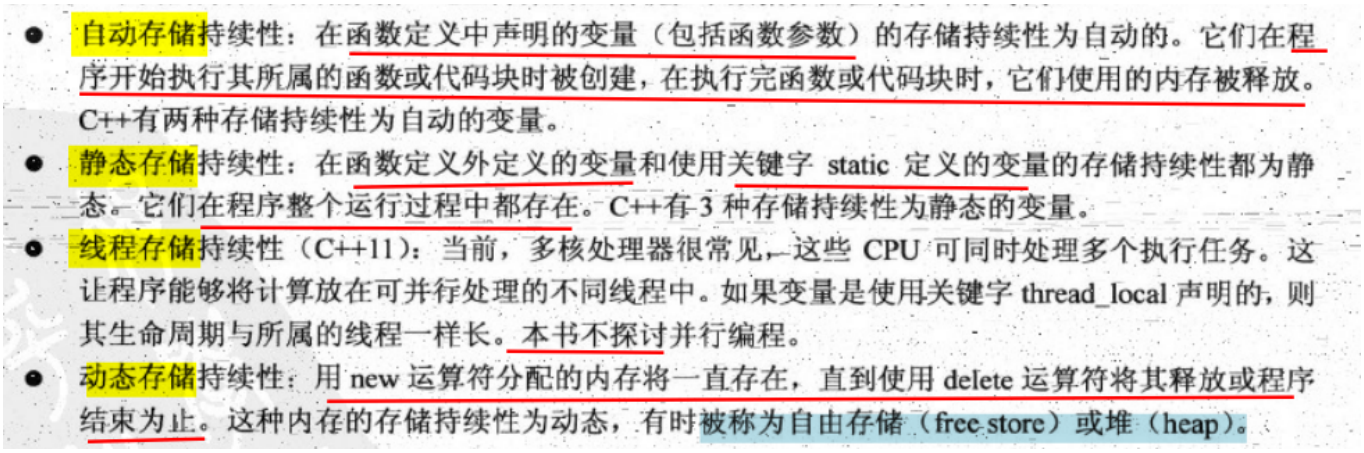
chapter9 内存模型和名称空间

1）单独编译：

理论知识见书，实战方法不会

2）存储持续性、作用域和链接性：

【1】c++中采取了四种不同的方案存储数据，它们区别在于：数据保留在内存中的时间



- 自动存储持续性：在函数定义中声明的变量（包括函数参数）的存储持续性为自动的。它们在程序开始执行其所属的函数或代码块时被创建，在执行完函数或代码块时，它们使用的内存被释放。C++有两种存储持续性为自动的变量。
- 静态存储持续性：在函数定义外定义的变量和使用关键字 static 定义的变量的存储持续性都为静态。它们在程序整个运行过程中都存在。C++有 3 种存储持续性为静态的变量。
- 线程存储持续性（C++11）：当前，多核处理器很常见，这些 CPU 可同时处理多个执行任务。这让程序能够将计算放在可并行处理的不同线程中。如果变量是使用关键字 thread_local 声明的，则其生命周期与所属的线程一样长。本书不探讨并行编程。
- 动态存储持续性：用 new 运算符分配的内存将一直存在，直到使用 delete 运算符将其释放或程序结束为止。这种内存的存储持续性为动态，有时被称为自由存储（free store）或堆（heap）。

【2】作用域与链接

（1）作用域(scope)：名称在文件的多大范围内可见

> [1] 函数中定义的变量可在该函数中使用，但是不能在其他函数中使用
> [2] 在文件的函数定义之前定义的变量，可以在所以函数中使用
> [3] 类中声明的成员的作用域是：整个类
> [4]名称空间中声明的变量的作用域是：整个名称空间
> [5]**自动变量（局部作用域变量）** 作用域是局部，静态变量作用域是全局/局部（取决于它的定义）

（2）链接性(linkage)：名称如何在不同单元之间共享

> [1] 链接性是外部的名称可以在文件之间共享
> [2] 链接性是内部的名称只能由一个文件中的函数共享
> [3] **自动变量（局部作用域变量）** 名称没有链接性，它们不可以共享

