C++ Templates

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

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

Administrivia

- Homework 2 due tomorrow!
  - Don't forget to tag `hw2-final` and double/triple check that it compiles!
- Late Policy Reminder:
  - You get 4 "free" late days for the quarter with no deduction
  - One day = due Friday @ 8:59pm
  - Two days = due Sunday @ 11:59pm
- Trying to post *draft* lecture slides before class ← *anon. f/b*
- No exercise assigned today
  - … and an extra 24h for Ex 11 (due *tomorrow* at 11am)
  - 

Lecture Outline

- **Namespaces**
- Templates

Namespaces

- Each namespace is a separate scope
  - Useful for avoiding symbol collisions!

- Namespace definition:
  - ```
    namespace name {
      // declarations go here
    }
    ```
  - Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace* (**!**)
    - This means that components (e.g. classes, functions) of a namespace can be defined in multiple source files
    ```

Classes vs. Namespaces

- They share similar syntax, but classes are *not* namespaces:
  - There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)

- To access a member of a namespace, you must use the fully qualified name (*i.e.* `nsp_name::member`)
  - Unless you are `using` that namespace

- Fully qualified name of a class member (`classname::member`) only needed when you are defining it outside of the class definition

Lecture Outline

- Namespaces
- **Templates**

Suppose that…

- You want to write a function to compare two ints

```
// 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;
}
```

Suppose that…

- You want to write a function to compare two ints

```
// 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;
}
```

- 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 string &value1, const string &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
```

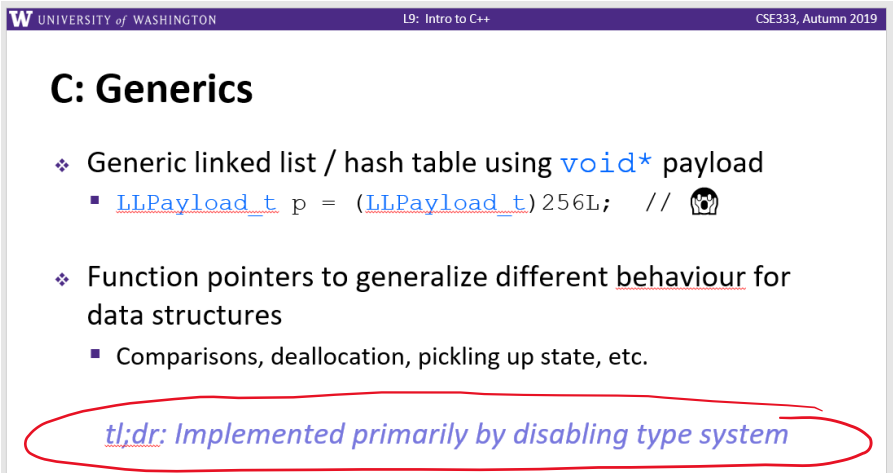
- The two implementations of `compare` are nearly identical!
  - What if we wanted a version of `compare` for *every* type that defines <?
  - We could write (many) more functions, but that's obviously wasteful and redundant

- What we'd prefer to do is write "*generic code*"
  - Code that is type-independent
  - Code that is compile-type polymorphic across types

Review: "Generic Code" in C

- "Code that is type-independent"? ✓
- "Code that is compile-type polymorphic across types"? ✗

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(void *value1, void *value2, CompareFn fn) {
  if (fn(value1, value2)) return -1;
  if (fn(value2, value1)) return  1;
  return 0;
}
```



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

Templating Functions

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::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) {
  string h("hello"), w("world");
  cout << compare<int>(10, 20) << endl;
  cout << compare<string>(h, w) << endl;
  cout << compare<double>(50.5, 50.6) << endl;
  return EXIT_SUCCESS;
}
```

prefer "typename" to "class" here

functiontemplate.cc

Compiler Type Inference

- Same thing, but letting the compiler infer the types:

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::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) {
  string h("hello"), w("world");
  cout << compare(10, 20) << endl;
  cout << compare(h, w) << endl;
  cout << compare(50.5, 50.6) << endl;
  return EXIT_SUCCESS;
}
```

functiontemplate_infer.cc

Template Non-types (we don't do this often)

- You can use constants as template parameters:

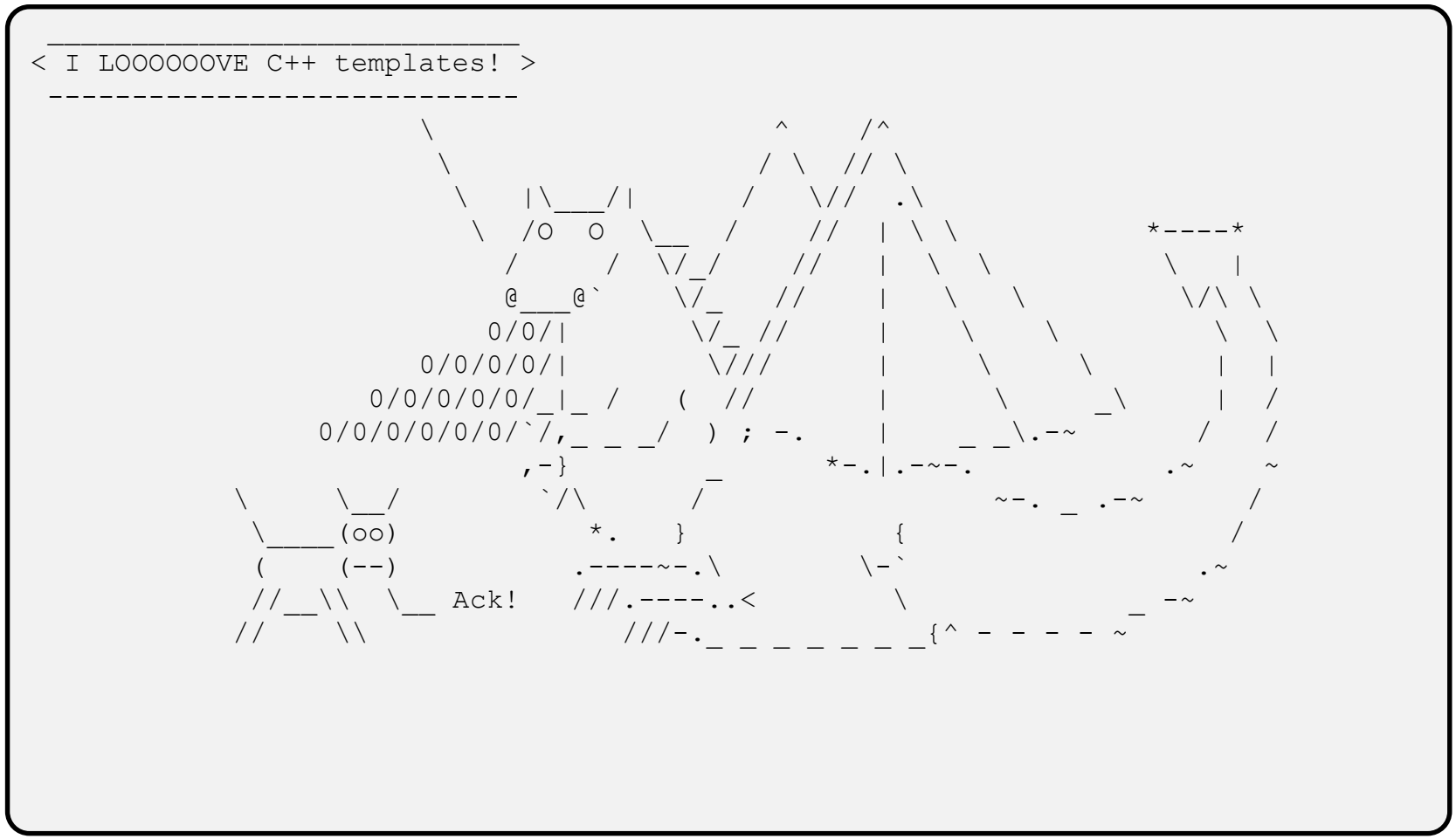
```
#include <iostream>
#include <string>
using std::string;

