第 3 章　C++变量的访问属性

如果把任何一个数据都称为对象，那么在 C++中，对象与其名字是在编译时被绑定在一起的，并将这个绑定体称为变量。变量有名字、类型、存储空间（地址）这样 3 个基本存在属性。还有生命周期、作用域、连接性和名字空间这样 4 个访问属性。

生命周期（lifetime）也称数据存储的持续性，指数据在内存中保留的时间长短。显然，只有在生命周期中的数据才是可以被访问的。

作用域（scope）指名字在编译单元（文件）中的哪个范围内可见——可访问。可访问与生命周期有些情况下是一致的，但也可能不一致，这说明，在生命周期中的变量不一定在任何代码区间都是可以访问的。

连接性（linkage）指可以被不同单元共享的能力。

名字空间（namespace）决定了当有两个同样的名字出现时，如何根据命名体系进行区分，以便正确地访问。当然，名字空间并非仅仅针对变量，它们还针对一切标识符（包括保留字）。

3.1　C++变量的生命周期与存储分配

3.1.1　C++变量的生命周期

变量的生命周期即变量从创建（分配存储空间——名字与存储空间相绑定）到撤销（名字与存储空间相分离）中间的一段时间。在程序中，变量的生存期有以下 3 种。

（1）自动生命周期也称临时生命周期。具有这种生命周期的变量往往被用于程序的一个局部，即在程序的某个域中需要时才用定义语句创建，这个域结束时自动撤销。因此，除非特别声明，一般定义（声明）在语句块（函数也是语句块）中的变量都具有自动生命周期。

（2）静态生命周期也称永久生命周期。具有这种生命周期的变量在编译时就被分配了存储空间并初始化。其生命周期是从程序运行开始到程序运行结束。在程序中，凡是声明（定义）在函数外部的实体（变量、函数、类等）都具有静态生命周期。具有静态生命周期的变量称为静态变量或静态持续变量。静态变量的一个特点是声明时需要初始化，若无显式初始化，编译器会以默认值——将其所有 bit 位都设置为 0，称为零初始化（zero-initiaized）。

（3）动态生命周期，也称自由生命周期、可控生命周期。具有这种生命周期的变量可被用于程序的一个局部，但与域无关，其创建与撤销完全由程序员掌控。

显然，不管哪一种生命周期都与存储分配有关。所以也有人把变量的生存期称为存储的持续性。

3.1.2 C++存储分配

为了管理方便，编译器将 3 种生命周期的变量分配在不同的存储区中。图 3.1 所示为典型的 C++程序存储空间布局。它分为 5 个部分。

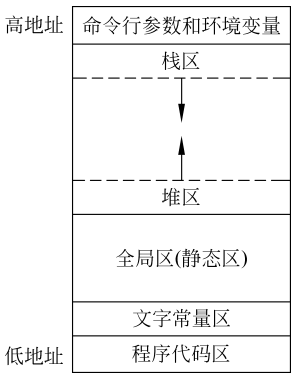


图 3.1 典型 C++程序存储空间布局

1. 全局区（静态区）

全局区（静态区，static area）也称永久区，存放永久生命周期的变量，也称静态持续变量或静态变量。这些静态变量的生命周期从程序运行开始被创建，直到程序运行结束时才被操作系统撤销。由于这个区的变量在程序运行期间不变，所以这个区是一个在程序运行期间不需要进行存储分配管理的区间。

2. 栈区

栈（stack）是一种先进后出或后进先出式的存储管理方式。栈区（stack area）用于存储运行函数所需要的局部变量、函数参数、返回数据、返回地址等。程序运行到一个变量声明时，才开始为其分配存储空间，并按照分配的先后顺序压入栈中；当程序运行到该变量所在的块结束时，编译器按照先进后出的原则执行弹出操作，自动释放该块内的局部变量所占用的系统资源。由于这些变量只生存在程序的某个局部运行期间，因此称为临时变量；这些过程是自动完成的，因此又称为自动变量；这样的生命周期是局部的，也被称为局部变量。局部变量在定义时如果不显式初始化，则值是一个未知数。

3. 堆区

堆区（heap area）也称自由存储区（free store area），用于存放由程序员分配并释放的变量，即它们可以由程序员需要时用一个操作符创建，不再需要时用一个回收操作符回收，它们的生命完全掌握在程序员手中，由程序员根据需要动态地掌控，不受块域的影响，所以它们的生命周期是可控的、动态的。如果程序员没有释放，程序结束时可能由操作系统回收。

堆与栈公用一个堆栈区，分别从这个区的两端开始分配。

4. 文字常量区

常量存储区是一块比较特殊的存储区，其中存放的是字符串常量，不允许修改，程序结束后由系统释放。

5. 程序代码区

程序代码区（code segment/text segment）用于存放函数体（类成员函数和全局区）的二

进制代码。这些代码具有只读属性，可以共享。有些编译器把代码段与文字常量区放在一起，也称为码段。

3.1.3 动态内存分配

动态存储区也称堆（heap）区或自由存储区，是供程序员在程序运行期间进行动态分配的区间。程序员可以在需要时分配，不再需要时回收，与代码块无关。在 C++中，用一元操作符 new 进行动态存储分配，用 delete 回收动态分配的内存空间。