【3】自动存储持续性

> [1] 默认情况下：函数中声明的函数参数和变量的存储持续性为自动，作用域为局部，没有链接性
> eg：如果在main()中声明了一个名为hbx的变量，在后续某函数wow()内也声明了一个名为hbx的变量，则创建了两个独立的变量:
> 对wow()中的hbx执行任何操作都不会影响main()中的hbx，反之亦然!
> ps：自动变量：执行到代码块时，将为变量分配内存（but：作用域的起点是其声明位置）

> [2] 代码块中定义了变量，则变量的存在时间和作用域将被限制在该代码块内
> eg：外部定义了变量teledeli，内部代码块定义的名称也是teledeli
> 这种情况下：程序将执行内部代码块的语句，将teledeli解释为局部代码块变量；当程序离开该代码块时，原定义又重新可见
> called：新的定义隐藏了以前的定义（hide）

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int teledeli = 6;
    {
        cout << "hello world!" <<endl;
        int teledeli = 3;
        cout << teledeli << endl;
```

```
    }
    cout << teledeli << endl;
    return 0;
}

结果:
hello world!
3
6
```

[3] 程序分析:

```
// autoscp.cpp -- illustrating scope of automatic variables
#include <iostream>
void oil(int x);
int main()
{
    using namespace std;
    int texas = 31;
    int year = 2011;

    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;

    oil(texas);
    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;
    // cin.get();
    return 0;
}

void oil(int x)
{
    using namespace std;
    int texas = 5;
    cout << "In oil(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In oil(), x = " << x << ", &x = ";
    cout << &x << endl;
    {                               // start a block
        int texas = 113;
        cout << "In block, texas = " << texas;
        cout << ", &texas = " << &texas << endl;
                cout << "In block, x = " << x << ", &x = ";
        cout << &x << endl;
    }                               // end a block
    cout << "Post-block texas = " << texas;
    cout << ", &texas = " << &texas << endl;
}

result:
In main(), texas = 31, &texas = 0x61fe1c
In main(), year = 2011, &year = 0x61fe18

In oil(), texas = 5, &texas = 0x61fddc      // 函数内部变量，遵循内部
In oil(), x = 31, &x = 0x61fdf0             // 实参texas值传递给x，因此x=31，但是地址不是texas的

In block, texas = 113, &texas = 0x61fdd8    // 模块内的hide原则
In block, x = 31, &x = 0x61fdf0             // 对于模块{}而言是“全局”
Post-block texas = 5, &texas = 0x61fddc     // 当程序离开该代码块时，原定义又重新可见
In main(), texas = 31, &texas = 0x61fe1c
In main(), year = 2011, &year = 0x61fe18
```

[4] 自动变量与栈(stack):
程序必须在运行时对自动变量进行管理，常用的方法是空出一段内存，并将其视为栈（新数据象征性地放在原有数据的上面；程序使用完后，将其从栈中删除；栈的默认长度取决于实现）

[5] 寄存器变量:
关键字 **register**

```
register int hbx;  // request for a register variable!

=> 旨在提高访问变量的速度！
```

【4】静态持续变量

[1] c++为静态存储持续性变量提供了3种链接性：
（1）外部链接性：可在其他文件中访问
（2）内部链接性：只能在当前文件中访问
（3）无链接性：只能在当前函数/代码块中访问

[2] 创建方式：

> [2.1] 外部链接性变量：在代码块外面声明它
> [2.2] 内部链接性变量：在代码块外面声明它，并且加上 **static 限定符**
> [2.3] 无链接性变量：在代码块内部声明它，并且加上 **static 限定符**
>
> ask：函数内部声明的“无链接性变量”作用域是局部，没有链接性，它就像自动变量一样，二者有何区别？
> 无链接性变量：在该函数没有被执行时，也留在内存中！
> 自动变量：执行到代码块时，将为变量分配内存（but：作用域的起点是其声明位置）

[3] 静态变量的初始化：
（1）静态初始化：零初始化 + 常量表达式初始化 （编译器处理文件时初始化）
（2）动态初始化： （在编译后初始化）

