More on Templates

- Templates for functions
- Other template arguments
- Specialization
- The basic_string template in STL
- Boost

Template arguments

- Templates can have more than one argument.
- Templates can have different kinds of arguments.
- Non-class arguments must be supplied with constants

```cpp
template<class T, int i>
class FixedSizeBuffer
{
    T m_buffer[i];
    int sz;
    FixedSizeBuffer() : sz(i) {}
};

FixedSizeBuffer<char, 80> line; // T is char, i is 80
FixedSizeBuffer<string, 10> sb; // T is string, i is 10
```
Default template arguments

- Templates arguments can default.
- As with function arguments, this is like overloading.

```cpp
template < class t1, class t2 = double >
class Point {
  public:
    Point<t1, t2> (t1 x, t1 y) : m_x(x), m_y(y) {} 
    t2 distance() 
    { 
      return (t2) sqrt(m_x*m_x + m_y*m_y); 
    }
  private:
    t1 m_x, m_y;
};
```

Specialization

- Sometimes, you want certain types treated specially.
- Usually, this is for efficiency.
- As with function names, this is like overloading.

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

// Specialize for char
template < class T>
class Stack<char> {
  ...
};
```
Specialization, cont.

• It’s also possible to treat entire groups of types separately:
• For example, you can treat all pointers differently from non-pointers:

```cpp
template <class T> // Most general version must be last!
class Stack<T> {
  ...
};

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

The Standard Template Library

• Initially written in the early 90s by Alexander Stepanov (and a couple others)
• Adopted as part of the ISO standard in 1994
• Now implemented correctly by all major compilers
• Based on three concepts: containers, iterators, and algorithms
• An good example of a form of programming known as “generic”
• The basic_string template is where we start…
The basic_string template

- basic_string owns its own memory
- Uses “position and count” system
- STL has typedef basic_string<char> string
- Construction:

```c++
#include <string>
using std::string;
string s1; // empty
string s2 = s1; // copy c’tor, of course
string s3 = "Hello" // from null terminated char*
string s4("Hello", 4) // first 4 chars from pointer
string s5(s3, 2, 5); // "llo"
string s6(6, ‘a’); // "aaaaaa"
```

“position and count” system

- Positions are 0-based
- string::npos is the “non-existent” position.
- string::npos is really -1.
- Counts count chars, no null terminator.
- As a count, npos means all or all the rest.
Assignment

```cpp
#include <string>
using std::string;

string s1;            // empty

s1 = "Fred";          // From null terminated char*
s1 = 'a';             // Becomes "a"
s1 = s2;              // Of course

// Also: (all like corresponding c'tors)
s1.assign( "Hello", 4);
s1.assign( 6, 'a');
s1.assign( s2, 2, 5);
```

Append

```cpp
#include <string>
using std::string;

string s1;            // empty

s1 += "Fred";         // From null terminated char*
s1 += 'a';            // Becomes "Freda"
s1 += s2;             // Of course

// Also: (all like corresponding c'tors)
s1.append( "Hello", 4);
s1.append( 6, 'a');
s1.append( s2, 2, 5);
```
Character access

```cpp
#include <string>
using std::string;
string s1 = "Hello";

char ch = s1[4];  // uses const version
s1[0] = 'J';      // uses non-const version
char ch2 = s1[5]; // Error!

// Corresponding named function at(size_type pos)
```

The dual operator[] idiom

- It’s common to define two operator[]s
- Allows use on left hand side, and on const objects

```cpp
template <class T>
class MyArray
{
    T& operator[](unsigned int i);
    T operator[](unsigned int i) const;
    ...
};

void myFunc( const MyArray<int>& src ) {
    MyArray<int> dest;
    // Use non-const op[] on left, const op[] on right
    dest[0] = src[1];
```
Information

#include <string>
using std::string;
string s1 = "Hello";

int len = s1.length();  // will be 5
len = s1.size();        // same as length
if ( s1.empty() )       // it isn’t, so false
Conversion to char*

#include <string>
using std::string;
string s1 = "Hello";

fopen( s1.c_str(), "r" );  // get null-terminated char*

// Note! The pointer returned by c_str is const, and
// can only be relied on until the next non-const member
// is called on s1!
char* backdoor = s1.c_str();  // illegal
const char* safeptr = s1.c_str(); // OK
s1.append("foo");            // invalidates safeptr
Search

```cpp
#include <string>
using std::string;
string s1 = "Hello world";

// find a substring (from front or back)
int pos = s1.find("worl"); // returns 6
pos = s1.find("foo"); // returns string::npos
pos = s1.find('o'); // returns 4
pos = s1.rfind('o'); // returns 7

// find any character (from front or back)
pos = s1.find_first_of( "aeiou" ); // returns 1;
pos = s1.find_first_not_of( "aeiou" ); // returns 0;
```

Comparison

```cpp
#include <string>
using std::string;
string s1 = "Hello world";

int res = s1.compare("Fred"); // returns positive (H>F)

// compare "world" with "Fred" w > F
res = s1.compare( 6, string::npos, "Fred" )
```
Non-members

- `operator+` taking 2 args, returning concatenation
- overloaded for 5 combinations
- `operator>` etc. comparing 2 strings or `char*`
- overloaded for 4 combinations

Inside the basic_string template

- A template with 3 template arguments
- The second two have defaults (whew!)
- `char_traits` allows you to specify how to treat `E` as a string component
- `allocator` allows you to handle allocation yourself

```cpp
template<class E, 
class T = char_traits<E>, 
class A = allocator<T> > 
class basic_string {
    ...
};
```
Boost libraries

• The Boost libraries provide useful additions to the STL.
• See boost.org for lots of information
• “TR1” includes a collection of components from the Boost libraries which have been accepted for the next C++ standard (C++0X)
• Both g++ (> 4.3.0) and MSVC++ (with a free add-on to > VC2008) have TR1 support.

TR1 library additions

• Reference wrappers
• Smart pointers
  • Resource Acquisition Is Initialization (RAII)
• Function objects:
  • bind generalizes bind1st
• Regexps
• Containers:
  • Hash tables
  • Tuples
  • Fixed-size arrays
• Metaprogramming
• Math and numerics:
  • Random numbers
  • Special functions
Boost regex library

- Supports a number of RegExp syntaxes (default is Perl’s)
- `#include<regex>` (or maybe `<boost/regex>`)  
- Find things in std::tr1 namespace (or maybe boost::)

```cpp
bool validate_card_format(const std::string s)
{
    // Four 4-digit groups separated by space or hyphen
    static const boost::regex e("((\d{4}[-\ ]){3}\d{4})");
    return regex_match(s, e);
}
```

Advice

- Try `std::string`, you’ll like it.
- Use `c_str` carefully to interact with older code.
- Don’t sweat all the details, just allow defaults.
- Be careful using `[]`, try to use `find`, etc. instead.