1. 用 new 进行动态内存分配

用 new 进行动态内存分配，需要下面两个参数。

（1）类型：决定所分配的内存空间以哪种类型单位为一个单位。

（2）数量：分配的内存空间为多少个分配单位，用方括号中的正整数表示。默认为 1 个单位。

分配成功，将返回一个指针——所分配空间的地址。这个地址是堆空间的一个地址，由操作系统给出。例如

```
int *ptrInt = new int;                  //分配 1 个 int 空间，地址赋给 ptrInt
int * ptrInt = new int [10];            //分配 10 个 int 空间，地址赋给 ptrInt
double *ptrDouble = new double;         //分配 1 个 double 空间，地址赋给 ptrDouble
double *ptrDouble = new double [3];     //分配 3 个 double 空间，地址赋给 ptrDouble
```

在分配存储空间的同时，还可以进行初始化。初始值放在圆括号内。若圆括号内为空，则初始化为默认值。若没有圆括号，则初始值不可知。例如

```
int *p5 = new int(5);                   //初始化为 5
int *p0 = new int();                    //初始化为 0
int *px = new int;                      //初始值不可预测
int *pd = new int[2]{3,5};              //分配两个 int 空间，分别初始化为 3 和 5
```

2. 用 delete 释放动态存储空间

当用 new 动态分配的存储空间不再使用时，应及时用操作符 delete 释放回收，否则就会造成内存空间的浪费。例如

```
delete ptrInt;                          //释放 ptrInt 所指向的动态存储空间
delete ptrDouble;                       //释放 ptrDouble 所指向的动态存储空间
```

使用 delete 应注意以下几点。

（1）delete 也是一个一元操作符，它作用于指向一个动态存储空间的指针，使所指向的地址不再有效，即释放这个指针所指向的动态存储空间。

（2）用 delete 释放一个指针所指向的堆空间，一定要是由 new 为该指针分配的堆空间。不可用来释放没有用 new 分配过堆空间的指针所指向的空间。否则，将产生不可预料的错误。也就是说，new 和 delete 是相互配合使用的。

（3）释放一个指针指向的动态存储空间时，不需要考虑该空间是否已被初始化过。

（4）尽量避免多个指针指向同一对象。例如

```
int *pi1;
int *pi2;
…
pi1 = new int;
pi2 = pi1;
…
```

这样，当 pi1 需要用 delete 释放时，必须同时也释放 pi2，否则 pi2 将“悬空”，对其操作将会导致内存混乱。

（5）当已经对一个指针使用过 delete，使其指向的内存释放，再次使用 delete 时，就会使运行的程序崩溃。克服的方法是将这个指针赋予 nullptr(NULL)值。这样，即使对其再次实施 delete，也可以保证程序是安全的。

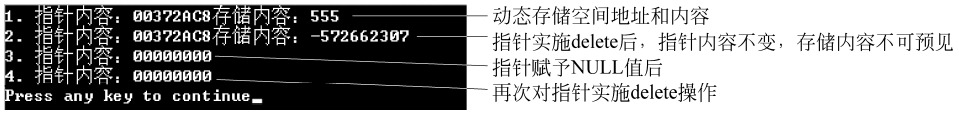
（6）也许有人认为，程序结束时所有内存都会被自动回收。这当然是真的，不过，等程序结束再回收一些本来可以再利用的存储空间，纯属亡羊补牢。在极端情况下，当所有可用的内存都用光时，再用 new 为新的数据分配存储空间就会造成系统崩溃。例如，在一个函数中使用 new，指向所分配的堆空间的指针是一个函数中的局部变量。这样，当函数返回时，指针被销毁，所指向的内存空间即被丢弃，这片内存空间将无法利用。

【代码 3-1】 用 delete 释放对象指针的例子。

```
#include <iostream>
int main() {
using namespace std;
    int *pInt;
    pInt = new int (555);
    cout << "1. 指针内容：" << pInt << "存储内容：" << *pInt << endl;
    delete pInt;
    cout << "2. 指针内容：" << pInt << "存储内容：" << *pInt << endl;
    pInt = nullptr;                  //给指针赋予 NULL 值
    cout << "3. 指针内容：" << pInt << endl;

    //cout << "存储内容：" << *pInt << endl;
    delete pInt;
    cout << "4. 指针内容：" << pInt << endl;
    return 0;
}
```

测试结果如下。



讨论：请读者测试指针经过一次 delete 操作后，不赋予 nullptr(NULL)，再实施一次 delete 时，程序运行将会出现什么情况。

在上述程序中，有一条被注释掉的语句，用于经过一次 delete 操作并赋予 nullptr(NULL)后，还想输出指针指向空间的内容。请读者考虑，如果不注释这条语句将出现什么问题。

3.2 C++标识符的作用域与连接性

3.2.1 作用域与连接性的概念

1. 标识符的声明域

标识符的声明域（declaration region）按存在结构可以形成的代码区间。按照代码范围的大小，从小到大，可以把标识符的声明域分为语句域、语句块域、函数域、类域和文件域。

