COMP 3522

Object Oriented Programming in C++

Week 9

Agenda

1. Measuring elapsed time
2. Function objects
3. Lambda
4. Template Programming AKA C++ generics

COMP 3522

LAST WEEKS

- Functors, Lambda, Templates
- Design week!
  - Design patterns
  - Design idioms
- Lvalue, rvalue, move, smart pointers
- Review

What if we want to measure duration?

- Developers often want to measure how long something takes
- There is a lo-rez and a high-rez way to do this
- We will try to use the high-rez where possible.

Measure duration – Low rez

```
std::clock_t c_start = std::clock();
//do something
std::clock_t c_end = std::clock();
std::cout << c_end-c_start << std::endl;
```

- c_end-c_start = time in milliseconds
- 1000.0 * (c_end-c_start) / CLOCKS_PER_SEC – clock ticks may be system dependent

Measure duration – Low rez

- std::clock time may advance faster or slower than the wall clock
- if CPU is shared by other processes, std::clock time may advance slower than wall clock
- if current process is multithreaded and more than one execution core is available, std::clock time may advance faster than wall clock

https://en.cppreference.com/w/cpp/chrono/c/clock

Measure duration – High rez

```
auto t_start = std::chrono::high_resolution_clock::now();
//do something
auto t_end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double, std::milli> dur = t_end-t_start;
std::cout << dur.count() << std::endl;
```

Check out time.cpp

FUNCTION OBJECTS

Regular member function call

```
class Greeter {
    public:
        // prints hello, returns length
        int sayHello(const std::string& name)
        {
            cout << "Hello " << name << endl;
            return name.length();
        }
};
Greeter hello;
int a = hello.sayHello("world");
```

Overloading function call operator

```
class Greeter {
    public:
        // Overloaded call operator
        int operator()(const std::string& name)
        {
            cout << "Hello " << name << endl;
            return name.length();
        }
};
Greeter hello;
int a = hello("world");
```

The C++ function object

- It is a generalization of a function
- Useful as predicates or comparison function (STL algorithms)
- Often called a **functor**
- It is an **object that acts like a function**
- **Accomplished by overloading the parentheses operator**

```
F f;
f(1); // like f.operator()(1);
```

A simple example from cplusplus.com

```
struct myclass {
    int operator()(int a) { return a;}
} myobject;

// Looks like a function call, so much neat!
int x = myobject(0);
```

Why use Functors?

- Let's compare functions and functors
- Functions don't maintain internal state
  - Ie: After calling a function, any local variables inside of the function are gone
- Functors are actual objects, meaning they contain state
  - Can have member variables that exist beyond the scope of member functions

Why use Functors?

```
struct myFunctor {
	int sum;
	void operator()(int x) {
		sum += x;
	}
};
```

```
myFunctor addNum, addNum2;

addNum(5);
addNum(10);
//addNum sum = 15

addNum2(100);
addNum2(200);
//addNum2 sum = 300
```

Review – Fibonacci sequence

- Series of numbers where the next number is found by adding the previous two numbers
- Start with: 0, 1
- 0,1,**1**
- 0,1,1,**2**
- 0,1,1,2,**3**
- 0,1,1,2,3,**5**
- …
- 0, 1, 1, 2, 3, 5, 8, 13, **21**…

Let’s convert a function to a functor

```
int fib(){
    static int a = 0, b = 1;
    int c = a + b;
    a = b;
    b = c;
    return a;}
int main(){
    std::cout << fib() << std::endl;
    std::cout << fib() << std::endl;
    std::cout << fib() << std::endl;
}
```

functor_fun.cpp

Analysis

- **Observation**: The function can only track one sequence
- **Question**: What if we want more than one sequence?

- **Solution: create a struct!**

Our next step: a struct

```
struct Fib{
    int a, b;
    Fib(): a{0}, b{1} {}
    int next() {
        int c = a + b;
        a = b;
        b = c;
        return a;
    }
};
```

Using our functor

```
Fib f1, f2;
std::cout << f1.next() << std::endl; // 1
std::cout << f1.next() << std::endl; // 1
std::cout << f1.next() << std::endl; // 2
std::cout << f2.next() << std::endl; // 1
```

Next Problem: I don’t want to type f1.next( ), I just want to type f1( )

Final step: our function object aka functor

```
class Fibonacci{
private:
    int a, b;
public:
    Fibonacci(): a{0}, b{1} { }
    int operator()() {
        int c = a + b;
        a = b;
        b = c;
        return a;
    }
};
```

Using our functor

```
Fibonacci final1;
std::cout << final1() << std::endl; // 1
std::cout << final1() << std::endl; // 1
std::cout << final1() << std::endl; // 2
std::cout << final1() << std::endl; // 3
std::cout << final1() << std::endl; // 5

// Don't do this - this declares a function
Fibonacci f();
```

A neat example

```
struct is_divisible_by{
    int divisor;
    is_divisible_by(int d): divisor{d} {}
    // const no change to divisor
    bool operator()(int number) const {
        return number % divisor == 0;
    }
}
is_divisible_by div5(5); // function object
div5(5); // returns true
div5(11); // returns false
```

Even neater

- We can do something like this, too!

```
list<int>::iterator it =
    find_if(list.begin(), list.end(), div5);
```

LAMBDA I, II

What is a lambda expression?