```
实例分析执行顺序：
#include<cmath>
int x;                                  // zero-initialization
int y = 5;                              // constant-expression initialization
long z = 13 * 13;                       // constant-expression initialization
const double pi = 4.0 * atan(1.0);      // dynamic initialization

[1] x,y,z,pi被零初始化
[2] 编译器计算常量表达式，并将y和z分别初始化成5和169
[3] 但要初始化pi，必须调用函数 atan() ，这需要等到函数被：链接且程序执行时

poi：常量表达式并非只能使用字面常量的算数表达式
eg：可以用 sizeof() 运算符
        int hbx = 2 * sizeof(long) + 1;
```

【5】静态持续性 + 外部链接性（外部链接性变量：在代码块外面声明它）

（1）外部变量是在函数外部定义的，因此对所有函数而言都是外部的，故**外部变量**也叫作**全局变量**

（2）单定义规则：变量只能有一次定义

> 【1】c++ 提供了两种变量声明
> [1] 定义声明（也叫作：定义，called：**definition**）：它给变量分配存储空间
> [2] 引用声明（called：**declaration**）：它不给变量分配存储空间，因为它引用已有的变量

> 【2】使用方式：
> **引用声明：使用关键字extern 且 不进行初始化**
> 否则，声明为定义，导致内存为其分配空间
> eg：

```
double up;            // definition , up is 0
extern int blem;      // declaration: blem is defined elsewhere
extern char gr = 'z'; // definition:  because initialized
```

> 【3】多文件编译时要点：
> 如果要在多个文件中使用外部变量，只需在一个文件中包含该变量的定义，但在使用该变量的所有其他文件中，都必须使用关键字extern声明它

```
// file01.cpp
extern int cats = 20;  // definition: because of initialization
int dogs = 22;         // definition: dogs is 22
int fleas;             // definition: fleas is 0 (automatically)

// file02.cpp
```

```
extern int cats;      // declaration
extern int dogs;      // declaration

// file03.cpp
extern int cats;      // declaration
extern int dogs;      // declaration
extern int fleas;     // declaration
```

ps：单定义规则并非意味着不能有多个变量的名称相同，eg：在 不同的函数 中声明的 同名自动变量 是彼此独立的，它们都有自己的地址

> 虽然程序中可包含多个同名的变量，但是每个变量只能有一个定义

【6】静态持续性 + 内部链接性（内部链接性变量：在代码块外面声明它，并且加上 static 限定符）

（1）知识：如果文件定义了一个 **静态外部变量** ，其名称与另一个文件中声明的 **常规外部变量** 相同，则在该文件中，静态变量将隐藏常规外部变量

（2）问题分析：

```
// file1
int errors = 20; // external declaration

// file2
int errors = 5;  // problem!!!
void f()
{
    cout << errors <<endl;  // fails
    ......
}
```

这种作法将失败！file2的定义试图创建一个外部变量，因此程序将包含errors的两个定义，这是错误！

（3）改进方法：

```
// file1
int errors = 20; // external declaration

// file2
static int errors = 5;  // known to file2 only
void f()
{
    cout << errors <<endl;  // use "errors" defined in file2
    ......
}
```

关键字static指出errors的链接性为内部，不需要提供外部定义

【7】静态存储持续性 + 无链接性（无链接性变量：在代码块内部声明它，并且加上 static 限定符）

无链接性的局部变量：该变量只在该代码块中可用，但是它在该代码块不处于活动状态时仍存在

【8】函数和链接性

（1）基本规则与使用方法：

> 类比上述，略之！

（2）对于内联函数：

> 内联函数不受这些约束，这使得 **内联函数的定义** 可以被放入头文件中

【9】存储方案和动态分配

具体过程见书即可，现阶段不需要掌握太多，搞清楚概念名词即可！

这里重点介绍下**初始化**：

```
[1] 标量类型【常规化处理】
int *pi = new int (6);              // *pi set to 6
double *pd = new double (99.99);    // *pd set to 99.99
```

```
--------------------------------------------------------------------------
[2]初始化结构体or数组【初始化列表】

1) 结构体:
struct hbx {
        double x;
        double y;
        double z;
};
hbx *one = new hbx {2.5, 1.9, 8.0};