（1）语句域。语句域是指标识符仅在一个语句中有效，是最小的作用域。函数原型声明具有语句域，即在函数原型声明中使用的参数名只在这个语句中有效。

（2）语句块域。语句块域即语句块作用域，简称块域，针对那些定义在用花括号括起的一组语句中、if 的子语句中、switch 的子语句中，以及循环体中的变量或对象而言，它们的作用范围仅在定义它的相应范围内，从定义时起是有效的。

（3）函数域。函数域也称为函数作用域，针对函数的形参、函数体内定义的变量或对象而言，但不包括在函数体内的语句块中定义的变量或对象。对于 C++来说，函数域也可以看成一种块域，并且是最大的块域。

（4）类域。一个类由一些成员组成，包括数据成员和成员函数。这些成员名字的作用域为所在的类（具体见第 5 章）。

（5）文件域。文件是程序进行编译的单位，所以文件域也称编译单元域。在函数和类之外定义的标识符具有文件作用域，其作用域从说明点开始，在文件结束处结束。文件作用域包含该文件中所有的其他作用域。

由于文件是编译的单位组织，因此文件域是一个特殊域：标识符可以在文件域或块域中定义，但在文件域中定义的标识符在文件所有的块中都可以使用，称之为全局变量。相对而言，在块域中定义的标识符具有局部性，称为局部变量。

在 C++程序中，以下几项属于文件域。

① 类的成员函数以外的其他函数。

② 宏名，除非文件中出现了 undef 取消定义。

③ 如果标识符出现在头文件的文件作用域中，则该标识符的作用域扩展到嵌入了这个头文件的程序文件中，直到该程序文件结束。

2. 标识符的潜在作用域

标识符的潜在作用域（potential scope）简称作用域（scope），是标识符可见、可以被使用的代码区间。潜在作用域位于声明域中，从声明点开始到声明域结束。因为，在 C++中，标识符要先定义（或声明）才可以使用。

【代码 3-2】 作用域示例。

```
#include <iostream>
int i;                                    //文件作用域
int main()
{
     using namespace std;
     i = 888;
     {
          int i = 666;                    //块作用域
          cout << "i = " << i << endl;     //输出 666
     }
     cout << "i = " << i << endl;          //输出 888
     return 0;
}
```

测试结果如下。

```
i = 666
i = 888
```

在这个程序中，最外层的 i 有文件作用域，最内层的 i 有块作用域，最内层的 i 隐藏最外层的 i，这时在最内层无法存取文件作用域的 i。使用作用域操作符::可以在块作用域中存取被隐藏的文件作用域中的名字。

【代码 3-3】 使用作用域操作符::在块域中访问具有文件域的变量。

```
#include <iostream>
int i;                                    //文件作用域
int main(){
     using namespace std;
     i = 888;
     {
          int i = 666;                    //块作用域
          cout << "i = " << ::i << endl;   //使用作用域操作符::
     }
     cout << "i = " << i << endl;
     return 0;
}
```

测试结果如下。

```
i = 888
i = 888
```

说明：可能读者已经注意到了，前面讲过，C++程序是从主函数开始执行，随主函数结束而结束，中间可以调用其他函数。那么定义全局变量的语句“int i;”是什么时间执行的呢？可以说，是编译时执行的。因为外部的定义都是静态的。

3. 局部变量与全局变量

在命令式编程中，以函数作用域为界，按作用域可以把 C++变量分为两类：局部变量和全局变量。当一个变量声明在函数——实际上是所有语句块外部（函数也是语句块）时，

它就具有全局作用域——实际上是文件作用域，即它在当前文件——编译单元的全局范围内，可以被任何一个函数或语句块访问。这种变量就称为全局变量。

当一个变量声明在一个函数内部时，它就只具有局部作用域。这样的变量称为局部变量。

4. 标识符的连接性

标识符的连接性（linkage）指标识符的作用域可以被扩展的能力，或者说是标识符在不同的单元间共享的能力。按照连接性，可以把标识符分为外部连接标识符、内部连接标识符和无连接标识符。由于连接是编译的一个阶段，因此连接性的划分以文件域（编译域）为基准进行划分。

（1）外部连接性：标识符的作用域有可能被扩充到其他文件域，即可以在文件间共享。

（2）内部连接性：标识符的作用域只限于当前文件域，即只能在一个文件的不同函数间共享。

（3）无连接性：标识符的作用域只限于当前函数域或块域，没有被共享的能力。

3.2.2 全局变量与 extern 关键字

1. 全局变量的定义

全局变量也称外部变量，其作用域也称外部作用域，实际上就是文件作用域。定义全局变量的语法如下。

```
extern 类型关键字 变量名 = 初始化表达式;
```

这种定义，不仅将一个变量的作用域定义为外部的，而且将所声明的变量存储在静态存储区，所以该变量的生命周期是永久的。例如

```
extern double d = 0.0;          //合法的全局变量声明：有 extern，也有初始化
double d = 0.0;                 //合法的全局变量声明：无 extern，但有初始化
double d;                       //合法的全局变量声明：无 extern，也无初始化
```

