21 - Functions as Parameters

April 10, 2023

COMP2404

Darryl Hill

Contents

Function Types

C++ is a strongly typed language

- All variables declared must have their type declared explicitly
- Even if you use `auto`, there is a type associated with it
- We cannot change the type of a variable, and can only point to a thing of that type.

In C++ we can think of functions as variables

- We can have pointers to functions
- We can have references to functions

These function variables are also strongly typed.

Function Types

Function signature:

- Name of the function
- Types of the parameters
- Order of the parameters

This is enough for the compiler to tell which functions is called.

- Return type does not affect the signature.
- Two matching functions with different return types confuse the compiler.

Function Types

When we make function variables, the type includes

- The function signature, and
- **The return value**

All of these must be specified in the type of a function handle (variable).

These variables can be used in place of a function.

Coding example <p1>

Functions as Parameters

Functions can be passed as parameters to other functions

- Since the parameters and return are well-defined, it is guaranteed to compile.

We have seen this with the Algorithms library

- We can determine at runtime functions for sorting, finding, or general processing.

Coding example <p2>

Functions as Parameters

There are multiple ways to declare a function as an argument.

- Declare the type directly
- Using a template and letting the compiler do the work
- Using the `functional` library

Functions as First Class Citizens

In some languages, functions are first class citizens

That means functions can be used or constructed like values

Functions can be

- Assigned to variables like values
- Passed as parameters like values
- Constructed from expressions like values

C++ does not necessarily have the third property.

Functions as First Class Citizens

In languages like Haskell, functions can be composed in expressions

- if we have a function g(x) and h(x), we can define g·h(x)
- this is akin to defining a function g(h(x)), and assigning it to a variable
- C++ does not do this

However, C++ does have **lambdas**

Lambdas

Lambdas are anonymous functions

- Anonymous - no name, just the signature and implementation
- Signature must match to be used in a variable or as a parameter.
- Can be written within another function to be executed later

Coding example <p3>

Lambdas

Lambdas have a number of different parts, some optional

- Capture clause
- Parameter list
- Mutable specification (optional, we will skip)
- Exception specification (optional, we will skip)
- Trailing return type (optional, the compiler can infer it)
- Body

```
[](int x, int y)->int { /*body*/}
```

Lambda Capture Clause

We have seen parameter list, return type, body

- In addition to these things, lambdas can **capture** variables from the surrounding context

```
int w, z;
auto func = [w, z](int x, int y)->int { /*body*/}
```

`x` and `y` are parameters, and don't exist until called, but `w` and `z` are variables captured from the surrounding context.

This can lead to problems with certain type declarations.

coding example <p3>

Lambda Capture Clause

In the previous example we captured `w` and `z` by **value**

- We can also capture by `reference`.

```
int w, z;
auto func = [&w, &z](int x, int y)->int { /*body*/}
```

- This can lead to problems - does this variable still exist when the lambda is called?

We can also choose to capture **everything**

- by value

```
auto func = [=](int x, int y)->int { /*body*/};
```

- or by reference

```
auto func = [&](int x, int y)->int { /*body*/};
```

Lambda Capture Clause

We can also choose to capture **everything** by value, with exceptions

```
auto func = [=, &w](int x, int y)->int { /*body*/};
```

Or **everything** by reference, with exceptions

```
auto func = [&, w](int x, int y)->int { /*body*/};
```

Functions as Objects

C++ also allows you do have objects behave as functions.

- We can overload the `operator()`
- Now we can declare an object and use it like a function
- It can be passed as a parameter
  - It will match the `function<>` type.
  - It cannot be matched to an actual function type
    - It is still a class / struct

coding example <p4>

The end