Computing at Scale

Lecture 10: Building Blocks for Generic Algorithms

Jacob Merson

February 13, 2025

Logistics

Today's Agenda

- Homework Code Review Recap
- Function pointers
- Templateed generic algorithms
- std::function
- Functors, lambdas
- IILE

Assignment 1 Recap

- Code Review Due today at 10PM.
- Make sure you are running the code and test cases.
- Some of the initial code reviews for assignment 0 looked a bit sparse.
- Let me know if there will be a delay as I want to go through these.
- Feel free to tag me in the PR if there are questions about feedback or strategies.
- How did it go?

Building Blocks For Generic Algorithms

Describing a Unit of Work

- A unit of work is a function or method that takes some input and produces some output.
- We saw how templates can be applied to create functions that are generic with respect to types.
- How can we build on this philosophy to create generic algorithms?

Example Algorithm Newton's Method

Newton's method in Python

```python
def newtons_method(func, dfunc, x0, tol=1e-6, max_iter=100):
  x = x0
  for i in range(max_iter):
    x_new = x - func(x)/dfunc(x)
    if abs(x_new - x) < tol:
      return x_new
    x = x_new
  return x
```

- What did we need to make this generic?
- A way to evaluate the user defined function and its derivative.
- How can we accomplish this in C++?

Function Pointer

C++ allows us to define a function that points to a *free* function or a *static* member function. Pointers to member functions are more complicated and we will not cover them in this course (see `https://isocpp.org/wiki/faq/pointers-to-members`).

Syntax:

```
return_type (*pointer_name)(arg_type1, arg_type2, ...);
```

Examples

```
double (*my_fp)(double) // function pointer to a function that takes
    a double and returns a double, pointer is named my_fp

std::string (*my_fp2)(int, int) // function pointer to a function
    that takes two ints and returns a string, pointer is named my_fp2

using my_fp_t = double (*)(double); // using alias for a function
    pointer that takes a double and returns a double

typedef double (*my_fp2_t)(double); // typedef for a function pointer
     that takes a double and returns a double
```

Newtons Method, Function Pointer

```
double my_function(double x) {return x*x - 2;}
double my_function_derivative(double x) {return 2*x;}

using FP = double (*)(double);

double newtons_method(FP func, double(* dfunc)(double), double x0,
    double tol=1e-6, int max_iter=100) {
  double x = x0;
  for (int i = 0; i < max_iter; i++) {
    double x_new = x - func(x)/dfunc(x);
    if (std::abs(x_new - x) < tol) {
      return x_new;
    }
    x = x_new;
  }
  return x;
}
```

Newton's Method Cont.

```
int main() {
  double root = newtons_method(my_function, my_function_derivative,
      1.0);
  std::cout << "Root: " << root << std::endl;
  return 0;
}
```

- Prefer to use a `using` asias with function pointers.
- Class member functions cannot be used with function pointers!
- What do we do when our functions may have cached data? Or are classes?

Passing State

Problem: We want to pass a function to our algorithm that requires state. Idea 1: Callback with callback with `void*` data! (C style)

```
using fp_state = double (*)(double, void*);

class MyClass {
  double state_{1.2};
  public:
  double square_state_value(){state_ *= state_; return state_;}
}

double my_function(double x, void* data) {
  MyClass* my_class = static_cast<MyClass*>(data);
  return my_class->square_state_value();
}
double my_algorithm(fp_state func, void* data) {
  return = func(1.0, data);
}
```

```
int main() {
  MyClass my_class;
  double result = my_algorithm(my_function, &my_class);
}
```

Passing State

Idea 2: Use a template!

```
class MyFunctor {
  double state_{1.2};
  public:
  double operator()(){state_ *= state_; return state_;}
}
double stateless() {
  return 1.0;
}

template <typename Func>
double my_algorithm(Func func) {
  static_assert(std::is_invocable_r_v<double, Func>, "Function␣must␣
     take␣no␣arguments␣and␣return␣a␣double");
  return func();
}
```

```
int main() {
  MyFunctor my_functor;
  double result = my_algorithm(my_functor);
  double result2 = my_algorithm(stateless);
}
```