2. 全局变量的初始化

1）全局变量的初始化缺省

当使用 extern 定义全局变量时，初始化是不可缺省的；只有 extern 缺省时，初始化才可以缺省。在初始化缺省时，编译器会自动给其进行零初始化。

2）全局变量的初始化阶段

从语言的层面来说，全局变量的初始化可以划分为以下两个阶段。

（1）静态初始化（static initialization）：指在程序加载的过程中用 0 或常量来对全局变量进行初始化。

（2）动态初始化（dynamic initialization）：指需要经过函数调用才能完成的初始化，例

如，int a = foo( )，或者是复杂类型（类）的初始化（需要调用构造函数）等。这些变量的初始化会在 main( )函数执行前由运行时调用相应的代码从而得以进行（函数内的 static 变量除外）。

需要明确的是，静态初始化执行先于动态初始化。只有当所有静态初始化执行完毕，动态初始化才会执行。显然，这样的设计是很直观的，能静态初始化的变量，它的初始值都是在编译时就能确定，因此可以直接硬编码（hard code）到生成的代码中，而动态初始化需要在运行时执行相应的动作才能进行。

3）初始化的顺序问题

初始化的顺序问题分如下几种情况考虑。

（1）静态初始化执行先于动态初始化。只有当所有静态初始化执行完毕，动态初始化才会执行。

（2）对于出现在同一个编译单元内的全局变量，它们初始化的顺序与它们声明的顺序是一致的。销毁的顺序则相反。

（3）对于不同编译单元间的全局变量，C++ 标准并没有明确规定它们之间的初始化（销毁）顺序是一种实现定义行为，即顺序完全由编译器自己决定。因此，一个比较普遍的认识是：不同编译单元间的全局变量的初始化顺序是不固定的，哪怕对同一个编译器，同一份代码来说，任意两次编译的结果都有可能不一样。因此，要避免不同编译单元间的全局变量相互引用的情况。

3. 全局变量的定义性声明与引用性声明

严格地说，定义与声明是两个不相同的概念。声明的含义更广一些，定义的含义稍窄一些，定义是声明的一种形式，定义具有分配存储的功能，凡是定义都属于声明，称为定义性声明（defining declaration）。另一种声明称为引用性声明（referencing declaration），它仅仅是对编译系统提供一些信息。总之，声明并不都是定义，而定义都是声明。

对于局部变量，声明与定义合二为一；对于全局变量，声明与定义各司其职。

在一个程序中，定义性声明只能有一个，而引用性声明可以有多个。定义性声明中可以有初始化表达式，但引用性声明中不可以有初始化表达式。

1）用 extern 引用性声明将全局变量的作用域向前扩展——连接到当前位置

对于 C++来说，全局变量具有内部连接性，当其定义位于一个文件的后部时，可以用引用性声明将其连接到前部。函数的原型声明就是一种引用性声明。相对于引用性声明，把外部类型的定义称为定义性声明。变量的引用性声明的格式如下。

```
extern 类型关键字 变量名;
```

注意：这里没有初始化部分。与引用性声明的区别是，定义性声明要么使用有关键字 extern 的，必须有初始化部分；要么省略关键字 extern，可以没有初始化部分。而引用性声明必须有 extern，并且不可以有初始化部分。

【代码 3-4】 使用引用性声明将全局变量的作用域向前扩展。

```
#include <iostream>
void gx( ),gy( );
using namespace std;
int main(){
     extern int x,y;           //引用性声明，将 x 和 y 的作用域扩充到主函数
     cout << ": x = " << x << "\t y = " <<  y << endl;
     y = 246;
     gx( );
     gy( );
     return 0;
}
void gx( ){
     extern int x, y;          //引用性声明，将 x 和 y 的作用域扩充到函数 gx()
     x = 135;
     cout"2:x = " << x << "\t y = " << y << endl;
}

int x, y;                      //定义性声明，定义 x,y 是全局变量
void gy( ){
     cout"3:x = " << x << "\t y = " << y << endl;
}
```

程序测试结果如下。

```
1: x = 0          y = 0
2:x = 135         y = 246
3:x = 135         y = 246
```

讨论：第一次输出 x=0 和 y=0，是全局变量初始化的结果（不给初值便自动赋以 0）。在执行 gx( )函数时，只对 x 赋值，没对 y 赋值，但在 main( )函数中已对 y 赋值，而 x 和 y 都是全局变量，因此可以引用它们的当前值，故输出“x=135, y=246”。同理，在函数 gy( ) 中，x 和 y 的值也是 135 和 246。

定义性声明与引用性声明除了形式不同外，全局变量的定义性声明只能有一次，但引用性声明可以有多次。

2）用 extern 引用性声明将全局变量连接到当前文件中

全局变量具有外部连接性。假设一个程序由两个以上的文件组成。当一个全局变量声明在文件 file1.cpp 中时，在另外的文件中使用 extern 声明，可以通知连接器一个信息：“此变量到外部去找”；或者说在连接时告诉连接器：“到别的文件中找这个变量的定义”。即，使用 extern 声明就可将其他源文件中定义的变量及函数连接到本源程序文件中。

