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
Relevant Course Information

❖ Exercise 7 released today!
  ▪ Due Wednesday (2/8) to give time for Homework 2
  ▪ Uses a lot of Monday’s lecture

❖ Homework 2 due tomorrow (2/2)
  ▪ 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

❖ Midterm: February 9 – 11 (see Ed post #543)
  ▪ Take home (Gradescope) and open notes
  ▪ Will involve reflecting on previous assignments
  ▪ Individual, but high-level discussion allowed (“Gilligan’s Island Rule”)
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!

```cpp
// 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;
}
```

```cpp
// 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! 

- What we’d prefer to do is write “generic code”
  - Code that is type-independent
  - Code that is compile-type 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”:

```cpp
#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;
}
```

explicit template arguments

(3 different instances of compare)
Compiler Inference

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

```cpp
#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;
}
```
Template Non-types

- You can use non-types (constant values) in a template:

```cpp
#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

```cpp
#include <iostream>
#include "compare.h"

using namespace std;

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

```cpp
// compare.h

#ifndef COMPARE_H_
#define COMPARE_H_

template <typename T>
int comp(const T& a, const T& b);
#endif // COMPARE_H_

// compare.cc

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

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

```cpp
// main.cc

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

g++ main.o compare.o → linker error
(no comp<int>)
```
# ifndef COMPARE_H_
# define COMPARE_H_

```cpp
# include <iostream>
# include "compare.h"

using namespace std;

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

```cpp
# 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

Less implementation hiding!
#include <iostream>
#include "compare.h"

using namespace std;

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

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

```cpp
#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_; }
        void set_first(const Thing& copyme);
        void set_second(const Thing& copyme);
        void Swap();

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

    } // end pair

#include "Pair.cc"  // following solution #2 (for ease of slide separation)

#endif // PAIR_H_
```

Pair.h
Pair Function Definitions

```cpp
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
}

// 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() << ")";
}
```
#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 accessor don't modify the class instance — the mutators and swap do.

- 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 accessor 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)?

<table>
<thead>
<tr>
<th>ctor</th>
<th>cctor</th>
<th>op=</th>
<th>dtor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>tmp p</td>
<td>first_, second_</td>
<td>tmp</td>
</tr>
</tbody>
</table>