Namespaces

• A simple mechanism for organizing large programs
• Recall the idea of a scope...
• Useful idea: allow named scopes with no other functionality - these are called *namespaces*.
• Good for avoiding collisions and making connections more explicit.

Namespace syntax

```cpp
nenamespace mylib {
    class IntStack {
    {
        ...
    };

    class Rational {
    {
        ...
    };

    } // Note no semicolon on closing

    mylib::Rational r1;
```
Namespaces can be re-opened

// IntStack.h
namespace mylib {
    class IntStack {
        ...;
    }
}

// Rational.h
namespace mylib {
    class Rational {
        ...;
    }
}

Namespace usage

• Classes and functions must be declared in a namespace to be placed inside the namespace
• Definitions can use the :: qualifier or be placed inside.
• using declarations provide local aliases for qualified names (this “imports” symbols):

  using mylib::IntStack;

  IntStack s1;
Namespace usage

• Can also import everything at once:

  using namespace mylib;

• Namespaces may have using declarations:

  namespace mylib {
    using herlib::IntStack;
    ...
  }

The std namespace

• All the standard “stuff” is in the std namespace.
• E.g. cout is really std::cout.
• iostream.h ends with “using namespace std;
• Which means that including it gets you everything without qualification.
• Doing #include<stdiostream> doesn’t, so you don’t.
• Keep older programs working, but allow migration to new organization.
• Other than I/O, what “stuff” is there?
• Lots, but all organized using templates!
Templates

• ...Or, how to get the compiler to write code for you.
• Recall the IntStack class.
• How do we produce DoubleStack?
• Global search/replace int with double?
• Build a reference copy with a dummy type marker, then use that to construct new implementations.
• Using templates, we’ll do the first part, the compiler will do the second part.

Template client syntax

```cpp
Stack<int> si1;
Stack<double> sd1;

// be careful about spacing!!
Stack< Stack < Rational > > ssr1;

• After that we’re home free!
  si1.push( 10 );
  sd1.push( 3.14 );
  sd1.push( si1.pop() );
  Rational rl(4,5);
  si1.push( rl ); // type safety??
```
Template syntax for classes

• Make this class declaration a template:

```cpp
template < class T >
class Stack {
....
};
```

• T is the template parameter, standing in for the type
• When the client declares `Stack<int>...`
• Template is *instantiated* for ints.

Template syntax for classes, cont.

• Places to use the template parameter:

```cpp
template < class T >
class Stack {
public:
  // Constructor name
  Stack<T>(int initialSize);
  // Arguments of the class type
  Stack<T>(const Stack<T>& other);
  // Other function arguments
  void push(const T& item);
private:
  // Types of members
  T* m_stack;
};
```
Template internals

• Templates need to be visible where they are used, so it’s usual to put them entirely in the header file.
• Compilers are required to instantiate only functions that will actually be used!!
• So good test driver coverage is really crucial.
• Clients of templates won’t want to know or care that they are using a template – they shouldn’t need to.
• A useful question for container templates: can this container be used on itself?

Template internals, cont.

• A template makes demands on the type the client wants to use.
• If they aren’t satisfied, it’s an error.
• So in a way, a template is a client of the class T, without knowing what T is yet.
• Native types allow lots of operations, so they should almost always be considered first.
• Templates are another reason to have operators…
• …and to have them behave like native types do.
A useful trick

• Sometimes, you want to construct an object of type T with “default” behavior:

```cpp
template < class T >
T Stack<T>::pop()
{
    if( is_empty() )
    {
        cerr << "Pop failed, stack is empty" << endl;
        return T();
    }
    return m_stack[--m_top];
}
```

• Happily, this works “right” when T is a native type!

A useful trick, cont.

```cpp
int x = int(); // x == 0
double d = double(); // d == 0.0
bool b = bool(); // b == false
int y; // y == ???
char* cp = char*(); // syntax error, but…

template class<T>
...
T thing = T(); // will be 0 if T is char*
```
Function templates

• Since we can build templates for classes, why not templates for functions?

    // return bigger int
    int max_int(int a, int b)
    {
        return (a>b)?a:b;
    }

    // return bigger T
    template<class T>
    const T& max(const T& a, const T& b)
    {
        return (a>b)?a:b;
    }

    string s1, s2;
    ...
    string bigger = max(s1,s2);

    // return bigger T
    template<class T>
    const T& max(const T& a, const T& b)
    {
        return (a>b)?a:b;
    }
Templates and friend

• The old way:

```cpp
template < class T >
class Stack {
    // This friend declaration is assumed to be a
    // template of T function.
    friend
    operator<< (ostream& os, const Stack<T>& s);
    ...};
```

• The new way:

```cpp
// Forward declare class template
template < class T > class Vect;

// Forward declare function template
template <class T> ostream& operator<<( ostream& os,
    const Vect<T>& v );

template < class T >
class Stack {
    // This friend declaration is a template of T
    // function because of the extra <> after the
    // name.
    friend
    operator<< <>(ostream& os, const Stack<T>& s)
};
```
Template philosophy

• Implement a concrete version first.
• Consider using something other than the type you are most interested in.
• When you convert to a template, be sure to test using several different classes.
• As always, try first to be a client!
Object Oriented Programming in C++
Assignment 3 -- Due March 26
In this assignment we will create a simple vector template. Copies of the (possibly relevant) Stack code are available on the course website. The driver for this assignment is available on the website.

Part 1.

1. Implement the following template. The public interface is shown, the private parts you should do as you think best. The operation of the push functions should automatically extend the vector by doubling if necessary. Once items are pushed, the only way for the vector to get smaller is by clearing the entire thing with clear(). Your stream operator should print the size of the vector and also show the entire vector contents (format as you think best).

```
template < class T > class Vector;

template class<T>
ostream& operator<<( ostream& os, const Vector<T>& v );

template < class T >
class Vector
{
friend ostream& operator<<( ostream& os, const Vector<T>& v );
public:
    Vector<T>();
    ~Vector<T>();
    size_t size() const;
    void push(T n);
    void push(T a[], size_t array_size);
    void clear();
};
```

2. Add member functions
   a) const T& operator[] (unsigned int index) const
   b) T& operator[] (unsigned int index)
   c) T sum() const;
   The first two functions should return the item at the given index or throw a std::out_of_range exception if the index is out of bounds. The second function should add up all the elements in the vector and return the result. Note that calls to this function on certain types may cause compiler errors [which ones?] Also note that you may want to use the default initialization trick here.

3. Add a member function size_t memUsage() const which returns the amount of memory used by your vector.

Part 2.

4. Add a member function void setAt(unsigned int index, const T& item) which stores item at the given index if it is in bounds or throws a std::out_of_range exception if the index is out of bounds

5. Add a member function void setAtGrow( unsigned int index, const T& item) which stores item at the given index and resizes the vector (again by doubling) if necessary so that index is in bounds. [Why does this seem like a not-all-that-useful functionality?]

6. Add a member function void minimize() that reduces the memory used by your object to the minimum possible without changing the contents.

Hand in your final Vector source code. You do not need to hand in the driver.