【代码 3-5】 将全局变量连接到其他文件的例子。

```
/*** file1.cpp ***/
#include <iostream>
int x, y;                  //定义全局变量 x,y
```

```
char ch;                                  //定义全局变量 ch
int main()
{
     using std::cout;
x = 12;
     y = 24;
     f1( );
     cout << ch;
     return 0;
}
```

```
/*** file2.cpp ***/
extern int x,y;                           //引用性声明
extern char ch;                           //引用性声明
f1( )
{
     using namespace std;
     cout << x << "," << y << endl;       //引用全局变量
     …
     ch ='a';                             //引用全局变量
…
}
```

说明：

（1）在 file2.cpp 文件中没有声明变量 x、y、ch，而是用 extern 声明 x、y、ch 是全局变量，因此在 file1.cpp 中定义的变量在 file2.cpp 中也可以引用。x、y 在 file1.cpp 中被赋值，它们在 file2.cpp 中也作为全局变量，因此输出 12 和 24。同样，在 file2.cpp 中对 ch 赋值'a'，在 file1.cpp 中也能引用它的值。当然要注意操作的先后顺序，只有先赋值才能引用。

注意：在 file2.cpp 文件中不能再定义“自己的全局变量”x、y、ch，否则就会犯“重复定义”的错误。

（2）如果一个程序包含有若干个文件，且不同的文件中都要用到一些共用的变量，可以在一个文件中定义所有的全局变量，而在其他有关文件中用 extern 来声明这些变量即可。

（3）在 C++程序中，函数都是全局的，也可以加上 extern 修饰，将其作用域扩展到其他文件。

3.2.3 C++的 static 关键字

static 是一个非常重要的存储属性关键字，它可以用来修饰局部变量，将其生命周期延长为永久的；也可以修饰全局变量，将全局变量的外部连接性限制为内部的；还可以用于定义类的成员，使其成为该类的所有对象共享的成员。

1. 用 static 将全局变量连接性限制为内部连接性

在多文件程序中，若用 static 声明全局变量的定义，则该全局变量的连接性被限制在当前文件内部，即不能连接到其他文件；而无 static 声明的全局变量，连接性是外部的。例如，

某个程序中要用到大量函数，其中有几个函数要共同使用几个全局变量时，可以将这几个函数组织在一个文件中，并将这几个全局变量声明为静态的，以保证它们不会与其他文件中的变量发生名字冲突，保证文件的独立性。

【代码 3-6】 采取表达式 r2 = (r1 * 123 + 59) % 65536，产生一个随机数序列。只要给出一个 r1，就能在 0～65535 范围内产生一个随机整数 r2。

```
static unsigned int r;           //将全局变量的连接性变为内部的
int random()
{
    r = (r * 123 + 59) % 65536;
    return (r);
}

/*产生 r 的初值*/
unsigned randomstart(unsigned int seed) {retrun r=seed; }
```

说明：r 是一个静态全局变量，初值为 0。在需要产生随机数的函数中先调用一次 randomstart()函数以产生 r 的第一个值，然后再调用 random()函数。每调用一次 random()，就得到一个随机数。

对于一个多文件程序来说，由于每个文件可能都是由不同的人单独编写的，这难免会出现不同文件中同名但含义不同的全局变量。这时，若采用静态全局变量，就可以避免因同名而造成的尴尬局面。所以，在程序设计时最好不用全局变量，非用不可时，也要尽量优先考虑使用静态全局变量。

【代码 3-7】 extern 与 static 综合应用实例。

```
extern int a;                   //声明变量 a：外部连接
extern void f(int x);           //声明函数 f，外部连接；x：局部，无连接
static int b = 999;             //声明变量 b：全局，内部连接
int main(){
    f(a);
    f(b);
    return 0;
}
```

```
//code2.cpp
#include <iostream>
extern int a;               //声明变量 a，使其作用域向前扩展到此
void f( int b)              //定义函数 f，全局，外部连接；变量 b，局部，无连接
{
    using namespace std;
cout << "a = " << a << ",b = " << b << endl;
}
int a = 888;                //声明变量 a：全局变量，外部连接
```

执行结果如下。

```
a=888, b=888
a=888, b=999
```

说明：

（1）关键字 extern 可以将作用域从定义域延伸到声明语句所在域。

（2）函数一般具有外部连接性，所以函数声明可以用关键字 extern 修饰，也可以将关键字 extern 省略。

（3）函数也可以用 static 修饰为文件内部的，以限制外部引用。

2. 用 static 将局部变量的生命周期延长为永久——创建静态局部变量

自动变量在使用中有时不能满足一些特殊的要求，特别是在函数中定义的自动变量，会随着函数的返回被自动撤销。但是有一些问题需要函数保存中间计算结果。解决的办法是将要求保存中间值的变量声明为静态的，使其生命周期成为永久性的。

【代码 3-8】 用 static 实现一个函数不同调用时的共享。

```
#include <iostream>
void getFact(int n);
int main()
{
    getFact (3);
    return 0;
}
void getFact(int n)
{
    using namespace std;
    for (int i = 1; i <= 3; i ++) {
        static long int fact = 1;   //fact 只在函数第一次调用时初始化，在以后调用中共用
        fact *= i;
        cout << i <<"! = " << fact << endl;
    }
}
```

