Review of Scope, Lifetime, and More on Functions (Chapter#6)

In prior chapters, we said that local variables are those declared within a block. The block does not have to be the body of a function—local identifiers can be declared within any block. In a large program, there may be several variables with the same identifier— name, for example. How do we know which variable is meant and where each variable is accessible? The answers to these questions are provided by scope rules.

Scope of an Identifier

There are four categories of scope for an identifier in C++: class scope, local scope, namespace scope, and global scope. Class scope is defined in Chapter II. Local scope is the scope of an identifier defined within a block and extends from the point where the identifier is declared to the end of the block. Global scope is the scope of an identifier declared outside all functions and extends to the end of the file containing the program.
Function names in C++ have global scope. Once a function name has been declared, any subsequent function can call it. In C++ you cannot nest a function definition within another function. When a function defines a local identifier with the same name as a global identifier, the local identifier takes precedence. That is, the local identifier hides the existence of the global identifier. For obvious reasons this principle is called name precedence or name hiding.

The rules of C++ that govern who knows what, where, and when are called scope rules.

1. A function name has global scope.
2. The scope of a global identifier extends from its declaration to the end of the file in which it is defined.
3. The scope of a parameter is the same as the scope of a local variable declared in the outermost block of the function body.
4. The scope of a local identifier includes all statements following the declaration of the identifier to the end of the block in which it is declared and includes any nested blocks unless a local identifier of the same name is declared in a nested block.
5. The scope of an identifier begins with its most recent declaration.

The last three rules mean that if a local identifier in a block is the same as a global or nonlocal identifier to the block, the local identifier blocks access to the other identifier. That is, the block's reference is to its own local identifier.
To summarize, any variable or constant declared outside all functions is a global identifier. Any global identifier is known (and can be accessed directly) by any function that does not declare a variable or constant with the same name. Local identifiers of a function are nonlocal (but accessible) to any block nested within it.
If global identifiers are accessible to all functions, why don't we just make all identifiers global and do away with parameter lists? Good programming practice dictates that communication between the modules of our program should be explicitly stated. This practice limits the possibility of one module accidentally interfering with another. In other words, each function is given only what it needs to know. This is how we write "good" programs.

The avoidance of global variables is especially critical in a team environment, where one person should not have to check with everyone else before using or changing a variable because it might be global.

Namespaces

As a concept, namespace is the same as scope. In C++, namespace is a language feature that allows a programmer to create a named scope. The authors of <iostream>, for example, enclosed the definitions and declarations within a namespace called std. To access streams cm and cout and the manipulator endl, we use the statement

using namespace std;

near the front of our programs. This statement tells the compiler that all identifiers defined in namespace std (defined in file io stream) are accessible to the parts of the program that are within the scope of the using directive.

Declarations and Definitions

In Chapter 6, we used a function prototype to tell the compiler that a function with certain parameters would be defined later. The function prototype is a declaration that is not also a definition. The function definition includes a heading and a body. One need for a function prototype comes about because most C++ programmers put function main physically first in the file. main invokes other functions, so the names of the other functions and their parameter types must be known before function main is compiled. In our example, if we had put the function definition physically before main, the prototype would not have been necessary, and the function definition would have been both a declaration and a definition. However, function prototypes cannot be eliminated simply by moving main: After all, we might have a function that calls a second function, with that second function also calling the first function.

The distinction between a declaration and a definition is that memory space is reserved in a definition. In a function definition, the space for the compiled code of the function body is reserved. There are times when we want to declare a variable but not define it. That is, we want to let the compiler know about a global variable defined in another file. Look at the following two statements:

```
int myValue;
extern int anotherValue;
```

The first statement is both a declaration and a definition. Space is reserved for the variable myValue of data type int. The second statement is a declaration only. Your program can reference anotherValue, but the compiler does not reserve space for it. The reserved word extern tells the compiler that anotherValue is a global variable located in another file.

More about Variables

The lifetime of a variable is the time during program execution when the variable has storage assigned to it. The lifetime of a global variable is the entire execution of the program. The lifetime of a local variable is the execution of the block in which it is declared. Sometimes we want to have the value of a local variable remain the same between calls to the same function. For example, to know how many times a function is called during the execution of a program, we would like to use a local function variable as a counter and increment it each time the function is called. However, if space is allocated each time the function is called and deallocated when the function finishes executing, we can't guarantee that the space for the local variable will be the same for each function invocation. In fact, it most likely would not be.

To get around this problem, C++ lets the user determine the lifetime of each local variable by assigning it a storage class: static or automatic. If you want the lifetime of a local variable to extend to the entire run of the program, you preface the data type identifier with the reserved word static when the variable is defined. The default storage class is automatic, in which storage is allocated on entry and deallocated on exit from a block.

Any variable may be initialized when it is defined. This is the fourth way to assign a value to a place in memory. (The first three were discussed in earlier chapters.) To initialize a variable, follow the variable identifier with an equal sign and an expression. The expression is called an initializer. The initializing process differs depending on whether the variable being initialized is static or automatic. For a static variable, the initializer must be a constant expression, and its value is stored only once. For an automatic variable, the initializer can be any expression, and its value is stored each time the variable is assigned storage.