STL Continued, Templates

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat
Jialin Cai
Mati Davis
Donglun He
Chandravaran Kunjeti
Heyi Liu
Shufan Liu
Eddy Yang

Poll Everywhere

pollev.com/tqm

- How do I declare an constructed, but empty, `vector` of `int`s in c++?

Logistics

- Check-in 06 Due Friday 3/17 @ 10 am
  - Releases later tonight
- Mid Semester Survey Due Monday 3/20 @ 11:59 pm
  - Graded, but all submissions get 100% credit
- Project details & HW3 to be released soon-ish
  - Project to be done in pairs
- Grades
  - Midterm grading is being worked on
  - HW2 grades are in progress
    still missing submissions from about 1/4th of the class

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
  - General form: `algorithm(begin, end, ...);`

Rest depends on the algo

Takes a range of a sequence to operate on

- Algorithms operate directly on *range elements* rather than the containers they live in
  - Make use of elements' copy ctor, =, ==, !=, <
  - Some do not modify elements
    - *e.g.* **`find`**, **`count`**, **`for_each`**, **`min_element`**, **`binary_search`**
  - Some do modify elements
    - *e.g.* **`sort`**, **`transform`**, **`copy`**, **`swap`**

Appropriate operator(s) must be defined for the element to use an STL algorithm

Algorithms Example (pt.1)

vector_algos.cc

```
#include <vector>
#include <algorithm>

using namespace std;

void print_out(const int& n) {
  cout << " print_out: " << n << endl;
}

int main(int argc, char** argv) {
  vector<int> vec = {73, 12, 22};

  cout << "sorting..." << endl;
  sort(vec.begin(), vec.end());
  cout << "done sort!" << endl;

  for_each(vec.begin(), vec.end(), &print_out);

  // continued on next slide

  return EXIT_SUCCESS;
}
```

Not in order ☹

Sort elements from [begin, end)

Runs function on each element. In this case, prints out each element

Algorithms Example (pt.2)

vector_algos.cc

```
int main(int argc, char** argv) {
  vector<int> vec = {73, 12, 22};

  // previous slide content cut out

  vec.push_back(73);

  int num = count(vec.begin(), vec.end(), 73);
  cout << "73 occurs " << num << " times in vec" << endl;

  auto it = find(vec.begin(), vec.end(), 5950);

  if (it == vec.end()) {
    cout << "5950 not found in vec" << endl;
  } else {
    cout << "5950 found in vec" << endl;
  }

  return EXIT_SUCCESS;
}
```

Counts all occurrences of the target in `[begin, end)`

Searches for first instance of specified value.

Returns end if not found

Could use the resulting it to do something like: `vec.erase(it);`

Algorithm Documentation:

- Very useful, more that I didn't cover in lecture:
- cplusplus.com: https://cplusplus.com/reference/algorithm/
- cppreference.com: https://en.cppreference.com/w/cpp/algorithm
- Even more useful if you know lambda expressions, but won't cover that in class

Lecture Outline

- C++ algorithm
- **C++ STL Cont**
  - **Map**
  - **Set**
- Templates

STL **map**

- One of C++'s *associative* containers: a key/value table, implemented as a search tree
  - http://www.cplusplus.com/reference/map/
  - General form: map<*key_type*, *value_type*> name;

    Independent types
  - Keys must be *unique*
    - **multimap** allows duplicate keys
  - Efficient lookup (O(log n)) and insertion (O(log n))
    - Access value via **operator[]** (example: **map_name[key]**)
      - if key doesn't exist in map, it is added to the map with a "default" value
  - Elements are type pair<key_type, value_type> and are stored in *sorted* order (key is field **first**, value is field **second**)
    - Key type must support less-than operator (<)

map Example

#include <map>

Map elements

map_example.cc

```
void print_out(const pair<int,string>& p) {
  cout << "printout: [" << p.first << "," << p.second << "]" << endl;
}

int main(int argc, char** argv) {
  map<int,string> table;
  map<int,string>::iterator it;

  table.insert(pair<int,string>(2, "hello"));
  table[4] = "NGNM";
  table[6] = "mutual aid";

  cout << "table[6]:" << table[6] << endl;

  it = table.find(4);

  cout << "print_out(*it), where it = table.find(4)" << endl;
  print_out(*it);
  cout << "iterating:" << endl;
  for_each(table.begin(), table.end(), &print_out);
  return 0;
}
```

