C/C++

存储类、链接和内存管理

张晓平

武汉大学数学与统计学院

Table of contents

存储类

存储类

- 存储时期(storage duration)

  变量在内存中保留的时间

- 作用域(scope)

  变量可被访问的一个或多个区域

- 链接(linkage)

  与变量的作用域一起来说明程序的哪些部分可以通过变量名来使用它。

在本节中，如不做特别说明，变量也可以是函数参数或函数名。

作用域

作用域

作用域是一个变量可被访问的一个或多个区域。

变量的作用域可以是

- 代码块作用域
- 函数原型作用域
- 文件作用域

作用域

代码块

代码块是包含在开始花括号与对应的结束花括号之间的一段代码。如

- 函数体
- 循环体
- 分支体
- 一个函数内的任一复合语句

作用域

代码块

代码块是包含在开始花括号与对应的结束花括号之间的一段代码。如

- 函数体
- 循环体
- 分支体
- 一个函数内的任一复合语句

代码块作用域（block scope）

代码块中定义的变量具有代码块作用域，从变量定义处到代码块的末尾该变量均可见。

作用域

注

函数的形参尽管在函数的开始花括号前被定义，但它同样也具有代码块作用域，隶属于包含函数体的代码块。

作用域

注

函数的形参尽管在函数的开始花括号前被定义，但它同样也具有代码块作用域，隶属于包含函数体的代码块。

```
double block(double x)
{
  double y = 0.0;
  ...
  return y;
}
```

作用域

注

函数的形参尽管在函数的开始花括号前被定义，但它同样也具有代码块作用域，隶属于包含函数体的代码块。

```
double block(double x)
{
  double y = 0.0;
  ...
  return y;
}
```

x 和 y 都有直到结束花括号的代码块作用域。

作用域

注

循环体中声明的变量，其作用域局限于该循环体。

作用域

注

循环体中声明的变量，其作用域局限于该循环体。

```
double block(double x)
{
  double y = 0.0;
  int i;
  for (i = 0; i < 10; i++) {
    double z = x + y;   // z作用域的开始
    ...
  }                     // z作用域的结束
  ...
  return y;
}
```

z 的作用域被限制在循环体内，只有循环体中的代码可以访问 z。

作用域

注

传统上，具有代码块作用域的变量都必须在代码块的开始处进行声明。C99 放宽了这一规则，允许在一个代码块中的任何位置声明变量。

可以这么写

```
for (int i = 0; i < 10; i++)
  printf("A C99 feature: i = %d'', i);
```

作用域

注

传统上，具有代码块作用域的变量都必须在代码块的开始处进行声明。C99 放宽了这一规则，允许在一个代码块中的任何位置声明变量。

可以这么写

```
for (int i = 0; i < 10; i++)
  printf("A C99 feature: i = %d'', i);
```

i 的作用域仅限于 for 循环，离开 for 循环后就看不到该变量了。

作用域

函数原型作用域(function prototype scope)

函数原型作用域从变量定义处一直到原型声明的末尾。

这意味着编译器在处理一个函数原型的参数时，只关心该参数的类型，而名字无关紧要。

作用域

函数原型作用域(function prototype scope)

函数原型作用域从变量定义处一直到原型声明的末尾。

这意味着编译器在处理一个函数原型的参数时，只关心该参数的类型，而名字无关紧要。

```
void show_array(int n, float * arr);
```

作用域

函数原型作用域(function prototype scope)

函数原型作用域从变量定义处一直到原型声明的末尾。

这意味着编译器在处理一个函数原型的参数时，只关心该参数的类型，而名字无关紧要。

```
void show_array(int n, float * arr);
```

n 和 arr 仅限于这条函数声明语句可见。

作用域

文件作用域（file scope）

一个在所有函数之外定义的变量具有文件作用域。

具有文件作用域的变量从变量定义处到文件结尾处都是可见的。

作用域

文件作用域（file scope）

一个在所有函数之外定义的变量具有文件作用域。

具有文件作用域的变量从变量定义处到文件结尾处都是可见的。

```
#include <stdio.h>
int a = 0;
void f(void);
int main(void) { ... }
void f(void) {  ... }
```

a 具有文件作用域，在 main() 和 f() 中都可以使用它。

作用域

文件作用域（file scope）

一个在所有函数之外定义的变量具有文件作用域。

具有文件作用域的变量从变量定义处到文件结尾处都是可见的。

```
#include <stdio.h>
int a = 0;
void f(void);
int main(void) { ... }
void f(void) {  ... }
```