- An **unnamed function object** (functor)
- A form of "**nested function**"
- You should use lambda expressions when the "function" is used a limited number of times

- Java lingo reminder: anonymous method

The general form of a lambda expression

```
[capture clause] (parameters)<specifiers> -> <return type>
{
   body
}
```

Capture clause

- Used to pass variables from the surrounding scope into the lambda expression:

1. **[ ]** empty there is no capturing
2. **[=]** outside variables are captured by value and cannot be modified inside the lambda expression
3. **[&]** outside variables are captured by reference
4. **[variable_name]** only variable_name is captured by value and cannot be modified inside the lambda
5. **[&variable_name]** only variable_name is captured by reference

Examples 1, 2, 3: capturing vs not capturing

Let's look at **lambda.cpp** together:

1. Compare **lambda 1** and **lambda 2**
2. Uncomment **lambda 3** and try to compile
3. Can you understand the crazy compiler message(s)?

Example 4: capturing by value with [=]

4. Look at **lambda 4** that uses [=] to capture all the variables by value

Example 5: capturing by value won't change

5. Look at **lambda 5** that uses [=] to capture all the variables by value
6. Note that when we uncomment it, we get a compiler error.
7. We cannot change a variable captured by value

Example 6: capturing by reference with [&]

8. Note that **lambda 6** does compile
9. It uses [&] to capture all variables by reference

Example 7: capturing some variables by value

10. Look at **lambda 7** that captures one variable by value
11. We have access to the specified variable
12. We cannot access any other variable outside the lambda

Example 8: capturing some variables by ref

13. Note that **lambda 8** captures one variable by ref
14. We can change that variable (and only that variable)

Example 9: we can mix our capture types

15. Look at **lambda 9** that captures one variable by value and one by reference
16. We **can** change the variable captured by reference
17. We **cannot** change the variable captured by value

What about parameters?

[capture clause] (**parameters**)<specifiers> → <return type>

{

body

}

- **Optional** (obviously)
- **It is bad practice to omit the empty parentheses if there are no parameters.**

`

Example 10: adding two values

18. No captured variables, but there are two parameters
19. We can access the lambda just like a function

Example 11: sorting (hint, hint!)

20. Here's a great use case for the lambda: sorting!
21. Check out **lambda 11** where we sort a std::vector

What about specifiers?

[capture clause] (parameters)<**<u>specifiers</u>**> → <return type>

{

body

}

- Optional
- Use "mutable" to permit a parameter captured by copy to be modified

Example 12: using the mutable specifier

22. Check out **lambda 12** which demonstrates the use of the mutable specifier to modify captured variables

And finally, what about the return type?

- Optional!
- If you don't specify it, the compiler will deduce it
- **<u>Use auto!</u>**

Advantages and disadvantages

- **Advantages**
  1. No functor inexplicably filling a namespace scope
  2. Define a function where we need it
  3. Inlined by the compiler (probably)
  4. Quick to write a simple function:

```
auto max = [](double a, double b) { return a > b ? a : b; };
```

- **Disadvantages**
  1. "Kinda" hard to debug
  2. Hard to find in our code
  3. May generate code duplication
  4. Low reusability

When should we lambda?

Lambdas are great for:

1. Functions passed to STL containers, i.e., something to compare elements in a queue
2. Short one-lined functions
3. Functions that are used in just one place.

Compare functor (function objects) vs lambda

```
struct some_functor {
  void operator()(int i) {
    std::cout << i << '\n';
  }
};
std::for_each( begin, end, some_functor() );

std::for_each(
  begin, end, [](int i){std::cout << i << '\n';} );
```

When to use capture clause vs parameters?

- Think of parameters in capture clause as analogous to member variables in a class
  - Member variables maintain state
- Input arguments are analogous to input arguments to a function
  - Input arguments in function do not maintain state. Lost value when leaving scope

```
int x;
auto myLambda = [&x](int y){x += y; return x;};
```

When to use capture clause vs parameters?

```
struct myFunctor {
    int x;
    myFunctor(int x) : x(x) {}
    int operator()(int y) {
        x += y;
        return x;
    }
};

int x;
auto myLambda = [&x](int y){x += y; return x;};
```

Lambda2.cpp

TEMPLATE PROGRAMMING

STOP! WHAT'S A TEMPLATE?

- We're talking about the Standard **Template** Library
- What's a template?
- It's like a Java generic.

- It's that easy.

