C++ Templates
The StringStack class only knows how to store strings

What if we need stacks of ints, floats, Elephants, ...

We could write stack classes for each different type
  - Results in lots of classes and maintenance effort

C++ has a feature called "templates" that allows us to write a class blueprint, and then create lots of real classes based on that blueprint

Different terms for the same idea:
  - Templates
  - Generic types (or just “generics”)
  - Parameterized types
template <typename T>
class Stack {
    ...
};

Stack<string> strStack(5);
strStack.Push("one");
cout << strStack.Pop() << endl;

Stack<int> intStack(5);
intStack.Push(1);
cout << intStack.Pop() << endl;

Stack<Elephant> eleStack(5);
eleStack.Push(Elephant(6000));
cout << eleStack.Pop() << endl;
Converting StringStack Into a Stack Template

- Add the line "template <typename T>" before the class declaration
- Change the name of the class to Stack
- Everywhere in the code that we used the type "string", change it to "T"
- This results in a class template that can be used to create a stack class for any type
template <typename T>
class Stack {

protected:
    int capacity;
    T * stack;
    int top;

};
template <typename T>
class Stack {

protected:
    int capacity;
    T * stack;
    int top;

public:
    Stack(int initialCapacity = 0) {
        capacity = initialCapacity;
        stack = new T[capacity];
        top = 0;
    }
};
template <typename T>
class Stack {

protected:
    int capacity;
    T * stack;
    int top;

public:
    Stack(const Stack<T> & other) {
        Init(other);
    }

protected:
    void Init(const Stack<T> & other) {
        capacity = other.capacity;
        top = other.top;
        stack = new T[capacity];
        for (int i=0; i < top; ++i) {
            stack[i] = other.stack[i];
        }
    }

};
template<typename T>
class Stack {

protected:
    int capacity;
    T * stack;
    int top;

public:
    ~Stack() {
        Free();
    }

protected:
    void Free() {
        delete [] stack;
        stack = 0;
    }

};
template <typename T>
class Stack {

protected:
    int capacity;
    T * stack;
    int top;

public:
    void operator =(const Stack<T> & other) {
        if (&other != this) {
            Free();
            Init(other);
        }
    }

};
template<typename T>
class Stack {

protected:
    int capacity;
    T * stack;
    int top;

public:
    bool IsEmpty() const {
        return (top == 0);
    }

    void Clear() {
        while (top > 0) {
            stack[--top] = T();
        }
    }

};
template <typename T>
class Stack {
protected:
    int capacity;
    T * stack;
    int top;
public:
    void Push(const T & value) {
        if (top == capacity) {
            Grow();
        }
        stack[top++] = value;
    }
protected:
    void Grow() {
        int newCapacity = capacity * 2;
        T * newStack = new T[newCapacity];
        for (int i=0; i < top; ++i) {
            newStack[i] = stack[i];
        }
        Free();
        capacity = newCapacity;
        stack = newStack;
    }
};
template <typename T>
class Stack {

protected:
    int capacity;
    T * stack;
    int top;

public:
    T Pop() {
        if (IsEmpty()) {
            throw CS240Exception("can't pop an empty stack");
        } else {
            T value = stack[--top];
            stack[top] = T();
            return value;
        }
    }
};
Templates

- Now, every time we declare a Stack of a different type, the compiler will automatically generate and compile a stack class for that type

```cpp
template <typename T>
class Stack {
    ...

};

Stack<string> strStack(5);
strStack.Push("one");
cout << strStack.Pop() << endl;

Stack<int> intStack(5);
intStack.Push(1);
cout << intStack.Pop() << endl;

Stack<Elephant> eleStack(5);
eleStack.Push(Elephant(6000));
cout << eleStack.Pop() << endl;
```