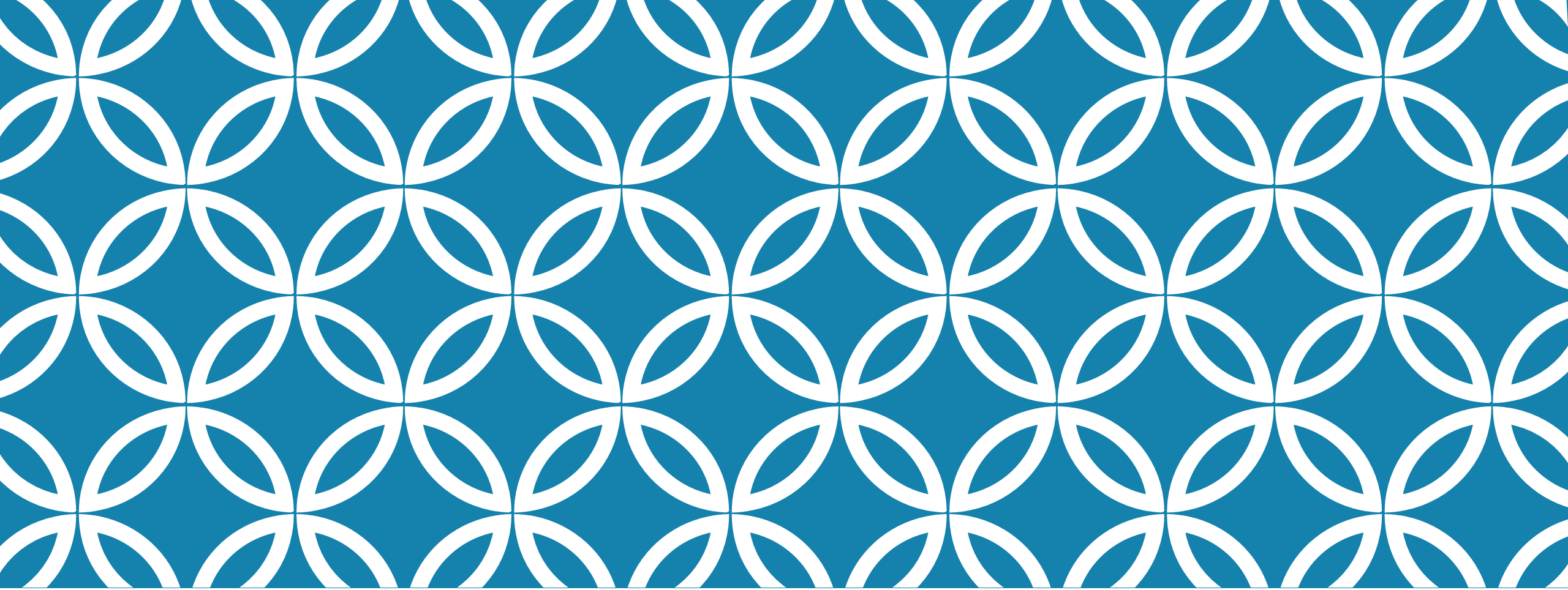

CS4414: RECITATION 7 – TEMPLATES

Ricky Takkar

Friday, March 10, 2023

MOTIVATION
RECAP: LECTURE 10 (2/22)

- Why devote the recitation to a topic (i.e., templates) previously introduced in lecture?
  - Ken Birman, 2/22/23:
    - "The most **awesome** feature of C++ … they (templates) are at the core of why we find C++ to be such a good systems language."
    - "(Templates) let us reprogram the behavior of the compiler in a really mind-bending way."

RECAP: LECTURE 10 (2/22)

- C++ templates are (*sort of*) analogous to Java generics
  - Benefit of former is that they are a **compile-time** construct unlike the latter which is a **run-time** construct
  - As a result, resulting code is super fast!
  - E.g., C++ can't determine in an object by object list which methods to apply on objects of subclass versus class because that requires runtime analysis (which Java *can* do)

C++ ADVANTAGE? (LECTURE SLIDE)

It centers on the compile-time type resolution. Impact? The resulting code is blazingly fast.

In fact, C++ wizards talk about the idea that at runtime, all the fancy features are gone, and we are left with “plain old data” and logic that touches that data mapped to a form of C.

The job of C++ templates is to be as expressive as possible without ever requiring any form of runtime reflection.

