

C++ Templates & STL (Intro)

CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano

Ian Hsiao

Ramya Challa

Allen Jung

Kyrie Dowling

Sylvia Wang



Poll Everywhere

pollev.com/cse33320su

About how long did Exercise 11 take?

- A. 0-1 Hours
- B. 1-2 Hours
- C. 2-3 Hours
- D. 3-4 Hours
- E. 4+ Hours
- F. I didn't submit / I prefer not to say

Side question:
how is hw2 looking?

Administrivia

- ❖ No exercise released today!
- ❖ Homework 2 due tomorrow (7/23)
 - Don't forget to clone your repo to double-/triple-/quadruple-check compilation!
 - Use Late days if you can't finish & polish your submission! They exist for a reason
- ❖ Mid Quarter Survey is out, due Monday (7/27)
 - Feedback will be used to try and better the rest of this quarter and future quarters!

Lecture Outline

- ❖ **Templates**
- ❖ **STL**

Suppose that...

- ❖ You want to write a function to compare two `ints`
- ❖ You want to write a function to compare two `strings`
 - 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; Only uses operator< to minimize requirements on T
}

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; Explicit type argument
}
```

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 Infers int
    std::cout << compare(h, w) << std::endl; // ok Infers string
    std::cout << compare("Hello", "World") << std::endl; // hm...
    return EXIT_SUCCESS; ↑ 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

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 "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

```
#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

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

```
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

```
#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



- ❖ 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/cse33320su

- ❖ 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" ← Using solution #2

#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> Definition of Member
void Pair<Thing>::set_first(Thing &copyme) { function of template
    first_ = copyme; class
}

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

template <typename Thing> Non member function
void Pair<Thing>::Swap() { to print out data in
    Thing tmp = first_; template class
    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() << ")";
}
```

Using Pair

usepair.cc

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

#include "Pair.h"

int main(int argc, char** argv) {
    Pair<std::string> ps;           // Invokes default ctor, which
    std::string x("foo"), y("bar"); // default constructs members
                                    // ("", "")

    ps.set_first(x);               // ("foo", "")
    ps.set_second(y);              // ("foo", "bar")
    ps.Swap();                    // ("bar", "foo")
    std::cout << ps << std::endl;

    return EXIT_SUCCESS;
}
```

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 are only `get_first()` and `get_second()` const?

These are the only MEMBER functions that do not modify the state of the Pair object

- ❖ 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

- ❖ Why is `operator<<` not a friend function?

Since we have getters, `operator<<` doesn't need direct access to private members of the class, and thus doesn't need to be a friend

- ❖ 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

temp

op= 2

first_second_

dtor 1

temp

Lecture Outline

- ❖ Templates
- ❖ **STL**

C++'s Standard Library

- ❖ C++'s Standard Library consists of four major pieces:
 - 1) The entire C standard library
 - 2) C++'s input/output stream library
 - std::cin, std::cout, stringstream, fstreams, etc.
 - 3) C++'s standard template library (**STL**) 
 - Containers, iterators, algorithms (sort, find, etc.), numerics
 - 4) 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
 - More info in *C++ Primer* §9.2, 11.2
- ❖ Several different classes of container
 - Sequence containers (vector, deque, list, ...)
 - Associative containers (set, map, multiset, multimap, bitset, ...)
 - Differ in algorithmic cost and supported operations

STL Containers 😞

- ❖ 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 constructor or copying is expensive)?
 - You can insert a wrapper object with a pointer to the object
 - We'll learn about these "smart pointers" soon

Our Tracer Class

- ❖ Wrapper class for an `unsigned int` value
 - Also holds unique `unsigned int id_` (increasing from 0)
 - Default ctor, cctor, dtor, `op=`, `op<` defined
 - `friend` function `operator<<` defined
 - Private helper method `PrintID()` to return
`"(id_, value_)"` as a string
 - Class and member definitions can be found in `Tracer.h` and `Tracer.cc`
- ❖ Useful for tracing behaviors of containers
 - All methods print identifying messages
 - Unique `id_` allows you to follow individual instances

