Review of Maps

- Association between types, the first type being the key type.
- Maps of vectors
- \texttt{operator[]} 
- \texttt{find} is a member function of the map class.
- \texttt{erase}

Today’s Class — Templates, Generic Functions, and Iterators

Koenig and Moo: Sections 8.1 and 8.2

- Motivating example
- Templated functions
- Templated functions involving iterators
- Requirements on types
- Different kinds of iterators
- More examples
- Passing functions as arguments
Example

- Here is a function to find the sum of the values in a vector of integers

```cpp
int Sum( const vector<int>& a )
{
    int sum = 0;
    for ( unsigned int i=0; i<a.size(); ++i )
        sum += a[i];
    return sum;
}
```

- Here is a function to find the sum of the values in a vector of doubles:

```cpp
double Sum( const vector<double>& a )
{
    double sum = 0.0;
    for ( unsigned int i=0; i<a.size(); ++i )
        sum += a[i];
    return sum;
}
```

- Here is a function to find the sum of the values in a list of doubles:

```cpp
double Sum( const list<double>& a )
{
    assert( a.size() > 0 );
    double sum = 0.0;
    for ( list<double>::const_iterator p = a.begin();
         p != a.end(); ++ p )
        sum += *p;
    return sum;
}
```

- Do we really need three functions?
- Think about the changes between functions.
- Can't we get the compiler to do this for us?
- The answer is yes — using templates. We will combine the first two functions in our first try, and all three in our second.
Tempating the Type Stored in the Vector

• A templated function is defined in terms of one or more templated types. Different specifications of these types causes the compiler to create different instances of the function.

• Here's an example definition of the two different vector Sum functions combined into a single templated function:

```cpp
template <class T>
T Sum( const vector<T>& a )
{
    T sum = 0;
    for ( unsigned int i=0; i<a.size(); ++i )
        sum += a[i];
    return sum;
}
```

• The line

```cpp
template <class T>
```

says that what follows is a templated function, using T to take the place of the class that will be named later. (By class, we mean structs as well as int, double. We will see “full-blown” classes next week.)

• Everywhere we reference the type stored in the vector (as an int or as a double), we have replaced this with T.
Here’s a demonstration main program that uses the templated function.

```cpp
int main()
{
    vector<int> a;
    vector<double> b;

    a.push_back(5); a.push_back(10); a.push_back(2);
    b.push_back(1.1); b.push_back(2.2); b.push_back(3.3);

    cout << Sum(a) << endl << Sum(b) << endl;
    return 0;
}
```

Up until the compiler reaches the line

```cpp
cout << Sum(a) << endl << Sum(b) << endl;
```

no functions called `Sum` have been created. (In fact, if we commented out this line and created a syntax error in `Sum`, it wouldn’t be caught!)

- When the compiler sees `Sum(a)`, it looks for a `Sum` function on a vector of ints. It sees that it can create it by substituting `int` for `T` in the templated function. It does so, generating an instance of the `Sum` function.

- When the compiler sees `Sum(b)`, it looks for a `Sum` function on a vector of doubles. It creates one from the templated function.

- We now have two templated functions that look almost exactly like our earlier functions.
  - If you create a single syntax error in `Sum`, some compilers will complain about it twice!
Templating Based on Iterators

- Here’s the sum version based on iterators:

```cpp
template <class Iter, class T>
T Sum( Iter beg, Iter end, T init )
{
    T sum = init;
    for ( Iter p = beg; p != end; ++p )
        sum += *p;
    return sum;
}
```

- Two templated types are now used, one for the iterator type, and one for the value type.

- The initial value is used for two reasons:
  - It gives a slight increase in generality.
  - Compilers can’t figure out the type from the iterators. We will see examples of this.

- Main program:

```cpp
int main()
{
    vector<int> a;
    vector<double> b;
    list<double> c;

    a.push_back( 5 ); a.push_back( 10 ); a.push_back( 2 );
    b.push_back( 1.1 ); b.push_back( 2.2 ); b.push_back( 3.3 );
    c.push_back( 4.5 ); c.push_back( 2.2 ); c.push_back( 7.5 );

    cout << Sum(a.begin(), a.end(), 0) << endl
         << Sum(b.begin(), b.end(), 0.0) << endl
         << Sum(c.begin(), c.end(), 0.0) << endl;

    return 0;
}
```
• This main program causes three different versions of the `Sum` function to be created. We will see what happens when we change the value 0 to 0.0 and vice-versa in the calls to `Sum`.

Why Iterators and Not Containers
• Iterators are what’s common between containers
• Use of iterators allows starting and ending anywhere in the container.

Different Types of Iterators
• Five different types of iterators: input, output, forward, bidirectional, random access.
• We will not worry too much any but the last two.
• List iterators are bidirectional
• Vector iterators and string iterators are random access
• The different iterator types have different capabilities.
• The sum function doesn’t need all of the capabilities of these iterators.

Requirements on Types
• Templated functions work when different object types have operators in common.
• The type `T` used in the `Sum` function requires assignment (`operator=`) and `operator+=`.
  – When one of these operators is missing a compiler error occurs.
• What types must `iter` have?

Two Other Examples
We will write the following in class:
• `equal`
• `remove`
Passing Functions as Arguments

- Many generic functions require passing a function as an argument.
- There is really nothing complicated about this. It just requires specifying the return type and argument types as part of the function parameters. Here’s an example:

```cpp
template <class iter, class T>
iter my_find_if( iter beg, iter end, bool tst( T ) )
{
    for( iter p = beg; p != end; ++p )
        if ( tst( *p ) )
            return p;
    return end;
}
```

- Any function passed as the third argument to this function must have a single value parameter of type T and return a bool.
- This generic function `my_find_if` works exactly like `find_if`.

Discussion of Academic Integrity

- There were several incidents of academic dishonesty on Week 3 and Week 4 homework.
- Reminders:
  - Don’t work together closely on the details of your code.
  - Don’t show anyone your homework! You are not helping them and could be hurting yourself.

Review the syllabus if you are at all in doubt!

- Dishonesty is easy to catch!

- If you want to work together, work on general design, and work on practice problems investigating the C++ and library techniques you need to use.

- If you need a lot of help on your program, seek it from the TAs or from the instructor.