Equivalent behavior

Returns iterator. (end if not found)
can also use map.count() to see if a key exists

STL `set`

- One of C++'s *associative* containers: a container of unique values, implemented as a search tree
  - http://www.cplusplus.com/reference/set/
  - General form: `set<element_type> name;`
  - elements must be *unique*
    - **`multiset`** allows duplicate elements
  - Efficient lookup (O(log n)) and insertion (O(log n))
  - Inserting an element that already exists does nothing
  - Can use `count(element)` to see if the element exists
  - Elements are stored in *sorted* order
    - element type must support less-than operator (<)

set Example

map_example.cc

```
#include <set>

void print_out(const string& s) {
  cout << "printout: " << s << endl;
}

int main(int argc, char** argv) {
  set<string> names;

  names.insert("bjarne");
  names.insert("ken");
  names.insert("dennis");
  names.insert("travis");
  names.insert("bjarne");

  int n = names.count("bjarne");
  cout << "number of times bjarne occurs: " << n << endl;

  numbers.erase("travis");

  for_each(names.begin(), names.end(), &print_out);
  return EXIT_SUCCESS;
}
```

Doesn't insert duplicate elements

prints "1"

Removes the element "travis"

Unordered Containers (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

Lecture Outline

- C++ algorithm
- C++ STL Cont
  - Map
  - Set
- **Templates**

Suppose that…

- You want to write a function to compare two `int`s
- You want to write a function to compare two `string`s
  - 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;
}
```

does something different in each case

Hm…

- The two implementations of `compare` are nearly identical!
  - What if we wanted a version of `compare` for *every* comparable type?
  - We could write (many) more functions, but that's obviously wasteful and redundant too much repeated code!

- What we'd prefer to do is 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 define 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***, the compiler will generate the "specialized" code from your template using the types you provided
    - Your template definition is NOT runnable code
    - Code is *only* generated if you use your template

Function Templates

- Template to **compare** two “things”:

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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // <...> can also be written <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<int>(10, 20) << std::endl;
  std::cout << compare<std::string>(h, w) << std::endl;
  std::cout << compare<double>(50.5, 50.6) << std::endl;
  return EXIT_SUCCESS;
}
```

Template parameter list

Only uses operator< to minimize requirements on T

Explicit type argument

functiontemplate.cc

Compiler Inference

- Same thing, but letting the compiler infer the types:

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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename 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; // ok
  std::cout << compare(h, w) << std::endl;   // ok
  std::cout << compare("Hello", "World") << std::endl;  // hm...
  return EXIT_SUCCESS;
}
```

Infers int

Infers string

No type specified

Infers char*? Does address integer comparison ☹

functiontemplate_infer.cc

Template Non-types

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

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

// return pointer to new N-element heap array filled with val
// (not entirely realistic, but shows what's possible)
template <typename T, int N>
T* valarray(const T &val) {
  T* a = new T[N];
  for (int i = 0; i < N; ++i)
    a[i] = val;
  return a;
}

int main(int argc, char **argv) {
  int *ip = valarray<int, 10>(17);
  string *sp = valarray<string, 17>("hello");
  ...
}
```

Fixed type template parameter

Use comma separated list to specify template arguments

valtemplate.cc

What's Going On?

- The compiler doesn't generate any code when it sees the template 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 (kind of like macro expansion)
    - The compiler generates template instantiations for *each* type used as a template parameter

This Creates a Problem

```
#ifndef COMPARE_H_
#define COMPARE_H_

template <typename T>
int comp(const T& a, const T& b);

#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 EXIT_SUCCESS;
}
```

main.cc

```
#include "compare.h"

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

compare.cc

Steps to compile

g++ -c compare.cc

Creates an empty .o file since comp<>() is not used!

g++ -c main.cc

No comp<int> definition, expects it to be linked in later

g++ -o main main.o compare.o

No comp<int> definition, compiler error!

Solution #1 (Google Style Guide prefers)