// dynamically allocate NxM matrix filled with val
template <typename T, int N, int M>
T* valmatrix(const T &val) {
  T* a = new T[N * M];
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j
      a[i * j] = val;
  return a;
}

int main(int argc, char **argv) {
  int *matrix = valmatrix<int, 10, 10>(17);
  string *asciiart = valmatrix<string, 100, 200>(" ");
  ...
}
```

valtemplate.cc

Template Non-types: Ascii Art



even more rare! (basically party trick)

Template Non-types: Computation

```
template <int n>
struct fibonacci {
  static constexpr int value =
    fibonacci<n-1>::value + fibonacci<n-2>::value;
};

template <>
struct fibonacci<0>
{
 static constexpr int value = 0;
};
template <>
struct fibonacci<1>
{
 static constexpr int value = 1;
};

int main(int argc, char **argv)
{
  int array[fibonacci<40>::value];  // compile-time constant!
  return 0;
}
```

fib_template.cc

Instantiation: How a Compiler Views Templates

- 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

Instantiation: How a Compiler Views Templates

```
#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 std::cout;
using std::endl;

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

main.cc

```
#include "compare.h"

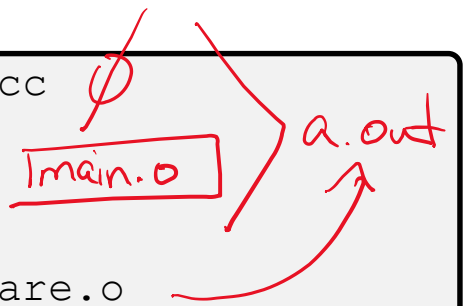
template <typename T>
int comp(const T &a, const T &b) {
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
```

compare.cc

```
$ g++ -c compare.cc

$ g++ -c main.cc

$ g++ main.o compare.o
```



Instantiation Option #1 (Preferred)

```
#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 std::cout;
using std::endl;

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

main.cc

Preferred by Google C++ Style Guide

Instantiation Option #2

```
#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 std::cout;
using std::endl;

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

- Assume we are using Option #2 (`.h` includes `.cc`); which is the ***best*** 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. I'm not sure…

they all work!

Templating Classes

- 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() { };

  const Thing& first() const { return first_; }
  const Thing& 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 Class: Implementation

Pair.cc

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

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

template <typename Thing>
void Pair<Thing>::Swap() {
  Thing tmp = first_;
  first_ = second_;
  second_ = tmp;
}

template <typename Thing>
ostream& operator<<(ostream &out, const Pair<Thing> &p) {
  return out << "Pair(" << p.first() << ", "
             << p.second() << ")";
}
```

templated class

templated function

Pair Class: Instantiation

usepair.cc

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

#include "Pair.h"

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

int main(int argc, char **argv) {
  Pair<string> ps;
  string x("foo"), y("bar");

  ps.set_first(x);
  ps.set_second(y);
  ps.Swap();
  cout << ps << endl;

  return EXIT_SUCCESS;
}
```

Review Questions (Classes and Templates)