测试结果如下。

```
1! = 1
2! = 2
3! = 6
```

3.2.4 C++变量存储属性小结

C++中，用存储属性综合变量的作用域、生命周期和连接性，提供了如下一些存储说明符（atorage class specifier）：

auto（在 C++11 中已改类型自动判定符）；

register（寄存器存储）；

static（静态存储）；

extern（外部连接）。

表 3.1 为 C++变量存储属性小结。

表 3.1　C++变量存储属性

分类		声明位置	声明关键字	生命周期	潜在作用域	连接性
局部变量	自动变量	语句块	无/register	自动	语句块	无
	静态局部变量	语句块	static	静态	语句块	无
全局变量	静态外部变量	文件域	无/extern	静态	文件	外部
	静态内部变量	文件域	static	静态	文件	内部

3.3　C++名字空间域

3.3.1　名字空间及其创建

1. 名字空间的概念

2007 年 7 月 31 日，一个网站发布了中国 13 亿人口中重复率最高的前 50 个名字，其中张伟（290 607 人）、王伟（281 568 人）、王芳（268 268 人）、李伟（260 980 人）位居前列。众多的重名现象，在某些情况下已经成为一个令人头疼的问题。但是，对于一个家庭来说，就不会出现这个现象。

在程序中同样会出现这样的问题。随着程序规模的扩大，程序中使用的具有全局作用域的名字会越来越多，例如全局变量名、函数名、类名、全局对象名等。大规模的程序一般是多人合作编写的。每一个人在自己涉及的那部分程序中可以做到名字不重，但很难保证与别人编写的那部分程序中没有名字冲突。此外，一个程序往往还需要包含一些头文件，这些头文件中也有大量的名字，如 cout、cin、ostream、istream 等。这么多名字的程序块，其中极有可能包含与程序的全局实体同名的实体，或者不同的块中有相同的实体名。它们分别编译时不会有问题。但是，在进行连接时，就会报告出错，因为在同一个程序中有两个同名的变量，认为是对变量的重复定义。这就是名字冲突（name clash）或称全局名字空间污染（global namespace pollution）。解决名字冲突的有效方法是引入名字空间（name space）机制。

名字空间的作用是将一个程序中的所有名称规范划分到不同的集合——名字空间中，确保每个名字空间中没有任何两个相同的名字定义。否则，将会引起重定义错误。

2. 名字空间的创建

名字空间用关键字 namespace 定义，格式如下。

```
namespace 名字空间名
{
    名字定义 1
    名字定义 2
      ⋮
    名字定义 n
}           //注意后面没有分号
```

在声明一个名字空间时，花括号内不仅含有变量（可以带有初始化表达式），也能含有常量、数（可以是定义或声明）、构造体、类声明、模板及一个嵌套名字空间。

【代码 3-9】 名字空间的定义。

```
namespace zhang1                                    //名字空间定义
{
     const double  PI = 3.14159;                    //全局常量定义
     double radius = 2.0;                           //全局变量声明
     double getCircumference()                      //外部函数定义
     {return 2 * PI * radius;}

     class C{                                       //类定义
          int a;                                    //类域变量声明
          int b;                                    //类域变量定义
     public:
          C (int aa, int bb):a (aa),b (bb){}        //类域函数定义
          int disp () {return (a + b);}
     };
     namespace zhang2                               //嵌套的名字空间定义
     {int age;}
}
namespace zhang1                                    //在 zhang1 中加入新成员
{
     int d = 123;
}
```

说明：

（1）namespace 是定义名字空间所必须写的关键字，zhangl 是用户自己指定的名字空间的名字，在花括号内是声明块，在其中声明的实体称为名字空间成员（namespace member）。

（2）名字空间的成员可以是全局变量名（如 radius、age）、全局常量名（如 PI），函数名（如 getCircumference）、类名（如 C）。它们在名字空间中以声明或定义的形式加入。

（3）名字空间的成员也可以是其他名字空间名（如 zhang2）。这样就形成嵌套名字空间。

（4）名字空间是开放的，允许随时通过重新声明或定义方法把新的成员名称追加到已有的名字空间中去。例如，加入 d。

（5）可以给名字空间取个别名。例如

```
namespace abc = std;
```

之后，凡是原来使用 std 的地方都可以改用 abc 了。

（6）对于大型程序来说，名字空间定义以头文件的形式保存。对于较小的程序则可以将上述代码与其他操作写在一起，用一个程序文件存储。

3.3.2 名字空间的使用

名字空间外部的代码不能直接访问名字空间内部的元素。要在某作用域中使用其他名字空间中定义的元素，首先要将定义该元素的头文件包含在当前文件中，然后可以使用下

面的 3 种方法之一将名字空间中的元素引入当前的代码空间中。

1. 直接使用名字空间限定方式

【代码 3-10】 主函数中对每一个名字，都用域解析限定（qualified）其名字空间。

```
#include <iostream>
int main(){
     std::cout << zhang1::PI << std::endl;
     std::cout << zhang1::radius << std::endl;
     std::cout << zhang1::getCircumference () << std::endl;
     std::cout << zhang1::zhang2::age << std::endl;

     zhang1::C c1 (2,3);
     std::cout << c1.disp () << std::endl;

     return 0;
}
```

