**Chapter 9 Memory Models and Namespace**

You have choices for how long data remains in memory (storage duration) and choices for which parts of a program have access to data (scope and linkage).

* **Separate Compilation**

You can divide the original program into three parts:

* A header file that contains the structure declarations and prototypes for functions that use those structures;
* A source code file that contains the code for the structure-related functions;
* A source code file that contains the code that calls the structure-related functions;

This is a useful strategy for organizing a program. If, for example, you write another program that uses those same functions, you can just include the header file and add the functions file to the project or make list.

You shouldn’t put function definitions or variable declarations into a header file. Here are some things commonly found in **header files**:

1. Function prototypes;
2. Symbolic constants defined using #define or const;
3. Structure declarations;
4. Class declarations;
5. Template declarations;
6. Inline functions;

#ifndef COORDIN\_H\_

#define COORDIN\_H\_

// place include file contents here

#endif

* **Storage Duration, Scope, and Linkage**

Recall that a block is a series of statements enclosed in braces.

**Automatic storage duration**: Variables declared **inside** a function definition, including function parameters, have automatic storage duration. The memory used for them is freed when execution leaves the function or block. C++ has two kinds of automatic storage duration variables.

**Static storage duration**: Variables defined **outside** a function definition or else by using the keyword **static** have static storage duration. They persist for the **entire time a program is running**. C++ has three kinds of static storage duration variables.

**Dynamic storage duration**:Memory allocated by the ***new*** operator persists **until it is freed** with the ***delete*** operator or until the **program ends**, whichever comes first.

* **Scope and Linkage**

**Scope** describes how **widely visible** a name is ***in a file*** (translation unit). For example, a variable defined in a function can be used in that function but not in another, whereas a **variable defined in a file above the function definitions can be used in all the functions**.

**Linkage** describes how a name can be shared in different units. A name with **external linkage** can be shared **across files**, and a name with **internal linkage** can be shared by functions **within a single file**. Names of automatic variables have no linkage because they are not shared.

* **Automatic Storage Duration**

Function parameters and variables declared inside a function have, by default, automatic storage duration. They also have local scope and no linkage. That is, if you declare a variable called *texas* in main() and you declare another variable *with the same name in a function called oil()*, **you’ve created two independent** **variables**, each known only in the function in which it’s defined. Anything you do to the *texas* in oil() has no effect on the *texas* in main(), and vice versa.

int main()

{

int teledeli = 5;

{ // **websight allocated here, not the line it is declared**

cout << "Hello\n";

int websight = -2; // **websight scope begins**

cout << websight << ' ' << teledeli << endl;

} // websight expires

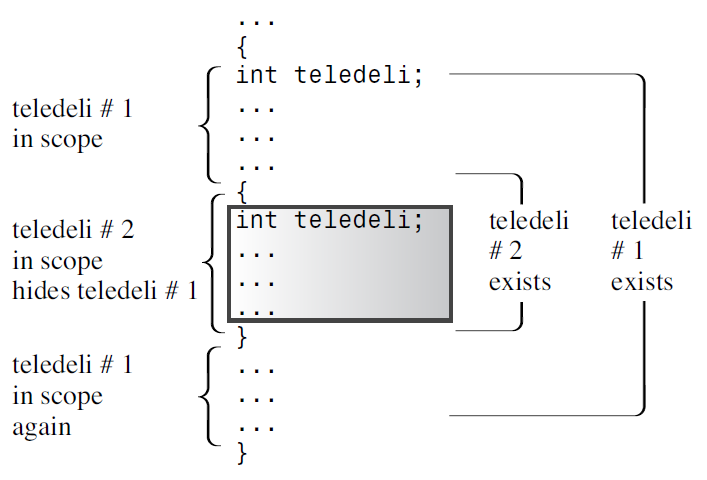
cout << teledeli << endl;

...

} // teledeli expires

Also each **variable is allocated** when program execution **enters the *innermost block*** containing the definition (**not the line the variable is declared**), but the **scope begins only after the point of declaration**.

**What if you name the variable in the inner block *teledeli* *instead of websight* so that you have two variables of the same name, with one in the outer block and one in the inner block**? In this case, the program interprets the *teledeli* name to mean the local block variable while the program executes statements **within the block**. **We say the new definition hides the prior definition**. The new definition is in scope, and the old definition is temporarily out of scope. When the program leaves the block, the original definition comes back into scope (see Figure 9.2).