```
#ifndef COMPARE_H_
#define COMPARE_H_

template <typename 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 EXIT_SUCCESS;
}
```

main.cc

Doesn't hide implementation ☹

Solution #2 (you'll see this sometimes)

```
#ifndef COMPARE_H_
#define COMPARE_H_

template <typename T>
int comp(const T& a, const T& b);

#include "compare.cc"

#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 EXIT_SUCCESS;
}
```

main.cc

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

compare.cc

Poll Everywhere

pollev.com/tqm

- Assume we are using Solution #2 (`.h` includes `.cc`)
- Which is the *simplest* way to compile our program (`a.out`)?

A. `g++ main.cc`

B. `g++ main.cc compare.cc`

C. `g++ main.cc compare.h`

D.
```
g++ -c main.cc
g++ -c compare.cc
g++ main.o compare.o
```

E. We're lost…

Poll Everywhere

pollev.com/tqm

- Assume we are using Solution #2 (`.h` includes `.cc`)
- Which is the *simplest* way to compile our program (`a.out`)?

A. `g++ main.cc`

B. `g++ main.cc ~~compare.cc~~`

C. `g++ main.cc ~~compare.h~~` → Template definition added via #include"compare,h"

D. `g++ -c main.cc`
`~~g++ -c compare.cc~~` → Empty object file
`g++ main.o ~~compare.o~~`

E. We're lost…

All of the commands will work, but crossed out parts are unnecessary.

Class Templates

- Templates are useful for classes as well
  - (In fact, that was one of the main motivations for templates!)

- Imagine we want a class that holds a pair of things that we can:
  - Set the value of the first thing
  - Set the value of the second thing
  - Get the value of the first thing
  - Get the value of the second thing
  - Swap the values of the things
  - Print the pair of things

Pair Class Definition

Pair.h

```
#ifndef PAIR_H_
#define PAIR_H_

template <typename Thing> class Pair {
 public:
  Pair() { };

  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void  set_first(Thing &copyme);
  void  set_second(Thing &copyme);
  void  Swap();

 private:
  Thing first_, second_;
};

#include "Pair.cc"

#endif  // PAIR_H_
```

Template parameters for class definition

Could be objects, could be primitives

Using solution #2

Pair Function Definitions

Pair.cc

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

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

template <typename Thing>
void Pair<Thing>::Swap() {
  Thing tmp = first_;
  first_ = second_;
  second_ = tmp;
}

template <typename T>
std::ostream &operator<<(std::ostream &out, const Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", "
             << p.get_second() << ")";
}
```

Definition of Member function of template class

Member of template class

Non member function to print out data in template class discussed later in semester

Using Pair

usepair.cc

```
#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.Swap();
  std::cout << ps << std::endl;

  return EXIT_SUCCESS;
}
```

- `Pair<std::string> ps;` ← Invokes default ctor, which default constructs members ("", "")
- `ps.set_first(x);` ← ("foo", "")
- `ps.set_second(y);` ← ("foo", "bar")
- `ps.Swap();` ← ("bar", "foo")

Class Template Notes (look in *Primer* for more)

- `Thing` is replaced with template argument when class is instantiated
  - The class template parameter name is in scope of the template class definition and can be freely used there
  - Class template member functions are template functions with template parameters that match those of the class template
    - These member functions must be defined as template function outside of the class template definition (if not written inline)
      - The template parameter name does *not* need to match that used in the template class definition, but really should
  - Only template methods that are actually called in your program are instantiated (but this is an implementation detail)

Review Questions (Classes and Templates)

- Why do the accessor methods return `Thing` and not references?

- What happens in the default constructor when `Thing` is a class?

- In the execution of **`Swap`**`()`, how many times are each of the following invoked (assuming `Thing` is a class)?

  ctor _____ cctor _____ op= _____ dtor _____

Review Questions (Classes and Templates)

- Why do the accessor methods return `Thing` and not references?
  To avoid the user being able to manipulate the state of the object indirectly via a reference
- What happens in the default constructor when `Thing` is a class?
  The default constructor for Thing is run on first_ and second_
- In the execution of **`Swap`**`()`, how many times are each of the following invoked (assuming `Thing` is a class)?

ctor __0__ cctor __1__ op= __2__ dtor __1__

temp first_ second_ temp