

[pollev.com/cse333a](https://pollev.com/cse333a)

## About how long did Exercises 9 and 10 take you? (two 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

### Instructors:

Amber Hu

Justin Hsia

### Teaching Assistants:

Ally Tribble

Blake Diaz

Connor Olson

Grace Zhou

Jackson Kent

Janani Raghavan

Jen Xu

Jessie Sun

Jonathan Nister

Mendel Carroll

Rose Maresh

Violet Monserate

# Relevant Course Information

- ❖ No exercise released today
  - Focus on HW2 and midterm prep
- ❖ Homework 2 due tomorrow (2/5)
  - 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, February 9 (see [Ed post #377](#))
  - [BAG 131](#) or [JHN 102](#), 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 #403](#)
  - 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;
}
```

*does something different in each case*

```
// 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 *too much repeated code!*
- ❖ 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; // -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;
}

```

template parameter list (also written as: <class T>)

template function definition

T only needs to implement < to work with compare()

explicit template arguments  
(3 different instances of compare)

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

no explicit template argument // infer char\*, (could be ±1, based on addresses)

# 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");
    ...
}
```

↑ separate template arguments with commas like normal

# 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

*g++ -c compare.cc → empty compare.o!*

```
#include "compare.h" (no usage of comp<>)
```

```
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

*g++ -c main.cc → main.o  
without  
definition of  
comp<int>*

*g++ main.o compare.o → linker error  
(no comp<int>)*

# 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

less implementation hiding! ☹️

# 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



# Poll Everywhere

pollev.com/cse333a



## 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`~~ → empty compare.o

~~`g++ main.o compare.o`~~

E. We're lost...

template definition added to main.cc  
via #include by preprocessor  
(main.cc → compare.h → compare.cc)

all of these command sets will work,  
but crossed out parts are  
unnecessary.

Make file

```
a.out: main.cc compare.h compare.cc
g++ main.cc
```

# 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() { }; // default constructor

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

private:
    Thing first_, second_;
}; // could be primitive or another class

#include "Pair.cc" ← following Solution #2 (for ease of slide separation)

#endif // PAIR_H_
```

template  
class  
definition

# Pair Function Definitions

Pair.cc

```

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

```

*template function for template class member function*

```

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;
}

```

*member of template class instantiation with Thing*

*// nonmember template function to print out Pair values*

```

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 constructor
    std::string x("foo"), y("bar");

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

    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?  
*the accessors don't modify the class instance — the mutators and swap do*  
*— operator << is a non-member function*
- ❖ Why do the accessor methods return `Thing` and not references?  
*returning a reference to a private member violates the 'private' modifier,*  
*so we instead return a copy of Thing*
- ❖ Why is `operator<<` not a `friend` function?  
*it doesn't need access to private data members because it uses the*  
*accessors instead*
- ❖ What happens in the default constructor when `Thing` is a class?  
*data members still get initialized — in this case, invoke the default constructor*  
*of Thing for first\_ and second\_*
- ❖ In the execution of `Swap()`, how many times are each of the following invoked (assuming `Thing` is a class)?

ctor 0cctor 1  
tmpop= 2  
first\_, second\_dtor 1  
tmp