



pollev.com/cse333

About how long did Exercise 6 take you?

- 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

C++ Templates

CSE 333 Summer 2023

Instructor: Timmy Yang

Teaching Assistants:

Jennifer Xu

Leanna Nguyen

Pedro Amarante

Sara Deutscher

Tanmay Shah

Relevant Course Information

- ❖ Exercise 7 due Wednesday (7/19) @ 1pm!
 - Builds off Exercise 6
 - Uses content from last Friday's lecture
- ❖ Homework 2 due Thursday (7/20) @ 11:59pm
 - Don't forget to clone your repo to double-/triple-/quadruple-check compilation!
 - Don't be afraid to use late days if you can't finish & polish your submission – they exist for a reason
- ❖ Quiz 2: Open Mon. (7/24) @ 2pm to Wed. (7/26) @ 11:59pm
 - Same format as Quiz 1
 - Individual, but high-level discussion allowed (“Gilligan’s Island Rule”)

Lecture Outline

❖ Templates

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; // -1
    std::cout << compare<std::string>(h, w) << std::endl; // -1
    std::cout << compare<double>(50.5, 50.6) << std::endl; // -1
    return EXIT_SUCCESS;
}
```

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 -1
    std::cout << compare(h, w) << std::endl; // ok -1
    std::cout << compare("Hello", "World") << std::endl; // hm... 1
    return EXIT_SUCCESS;
}
```

*Compiler infers char**

*comparing char**

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");
    ...
}
```


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

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

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

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/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 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(const Thing& copyme);
    void set_second(const Thing& copyme);
    void Swap();

private:
    Thing first_, second_;
};

#include "Pair.cc"

#endif // PAIR_H_
```

Pair Function Definitions

Pair.cc

```
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() << ")";
}
```


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;
}
```

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`?
- ❖ 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 _____

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`, `stringstreams`, `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