, except for information related to the type of the thing, or object, that is kept in the list. The same goes for a stack or queue.

The two key data structuring concepts in this course are:

### 1. Separation of an interface from an implementation for that interface

For example, a stack or queue can be implemented with either an array (fixed size) or a linked list (variable size). The interface is the same. The choice of an array or a linked list as the underlying implementation is a separate decision that is made depending on how the data structure is to be used.

### 2. Reuse of an interface and an implementation through type parameterization.

That is, we can define a list or stack or queue interface once, and provide a way to parameterize the type of the thing that will be held by the container data structure. We can also build the implementation part so that it is parameterized with some information as well, to give us more *flexibility*. A flexible data structure is one that allows the user to make choices about the kind of thing stored in the data structure, and even the type of the implementation that is behind the interface.

## A Template Interface for a Stack of T

```
template<class T> class Stack {
public:
    ... // as before

private:
    T    list[100];
    int  top;
};
```

To use the `Stack<T>` template, we must declare a variable of `Stack<T>`, where `T` is some already defined type: either builtin, or user-defined.

```
Stack<int>  int_stack;
Stack<String> *string_stack = new Stack<String>();
...
int_stack.push(5);
string_stack->push(String("hello, world"));
```

Note that we have “hardcoded” a fixed size array implementation of 100 things of type `T` for this stack.

Can we do better?

That is, can we provide additional parameterization so that the choice of an array implementation is not so rigidly defined?

## A Template Stack Interface

```
/* Stack.h */
template<class T> class Stack {
public:
    // constructors and destructors
    Stack ();
    Stack (T x);
    Stack (const Stack<T>& s);

    ~Stack ();

    // boolean predicates (truth functions)
    bool empty() const;
    bool full() const;

    // the stack operations
    void push (T x);
    T pop();

    // don't forget the assignment operator!
    Stack& operator=(const Stack<T>& s);

    // We might want to allow for input and output too
    friend ostream& operator>> (ostream& os, const Stack<T>& s);
    friend ostream& operator<< (ostream& os, const Stack<T>& s);

private:
    ... // we do this next
};
```

## Parameterizing Implementation Information

You can parameterize a template with a type name, or a constant value. Like type names, constants are known at compile-time, so the compiler can instantiate a new class that is defined by the class name AND all of its type parameters.

```
template<class T, int SIZE> class Stack {
public:
    ... // as before

private:
    T    list[SIZE];
    int  top;
};

Stack<int, 10> s1;
Stack<int, 10> s2;
Stack<int, 1000> s3;
Stack<String, 10> *s4 = new Stack<String, 10>();
```

Note that `s1` and `s2` are type compatible, i.e., you can assign one to the other. The compiler treats `s3` as having a different type from `s1` and `s2` because the `SIZE` const parameter is different. That's just a “feature” of C++.

As mentioned before, it is often better to use a typedef to define a type alias for a template type:

```
typedef Stack<int, 10> SmallIntStack;
typedef Stack<int, 1000> LargeIntStack;
...
```

## Parameterizing the Complete Stack Implementation

We can define the stack template parameter to be some implementation of a container. For example, a fixed size list, or a linked list. The only requirement is that each type that is given as the type parameter to the Stack interface must implement a well-defined set of operations. So, a Stack<L> is just an interface that guarantees that some *sequence container* implementation is used as a FIFO list.

```
template<class L> class Stack {
private:
    L list;

public:
    typedef List::Type T;    // nested typedef for the type T held by the List

    Stack () {}
    Stack (const Stack<L>& s) { list = s.list; }
    ~Stack ();

    bool empty() const { return list.empty(); }
    bool full() const { return list.full(); }

    void push (const T& x) { return list.insert_front(x); }
    T pop() { return list.remove_front(); }

    Stack<L>& operator=(const Stack<L>& s) { list = s.list; }
};

Stack< FixedList<int, 100> > s1;
Stack< SinglyLinkedList<int> > s2;
```

Since the Stack<L> template is just a simple inline interface as shown above, all the implementation details are provided in some List implementation. All the list implementation is required to do is implement the empty, full, insert\_front, remove\_front operations, and assignment operations.

## A FixedList Container

```
template<class T, int SIZE> class FixedList {
private:
    T list[SIZE];
    int n;

public:
    typedef T Type;    // advertise the type T parameter

    FixedList() : n(0) {}
    ~FixedList() {}

    bool empty() const { return n == 0 ? true : false; }
    bool full() const { return n == SIZE ? true : false; }

    void insert_front (const T& x) {
        if (!full())
            list[n++] = x;
        else
            overflow();
    }

    T remove_front () {
        if (!empty())
            return list[--n];
        else
            underflow();
    }

    ... // other methods, like insert_back, remove_back, operator=, etc.
};
```

## A LinkedList Container

```
template<class T> class SinglyLinkedList {
    struct Node {
        T elem;
        Node* next;

        Node() : next(NULL) {}
        Node(const T& x, Node* n = NULL) : elem(x), next(n) {}
    };

    Node* head;    // pointer to front of the list

public:
    typedef T Type;

    SinglyLinkedList() { head = NULL; }
    ~SinglyLinkedList();

    bool empty() const { return head == NULL ? true : false; }
    bool full() const { return false; }

    void insert_front (const T& x) { head = new Node(x, head); }

    T remove_front () {
        if (!empty()) {
            Node* n = head;
            T tmp = n->elem;
            head = head->next;
            delete n;
            return tmp;
        }
        else
            underflow();
    }

    ... // other methods, like insert_back, remove_back, operator=, etc.
};
```

## Things to Think About

1. How would you implement insert\_back and remove\_back operations for both the FixedList and the LinkedList template classes? You can change the private data part to have additional instance variables if you need them (e.g., front and back markers).
2. Read Chapter 4 - Inheritance for this week. This chapter will conclude our introduction to C++. Homework 4.1-4.10 due in class on Tuesday, July 2.
3. Programming assignment #2 due on Thursday.
4. Mid-term Exam on Tuesday, July 2 during class.

The exam will be on the C++ material in Chapters 1-4, and the lecture notes.

I will be away next week, and the TA will administer the exam.

Thursday will be a review of Chapters 1-4. Bring your questions!

You can send me any questions you have via email and I will respond while I am away