2) 数组:
int *arr = new int[4] {1, 2, 3, 4};
--------------------------------------------------------------------------
[3]单值元素【初始化列表】
int *pi = new int {6};       // *pi set to 6
int *pd = new double{99.99}; // *pd set to 99.99
```

3）名称空间：

（1）传统的 c++ 名称空间

【1】声明区域：可以在其中进行声明的区域

> eg1: 在函数外面声明全局变量，其声明区域是声明所在的文件
> eg2: 在函数中声明的变量，其声明区域是其声明所在的代码块

【2】潜在作用域：从声明开始，到声明区域的结尾

> hence：潜在作用域比声明区域小，这是由于变量必须定以后才能使用
> poi：变量并非在其潜在作用域内的任何位置都是可见的，比如：它可能被另一个嵌套声明区域中声明的同名变量隐藏

（2）新的名称空间特性

【1】一个名称空间中的名称不会和另一个名称空间的相同名称发生冲突

```
namespace Jack {
    int pail;
    void fetch();
    struct node (......);
}

namespace Jill {
    int pail;
    void fetch();
    struct node (......);
}
```

【2】名称空间：1）用户自定义的；2）全局名称空间 (global namespace)

【3】名称空间是开放(open)的，即：可以把名称加入到已有的名称空间

【4】访问 **给定名称空间中的名称** 的方式：作用域解析运算符 : :

```
Jack :: pail = 12;  // use a variable
Jill :: node mole;  // create a type node_structure
Jack :: fetch();    // use a function
```

（3）using 声明

【1】组成：被限定的名称 + 关键字 using

```
using Jill :: fetch; // a using declaration
```

【2】在函数内使用 using 声明：实现 (local namespace) 名称替代

```
namespace Jill {
        double bucket (double n) {......};
        double fetch;
        struct Hill {......};
}

char fetch;     // global variable
int main()
{
    using Jill :: fetch;  // put fetch into local namespace
    double fetch;         // ERROR! it tries to set up a local name, however! Already have a local fetch!
    cin >> fetch;         // read a value into Jill::fetch
    cin >> ::fetch;       // read a value into global fetch
}

(1) using Jill::fetch 将 fetch 添加到 main() 定义的声明区域内。完成该声明后，便可以用 fetch 代替 Jill::fetch
(2) using 的局部声明避免了将另一个局部变量也命名成fetch
(3) fetch 作为局部变量，也会覆盖同名的全局变量
```

【3】在函数外使用 using 声明：实现 (global namespace) 名称替代

```
void other();
namespace Jill {
    double bucket (double n) {......};
        double fetch;
        struct Hill {......};
}

using Jill :: fetch;    // put fetch into global namespace
int main()
{
        cin >> fetch;       // read a value into Jill::fetch
        other();
        ......
}

void other()
{
        cout << fetch <<endl;  // display Jill::fetch
        ......
}
```

（4）using 编译指令

【1】组成：关键字 using namespace + 名称

```
using namespace Jack; // make all the names in Jack available
```

【2】在全局声明区域中使用 using 编译指令，将使该名称空间的名称全局可用

```
#include<iostream>
using namespace std; // making names available globally
```

【3】在函数中使用 using 编译指令，将使其中的名称在该函数中可用

```
#include<iostream>
int main()
{
    using namespace Jack; // making names available in main()
    ......
}
```

【4】实例分析：

```
namespace Jill {
        double bucket(double n) {...}
        double fetch;
        struct Hill {...};
```

