Poll Everywhere

pollev.com/cse333

What value will y have before z is constructed?

A. x_=0, y_=0
B. x_=1, y_=2
C. compile error
D. x_=2, y_=2
E. I’m lost

```
Point::Point(const int x, const int y)
  : x_(x), y_(y) { }

Point& Point::operator=(const Point& rhs) {
    x_ = 2;
    y_ = 2;
  return *this;
}

// copy constructor
Point::Point(const Point copyme) {
  x_ = copyme.x_;
  y_ = copyme.y_;
}

void foo() {
  Point x(1, 2);
  Point y(x);
  Point z(2, 1);
}
// other details of Point class as before
```

C++ Templates

CSE 333 Autumn 2021

Instructor: Chris Thachuk

Teaching Assistants:

Arpad (John) Depaszthory
Ian Hsiao
Logan Gnanapragasam
Mengqi (Frank) Chen
Angela Xu
Khang Vinh Phan
Maruchi Kim
Cosmo Wang

Administrivia

- Exercise 10 released today!
  - Due Monday Nov 1st to give time for HW2
- Homework 2 due Tomorrow (Oct. 28th)
  - Don't forget to clone your repo, and checkout `hw2-final` tag to double-/triple-/quadruple-check compilation on attu!
  - Use Late days if you can't finish & polish your submission! They exist for a reason.
  - No submissions accepted after two late days.

Lecture Outline

- **Templates**

Suppose that…

- You want to write a function to compare two `int`s
- You want to write a function to compare two `string`s
  - 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;
}
```

How values are compared differs in these cases.

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 DRY: "Don't Repeat Yourself"
- What we'd prefer to do is write "*generic code*"
  - Code that is type-independent
  - Code that is compile-time 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
  - 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;
}
```

Template parameter list

Only uses operator< to minimize requirements on T

Explicit type argument

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;
}
```

Infers int

Infers string

No type specified

Infers char*? Does address integer comparison ☹

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");
  ...
}
```

Fixed type template parameter

Use comma separated list to specify template arguments

valtemplate.cc

What's Going On?

- 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 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

Steps to compile

g++ -c compare.cc

Creates an empty .o file since comp<>() is not used!

g++ -c main.cc

No comp<int> definition, expects it to be linked in later

g++ -o main main.o compare.o

No comp<int> definition, compiler 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

Doesn't hide implementation ☹

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

- Assume we are using Solution #2 (`.h` includes `.cc`)
- Which is the *simplest* 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. We're lost…

Poll Everywhere

pollev.com/cse333

- Assume we are using Solution #2 (`.h` includes `.cc`)
- Which is the *simplest* way to compile our program (`a.out`)?

A. `g++ main.cc`

B. `g++ main.cc ~~compare.cc~~`

C. `g++ main.cc ~~compare.h~~`

Template definition added via #include "compare.h"

D. `g++ -c main.cc`
~~`g++ -c compare.cc`~~ → Empty object file
`g++ main.o ~~compare.o~~`

E. We're lost…

All of the commands will work, but crossed out parts are unnecessary.

Still would want these as sources in Makeflie:

```
main: main.cc compare.cc compare.h
      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() { };

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

 private:
  Thing first_, second_;
};

#include "Pair.cc"

#endif  // PAIR_H_
```

Template parameters for class definition

Could be objects, could be primitives

Using solution #2 (easier for separating on slides)

Pair Function Definitions

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 T>
std::ostream &operator<<(std::ostream &out, const Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", "
             << p.get_second() << ")";
}
```

Need to remind compiler we are a template function

Member of template class

Needed for Non-members too
Note: different name 'T' for space

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;
}
```

Invokes default ctor, which default constructs members ("", "")

ps.set_first(x); → ("foo", "")

ps.set_second(y); → ("foo", "bar")

ps.Swap(); → ("bar", "foo")

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 _____

Review Questions (Classes and Templates)

- Why are only **`get_first`**`()` and **`get_second`**`()` const?
  These are the only MEMBER functions that do not modify the state of the Pair object
- Why do the accessor methods return `Thing` and not references?
  To avoid the user being able to manipulate the state of the object indirectly via a reference
- Why is **`operator<<`** not a `friend` function?
  Since we have getters, operator<< doesn't need direct access to private members of the class, and thus doesn't need to be a friend
- What happens in the default constructor when `Thing` is a class?
  The default constructor for Thing is run on first_ and second_
- In the execution of **`Swap`**`()`, how many times are each of the following invoked (assuming `Thing` is a class)?

  ctor __0__ cctor __1__ (temp) op= __2__ (first_ second_) dtor __1__ (temp)