



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



**We've talked about C++. Let's talk about C--.  
What feature of C do we cast into the sea?**

- A. **Pointer types**
- B. **Dynamic memory allocation**
- C. **Structs**
- D. **Arrays**
- E. Preprocessor macros
- F. **None. They're all my precious babies and I'm keeping them.**

# CSE333 Systems Programming

## C++: Templates

### Instructors:

Naomi Alterman

### Teaching Assistants:

Ann Baturytski

Barbora Buzkova

Mendel Carroll

Camden Harris

Hannah Hempstead

Rishabh Jain

Irene Lau

Jonathan Nister

Advay Patil

Aviel Wood



Angela Wu


Jennifer Xu

# Administrivia

❖ Psst! Hey! Nice Ex6! Trade you for an Ex7?

- Due next Wednesday (post-midterm)
- Take your code from Ex6 and “heapify” it

 **Homework 2** due tomorrow (2/5) 

 **Midterm** next **Monday 5/4 11:30-12:20** in **Gates G20** (this room)

- Covers material through end-of-lecture today
- Reference sheet and two-sided 8.5x11” *handwritten* cheat sheet!
- Midterm **review session** this **Friday 5/1 5:30-7:30p** in **Gates G04**
- Practice midterms and solutions on the course website

```
class Foo {
public:
    Foo(int val) { Init(val); }
    ~Foo() { delete foo_ptr_; }
    Foo& operator=(const Foo& rhs) {
        delete foo_ptr_;
        → Init(*rhs.foo_ptr_);
        return *this;
    }

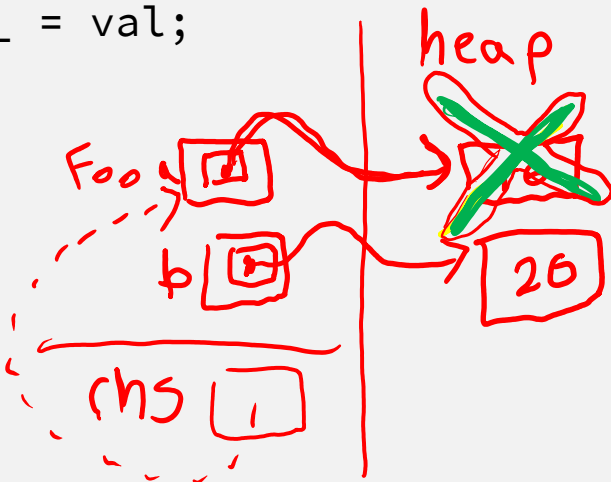
```

```
private:
    int *foo_ptr_;
    void Init(int val) {
        → foo_ptr_ = new int;
        *foo_ptr_ = val;
    }
}