a 具有文件作用域，在 main() 和 f() 中都可以使用它。

注

具有文件作用域的变量可以在不止一个函数中使用，故它也被称为全局变量（global variable）。

链接

问题

当多个源文件链接在一起，我们如何处理相同名字的标识符？

假设多个源文件均含有变量 a，那么它的值到底采用哪个源文件定义的值呢？这就涉及到标识符的链接属性。

链接

问题

当多个源文件链接在一起，我们如何处理相同名字的标识符？

假设多个源文件均含有变量 a，那么它的值到底采用哪个源文件定义的值呢？这就涉及到标识符的链接属性。

链接（linkage）

C 变量有如下三种链接：

- 外部链接（external linkage）
- 内部链接（internal linkage）
- 空链接（no linkage）

链接

空链接

具有代码块作用域与函数原型作用域的变量有空链接，这意味着它们是由其定义所在的代码块或函数原型所私有的。

链接

空链接

具有代码块作用域与函数原型作用域的变量有空链接，这意味着它们是由其定义所在的代码块或函数原型所私有的。

外部与内部链接

局变量可能有内部或外部链接。

- 一个具有外部链接的变量可以在一个多文件程序的任何地方使用；
- 一个具有内部链接的变量可以在一个文件的任何地方使用。

链接

注

要区分一个全局变量是具有内部链接还是外部链接，可以看看定义它时是否被关键字 `static`修饰。

- 若用了 `static`，则它具有内部链接，只能被当前文件使用；
- 否则具有外部链接，程序中的其他文件可以使用它。

链接

注

要区分一个全局变量是具有内部链接还是外部链接，可以看看定义它时是否被关键字 static修饰。

- 若用了 static，则它具有内部链接，只能被当前文件使用；
- 否则具有外部链接，程序中的其他文件可以使用它。

```
int a = 5;
static int b = 3;
int main(void)
{
   ...
}
...
```

链接

注

要区分一个全局变量是具有内部链接还是外部链接，可以看看定义它时是否被关键字 static修饰。

- 若用了 static，则它具有内部链接，只能被当前文件使用；
- 否则具有外部链接，程序中的其他文件可以使用它。

```
int a = 5;
static int b = 3;
int main(void)
{
   ...
}
...
```

- 和该文件属于同一程序的其他文件可以使用变量 a。
- 变量 b是该文件私有的，但可以被该文件的任一函数使用。

存储时期

存储期（storage duration）

也称生存期，指的是变量在内存中的时间。

- 静态存储时期（static storage duration）
- 自动存储时期（auto storage duration）

存储时期

静态存储时期

若一个变量有静态存储时期，则它将在程序执行期间一直存在。

存储时期

静态存储时期

若一个变量有静态存储时期，则它将在程序执行期间一直存在。

例

- 全局变量有静态存储时期。

  注意，对于全局变量，关键字 `static`表明其链接类型，而非存储时期。

- 关键字 `static` 修饰的代码块作用域变量也具有静态存储时期。

存储时期

自动存储时期（auto storage duration）

一般来说，具有代码块作用域的变量具有自动存储时期。程序进入定义这些变量的代码块时，为其分配内存；当退出该代码块时，将释放内存。

存储时期

自动存储时期（auto storage duration）

一般来说，具有代码块作用域的变量具有自动存储时期。程序进入定义这些变量的代码块时，为其分配内存；当退出该代码块时，将释放内存。

```
void bore(int number)
{
  int index;
  for (index = 0; index < number; index++)
    ...
}
```

存储时期

自动存储时期（auto storage duration）

一般来说，具有代码块作用域的变量具有自动存储时期。程序进入定义这些变量的代码块时，为其分配内存；当退出该代码块时，将释放内存。

```
void bore(int number)
{
  int index;
  for (index = 0; index < number; index++)
    ...
}
```

number 和 index 在每次调用 bore() 时被创建，每次退出函数时消失。

存储类

存储类

C 使用作用域、链接和存储时期来定义 5 种存储类：

1. 自动
2. 寄存器
3. 具有代码块作用域的静态
4. 具有外部链接的静态
5. 具有内部链接的静态

存储类

存储类	存储时期	作用域	链接	声明方式
自动	自动	代码块	空	代码块内
寄存器	自动	代码块	空	代码块内，使用 `register`
具有外部链接的静态	静态	文件	外部	所有函数之外
具有内部链接的静态	静态	文件	内部	所有函数之外，使用 `static`
具有代码块作用域的静态	静态	代码块	空	代码块内，使用 `static`