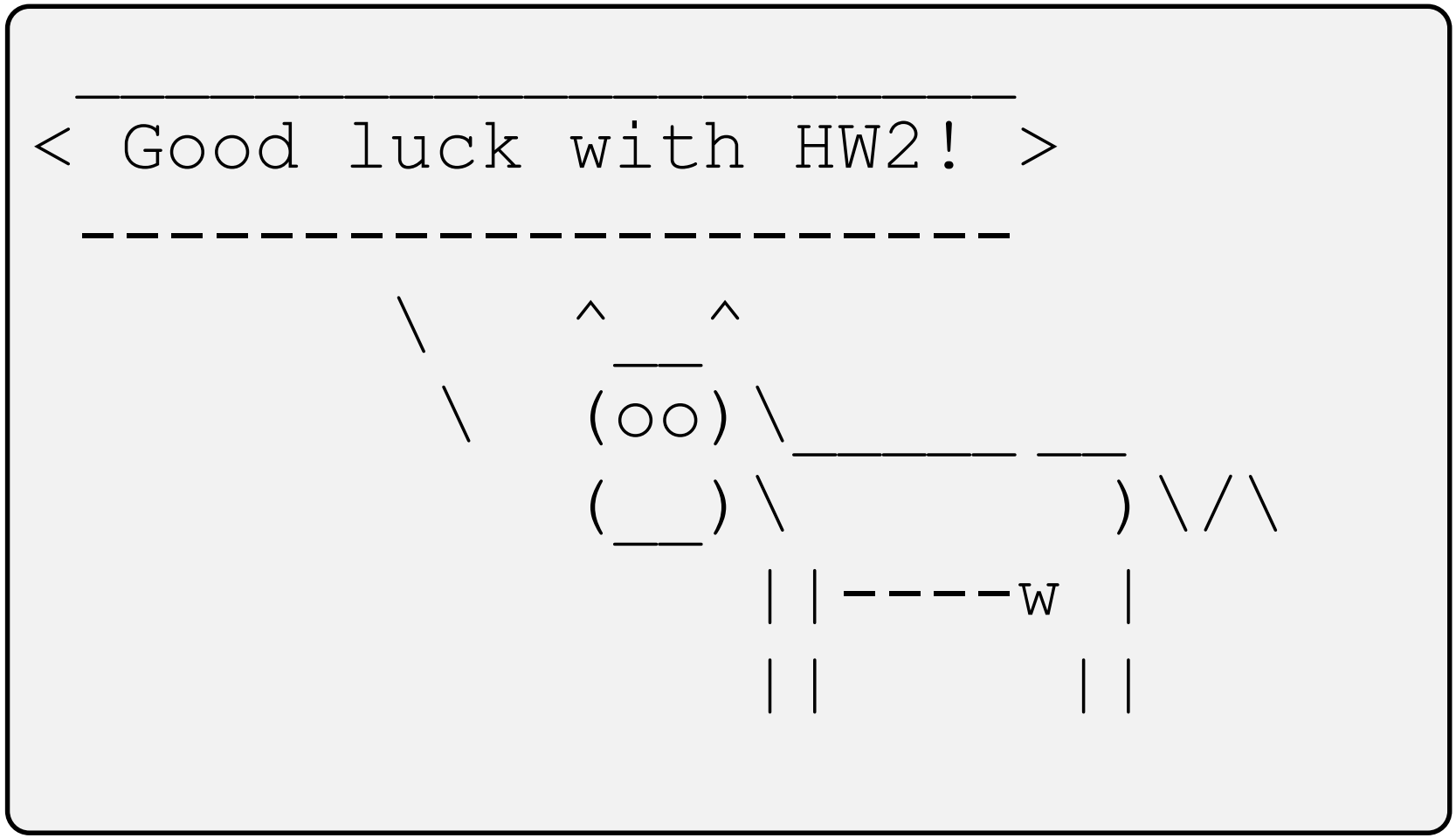
see piazza @309

- Why are only `first()` and `second()` const?
- Why do the accessors return `const Thing&` and not a copy?
- Why is `operator<<` not a `friend` function?
- What happens in `Pair`'s 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 _____

Template Notes (look in *Primer* for more)

- `Thing` is replaced with template argument when function/class is instantiated
  - The template parameter name (`Thing`) is in scope of the function/class definition and can be freely used there
  - *Class* member functions' template parameters match the class's
    - If not inline, these member functions must be defined as templated functions outside of the class template definition
    - The template parameter name does *not* need to match that used in the template class definition, but really should
- Only templated methods that are actually called in your program are instantiated (but this is an implementation detail)



https://textart.io/cowsay