```

```
void Bar() {
    Foo a(10);
    Foo b(20);
    → a = a;
}

```



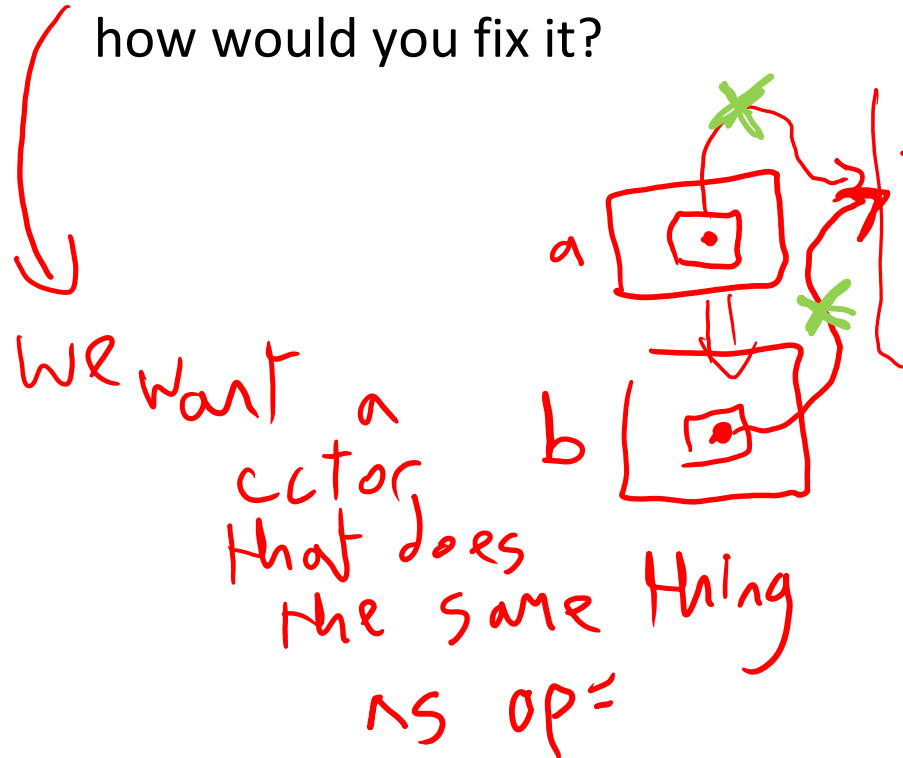
# Last class: what will happen when we invoke **Bar()**?

And if there is an error, how would we fix it?

- A.** Bad dereference
- B.** Bad delete
- C.** Memory leak
- D.** “Works” fine
- E.** Look hon, how am I supposed to know?

# Rule of Three, Revisited

- ❖ Now what will happen when we invoke **Bar()**?
  - If there is an error, how would you fix it?



```

Foo::Foo(int val) { Init(val); }
Foo::~~Foo() { delete foo_ptr_; }

void Foo::Init(int val) {
    foo_ptr_ = new int;
    *foo_ptr_ = val;
}

Foo& Foo::operator=(const Foo& rhs) {
    if (&rhs != this) {
        delete foo_ptr_;
        Init(*(rhs.foo_ptr_));
    }
    return *this;
}

void Bar() {
    Foo a(10);
    Foo b = a; => b(a);
}
    
```

no ctor ;)

# 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!

looks the same

```
// 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;
}
```

asm compare instrs

```
// 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;
}
```

whole func. w/ loops and stuff

# 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**

resolved at compile time

- 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”:

*could be 'class'*

```
#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
template <typename T, int N>
T* initialize_array_on_heap(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 = initialize_array_on_heap<int, 10>(17);
    string* sp = initialize_array_on_heap <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
- ❖ 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

*decl*  
→

```
#ifndef COMPARE_H_
#define COMPARE_H_

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

#endif // COMPARE_H_
```

compare.h

*defn*  
↓

```
#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.o → empty compare.cc*

*g++ -c main.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

*use*  
↑

*g++ -c compare.cc*

*g++ main.o compare.o*

*x\_x linker 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

## 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/naomila](https://pollev.com/naomila)

Which is the *simplest* command (or set of commands) that will compile our program?

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

- A. `g++ -o tragedy main.cc #include → compare.h → compare.cc`
- B. `g++ -o tragedy main.cc compare.cc`
- C. `g++ -o tragedy main.cc compare.h`
- D. `g++ -c main.cc`  
`g++ -c compare.cc`  
`g++ -o tragedy main.o compare.o`
- E. I'm not even gonna try...

# 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:
  - Get and set the value of the first thing
  - Get and set 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

Full type:  
template type name w/ args for all template  
params

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
- ❖ 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 gotta 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

1. Why are only `get_first()` and `get_second()` const?
2. Why do the accessor methods return `Thing` and not references?
3. Why is `operator<<` not a `friend` function?
4. What happens in the default constructor for `Pair` when `Thing` is a class?
5. In the execution of `Swap()`, how many times are each of the following invoked (assuming `Thing` is a class)?

ctor \_\_\_\_\_

cctor \_\_\_\_\_

op= \_\_\_\_\_

dtor \_\_\_\_\_

Code for "Pair"

