Object Oriented Programming—C++
Lecture8 Template Functions

Qijun Zhao
College of Computer Science
Sichuan University
Spring 2023

Why do we want generic C++?

C++ is strongly typed, but generic C++ lets you parametrize data types!

- Ex. variable return type or input in a class (template classes)

Can we parametrize even more?

Can we write a function that works on **any data type**?

Why not!

Let’s say we want a function to return the min of two ints!

We take in two ints…

```
int myMin(int a, int b) {
    return a < b ? a : b;
}
```

And return an int!

What about doubles? Floats? Longs?

What about function overloading?

Sure, we could…

```
int myMin(int a, int b) {
  return a < b ? a : b;
}

// exactly the same except for types
std::string my_min(std::string a, std::string b) {
  return a < b ? a : b;
}

int main() {
  auto min_int = myMin(1, 2);                  // 1
  auto min_name = myMin("Sathya", "Frankie");  // Frankie
}
```

What about other types?

Template functions are completely generic functions!

Just like classes, they work regardless of type!

Let’s break it down:

Indicating this function is a template

Specifies that Type is generic

List of your template variables

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
```

Default Types

We can define default parameter types!

```
template <typename Type=int>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
```

What does it look like to use a template function?

Calling template functions

We can explicitly define what type we will pass, like this:

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}

// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin<int>(3, 4) << endl; // 3
```

Just like in
template classes!

Calling template functions

We can also **implicitly** leave it for the compiler to deduce!

```
template <typename T, typename U>
auto smarterMyMin(T a, U b) {
  return a < b ? a : b;
}

// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin(3.2, 4) << endl; // 3.2
```

Behind the Instantiation Scenes

Remember: like in template classes, **template functions are not compiled until used!**

- For each instantiation with different parameters, the compiler generates a new specific version of your template
- After compilation, it will look like you wrote each version yourself

Wait a minute…

The code doesn’t exist until you instantiate it, which runs quicker.

Can we take advantage of this behavior?

Templates can be used for efficiency!

Normally, code runs during **runtime**.

With template metaprogramming, code runs **once** during **compile time**!

```
template<unsigned n>
struct Factorial {
  enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
  enum { value = 1 };
};

std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

How?

```
template<unsigned n>
struct Factorial {
  enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
  enum { value = 1 };
};

std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

Why?

Overall, can increase performance for these pieces!

- Compiled code ends up being smaller
- Something runs once during compiling and can be
  used as many times as you like during runtime

TMP was an accident; it was discovered, not invented!

Applications of TMP

TMP isn’t used that much, but it has some interesting implications:

- Optimizing matrices/trees/other mathematical structure operations
- Policy-based design
- Game graphics

Solving problems with generics

What if we wanted to count all the occurrences of a character in a string?

Or a number in a vector?

Or a word in a stream?

These are all the same problem!

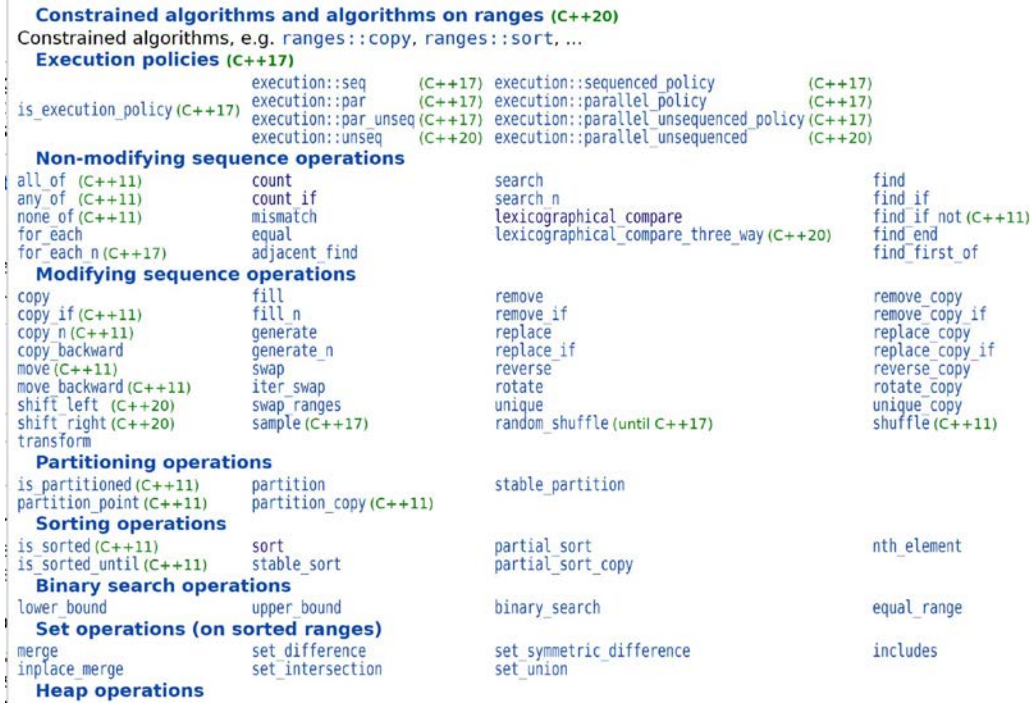
Summary

- Template functions allow you to parametrize the type of a function to be anything without changing functionality
- Generic programming can solve a complicated conceptual problem for any specifics – powerful and flexible!
- Template code is instantiated at compile time; template metaprogramming takes advantage of this to run code at compile time

New toys!

The STL implements an entire library of algorithms written by C++ developers!

- To utilize, #include <algorithm> in your file!
- All algorithms are fully generic, templated functions!