作用域解析符“::”表明所使用名字来自哪个名字空间。

2. 使用 using 声明将一个名字空间成员引入当前作用域

【代码 3-11】 在主函数中，用 using 作为关键字，声明某个名字空间中的某个名字，使其进入当前作用域，包括标准名字空间 std 中的 cout、cin、endl 等名字的应用。

```
#include <iostream>

int main() {
     using std::cout;
     using std::endl;

     using zhang1::PI;
     using zhang1::radius;
     using zhang1::getCircumference;
     using zhang1::C;                    //类名的引入
     using zhang1::zhang2::age;          //嵌套名字域的引入

     cout << PI << endl;
     cout << radius << endl;
     cout << getCircumference () << endl;

     C c1 (2,3);
     cout << c1.disp () << endl;         //成员函数的使用

     return 0;
}
```

说明：

（1）using 声明遵循作用域规则，超出了作用域就不再有效。这里，所引入的名字都是

在主函数 main()中。如果在函数外部，则将这些名字引入到全局作用域中，并且局部变量能够覆盖同名的全局变量。

对于类中的成员来说，只引入类名即可。

（2）不合理的 using 声明有时会引发名字冲突。如下面的代码将导致二义性。

```
using zhang1::val1;
using wang2::val1;
val1 = 5;
```

3. 使用 using 编译预处理命令将一个名字空间中的所有元素引入到当前作用域中

如对于代码 3-10 和代码 3-11 来说，可以使用下面的语句。

```
using namespace zhang1;
…
```

使用于标准名字空间 std 时的方法为

```
#include <iostream>
using namespace std;
…
```

说明：

（1）using 命令也遵循作用域规则，超出了作用域就不再有效。

（2）using 命令使一个名字空间中的所有元素都可使用，不需要再用名字空间限定符。

（3）using 命令导入了一个名字空间中的所有名字，并且当其中某个名字与局部名字发生冲突时，局部名字将覆盖名称空间版本，而编译器不会发出警告。所以，不如 using 声明安全。因为 using 声明只导入指定的名字，并且当与局部名字冲突时，编译器会发出指示。

4. 名字空间的使用指导原则

（1）尽量使用已定义名字空间中的名字，尽量避免使用全局变量和静态全局变量。

（2）导入名字时，优先使用名字域解析和 using 声明，尽量不用 using 命令。

（3）使用 using 声明时，首先将其作用域设置为局部，即在局部域中声明。

（4）不要在头文件中使用 using 命令。

（5）非使用 using 命令不可时，应当将其放在所有编译预处理命令之后。

3.3.3 C++特别名字空间

1. 标准名字空间 std

C++ 是在 C 语言的基础上开发的，早期的 C++ 还不完善，不支持名字空间，没有自己的编译器，而是将 C++ 代码翻译成 C 代码，再通过 C 编译器完成编译。这时，C++仍然在使用 C 语言的库，如 stdio.h、stdlib.h、string.h 等头文件依然有效。此外 C++ 也开发了一些新的库，增加了自己的头文件，例如

iostream.h：用于控制台输入输出头文件。

fstream.h：用于文件操作的头文件。

complex.h：用于复数计算的头文件。

这些头文件所包含的类、函数、宏等都是全局范围的。

后来 C++引入了名字空间的概念，计划重新编写库，将类、函数、宏等都统一纳入一个名字空间，这个名字空间的名字就是 std。std 是 standard 的缩写，意思是“标准名字空间”。

2. 无名名字空间

C++还允许使用没有名字的名字空间，即名字空间定义时不给出名字。由于该名字空间没有名字，因此在其他编译单元（文件）中无法引用，只能在本编译单元（文件）的作用域内有效，其成员可以不必（也无法）用名字空间名限定。

【代码 3-12】 无名名字空间应用示例。

```
#include <iostream>
using namespace std;

namespace {                    //定义无名名字空间
     int a = 5;
     void fun1( )
     { cout << "OK. " << endl;}
}

int fun2(){ return a + 3;}

int main(){
fun1();
     cout << a << endl;
     cout << fun2 () << endl;
     return 0;
}
```

无名名字空间的成员 fun1()函数和变量 a 的作用域仅为文件（从声明无名名字空间的位置开始到所在文件结束）。在代码 3-12 中使用无名名字空间的成员，不需要任何限定。

