

pollev.com/cse333

About how long did Exercises 5 and 6 take you? (2 polls)

- A. [0, 2) hours
- B. [2, 4) hours
- C. [4, 6) hours
- D. [6, 8) hours
- E. 8+ Hours
- F. I didn't submit / I prefer not to say

Systems Programming

C++ Templates

Instructor: Justin Hsia

Teaching Assistants:

Aadi Adgaonkar

Advaith Vijayakumar

Alice Fu

Andrew Vlahovich

Aviel Wood

Camden Harris

Esha Gore

Hannah Hempstead

Helena Wang

Laura Wang

Megan Frisella

Nina Tharamal

Shrihun Sankepally

Shriya Sateesh

Yusong Yan

Relevant Course Information

- ❖ Exercise 7 not due until next Wed (11/4); focus on HW2 and Midterm
- ❖ Homework 2 due tomorrow (10/29)
 - Don't forget to clone your repo to double-check compilation!
 - Feel free to use late days if you can't finish & polish your submission – they exist for a reason
 - Don't modify header files, we will test with the original .h files
- ❖ Midterm: Monday, November 2 (see [Ed post #300](#))
 - [GUG 220](#) or [ARC 147](#), depending on your *registered* quiz section
 - [Reference sheet](#) and two-sided *handwritten* cheat sheet!
 - Midterm review session on Friday (2/6) from 4:30–6:20 PM (See [Ed post #304](#))
 - Practice midterms and solutions on the course website

Lecture Outline

❖ Templates

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;
}
```

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
- ❖ What we'd prefer to do is write "*generic code*"
 - Code that is **type-independent** (*i.e.*, **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
 - This is in addition to the normal function parameters
- 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;
}
```

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;
}
```

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");
    ...
}
```

valtemplate.cc

What's Going On?

- ❖ The compiler doesn't generate any code when it sees the template function definition
 - It doesn't know what code to generate yet, since it doesn't know what types are involved – **different behavior for different types!**
- ❖ 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

The Great Tragedy of C++ Templates

```
#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 <cstdlib>
#include <iostream>

#include "compare.h"

using std::cout;
using std::endl;

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

main.cc



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 <cstdlib>
#include <iostream>

#include "compare.h"

using std::cout;
using std::endl;

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

main.cc

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 <cstdlib>
#include <iostream>

#include "compare.h"

using std::cout;
using std::endl;

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

main.cc

pollev.com/cse333

Which is the *simplest* way to compile our program (a.out)?

- Assume we are using Solution #2 (.h includes .cc)

- 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...

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. Users of the class 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

```
#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(const Thing& copyme);
    void set_second(const Thing& copyme);
    void Swap();

private:
    Thing first_, second_;
};

#include "Pair.cc"

#endif // PAIR_H_
```

Pair.h

Pair Function Definitions

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

template <typename Thing>
void Pair<Thing>::set_second(const 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() << ")";
}
```

Pair.cc

Using Pair

```
#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;
}
```

usepair.cc

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

- ❖ **Thing** is replaced with template argument when class is instantiated
- ❖ Class template parameters are in scope everywhere within the class's curly braces
 - Inline methods in the class's `{}`s don't need a `template<>` line above them.
 - Method definitions outside of the `{}`s must have a `template<>` line above them, with the same parameters
 - The names *should* match, but don't need to
- ❖ Only template methods that actually get 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`?
- ❖ Why do the accessor methods return `Thing` and not references?
- ❖ Why is `operator<<` not a `friend` function?
- ❖ 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 _____