Templates

CMSC 240 Software Systems Development

Today – Templates

- Function Templates
- Class Templates
- In-Class Exercise
- Module 9

RICHMOND

Today – Templates

- Function Templates
- Class Templates
- In-Class Exercise
- Module 9

RICHMOND

Redundant Code

- You want to write a function to compare two **`int`** values
- And you want to write a function to compare two **`string`** values
  - You could use function overloading

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(int value1, int value2)
{
    if (value1 > value2) return 1;
    if (value1 < value2) return -1;
    return 0;
}

int compare(string value1, string value2)
{
    if (value1 > value2) return 1;
    if (value1 < value2) return -1;
    return 0;
}
```

Generic code

- 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-time polymorphic** across types

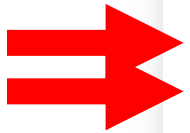
Templates

- 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

Templates

- Function template to **`compare`** two "things"



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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
int compare(T value1, T value2)
{
    if (value1 > value2) return 1;
    if (value1 < value2) return -1;
    return 0;
}

int main()
{
    cout << compare<int>(1, 2) << endl;
    cout << compare<string>("two", "one") << endl;
    cout << compare<double>(50.5, 50.6) << endl;
    return 0;
}
```

Templates

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

```cpp
#include <iostream>
#include <string>
using namespace std;

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
int compare(T value1, T value2)
{
    if (value1 > value2) return 1;
    if (value1 < value2) return -1;
    return 0;
}

int main()
{
    cout << compare(1, 2) << endl;
    cout << compare("two", "one") << endl;
    cout << compare(50.5, 50.6) << endl;
    return 0;
}
```

Templates – What the compiler does

- 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 (i.e., 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
    - The compiler generates template instantiations for each type used as a template parameter

Demo

Templates

- You can use non-types in a template

```
// return a pointer to new N-element heap array filled with value
template <typename T, int N>
T* setupArray(T value)
{
    T* array = new T[N];
    for (int i = 0; i < N; i++)
    {
        array[i] = value;
    }
    return array;
}

int main()
{
    int* intPointer = setupArray<int, 10000>(42);
    string* stringPointer = setupArray<string, 10>("hello");
    delete[] intPointer, stringPointer;
    return 0;
}
```

Today – Templates

- ~~Function Templates~~
- Class Templates
- In-Class Exercise
- Module 9

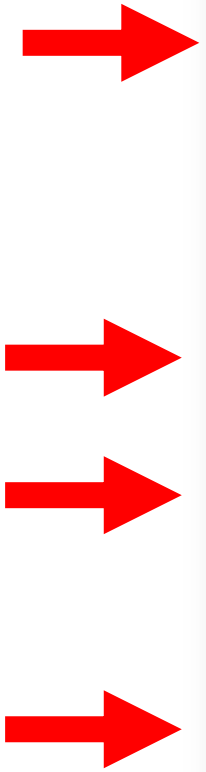
RICHMOND

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

Class Templates

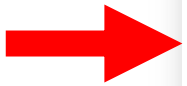


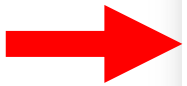
```
#ifndef PAIR_H
#define PAIR_H

template <typename T>
class Pair
{
public:
    Pair() { };
    T get_first() { return first; }
    T get_second() { return second; }
    void set_first(T value);
    void set_second(T value);
    void swap();
private:
    T first;
    T second;
};

#include "Pair.cpp"

#endif
```





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

template <typename T>
void Pair<T>::set_first(T value) { first = value; }

template <typename T>
void Pair<T>::set_second(T value) { second = value; }

template <typename T>
void Pair<T>::swap()
{
    T tmp = first;
    first = second;
    second = tmp;
}

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

Using Pair

```cpp
#include <iostream>
#include <string>
#include "Pair.h"
using namespace std;

int main()
{
    Pair<string> stringPair;

    stringPair.set_first("Hello");
    stringPair.set_second("Goodbye");
    stringPair.swap();

    cout << stringPair << endl;

    return 0;
}
```

Demo

Today – Templates

- ~~Function Templates~~
- ~~Class Templates~~
- In-Class Exercise
- Module 9

RICHMOND