Templated Algorithm

- With templated algorithm, we can pass in any callable type! i.e., function, functor, lambda, etc.
- Template will deduce to the most appropriate type. (check with cppinsights.io)

Functor

Functor: A class that defines the `operator()` method. This allows the class to be called like a function. A functor can have multiple call operators with different signatures.

```
class MyFunctor {
  public:
  double operator()(double x){return x*x - 2;}
  // int operator()(double x){return x*x - 2;} // error, cannot
      overload on return type
  int operator()(int x){return x*x - 2;} // ok, different signature
}
```

lambdas

- lambdas are syntactic sugar for defining a functor.
- lambdas can automatically capture variables from the enclosing scope by reference, or by value.
- captureless lambdas can be converted to function pointers.
- lambdas are called anonymous functions, because you don't need to give them a name, an their type cannot be named[1].

[1] technically you can retrieve the type with `decltype`, but you shouldn't

lambda syntax

```
// captureless lambdas
[](double x){return x*x - 2;};
[](double x, double y){return x*y;};

// capture by value
int a = 1, b=2;
[=](double x){return x*x - a;};

// capture by reference
[&](double x){return x*x - a;};

// capture this pointer
[this](double x){return x*x - a;};
[*this](double x){return x*x - a;};

// mixed capture
[a, &b](double x){return x*x - a*b;};
```

- Mental model for a lambda is a functor class with member variables for any of the captures.
- Be careful to avoid dangling references with capture by refereence.

Exercise: Implement the following lambdas in cppinsights.io and investigate the generated code.

```
int a = 1, b=2;
auto l1 = [](double x){return x*x - 2;};
auto l2 = [=](double x){return x*x - a;};
auto l3 = [=](double x){return x*x - a*b;};
auto l4 = [&](double x){return x*x - a*b;};
```

IILE

- Another use for lambda functions is to intitialize complex variables as const.
- This method is called the *Immediately Invoked Lambda Expression* (IILE).
- `std::invoke` can be used instead of trailing `()` which can make intent clearer.

Example:

```
const auto my_complex_variable = [](){
  std::vector<int> vec;
  for (int i = 0; i < 10; i++) {
    vec.push_back(i);
  }
  return vec;
}(); // note the () at the end to call the lambda
```

C++ Standard Algorithms

- C++ provides a number of general algorithms.
- See `https://en.cppreference.com/w/cpp/algorithm` for a list of algorithms.
- C++ implements algorithms as templates because they provide the best possible performance with any callable types.

std::function

- What problems do we have with templates? all the code must be in the header file, and exposed to the user!
- C++ has a solution: `std::function`.
- `std::function` is a type-erased wrapper around a callable object.
- `std::function` can be used to store any callable object, including function pointers, functors, and lambdas.
- it is slower than a template. How much? we will find out when we practice profiling.

std::function

Syntax:

```
std::function<return_type(arg_type1, arg_type2, ...)> my_function;
```

Examples:

```
std::function<double(double)> my_function = [](double x){return x*x -
    2;};
Struct MyFunctor {
  double operator()(double x){return x*x - 2;}
};
std::function<double(double)> my_function2 = MyFunctor{};
```

std::function

```
// .h file
void newtons_method(std::function<double(double)> func, std::
    function<double(double)> dfunc, double x0, double tol=1e-6, int
    max_iter=100);

// .cpp file
void newtons_method(std::function<double(double)> func, std::
    function<double(double)> dfunc, double x0, double tol=1e-6, int
    max_iter=100) {
  ...
}
```

Summary

- Generic algorithms need a way to describe a generic unit of work that they can call.
- we can use functions, functors, lambdas to describe this work.
- Function pointers can be used to pass functions to algorithms, but stateful functions take effort.
- Templates can be used to pass any callable object, but all code must be in the header file.
- std::function can be used to store any callable object, but they are slower than templates.
- virtual functions require the user to inherit from a base class and are not flexible enough for fully generic algorithms.

Exercise: Implement a templated version of Newton's method. Make sure to use `static_assert` to ensure that the function passed in has the correct signature. Use newtons method to find find the square root of a number with a function, a lambda, and a functor.