**Changes to auto in C++11**

In **C++11**, the keyword **auto** is used for **automatic *type*** deduction, while in **C and in prior versions of C++**, auto has an entirely different meaning. It’s used to explicitly identify a variable as having **automatic *storage.***

**Register Variables**

C originally introduced the register keyword to suggest that the compiler use a CPU register to store an **automatic variable**:

register int count\_fast; // request for a register variable

**Prior to C++11**, C++ used the keyword in the same fashion, except that as hardware and compilers developed in sophistication, the hint was generalized to mean that the variable was heavily used and perhaps the compiler could provide some sort of special treatment.

**With C++11**, even that hint is being deprecated, leaving register as just a way to explicitly identify a variable as being automatic.

The more important reason for retaining register, however, is to avoid invalidating existing code that uses that keyword. *Register* can only be used with variables that would be automatic.

* **Static Duration Variables**

C++, like C, provides **static storage *duration*** variables with **three kinds of *linkage***: *external linkage* (accessible across files), *internal linkage* (accessible to functions within a single file), and *no linkage* (accessible to just one function or to one block within a function).

All three last for the duration of the program. Also if you don’t explicitly initialize a static variable, the compiler sets it to 0. Static arrays and structures have all the bits of each element or member set to 0 by default.

To create a static duration variable with external linkage, you declare it outside any block. To create a static duration variable with internal linkage, you declare it outside any block and use the static storage class modifier. To create a static duration variable with no linkage, you declare it inside a block, using the static modifier.

int global = 1000; // static duration, **external linkage**

**static** int one\_file = 50; // static duration, **internal linkage**

int main()

{

...

}

void funct1(int n)

{

static int count = 0; // static duration, no linkage

int llama = 0;

...

}

void funct2(int q)

{

...

}

**Static Duration, External Linkage**

Variables with external linkage are often simply called external variables. **If you use an external variable** in several files, **only one file can contain a definition** for that variable (per the one definition rule). But every **other file using the variable needs to declare** that variable using the keyword *extern*:

// file01.cpp

**extern int cats** = 20; // definition because of initialization,**定义的时候也可以加extern**

int dogs = 22; // also a definition

int fleas; // also a definition

...

// file02.cpp

// use cats and dogs from file01.cpp

**extern** int cats; // not definitions because they use

**extern** int dogs; // extern and have no initialization

...

// file98.cpp

// use cats, dogs, and fleas from file01.cpp

extern int cats;

extern int dogs;

extern int fleas;

What if you *define an external variable* and then *declare a variable by the same name inside a function*? The *second declaration* is interpreted as a definition for an **automatic variable**. The automatic variable is the one that is in scope when the program executes that particular function.

// **external.cpp**

#include <iostream>

using namespace std;

**double warming = 0.3**; // warming defined

void update(double dt);

void local();

int main()

{

cout << "Global warming is " << warming << " degrees.\n";

update(0.1);

cout << "Global warming is " << warming << " degrees.\n";

local();

cout << "Global warming is " << warming << " degrees.\n";

return 0;

}

// **support.cpp**

#include <iostream>

**extern double warming;**

void update(double dt);

void local();

using std::cout;

void update(double dt)

{

**extern double warming;** // **optional re-declaration, 仍然是同一个**

warming += dt; // uses global warming

cout << "Updating global warming to " << warming;

cout << " degrees.\n";

}

void local()

{

**double warming** = 0.8; // new variable hides external one

cout << "Local warming = " << warming << " degrees.\n";

// **Access *global* variable with the scope resolution operator**

cout << "But global warming = " << ::warming;

cout << " degrees.\n";

}

Here is the output from the program:

Global warming is 0.3 degrees.

Updating global warming to 0.4 degrees.

Global warming is 0.4 degrees.

Local warming = 0.8 degrees.

But global warming = 0.4 degrees.

Global warming is 0.4 degrees.

The update() function re-declares the warming variable by using the keyword extern. This keyword means “Use the variable by this name previously defined externally.” Because that is what update() would do anyway if you omitted the entire declaration, this declaration is optional. It serves to document that the function is designed to use the external variable.

C++ goes a step beyond C by offering the **scope-resolution operator (::)**. When it is prefixed to the name of a variable**, this operator means to use the global version of that variable**. Thus, local() displays warming as 0.8, but it displays ::warming as 0.4.

The external storage class is particularly suited to representing constant data because you can use the keyword const to protect the data from change:

const char \* const months[12] =

{

"January", "February", "March", "April", "May",

"June", "July", "August", "September", "October",

"November", "December"

};

In this example, the **first const** protects the **strings from change**, and the **second const** makes sure that each **pointer in the array remains pointing to the same string to which it pointed initially**. 每一个””中的string本身地址是不会变的.

**Static Duration, Internal Linkage**

What if you want to use the same name to denote different variables in different files? Can you just omit the extern?