Look familiar?

count_occurrences

```
template <typename InputIt, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred);
```

std::count_if

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
    count( InputIt first, InputIt last, const T& value );
```

Algorithms

All standard algorithms work on iterators.

- Efficient searching, sorting, complex data structure operations, smart pointers, and more are all there for you to use!
- Check out the documentation to get more information!

Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

1. Look both ways before crossing the street.
2. Never tell a pre-med you’ re stressed.
3. When coding, never reinvent the wheel.

This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}
```

isVowel(*iter) ?

Usage

```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```

What if we wanted to find all the vowels in "Xadia" ?

Predicate Functions

Any function that returns a boolean value is a predicate!

- isVowel() is an example of a predicate, but there are

tons of others we might want!

- A predicate can have any amount of parameters…

Unary

```
bool isLowercaseA(char c) {
    return c == 'a';
}

bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Binary

```
bool isMoreThan(int num, int limit) {
    return num > limit;
}

bool isDivisibleBy(int a, int b) {
    return (a % b == 0);
}
```

Let's use that!

```cpp
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val pred(*iter)) count++;
    }
    return count;
}

bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Usage:
```cpp
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), isV
```

What type is UniPred???

Function Pointers

UniPred is what's called a **function pointer!**

- Function pointers can be treated just like other pointers
- They can be passed around like variables as parameters or in template functions!
- They can be called like functions!

Is this good enough?

Are there any ways this could be an issue?

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val pred(*iter)) count++;
    }
    return count;
}

bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Usage:

```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), isVowel);
```

Poor Generalization

Unary predicates are pretty limited and don’ t generalize well.

Ideally, we’ d like something like this!


```
bool isMoreThan3(int num) {
    return num > 3;
```


```
bool isMoreThan3(int num) {
    return num > 3;
}

bool isMoreThan4(int num) {
    return num > 4;
}

bool isMoreThan5(int num) {
    return num > 5;
}

// a generalized version of the above
bool isMoreThan(int num, int limit) {
    return num > limit;
}
```

Can we use binary predicates?

If we could, it would be nice to use a binary predicate to handle this!

```
template <typename InputIt, typename BinPred>
int count_occurrences(InputIt begin, InputIt end, BinPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (pred(*iter, ???)) count++;
    }
    return count;
}
```

Can we use binary predicates?

How do we know what value to use? What about unary (or any other number of arguments) predicates?

```
template <typename InputIt, typename BinPred>
int count_occurrences(InputIt begin, InputIt end, BinPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (pred(*iter, ???)) count++;
    }
    return count;
}
```

We can' t pass this in from the predicate!

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), isVowel);
```

The Catch-22

- We want our function to know more information about our predicate.
- However, we can’t pass in more than one parameter.
- How can we pass along information without needing another parameter?

Let’s use lambdas!

Lambdas are inline, anonymous functions that can know about functions declared in their same scope!

```
auto var = [capture-clause] (auto param) -> bool
{
    ...
}
```

Let’ s use lambdas!

Lambdas are **inline**, anonymous functions that can know about functions declared in their same scope!

```
auto var = [capture-clause] (auto param) -> bool
{
    ...
}
```

Let’ s use lambdas!

Lambdas are inline, **anonymous** functions that can know about functions declared in their same scope!

```
auto var = [capture-clause] (auto param) -> bool
{
    ...
}
```

Let’ s use lambdas!

Lambdas are inline, **anonymous** functions that can know about functions declared in their same scope!

Outside parameters go here

Specifies that Type is generic

```
auto var = [capture-clause] (auto param) -> bool
{
      ...
}
```

Function body goes here!

Let’ s use lambdas!

It might look something like this!

```
int limit = 5;
auto isMoreThan = [limit] (int n) { return n > limit; };
isMoreThan(6); // true
```

Capture Clauses

You can capture any outside variable, both by reference and by value.

- Use just the = symbol to capture everything by value,

and just the & symbol to capture everything by

reference

```
[]                  // captures nothing
[limit]             // captures lower by value
[&limit]            // captures lower by reference
[&limit, upper]     // captures lower by reference, higher by value
[&, limit]          // captures everything except lower by reference
[&]                 // captures everything by reference
[=]                 // captures everything by value
```

We' ve solved our problem!

```
template <typename InputIt, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (pred(*iter)) count++;
    }
    return count;
}
```

Usage:

```
int limit = 5;
auto isMoreThan = [limit] (int n) { return n > limit; };
std::vector<int> nums = {3, 5, 6, 7, 9 ,13};

count_occurrences(nums.begin(), nums.end(), isMoreThan);
```

Using Lambdas

Lambdas are pretty computationally cheap and a great tool!

- Use a lambda when you need a short function or to access local variables in your function.
- If you need more logic or overloading, use function pointers.

Aside: What the Functor?

A **functor** is any class that provides an implementation of operator().

- They can create **closures** of "customized" functions!
- Lambdas are just a reskin of functors!

```
class functor {
public:
    int operator() (int arg) const { // parameters and function body
        return num + arg;
    }
private:
    int num; // capture clause

};

int num = 0;
auto lambda = [&num] (int arg) { num += arg; };
lambda(5);
```

Closure: a single instantiation of a functor object

Tying it all together

So far, we' ve talked about lambdas, functors, and function pointers.

The STL has an overarching, standard function object!

```
std::function<return_type(param_types)> func;
```

Everything (lambdas, functors, function pointers) can be cast to a standard function!

Much bigger and more expensive than a function pointer or lambda!

Summary

- Lambda functions are inline functions that let you pass outside variables in using capture clauses!
- Lambdas can be used to pass predicate function pointers to template functions for more generalizability.
- The STL implements tons of cool alg

四川大學
SICHUAN UNIVERSITY

Coding for love,
Coding for the world

Qijun Zhao
qjzhao@scu.edu.cn