CS4414 Recitation 10
All about templates

10/29/2021
Sagar Jha
Metaprogramming with templates

• Metaprogramming: Metaprogramming is when a program takes as input another program. E.g., g++ takes your C++ program and transforms it into machine code

• With templates, we write a template for the actual source code
Metaprogramming with templates

• Metaprogramming: Metaprogramming is when a program takes as input another program. E.g., g++ takes your C++ program and transforms it into machine code
• With templates, we write a template for the actual source code

```cpp
template <class T>
void print(std::vector<T> vec) {
    for(const T& t : vec) {
        std::cout << t << "\n";
    }
}

int main() {
    std::vector<int> nums{1, 3, 5, 7, 9};
    print(nums);
}
```
Metaprogramming with templates

• The compiler generates code, in this case a function print that takes a
  std::vector<int> (since print is called on a std::vector<int>)

• This is called **template instantiation**

• If print was called on a std::vector<std::string>, another function print
  (overload) will be generated with T = std::string

```cpp
void print(std::vector<int> vec) {
    for(const int& t : vec) {
        std::cout << t << "\n";
    }
}
```
Templates enable generic programming

• A vector of objects, no matter what type, need similar memory management, indexing etc.
Templates enable generic programming

- A vector of objects, no matter what type, need similar memory management, indexing etc.

### std::vector<T> can store

<table>
<thead>
<tr>
<th>T</th>
<th>std::vector&lt;T&gt; can store</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>a finite sequence of integers</td>
</tr>
<tr>
<td>double</td>
<td>a point in the n-dimensional space</td>
</tr>
<tr>
<td>controller::traffic_controller</td>
<td>a collection of traffic controllers in a city</td>
</tr>
<tr>
<td>std::string</td>
<td>fields in a line of a csv file</td>
</tr>
<tr>
<td>std::vector&lt;double&gt;</td>
<td>a real matrix</td>
</tr>
</tbody>
</table>
Templates in the perspective of programming

• One larger goal of programming is to automate tasks. Reflexively, we want to avoid code copying in programming itself
• Functions are blocks of organized, reusable code that model a particular action
• Classes model similar set of objects
• Libraries provide a consistent set of features
• With templates, we can write functions or classes or variables that can work with different types. Templates *abstract away* the type.
Three types of templates

- Variable templates

```cpp
namespace math {
    template<class T>
    T pi = 3.14159265359;
}

int main() {
    std::cout << math::pi<int> << "\n"    // prints 3
    << math::pi<float> << "\n"    // prints 3.14159
    << math::pi<double> << "\n";    // prints 3.14159
}
Three types of templates

• Function templates

```cpp
template <class T>
void print(std::vector<T> vec) {
    for(const T & t : vec) {
        std::cout << t << "\n";
    }
}

int main() {
    std::vector<int> nums{1, 3, 5, 7, 9};
    print(nums); // or print<int>(nums);
}
```
Three types of templates

• Class templates

template <class T>
class my_vector {
public:
    my_vector(size_t _size);
    T& operator[](size_t i);
    const T& operator[](size_t i) const;
    size_t size() const;
private:
    T* elem;
    size_t _size;
    size_t cap;
};
Implementation of a template class function cannot be in .cpp files, in general!

• Suppose we define the `my_vector` class in `my_vector.hpp`. Then, I implement the functions `my_vector::size` and `my_vector::operator[]` etc. in `my_vector.cpp`

• Then, I use a `my_vector<int>` in `main.cpp` which includes `my_vector.hpp`

• This will not compile. Let’s see why.
First, let’s see how to implement template class functions outside the class

```cpp
#include "my_vector.hpp"

template <class T>
my_vector<T>::my_vector(size_t _size)
    : elem(new T[_size]),
    _size(_size) {}

template <class T>
size_t my_vector<T>::size() const {
    return size;
}

template <class T>
T& my_vector<T>::operator[](size_t i) {
    return elem[i];
}

template <class T>
const T& my_vector<T>::operator[](size_t i) const {
    return elem[i];
} // contents of my_vector.cpp
```
We get a linking error at compile time

```
-- mode: compilation; default-directory: "/tmp/" --
Compilation started at Fri Oct 29 13:10:46

```

```
g++ -std=c++2a main.cpp my_vector.cpp -o main
/usr/bin/ld: /tmp/ccn3sT0.o: in function `main':
main.cpp(.text+0x28): undefined reference to `my_vector<int>::my_vector(unsigned long)'
collect2: error: ld returned 1 exit status
```

Compilation **exited abnormally** with code **1** at Fri Oct 29 13:10:46
Why do we get an error?

• Recall that a template class is not a class, but a template for creating a class.

• When the compiler creates an object file from my_vector.cpp which includes my_vector.hpp, my_vector<int> is never used. So, code is not generated for the class my_vector<int>

• When main.cpp is compiled to an object file, the compiler generates code for the definition of my_vector<int>

• When main.o is linked against my_vector.o, the implementation of functions of my_vector<int> is, therefore, not found

• Fun fact: If your template code is not used, it will not be instantiated with real types. You can even have syntactic errors in such code!
What’s the fix?

• In my_vector.cpp, add the following line:

```cpp
template class my_vector<int>;
```

• This requires my_vector.cpp to know that \( T = \text{int} \) will be used.

• What if the user supplies the template parameter for our library class? For e.g., what if we wanted `std::vector<traffic_controller>`?

• The entire implementation must be in the header file 😞
A practical workaround 😊

- Implement the functions in my_vector_impl.hpp (or my_vector.tpp)
- At the end of my_vector.hpp, include my_vector_impl.hpp

```cpp
template <class T>
class my_vector {
public:
    my_vector(size_t _size);
    T& operator[](size_t i);
    const T& operator[](size_t i) const;
    size_t size() const;
private:
    T* elem;
    size_t _size;
    size_t capacity;
};
#include "my_vector_impl.hpp"
```
Question: What happens if I call my print function with an `std::vector<std::vector<int>>`?

template <class T>
void print(std::vector<T> vec) {
    for(const T& t : vec) {
        std::cout << t << "\n";
    }
}

int main() {
    std::vector<std::vector<int>> nums{{{1, 3}, {5, 7}, {9}}};
    print(nums);
}
The compiler writes me an essay (400 lines, 4 pages with very tiny font size, 15-inch Dell XPS)

```
g++-10 with -std=c++2a
```
Concepts: Model constraints on template parameters (C++-20)

• We want the type T to work with the ostream (<<) operator
• We write a Streamable concept requiring just that
• The print function is then templated on a Streamable T

```cpp
template <class T>
concept Streamable = requires(std::ostream& os, T t) {
    { os << t } -> std::convertible_to<std::ostream&>;
};
template <Streamable T>
void print(std::vector<T> vec) {
    for(const T& t : vec) {
        std::cout << t << "\n";
    }
}
```
The error after formally defining the streamable constraint on type T
Template specialization

• Our print function does not work with a vector of vectors
• I can write a specialization of print for a vector of vector of ints
• Another use case: What if I wanted to prints a vector of strings with spaces between them (like a sentence), but a vector of integers with new line?
Template specialization

template <>
void print(std::vector<std::string> vec) {
    for(const std::string& s : vec) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}

template <>
void print(std::vector<std::vector<int>> vec) {
    for(const std::vector<int>& inner_vec : vec) {
        for(const int i : inner_vec) {
            std::cout << i << "\n";
        }
        std::cout << "\n";
    }
}

template <class T>
void print(std::vector<T> vec) {
    for(const T& t : vec) {
        std::cout << t << "\n";
    }
}
std::vector<std::vector<int>> nums{{{1, 3}, {5, 7}, {9}}};
print(nums);
std::vector<std::string> strs{"This", "is", "a", "sentence"};
print(strs);
The SFINAE rule (Substitution Failure is Not An Error)

• If multiple overloads of function templates exist and substitution fails for one particular specialization, a compile time error is not produced. Instead, the compiler tries to match with the next specialization.

• If no specialization matches the invocation, then the compiler (of course) gives an error

• What if there are multiple matches? The compiler picks the most specialized version. If there is a tie, again you get an error.
Value template arguments

template<
    class T,
    std::size_t N
>
struct array; // defined in C++’s header array

std::array<int, 27> arr; // in user code, e.g., main
Default template arguments

• Ever wondered why

    std::priority_queue<controller::traffic_controller> queue;

    works if controller::traffic_controller defines the operator <?

• But, otherwise, you supply a comparator functions like this:

    std::priority_queue<controller, std::vector<controller>, compare> queue;
Default template arguments

• This is because `std::priority_queue` specifies default arguments for the second and the third template parameter

• It works as long as `std::less` which invokes the `<` operator is defined for the type

```cpp
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```
Variadic templates

class car {
public:
    int price;
    car(int price) : price(price) {}
};
class pc {
public:
    int price;
    pc(int price) : price(price) {}
};
class pen {
public:
    int price;
    pen(int price) : price(price) {}
};
Variadic templates

```cpp
int sum() {
    return 0;
}

template <typename T, typename... Args>
int sum(T item, Args... rest) {
    return item.price + sum(rest...);
}

int main() {
    car c(100);
    pc pc(10);
    pen p(1);
    std::cout << "The sum is " << sum(c, pc, p);
}
```