// file1

int errors = 20; // external declaration

...

---------------------------------------------

// file2

int errors = 5; // known to file2 only? **No!**

No, this attempt fails because it **violates the one definition rule**. The file2 definition attempts to create an external variable, so **the program winds up with two definitions of errors**, which is an error.

But if a file declares a **static external** variable that has the same name as an ordinary external variable declared in another file, **the static version is the one in scope for that file**:

// file1

int errors = 20; // external declaration

...

---------------------------------------------

// file2

static int errors = 5; **// known to file2 only**

void froobish()

{

cout << errors; // uses errors defined in file2

...

This doesn’t violate the one definition rule because the keyword static establishes that the identifier errors has internal linkage, so no attempt is made to bring in an external definition.

In a multifile program, you can define an external variable in one and only one file. All other files using that variable have to declare that variable with the *extern* keyword.

You can use an external variable to share data among different parts of a multi-file program. You can use a static variable with internal linkage to share data among functions found in just one file. Also if you make a file-scope variable static, you needn’t worry about its name conflicting with file-scope variables found in other files.

**Static Storage Duration, No Linkage**

You create such a variable by applying the static modifier to a variable defined inside a block. This means that even though the variable is known within that block, it exists even while the block is inactive. Thus a static local variable can preserve its value between function calls. Also if you initialize a static local variable, the program initializes the variable once, when the program starts up. Subsequent calls to the function don’t reinitialize the variable the way they do for automatic variables.

// **List9-9.cpp** -- using a static local variable

#include <iostream>

const int ArSize = 10;

void strcount(const char \* str);

int main()

{

using namespace std;

char input[ArSize];

char next;

cout << "Enter a line:\n";

**cin.get**(input, ArSize);

cout << "After 1st cin.get.\n";

while (cin)

{

**cin.get**(next);

cout << "next before while is :" << next << endl;

while (next != '\n') // string didn't fit!

{

**cin.get**(next); **// dispose of remainder**

cout << "next in while is :" << next << endl;

}

strcount(input);

cout << "Enter next line (empty line to quit):\n";

cin.get(input, ArSize);

}

cout << "Bye\n";

return 0;

}

void strcount(const char \* str)

{

using namespace std;

static int total = 0; // static local variable

int count = 0; // automatic local variable

cout << "\"" << str <<"\" contains ";

while (\*str++) // go to end of string

count++;

total += count;

cout << count << " characters\n";

cout << total << " characters total\n";

}

自己的测试结果是：

momo@HMI:~/C++PrimerPlus/Chapter9$ ./List9-9

Enter a line:

***ABCDEFGHIJKLMN(输入回车)***

After 1st cin.get.

next before while is :J

next in while is :K

next in while is :L

next in while is :M

next in while is :N

next in while is :

"ABCDEFGHI" contains 9 characters

9 characters total

Enter next line (empty line to quit):

***123456789123456(输入回车)***

next before while is :1

next in while is :2

next in while is :3

next in while is :4

next in while is :5

next in while is :6

next in while is :

"123456789" contains 9 characters

**18** characters total

Enter next line (empty line to quit):

***(输入回车)***

Bye

**Note that because the array size is 10, the program does not read more than nine characters per line**. Recall that the *cin.get(input,ArSize)* input method reads up to the end of the line or up to ArSize - 1 characters, whichever comes first. ***It leaves the newline character in the input queue***. This program uses cin.get(next) to read the character that follows the line input. If next is a newline character, then the preceding call to cin.get(input, ArSize) must have read the whole line. If next isn’t a newline character, there are more characters left on the line. This program then uses a loop to reject the rest of the line, but you can modify the code to use the rest of the line for the next input cycle. **The program also uses the fact that attempting to read an empty line with get(char \*, int) causes cin to test as *false***.

**More About *const***

In C++ (but not C), the ***const*** modifier alters the default storage classes slightly. Whereas a **global variable has external linkage by default**, a ***const global variable* has internal linkage by default**. That is, C++ treats a **global const definition**, such as in the following code fragment, as if the **static** specifier had been used:

const int fingers = 10; **// same as static const int fingers = 10;**

int main(void)

{

因为global const等同于static, 所以在许多文件中需要共享的const变量应该放在头文件里.

After the preprocessor **includes the header file contents in each source file**, each source file will contain definitions like this:

const int fingers = 10;

const char \* warning = "Wak!";

**If global const declarations had external linkage as regular variables** do, this would be an error because of the one definition rule.

**Internal linkage also means that each file gets its own set of constants** rather than sharing them. Each definition is private to the file that contains it. This is why it’s a good idea to put constant definitions in a header file. That way, as long as you include the same header file in two source code files, they receive the same set of constants.