Two fields:

value

id (unique to the instance)

Sets `id_` to be unique
for each instance

STL `vector`

- ❖ A generic, dynamically resizable array

- <http://www.cplusplus.com/reference/stl/vector/> Like a normal C array!
- Elements are store in contiguous memory locations
 - Elements can be accessed using pointer arithmetic if you'd like
 - Random access is $O(1)$ time ← Pointer arithmetic, then acces
- Adding/removing from the end is cheap (amortized constant time)
- Inserting/deleting from the middle or start is expensive (linear time)

Need to shift all of the elements in the array

vector/Tracer Example

vectorfun.cc

```
#include <iostream>
#include <vector> ← Most containers are declared in library of same name
#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c; ← Construct three tracer instances & empty vector
    vector<Tracer> vec;

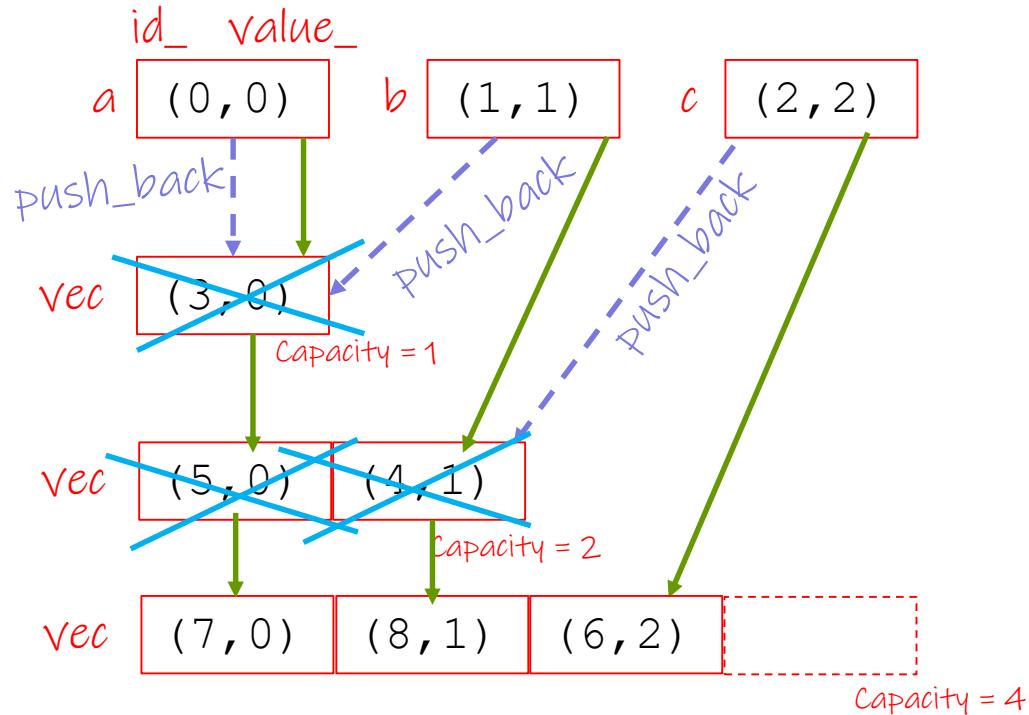
    cout << "vec.push_back " << a << endl;
    vec.push_back(a); ← Add tracers to
    cout << "vec.push_back " << b << endl; end of vector
    vec.push_back(b); ←
    cout << "vec.push_back " << c << endl;
    vec.push_back(c); → Array syntax to access elements

    cout << "vec[0]" << endl << vec[0] << endl;
    cout << "vec[2]" << endl << vec[2] << endl;

    return EXIT_SUCCESS;
}
```

Why All the Copying?

Construct three tracer instances



Key:

Copy constructor

Destructed

Push back calls	Tracers constructed
0	3 (a,b,c)
1	4
2	6
3	9
4	10
5	15

Note:

- Capacity doubles each time capacity is reached
- Exact construction order when resizing is not important