【C++】Day20

Class	C++
Date	@December 10, 2021
Material	
Series Number	
Summary	

【Ch6】Functions

6.7 Poiners to Functions

A function pointer is just that-a pointer that denotes a function rather than an object. Like any other pointer, a function pointer points to a particualr type.

A function's type is determined by its return type and the types of its parameters. The function's name is not part of its type. For example:

```
//compares lengths of two strings
bool lengthCompare(const string&, const string&);
```

has type `bool (const string&, const string&)`.

To declare a pointer that can point at this function, we declare a pointer in place of the function name:

```
//pf points to a function returning bool that takes two constr string references
bool (*pf)(const string&, const string&); //uninitialized
```

Using Function Pointers

When we use the name of a function as a value, the function is automatically converted to a pointer.

For example, we can assign the address of `lengthCompare` to pf as follows:

```
pf = lengthCompare; //pf now points to the function named lengthCompare
pf = &lengthCompare; //equivalent assignment: address-of operator is optional
```

Moreover, we can use a pointer to a function to call the function to which the pointer points. We can do so directly-there is no need to dereference the pointer:

```
bool b1 = pf("hello", "goodbye"); //calls lengthsCompares
bool b2 = (*pf)("hello", "goodbye"); //equivalent call
bool b3 = lengthCompare("hello", "goodbye"); //equivalent call
```

Note: There is no conversion between pointers to one function type and pointers to another function type.

However, as usualy, we can assign nullptr or a zero-valued integer constant expression to a function pointer to indicate that the pointer does not point to any function:

```
pf = 0;
pf = nullptr;
```

Pointers to Overloaded Functions

As usually, when we use an overloaded function, the context must make it clear which version is being used. When we declare a pointer to an overloaded function:

```
void ff(int*);
void ff(unsigned int);
void(*pf1)(unsigned int) = ff; //pf1 points to ff(unsigned int)
```

the compiler uses the type of the pointer to determine which overloaded function to use. The type of the pointer must match one of the overloaded functions exactly:

```
void (*pf2)(int) = ff; //error: no ff with a matching parameter list
double (*pf3)(int*) = ff; //error: return type of ff and pf3 don't match
```

Function Pointer Parameters

Just as with arrays, we cannot define parameters of function type but can have a parameter that is a pointer to function.

As with arays, we can write a parameter that looks like a function type, but it will be treated as a pointer:

```
//thrid parameter is a function type and is automatically treated as a pointer to function
void useBigger(const string &s1, const string &s2, bool pf(const string&, const string&));
//equivalent declaration: explicitly define the parameter as a pointer to function
void useBigger(const string &s1, const string &s2, bool (*pf)(const string&, const string&));
```

Writing function pointer types quickly gets tedious. Type aliases, along with `decltype` and `using`, let us simplify code that uses function pointers:

```
//Func and Func2 have function type
typedef bool Func(const string&, const string&);
typedef decltype(lengthCompare) Func2; //equivalent type
using Func3 = bool (const string&, const string&);

//FuncP and FuncP2 have pointer to funcion type
typedef bool(*FuncP)(const string&, const string&);
typedef decltype(lengthCompare) *FuncP2; //equivalent type
using FuncP3 = bool (*)(const string&, const string&);
```

Both `Func` and `Func2` are function types, wehereas `FuncP` and `FuncP2` are pointer types. It is important to note that decltype returns the function type; the automatic conversion to pointer is not done. Because decltype returns a function type, if we want a pointer we must add the * ourselves.

We can redeclare `useBigger` using any of these steps:

```
//equivalent declarations of useBigger using type aliases
void useBigger(const string&, const string&, Func);
void useBigger(const string&, const string&, FuncP2);
```

Both declarations declare the same function. In the first case, the compiler will automatically convert the function type represented by `Func` to a pointer.

Returning a Pointer to Function

As with arrays, we can't return a function type but can return a pointer to a function type. Similarly, we must write the return type as a pointer type; the compiler will not automatically treat a function return type as the corresponding pointer type.

By fat the easiest way to declare a function that returns a pointer to function is by using a type alias:

```
using F = int(int*, int); //F is a function type, not a pointer
using PF = int (*)(int*, int); //PF is a pointer type
```

We must explicitly specify that the return type is a pointer type:

```
PF f1(int); //ok: Pf is a pointer to function
F f1(int); //error: F is a function type; f1 can't return a function type
F *f1(int); //ok: explicitly specify that the return type is a pointer to function
```

Or we can also declare f1 directly, which we'd do as:

```
int (*f1(int))(int*, int);
```

Or using a trailing return:

```
auto f1(int) -> int (*)(int*, int);
```

Using auto or decltype for Function Pointer Types

If we know which functions we want to return, we can use `decltype` to simplify writing a function pointer return type.

For example, assume we have two functions, both of which return a `string::size_type` and have two `const string&` parameters. We can write a third function that takes a string

parameter and returns a pointer to one of these two functions as follows:

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);

// depending on the value of its string parameter
//getFcn returns a pointer to sumLength or to largerLength
decltype(sumLength) *getFcn(const string&);
```

Exercise

Exercises Section 6.7

Exercise 6.54: Write a declaration for a function that takes two `int` parameters and returns an `int`, and declare a `vector` whose elements have this function pointer type.

Exercise 6.55: Write four functions that add, subtract, multiply, and divide two `int` values. Store pointers to these values in your `vector` from the previous exercise.

Exercise 6.56: Call each element in the `vector` and print their result.

```
vector<int (*)(int, int)> vec;

int add(int n, int a) { return n + a; }

int sub(int n, int a) { return n - a; }

int mul(int n, int a) { return n * a; }

int divi(int n, int a) { return n / a; }

int main(int argc, char **argv) {
  vec.push_back(add);
  vec.push_back(sub);
  vec.push_back(mul);
  vec.push_back(divi);
  for(auto beg = vec.begin() ; beg != vec.end(); ++beg) {
    cout << (*beg)(4, 4) << endl;
  }
  return 0;
}
```