【C++】Day15

Class	C++
Date	@December 5, 2021
Material	
Series Number	
Summary	

【Ch6】Functions

6.1 Function Basics

A function definition typically consists of a return type, a name, a list of zero or more parameters, and a body.

The actions that the function performs are specified in a statement block, referred to as the function body.

We execute a function through the call operator, which is a pair of parentheses. The call operator takes an expression that is a function or points to a function. Inside the parentheses is a comma-separated list of arguments. The arguments are used to initialize the function's parameters.

Writing a Function

A function call does two things: It initializes the function's parameters from the corresponding arguments, and it transfers control to that function. Execution of the calling function is suspended and the execution of the called function begins.

Execution of a function ends when a return statement is encountered. Like a function call, the return statement does two things: It returns the value in the return, and it transfers control out of the called function back to the call expression.

Parameters and Arguments

Although we know which argument initializes which parameter, we have no guarantees about the order in which arguments are evaluated. The compiler is free to evaluate the arguments in whatever order it prefers.

Warning: Local variables at the outermost scope of the function may not use the same name as any parameter.

Parameter names are optional. However, there is no way to use an unnamed parameter.

6.1.1 Local Objects

In C++, names have scope and objects have lifetimes. It is important to understand both of these concepts:

- The scope of a name is the part of the program's text in which that name is visible.
- The lifetime of an object is the time during the program's execution that the object exists.

Local static Objects

It can be useful to have a local variable whose lifetime continues across calls to the function. We obtain such objects by defining a local variable as `static`. Each local static object is initialized before the first time execution passes through the object's definition. Local statics are not destroyed when a function ends; they are destroyed when the program terminates.

If a local static has no explicit initializer, it is value initialized meaning that local statics of built-in type are initialized to zero.

6.1.2 Function Declarations

Like any other name, the name of a function must be declared before we can use it. A function may be defined only once but may be declared multiple times.

A function declaration is just like a function definition except that a declaration has no function body. In a declaration, a semicolon replaces the function body.

Because a function declaration has no body, there is no need for parameter names. Hence, parameter names are often omitted in a declaration. Although parameter names are not required, they can be used to help users of the function understand what the function does.

Recall that variables are declared in header files and defined in source files. For the same reasons, functions should be declared in header files and defined in source files.

Best Practices: The header that declares a function should be included in the source file that defines that function.

6.1.3 Separate Compilation

Assume that the definition of our fact function is in a file named `fact.cc` and its declaration is in a header named `Chapter6.h`. Our fact.cc file, like any file that uses these functions, will include the Chapter6.h header. We'll store a main function that calls fact in a second file named factMain.cc.

To produce an executable file, we must tell the compiler where to find all of the code we use.

```
> CC factMain.cc fact.cc #generates factMain.exe or a.out
> CC factMain.cc fact.cc -0 main #generates main or main.exe
```

Here `CC` is the name of our compiler and `#` begins a command-line comment.

If we have changed only one of source files, we'd like to recompile only the file that actually changed .Most compilers provide a way to separately compile each file. This process usually yields a file with the .obj(Windows) or .o(Unix) file extension, indicating that the file contains object code.

We would separately compiler our program as follows:

```
> CC -c factMain.cc #generates factMain.o
> CC -c fact.cc #generates fact.o
> CC factMain.o fact.o #generates factMain.exe or a.out
> CC factMain.o fact.o -o main #generates main or main.exe
```