The C++ template is like Java's generic

Our function is prefaced with either:

```
template <typename T>
```

or:

```
template <class T>
```

These are equivalent **template parameters**.

This works very nicely with functions!

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

// Suppose we have two doubles first and second
double maximum = get_max<double>(first, second);
```

But what actually happens?

1. The compiler uses the template to generate a new function
2. Each template parameter is replaced with the type passed as the actual template parameter
3. The function is called.

- This is automatic and invisible
- The compiler will often be able to determine the correct instantiation
- “Compiled on demand”

But what actually happens?

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
  cout << myMax<int>(3, 7) << endl;
  cout << myMax<char>('g', 'e') << endl;
  return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
   return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
   return (x > y)? x: y;
}
```

https://www.geeksforgeeks.org/templates-cpp/

What about types?

Will this compile? What is the output?

```
int first{1};
double second{3.14};
double maximum = get_max(first, second);
// auto maximum = get_max(first, second);
```

templateFun.cpp

How about this?

```
int first{1};
double second{3.14};
double maximum = get_max<double>(first, second);
```

What about this?

```
int first{1};
double second{3.14};
int maximum = get_max<int>(first, second);
```

We can write class templates too!

- We can define class with generic types, too!
- Same syntax as a regular class, except that it is preceded by the **template** keyword and a series of template parameters enclosed in angle brackets
- It makes no difference whether the generic type is specified with keyword **class** or keyword **typename** in the template argument list (they are 100% synonyms in template declarations).

An important note

- **The entire template must be in the header file**
- We cannot separate the interface (header file) from the implementation (source file)
- This is because the templates are **compiled as required**

entry.cpp

Multiple typename identifiers

```
template <typename K, typename V>
class Entry{
    K key;
    V value;
public:
    Entry(K key, V value) :
                key{key}, value{value} {}
};
```

Specific typename identifiers

```
template <typename T, int N>
class MySet{
     T set [N];
public:
     void set_member(int index, T member);
     T get_member(int index);
};

template <typename T, int N>
void MySet<T, N>::set_member(int index, T member)
{
     set[index] = member;
}
```

set.cpp

Default typename identifiers

```
template <typename T = string, int N = 25>
class MySet{
     T set [N];
public:
     void set_member(int index, T member);
     T get_member(int index);
};

template <typename T, int N>
void MySet<T, N>::set_member(int index, T member)
{
     set[index] = member;
}
```

set2.cpp

Let's develop a generic printing template

```
// print.h
#include <iostream>

template <typename C>
void print(const C& c)
{
    for (typename C::const_iterator it = c.begin();
         it != c.end(); ++it)
    {
        std::cout << *it << std::endl;
    }
}
```

Think about printing

- Let's use templates in an interesting way
- Use abstraction
- Printing is like copying
- Copying is more abstract than printing
- We can develop an algorithm to copy things from one location to another
- We can even think of printing as copying a range defined by some iterators to some ostream

A copy algorithm (step by step!)

```
void copy(int a[], size_t n, int b[])
{
    size_t i;
    for (i = 0; i < n; ++i) {
        b[i] = a[i];
    }
}
```

A copy algorithm (step 2)

```
void copy(int a[], size_t n, int b[])
{
    size_t i;
    //start at the beginning
    //while we're not at end
    for (i = 0; i < n; ++i) {
        //do copy, increment next index
        b[i] = a[i];
    }
}
```

Our copy algorithm (step 3)

```
void copy(int* first, int* last, int* result)
{
    while (first != last) {
        *result = *first;
        result++;
        first++;
    }
}
// int a [] = { 1, 2, 3, 4, 5 };
// int b [5];
// copy(a, a + 5, b);
```

Our generic copy template (final step)

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
     OutputIterator result)
{
    while (first != last) {
        *result = *first;
         result++;
         first++;
    }
    return result;
}
```

Our generic copy template (final step+)

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
     OutputIterator result)
{
    while (first != last) {
        *result++ = *first++;
    }
     return result;
}
```

Checking if a list is in ascending order

- What if we want to ensure a list is in ascending order
- **What a great opportunity for a template!**

- Check out **ascending.cpp**

Remember functors

```
struct is_divisible_by{
    int divisor;
    is_divisible_by(int d): divisor{d} {}
    // const no change to divisor
    bool operator()(int number) const {
        return number % divisor == 0;
    }
}
is_divisible_by div5(5); // function object
div5(5); // returns true
div5(11); // returns false
```

Let’s convert it to a template

```
template<class T>
struct is_div_by{
    T divisor;
    is_div_by(T d) : divisor{d} {}
    bool operator()(T n) const {
        return n % divisor == 0;
    }
};

is_div_by<int>div10(10);
std::cout << div10(10) << std::endl;
```

functor_fun.cpp
templateStatic.cpp
templateFunctionStatic.cpp

IN CLASS ACTIVITY

1. The Learning Hub -> Content -> Activities **“Templates In Class Activity”**