CSE 333
Lecture 12 - templates, STL

Hal Perkins
Department of Computer Science & Engineering
University of Washington
Administrivia

HW2 due Thursday night, 11 pm.

<panic>if not started yet</panic>

<celebrate>if done</celebrate>

**BE SURE TO CHECK YOUR WORK** (git clone/checkout, make, tests, …)

No new exercise for Friday
Administrivia (Friday)

New exercise out today, due Monday morning

Exam next Friday, in class

Closed book, no notes — exam questions can be more straightforward that way; reference info on test as needed

Topics: everything from lectures, exercises, project, etc. up to HW2 & basics of C++ (including references, const, classes, constructors, destructors, new/delete, nothing after that)

Old exams and topic list on the web now

Review in sections next week
Today’s goals

Templates and type-independent code

C++’s standard library

   STL containers, iterators, algorithms

   A few core ones only - see docs & Primer for others
Suppose that...

You want to write a function to compare two ints:

```cpp
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int &value1, const int &value2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```
Suppose that...

You want to write a function to compare two ints, and you also want to write a function to compare two strings:

```cpp
// note the cool use of function overloading!

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int &value1, const int &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string &value1, const string &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```
Hmm....

The two implementations of compare are nearly identical.

- we could write a compare for every comparable type
  but, that’s obviously a waste; lots of redundant code!

Instead, we’d like to write “generic code”

- code that is **type-independent**
- code that is **compile-time polymorphic** across types
C++: parametric polymorphism

C++ has the notion of **templates**

- a function or class that accepts a **type** as a parameter
  - you implement the function or class once, in a type-agnostic way
  - when you invoke the function or instantiate the class, you specify (one or more) types, or values, as arguments to it

at **compile-time**, when C++ notices you using a template...

- the compiler generates specialized code using the types you provided as parameters to the template
Function template

You want to write a function to compare two things:

```cpp
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <class T>
int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare<std::string>(h, w) << std::endl;
    std::cout << compare<int>(10, 20) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    return 0;
}
```
Function template

Same thing, but letting the compiler infer the types:

```cpp
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <class T>
int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare(10, 20) << std::endl;
    std::cout << compare("Hello", "World") << std::endl;  // hmmm...
    std::cout << compare(h, w) << std::endl;  // ok
    return 0;
}
```
Function template

You can use non-types (constant values) in a template:

```cpp
#include <iostream>
#include <string>

template <class T, int N>
void printmultiple(const T &value1) {
    for (int i = 0; i < N; ++i)
        std::cout << value1 << std::endl;
}

int main(int argc, char **argv) {
    std::string h("hello");
    printmultiple<std::string,3>(h);
    printmultiple<const char *,4>("hi");
    printmultiple<int,5>(10);
    return 0;
}
```
What’s going on underneath?

The compiler doesn’t generate any code when it sees the templated function

it doesn’t know what code to generate yet, since it doesn’t know what types are involved

When the compiler sees the function being used, then it understands what types are involved

it generates the instantiation of the template and compiles it

the compiler generates template instantiations for each type used as a template parameter

kind of like macro expansion
This creates a problem...

```cpp
#ifndef _COMPARE_H_
define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b);
#endif // COMPARE_H_

#include "compare.h"

#ifndef _COMPARE_H_
define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
#endif // COMPARE_H_

#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

CSE333 lec 12 C++.4 // 04-26-17 // Perkins
#ifndef _COMPARE_H_
#define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
#endif  // _COMPARE_H_  compare.h

#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20); 
    cout << endl;
    return 0;
}

main.cc
Another solution

```cpp
#ifndef _COMPARE_H_
define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b);
#include "compare.cc"
#endif // COMPARE_H_

#include <iostream>
#include "compare.h"
using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

```cpp
#include <iostream>
#include "compare.h"
using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

```cpp
template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```
Class templates

 Templating is useful for classes as well! Imagine we want a class that holds a pair of things

 we want to be able to:

 set the value of the first thing, second thing

 get the value of the first thing, second thing

 reverse the order of the things

 print the pair of things
Pair class

```cpp
#include <iostream>
#include <string>

template <class Thing> class Pair {
    public:
        Pair() { }
        Thing &get_first() { return first_; }
        Thing &get_second();
        void set_first(Thing &copyme);
        void set_second(Thing &copyme);
        void Reverse();

    private:
        Thing first_, second_;
    }

#include "Pair.cc"
```
class Pair

```cpp
template <class Thing> Thing &Pair<Thing>::get_second() {
    return second_
}

template <class Thing> void Pair<Thing>::set_first(Thing &copyme) {
    first_ = copyme;
}

template <class Thing> void Pair<Thing>::set_second(Thing &copyme) {
    second_ = copyme;
}

template <class Thing> void Pair<Thing>::Reverse() {
    // makes *3* copies
    Thing tmp = first_
    first_ = second_
    second_ = tmp;
}
```
Pair class

```cpp
#include <iostream>
#include <string>
#include "Pair.h"

int main(int argc, char **argv) {
    Pair<std::string> ps;
    std::string x("foo"), y("bar");

    ps.set_first(x);
    ps.set_second(y);
    ps.Reverse();
    std::cout << ps.get_first() << std::endl;

    return 0;
}
```

C++’s standard library

Consists of four major pieces:

the entire C standard library

C++’s input/output stream library

std::cin, std::cout, stringstreams, fstreams, etc.