存储类

自动变量

自动变量

自动变量

- 默认情况下，在代码块或函数头中定义的任何变量都属于自动存储类。可显示地使用 auto 使此意图更清晰。
- 代码块作用域与空链接意味着只有代码块才能访问该变量，其他函数中的同名变量与它无关。
- 自动存储时期意味着：当程序进入包含变量声明的代码块时，变量开始存在；当程序离开该代码块时，自动变量马上消失。

```
int main(void)
{
  auto int i;
  ...
}
```

自动变量

```
int loop(int n)
{
  int m;                    // scope of m
  scanf("%d'', &m);
  {
    int i;                  // scope of m and i
    for (i = m ; i < n; i++)
      puts("i is local to a sub-block\n'');
  }
  return m;                 // scope of m,  i vanished
}
```

自动变量

```
int loop(int n)
{
  int m;                    // scope of m
  scanf("%d'', &m);
  {
    int i;                  // scope of m and i
    for (i = m ; i < n; i++)
      puts("i is local to a sub-block\n'');
  }
  return m;                 // scope of m,  i vanished
}
```

- i 仅在内层代码块中可见，程序运行至定义处被创建，离开代码块时消失；
- n 和 m 在整个函数中均可用，函数调用时被创建，函数退出时消失。

自动变量

问题

如果内层代码块有变量与外层代码块中的变量重名，会发生什么？

自动变量

问题

如果内层代码块有变量与外层代码块中的变量重名，会发生什么？

内层定义将覆盖外部定义，但当程序离开内层代码块时，外部变量将重新恢复作用。

自动变量

```c
#include <stdio.h>
int main(void)
{
  int x = 30;
  printf("x in outer block: %d\n", x);
  {
    int x = 77;
    printf("x in inner block: %d\n", x);
  }
  printf("x in outer block: %d\n", x);
  while (x++ < 33) {
    int x = 100;
    x++;
    printf("x in while loop: %d\n", x);
  }
  printf("x in outer block: %d\n", x);
  return 0;
}
```

自动变量

```
x in outer block: 30
x in inner block: 77
x in outer block: 30
x in while loop: 101
x in while loop: 101
x in while loop: 101
x in outer block: 34
```

自动变量

```c
#include<stdio.h>
int main(void)
{
  int n = 10;

  printf("Initially, n = %d\n", n);
  for (int n = 1; n < 3; n++)
    printf("loop 1: n = %d\n", n);
  printf("After loop 1, n = %d\n", n);
  for (int n = 1; n < 3; n++) {
    printf("loop 2: index n = %d\n", n);
    int n = 30;
    printf("loop 2: n = %d\n", n);
    n++;
  }
  printf("After loop 2, n = %d\n", n);
}
```

自动变量

```
Initially, n = 10
loop 1: n = 1
loop 1: n = 2
After loop 1, n = 10
loop 2: index n = 1
loop 2: n = 30
loop 2: index n = 2
loop 2: n = 30
After loop 2, n = 10
```

自动变量的初始化

除非你显式地初始化自动变量，否则它不会被自动初始化。

```
int main(void)
{
  int i;
  int j = 5;
  ...
}
```

变量 `j` 初始化为 5，而变量 `i` 的初值则是先前占用分配给它的空间的任意值。

存储类

寄存器变量

寄存器变量

寄存器变量

通常，变量存储在内存中。如果幸运，变量可以被存储在 CPU 寄存器中，从而可以比普通变量更快地被访问和操作，这样的变量被称为寄存器变量。

寄存器变量

寄存器变量

通常，变量存储在内存中。如果幸运，变量可以被存储在 CPU 寄存器中，从而可以比普通变量更快地被访问和操作，这样的变量被称为寄存器变量。

注

因为寄存器变量多是存放在一个寄存器而非内存中，故无法获得寄存器变量的地址。

寄存器变量

寄存器变量同自动变量一样，有代码块作用域、空链接以及自动存储时期。通常使用关键字 register 声明寄存器变量：

```
int main(void)
{
  register int quick;
  ...
}
```