由代码 3-12 可以看出，使用无名名字空间可以把一些名字限定于一个编译单元（文件）的范围内，显然与用 static 声明全局变量具有异曲同工之效。

3. 全局名字空间

全局名字空间是一个默认的名字空间，即当一个名字不被明确地声明或限定在特定的名字空间时，就默认其为全局名字空间中的名字。

注意，无名名字空间成员和全局名字空间成员都可以在没有任何限定的条件下直接使用，但两者还是有一些明显不同，如表 3.2 所示。

表 3.2　无名名字空间与全局名字空间的明显不同

比较项	定义形式	作用域
全局名字空间	无名字空间显式定义	程序所有文件
无名名字空间	有名字空间显式定义，但没有名字空间名	仅用在当前编译单元

习　题　3

概念辨析

1. 判断题。

（1）自动变量在程序执行结束时才释放。（　　）

（2）声明一个全局变量，其前必须加关键字 extern。（　　）

（3）可以用静态变量代替全局变量。（　　）

（4）在程序中使用全局变量比使用静态变量更安全。（　　）

（5）若 i 为某函数 func()之内说明的变量，则当 func()执行完后，i 值无定义。（　　）

（6）delete 操作符只可以在内存值已经清零后使用。（　　）

（7）程序执行过程中，不及时释放动态分配的内存，有造成内存泄露的危险。（　　）

（8）使用 new 操作符，可以动态分配全局堆中的内存资源。（　　）

（9）实现全局函数时，new 和 delete 通常成对地出现在由一对匹配的花括号限定的语句块中。（　　）

（10）delete 必须用于 new 返回的指针。（　　）

（11）名字空间可以多层嵌套。类 A 中的函数成员和数据成员，它们都属于类名 A 代表的一层名字空间。（　　）

2. 选择题。

（1）局部变量_____。

A. 在其定义的程序文件中，所有函数都可以访问

B. 可用于函数之间传递数据

C. 在其定义的函数中，可以被定义处以下的任何语句访问

D. 在其定义的语句块中，可以被定义处以下的任何语句访问

（2）全局变量_____。

A. 可以被一个系统中任何程序文件的任何函数访问

B. 只能在它定义的程序文件中被有关函数访问

C. 只能在它定义的函数中，被有关语句访问

D. 可用于函数之间传递数据

（3）_____是函数作用域变量。

A. 函数中的参数　　B. 函数体中定义的变量

C. 函数调用表达式中的变量　　D. 函数原型声明中的形式参数

（4）自动变量的存储空间分配在_____。

A. 堆区　　B. 栈区　　C. 自由区　　D. 静态区

（5）在某文件中定义的静态全局变量（或称静态外部变量），其作用域_____。

A. 只限于某个函数　B. 只限于本文件　C. 可以跨文件　D. 不受限制

（6）静态变量_____。

A. 生命周期一定是永久的

B. 一定是外部变量

C. 只能被初始化一次

D. 是在编译时被赋初值的，只能被赋值一次

（7）以下叙述中，错误的是_____。

A. 局部变量的定义可以放在函数体或复合语句的内部

B. 外部变量的定义可以放在函数以外的任何地方

C. 在同一程序中，局部变量与外部变量不可以重名

D. 函数的形式参数属于局部变量

（8）以下叙述中，正确的是_____。

A. 局部变量说明为 static 存储类，其生命周期将被延长

B. 外部变量说明为 static 存储类，其作用域将被扩大

C. 任何变量在未初始化时，其值都是不确定的

D. 形参可以使用的存储类说明其与局部变量完全相同

（9）以下叙述中，正确的是_____。

A. 外部变量的作用域一定比局部变量的作用域范围大

B. 静态（static）类别变量的生存期贯穿于整个程序运行期间。

C. 函数的形参都属于外部变量

D. 未在定义语句中赋初值的 auto 变量和 auto 变量的初值都是随机的

（10）对于外部变量 r，下面的叙述中错误的是_____。

A. 只有添加 static 属性，r 才被分配在静态存储区

B. 加不加 static 属性，r 都被分配在静态存储区

C. 加 static 属性，是限制其他文件使用 r

D. 若加 static 属性，则 r 不是本文件中定义的变量

（11）操作符 new_____。

A. 不会为一个指针指向的对象分配存储空间并初始化

B. 不会为一个指针变量分配需要的存储空间并初始化

C. 创建的对象要用操作符 delete 删除

D. 用于创建对象时，必须显式调用构造函数

（12）操作符 delete_____。

A. 仅可用于用 new 返回的指针

B. 可以用于空指针

C. 可以对一个指针使用多次

D. 所删除的堆空间与是否初始化过无关

（13）名字空间可以_____。

A. 限制一个程序中使用的变量名过多　　B. 限制一个名字太长

C. 用来限制程序元素的可见性　　D. 给不同的文件分配不同的名字

（14）用于指定名字空间时，std 是一个_____。

A. 定义在<iostream>中的标识符　　B. 系统定义的关键字

C. 定义在<iostream>中的操作符　　D. 系统定义的变量名

（15）用于指定名字空间时，using 是一个_____。

A. 定义在<iostream>中的标识符　　B. 系统定义的操作符

C. 系统定义的预编译命令　　D. 系统定义的变量名

（16）用于指定名字空间时，namespace 是一个_____。

A. 系统定义的变量名　　B. 系统定义的操作符

C. 预编译命令　　D. 系统定义的关键字

代码分析

1．找出下面各程序段中的错误并说明原因。

```
//file1.cpp
int a = 1;
int func(){
…
}
```

```
//file.,cpp
extern int a = 1;
int func();
void g(){
   a = func();
}
```

```
//file3.cpp
extern int a = 2;
int g();
int main() {
   a = g();
   …
}
```

2．找出下面各程序段中的错误并说明原因。

```
//file1.cpp
int a = 5, b = 8;
extern int a ;
```

```
//file2,cpp
extern double b;
extern int c ;
```