C++’s standard template library (STL)

containers, iterators, algorithms (sort, find, etc.), numerics

C++’s miscellaneous library

strings, exceptions, memory allocation, localization
STL :)  

Containers!

a container is an object that stores (in memory) a collection of other objects (elements)

    implemented as class templates, so hugely flexible

several different classes of container

    sequence containers (vector, deque, list)

    associative containers (set, map, multiset, multimap, bitset)

differ in algorithmic cost, supported operations
STL :( 

STL containers store by value, not by reference 

when you insert an object, the container makes a copy 

if the container needs to rearrange objects, it makes copies 

  e.g., if you sort a vector, it will make many many copies 
  e.g., if you insert into a map, that may trigger several copies 

what if you don’t want this (disabled copy ctr, or copy is $$)? 

you can insert a wrapper object with a pointer to the object 

we’ll learn about these “smart pointers” later
STL vector

A generic, dynamically resizable array

- elements are stored in contiguous memory locations
- elements can be accessed using pointer arithmetic if you like
- random access is $O(1)$ time
- adding / removing from the end is cheap (constant time)
- inserting / deleting from middle or start is expensive ($O(n)$)

Example

see Tracer.cc, Tracer.h, vectorfun.cc
STL iterator

Each container class has an associated iterator class used to iterate through elements of the container (duh!)

some container iterators support more operations than others

- all can be incremented (++ operator), copied, copy-cons’ed
- some can be dereferenced on RHS (e.g., x = *it;)
- some can be dereferenced on LHS (e.g., *it = x;)
- some can be decremented (-- operator)
- some support random access ([ ], +, -, +=, -=, <, > operators)

http://www.cplusplus.com/reference/std/iterator/
Example

see vectoriterator.cc
Type inference [C++11]

the ‘auto’ keyword can be used to infer types

simplifies your life if, for example, functions return complicated types

the expression using auto must contain explicit initialization for it to work

```cpp
// Calculate and return a vector containing all factors of n
std::vector<int> Factors(int n);

void foo(void) {
    // Manually identified type
    std::vector<int> facts1 = Factors(324234);

    // Inferred type
    auto facts2 = Factors(12321);

    // Compiler error here
    auto facts3;
}
```
Type inference [C++11]

Auto and iterators

life becomes much simpler!

```
for (vector<Tracer>::iterator it = vec.begin(); it < vec.end(); it++) {
    cout << *it << endl;
}
```

```
for (auto it = vec.begin(); it < vec.end(); it++) {
    cout << *it << endl;
}
```
Range “for” statements [C++11]

Syntactic sugar that emulates Java’s “foreach”

works with any sequence-y type

strings, initializer lists, arrays with an explicit length defined, STL containers that support iterators

```cpp
// Prints out a string, one character per line
std::string str("hello");
for (auto c : str) {
    std::cout << c << std::endl;
}
```
Combining auto with range for

see vectoriterator_2011.cc
STL algorithms

A set of functions to be used on ranges of elements

- **range**: any sequence that can be accessed through iterators or pointers, like arrays or some of the containers

- algorithms operate directly on values using assignment or copy constructors, rather than modifying container structure

- some do not modify elements
  - find, count, for_each, min_element, binary_search, etc.

- some do modify elements
  - sort, transform, copy, swap, etc.

http://www.cplusplus.com/reference/algorithm/
Example

see vectoralgos.cc
STL list

A generic doubly-linked list

- elements are *not* stored in contiguous memory locations
  - does not support random access (cannot do list[5])
- some operations are much more efficient than vectors
  - constant time insertion, deletion anywhere in list
  - can iterate forward or backwards
- has a built-in sort member function
  - no copies; manipulates list structure instead of element values
Example

see listexample.cc
STL map

A key/value table, implemented as a tree

- elements stored in sorted order
  - key value must support less-than operator
- keys must be unique
  - multimap allows duplicate keys
- efficient lookup \(O(\log n)\) and insertion \(O(\log n)\)

http://www.cplusplus.com/reference/stl/map/
Example

see mapexample.cc
New in C++ 11

unordered_map, unordered_set

and related classes: unordered_multimap, unordered_multiset

average case for key access is $O(1)$

But range iterators can be less efficient than ordered map/set

See C++ Primer, online references for details
Exercise 1

Take one of the books from HW2’s test_tree, and:

- read in the book, split it into words (you can use your HW2)
- for each word, insert the word into an STL map
  - the key is the word, the value is an integer
  - the value should keep track of how many times you’ve seen the word, so each time you encounter the word, increment its map element
  - thus, build a histogram of word count
- print out the histogram in order, sorted by word count
- bonus: plot the histogram on a log/log scale (use excel, gnuplot, ...)
  - xaxis: log(word number), y-axis: log(word count)
Exercise 2

Using the Tracer.cc/.h file from lecture:

construct a vector of lists of Tracers

i.e., a vector container, each element is a list of Tracers

observe how many copies happen. :) 

use the “sort” algorithm to sort the vector

use the “list.sort( )” function to sort each list
See you on Monday!