【C++】Day19

Class	C++
Date	@December 9, 2021
Material	
Series Number	
Summary	Function Matching

【Ch6】Functions

6.6 Function Matching

In many cases, it is easy to fugure out which overloaded function matches a given call. However, it is not so simple when the overloaded functions have the same number of parameters and when one ore more of the parameters have tyeps that are related by conversions.

Consider the following set of functions and function call:

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); //calls void f(double, double)
```

Determining the Candidate and Viable Functions

The first step of function matching identifies the set of overloaded functions considered for the call. The functions in this set are the candidate functions.

A candidate function is a function with the same name as the called function and for which a declaration is visible at the point of the call.

The second step selects from the set of candidate functions those functions that can be called with the arguments in the given call. The selected functions are the viable functions.

To be viable, a function must have the same number of parameters as there are arguments in the call, and the type of each argument must match-or be convertible to-the type of its corresponiding parameter.

Note: When a function has default arguments, a call may appear to hvae fewer arguments that it actually does.

Having used the number of arguments to winnow the candidate functions, we next look at whether the argument types match those of the parameters. As with any call, an argument might match its parameter either because the types match exactly or because there is a conversion from the argument type to the type of the parameter.

In this example, both of our remaining functions are viable:

- `f(int)` is viable because a conversion exists that can convert the argument of type double to the parameter of type int
- `f(double, double)` is viable because a default argument is provided for the function's second parameter and its first parameter is of type double, which exactly matches the type of the parameter

Note: If there are no viable functions, the compiler will complain that there is no matching function.

Finding the Best Match, if Any

The third step of function matching determines which viable function provides the best match for the call. This process looks at each argument in the call and selects the viable function fo which the corresponding parameter best matches the argument.

An exact match is better than a match that requires a conversion.

6.6.1 Argument Type Conversions

In order to determine the best match, the compiler ranks the conversions that could be used to convert each argument to the type of its corresponding parameter. Conversions are ranked as follows:

1. An exact match. An exact match happens when:
    a. The argument and parameter types are identical
    b. The argument is converted from an array or function type to the corresponding pointer type
    c. A top-level const is added to or discarded from the argument
2. Matching through a const conversion
3. Match through a promotion
4. Match through an arithmetic or pointer conversion
5. Match through a class-type conversion

In order to analyze a call, it is important to remember that the small integral types always promote to int or to a larger integral type. Given two functions, one of which takes an int and the otehr a short, the short version will be called only on values of type short. Even though the smaller integral values might appear to be a closer match, those values are promoted to int, whereas calling the short version would require a conversion:

```
void ff(int);
void ff(short);
ff('a'); //char promoted to int; calls ff(int)
```

Function Matching and const Arguments

When we call an overloaded function that differs on whether a reference or pointer parameter refers or points to const, the compiler uses the constness of the argument to decide which function to call:

```
Record lookup(Account&); //function that takes a reference to Account
Record lookup(const Account&); //new function that takes a const reference
const Account a;
Account b;
lookup(a);
lookup(b);
```

In the first call, we pass the const object a. We cannot bind a plain reference to a const object. In this case, the only viable function is the version that takes a reference to const.

In the second call, we pass the nonconst object b. For this call, both functions are viable. We can use b to initialize a reference to either const or noncnost type. Hoever, initializing a reference to const from a nonconst object requries a conversion.

The version that takes a nonconst parameter is an exact match for b.