SUMMARY OF TEMPLATE GOALS (LECTURE SLIDE)

Compile time type checking and type-based specialization.

A way to create classes that are specialized for different types

Conditional compilation, with dead code automatically removed

Code polymorphism and varargs without runtime polymorphism

THE BASIC IDEA IS EXTREMELY SIMPLE (LECTURE SLIDE)

As a concept, a template could not be easier to understand.

Suppose we have an array of objects of type int:

```
int      myArray[10];
```

With a template, the user supplies a type by coding something like Things<long>. Internally, the class might say something like:

```
template<Typename T>
T        myArray[10];
```

THE BASIC IDEA IS EXTREMELY SIMPLE (LECTURE SLIDE)

As a concept, a template could not be easier to understand.

Suppose we have an array of objects of type int:

```
int     myArray[10];
```

With a templat nd we express this by just codi

```
T       myArray[10];
```

T behaves like a variable, but the "value" is some type, like int or Bignum

YOU CAN ALSO TEMPLATE A CLASS (LECTURE SLIDE)

```
template<typename T>
class Things {
    T       myArray[10];
    T       getElement(int);
     void   setElement(int,T);
}
```

TEMPLATED FUNCTIONS (LECTURE SLIDE)

Templates can also be associated with individual functions. The entire class can have a type parameter, but a function can have its own (perhaps additional) type parameters

```
Template<typename T>
T max(T a, T b)
{
    return a>b? a : b;                    // T must support a > b
}
```

This really should require that T be a type supporting "comparable".

QUICK ASIDE: TEMPLATE HPP FILES DON'T COME WITH ASSOCIATED CPP FILES

- C++ often *generates* implementation file code internally for each type parameter from the template code in hpp file
  - Remember: the compiler generates for each different type parameter that got used

FUNCTION TEMPLATES (LECTURE SLIDE)

Nothing special has to be done to use a function template

```
int main(int argc, char* argv[]) {
    int     a = 3, b = 7;
    double  x = 3.14, y = 2.71;

    cout << max(a, b) << endl;   // Instantiated with type int
    cout << max(x, y) << endl;   // Instantiated with type double
    cout << max(a, x) << endl;   // ERROR: types do not match
}
```

cout is templated. The type is automatically inferred by C++

MOTIVATION

- Your boss wants you to build a digital calculator
- You come up with something like this

recitation > 7 > func_t.cpp > main()

```cpp
#include <iostream>
using namespace std; // don't do this, it's lazy!

int subtract(int a, int b) {
    return a-b;
}

int main () {
    int x=10, y=7, z;
    z=subtract(x,y);
    cout << z << endl;
}
```

MOTIVATION

- But calculators should be able to subtract floats and doubles too! And much more…
- So you come up with

this…

- 😭

recitation > 7 > func_t.cpp > ...

```cpp
#include <iostream>
using namespace std; // don't do this, it's lazy!

int subtract(int a, int b) {
    return a-b;
}

double subtractDouble(double a, double b) {
    return a-b;
}

float subtractFloat(float a, float b) {
    return a-b;
}

int main () {
    int x=10, y=7, z;
    z=subtract(x,y);
    cout << z << endl;
}
```

SOLUTION: FUNCTION TEMPLATES

- What if you could just replace int with a *generic* data type
- How? Let's code!
- Limitation in shown example: parameters in subtract() must share type
- Uh-oh!
  - No worries actually – let's code again!

Break (15 minutes)

1. TA feedback form (check email) ~10 mins
2. Break ~5 mins

CLASS TEMPLATES

- Let's code!

MOTIVATION

- Your boss tells you that you're going to be streaming data of various types, but you need to treat data of type char in a special manner
  - Step 1: You need to be able to identify which data is of type char compared to other types
  - But how?
    - Let's code!

HOW DO WE USE TEMPLATES WHEN OUR FUNCTION HAS AN ARBITRARY NUMBER OF PARAMETERS?

- Common issue…
  - Solution: **Variadic templates**
  - Let's code!

SUMMARY

- Templates let us move away from hardcoding types earlier on in our code so that our code can be more *generic*
- Templates allow us to *specialize* the treatment of select types while applying the default operations on all others
- C++ templates are *compile-time constructs* and thus must be implemented in a manner supporting such constraints
- *Variadic* templates let us leverage template benefits despite arbitrary number of parameters in function