```
}

char fetch;                    // global namespace

int main()
{
    using namespace Jill;      // import all namespace names
    Hill Thrill;               // create a type Jill::Hill structure
    double water = bucket(2);  // use Jill::bucket()
    double fetch;              // 局部声明的fetch会(local name)隐藏Jill::fetch 和 全局fetch
    cin >> fetch;              // read a value into the local fetch
    cin >> ::fetch;            // read a value into the global fetch
    cin >> Jill :: fetch;      // read a value into Jill::fetch
    ......
}

int foom()
{
    Hill top;              // Error!
    Jill :: Hill crest;    // valid!
}
```

在 main( )中，名称 Jill::fetch 被放在局部名称空间中，但其作用域不是局部的，因此不会覆盖全局的 fetch。然而，局部声明的 fetch 将隐藏 Jill::fetch 和全局 fetch。然而，如果使用作用域解析运算符，则后两个 fetch 变量都是可用的。读者应将这个示例与前面使用 using 声明的示例进行比较。

需要指出的另一点是，虽然函数中的 using 编译指令将名称空间的名称视为在函数之外声明的，但它不会使得该文件中的其他函数能够使用这些名称。因此，在前一个例子中，foom( )函数不能使用未限定的标识符 Hill。

point:

> [1] using编译指令 和 using声明：它们增加了名称冲突的可能性

```
1) 标识符
jack :: pal = 3;
jill :: pal = 6;
                           valid!

2) using声明
using jack::pal;
using jill::pal;
pal = 4;
                           which one? now have a conflict!
```

> [2]综合说明：
>
> 一般说来，使用 using 声明比使用 using 编译指令更安全，这是由于它只导入指定的名称。如果该名称与局部名称发生冲突，编译器将发出指示。using 编译指令导入所有名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

（5）名称空间的其他特性

可以将名称空间声明进行嵌套：

```
namespace elements
{
    namespace fire
    {
        int flame;
        ...
    }
    float water;
}
```

这里，flame 指的是 element::fire::flame。同样，可以使用下面的 using 编译指令使内部的名称可用：

```
using namespace elements::fire;
```

另外，也可以在名称空间中使用 using 编译指令和 using 声明，如下所示：

```
namespace myth
{
    using Jill::fetch;
    using namespace elements;
    using std::cout;
    using std::cin;
}
```

假设要访问 Jill::fetch。由于 Jill::fetch 现在位于名称空间 myth（在这里，它被叫做 fetch）中，因此可以这样访问它：

```
std::cin >> myth::fetch;
```

当然，由于它也位于 Jill 名称空间中，因此仍然可以称作 Jill::fetch：

```
Jill::fetch:
std::cout << Jill::fetch;  // display value read into myth::fetch
```

如果没有与之冲突的局部变量，则也可以这样做：

```
using namespace myth;
cin >> fetch;       // really std::cin and Jill::fetch
```

现在考虑将 using 编译指令用于 myth 名称空间的情况。using 编译指令是可传递的。如果 A op B 且 B op C，则 A op C，则说操作 op 是可传递的。例如，>运算符是可传递的（也就是说，如果 A>B 且 B>C，则 A>C）。在这个情况下，下面的语句将导入名称空间 myth 和 elements：

```
using namespace myth;
```

这条编译指令与下面两条编译指令等价：

```
using namespace myth;
using namespace elements;
```

可以给名称空间创建别名。例如，假设有下面的名称空间：

```
namespace my_very_favorite_things { ... };
```

则可以使用下面的语句让 mvft 成为 my_very_favorite_things 的别名：

```
namespace mvft = my_very_favorite_things;
```

可以使用这种技术来简化对嵌套名称空间的使用：

```
namespace MEF = myth::elements::fire;
using MEF::flame;
```

（6）未命名的名称空间

可以通过省略名称空间的名称来创建未命名的名称空间：

```
namespace        // unnamed namespace
{
    int ice;
    int bandycoot;
}
```

这就像后面跟着 using 编译指令一样，也就是说，在该名称空间中声明的名称的潜在作用域为：从声明点到该声明区域末尾。从这个方面看，它们与全局变量相似。然而，由于这种名称空间没有名称，因此不能显式地使用 using 编译指令或 using 声明来使它在其他位置都可用。具体地说，不能在未命名名称空间所属文件之外的其他文件中，使用该名称空间中的名称。这提供了链接性为内部的静态变量的替代品。例如，假设有这样的代码：

```
static int counts;   // static storage, internal linkage
int other();
int main()
{
...
}

int other()
{
...
}
```

采用名称空间的方法如下：

```
namespace
{
    int counts;   // static storage, internal linkage
}
int other();
int main()
{
...
}

int other()
{
...
}
```