If, for some reason, you want to make a constant have external linkage, you can use the extern keyword to override the default internal linkage:

extern const int states = 50; // definition with external linkage

You then must use the extern keyword to declare the constant in all files that use the constant. This differs from regular external variables, in which you don’t have to use the keyword extern when you define a variable, but you use extern in other files using that variable.

**When you declare a *const* within a function or block**, it has block scope, which means the constant is usable only when the program is executing code within the block. This means that you can create constants within a function or block and not have to worry about the names conflicting with constants defined elsewhere.

**Functions and Linkage**

C++, like C, does not allow you to define one function inside another, so all **functions automatically have *static storage*** duration, meaning they are all present as long as the program is running.

By default, functions have external linkage, meaning they can be shared across files. You can, in fact, use the keyword extern in a function prototype to indicate that the function is defined in another file, but that is optional. You can also use the keyword static to give a function internal linkage, confining its use to a single file.

Inline functions are excepted from this rule to allow you to place inline function definitions in a header file. Thus, each file that includes the header file ends up having the inline function definition.

**Language Linking**

由于C和C++对函数有不同的命名方式, suppose you want to use a precompiled function from a C library in a C++ program? For example, suppose you have this code:

spiff(22); // want spiff(int) from a C library

Its hypothetical symbolic name in the C library file is \_spiff, but for our hypothetical linker, the C++ look-up convention is to look for the symbolic name \_spiff\_i. To get around this problem, you can use the function prototype to indicate which protocol to use:

extern "C" void spiff(int); // **use C protocol for name look-up**

extern void spoff(int); // use C++ protocol for name look-up

extern "C++" void spaff(int); // use C++ protocol for name look-up

**Storage Schemes and Dynamic Allocation**

Dynamic memory is controlled by the ‘*new’* and ‘*delete’* operators, **not by scope and linkage rules**. Thus, **dynamic memory can be allocated from one function and freed from another function.** Although the storage scheme concepts don’t apply to dynamic memory, they do apply to automatic and static pointer variables used to keep track of dynamic memory.

int \*pi = new int (6); // \*pi set to 6

double \* pd = new double (99.99); // \*pd set to 99.99

struct where {double x; double y; double z;};

where \* one = new where {2.5, 5.3, 7.2}; // C++11

int \* ar = new int [4] {2,4,6,7}; // C++11

int \*pin = new int {6}; // \*pin set to 6

double \* pdo = new double {99.99}; // \*pdo set to 99.99

**new: Operators, Functions, and Replacement Functions**

void \* operator new(std::size\_t); // used by new

void \* operator new[](std::size\_t); // used by new[]

void operator delete(void \*);

void operator delete[](void \*);

**int \* pi = new int**;

gets translated into something like this:

int \* pi = new(sizeof(int));

And the statement

**int \* pa = new int[40];**

gets translated into something like this:

int \* pa = new(40 \* sizeof(int));

Similarly,

delete pi;

invokes the following function call:

delete (pi);

后面还讲了placement new, 就是在一个已有的空间里用new获取, 用到时再看.

* **Namespaces**

When you use class libraries from more than one source, you can get name conflicts. For example, two libraries might both define classes named List, Tree, and Node, but in incompatible ways. You might want the List class from one library and the Tree from the other, and each might expect its own version of Node. Such conflicts are termed ***namespace problems***.

The following code, for example, uses the new keyword namespace to create two namespaces, **Jack** and **Jill**:

namespace **Jack**

{

double pail; // variable declaration

void fetch(); // function prototype

int pal; // variable declaration

struct Well { ... }; // structure declaration

}

namespace **Jill**

{

double bucket(double n) { ... } // function definition

double fetch; // variable declaration

int pal; // variable declaration

struct Hill { ... }; // structure declaration

}

**Namespaces are open, meaning that you can add names to existing namespaces**. For example, the following statement adds the name goose to the existing list of names in Jill:

namespace **Jill**

{

char \* goose(const char \*);

}

Similarly, the original Jack namespace provides a prototype for a fetch() function. You can provide the code for the function later in the file (or in another file) by using the Jack namespace again:

namespace **Jack**

{

void fetch()

{

...

}

}

Of course, you need a way to access names in a given namespace. The simplest way is to use **::**, the scope-resolution operator, to *qualify* a name with its namespace:

Jack::pail = 12.34; // use a variable

Jill::Hill mole; // create a type Hill structure

Jack::fetch(); // use a function

***using* Declarations and *using* Directives**

这里的没看，用到时再看.