寄存器变量

注

所谓“幸运”，是因为声明一个寄存器变量仅仅是一个请求，而非命令。编译器必须在你的请求与可用寄存器的个数之间做出权衡，所以你可能达不成愿望。在此情况下，变量会变成普通的自动变量，但依然不能使用地址运算符。

寄存器变量

可以把一个形参请求为寄存器变量，只需在函数头使用 register 关键字：

```
void foo(register int n)
{
   ...
}
```

存储类

具有代码块作用域的静态变量

具有代码块作用域的静态变量

静态变量（static variable）

所谓“静态”，指的是变量在内存中的位置固定不变。

- 全局变量具有静态存储时期。
- 也可创建具有代码块作用域，兼具静态存储的局部变量，使用关键字 `static` 在代码块中声明创建。

  它们和自动变量具有相同的作用域，但当包含这些变量的函数完成工作时，它们并不消失。从一次函数调用到下一次函数调用，计算机都记录着它们的值。

具有代码块作用域的静态变量 i

```
#include <stdio.h>
void trystat(void);
int main(void)
{
  int count;
  for (count = 1; count <= 3; count++) {
    printf("iteration %d: \n", count);
    trystat();
  }
  return 0;
}
void trystat(void)
{
  int fade = 1;
  static int stay = 1;
  printf("fade = %d and stay = %d\n", fade++, stay++);
}
```

具有代码块作用域的静态变量

```
iteration 1:
fade = 1 and stay = 1
iteration 2:
fade = 1 and stay = 2
iteration 3:
fade = 1 and stay = 3
```

具有代码块作用域的静态变量

每次调用时，静态变量 stay 的值都被加 1，而变量 fade 每次都重新开始。这表明了初始化的不同：

每次调用 trystat 时， fade 都被初始化，而 stay 只在编译时被初始化一次。如果不显式地对静态变量进行初始化，它们将被初始化为 0。

具有代码块作用域的静态变量

以下两个声明看起来相似：

```
int fade = 1;
static int stay = 1;
```

- 第一条语句确实是 trystat 的一部分，每次调用时都会被执行，它是个运行时的动作。
- 第二条语句实际上并不是 trystat 的一部分。调试并逐步执行程序时，你会发现程序看起来跳过了这条语句，因为静态变量和外部变量在程序调入内存时就已经到位了。

  把这个语句放在 trystat 函数中是为了告诉编译器只有函数 trystat 可以看到该变量。它不是运行时执行的语句。

具有代码块作用域的静态变量

注

对函数形参不能使用 static：

```
int wontwork(static int flu);   // not
allowed
```

存储类

具有外部链接的静态变量

具有外部链接的静态变量

具有外部链接的静态变量

它具有文件作用域、外部链接和静态存储时期。该类型也被称为外部存储类（external storage class），该类型的变量被称为外部变量（external variable）。

注

- 把变量的定义声明放在所有函数之外，即创建了一个外部变量。
- 为了使程序更加清晰，可以在使用外部变量的函数中使用关键字 extern 再次声明它。
- 如果变量是在别的文件中定义的，使用 extern 来声明该变量就是必须的。

具有外部链接的静态变量

```
int num;              // external variable
double arr[100];      // external array
extern char ch;       //
void next(void);
int main(void)
{
  extern int num;         // optional declare
  extern double arr[]; // optional declare
  ...
}
void next(void){ ... }
```

具有外部链接的静态变量

- num 的两次声明是链接的例子，它们指向同一变量。外部变量具有外部链接。
- 无需在 double arr[] 中指明数组大小，因第一次声明已提供了这一信息。外部变量具有文件作用域，它们从被声明处到文件结尾都是可见的，故 main() 中的一组 extern 声明完全可以省略。如果它们出现在那，仅表明 main() 使用这些变量。

具有外部链接的静态变量

- 若函数中的声明不写 extern，则创建一个独立的自动变量。

  在 main() 中，若用 extern int num; 替换 int num; ，则创建一个名为 num 的自动变量，它是一个独立的局部变量，而不同于初始的 num。程序执行时， main() 中该局部变量起作用；而 next() 中，外部的 num 将起作用。

  简言之，**程序执行代码块内语句时，代码块作用域的变量将覆盖文件作用域的同名变量。**
- 外部变量具有静态存储时期，因此数组 arr 一直存在并保持其值。

具有外部链接的静态变量

```
// example 1
int num;
int magic();
int main(void)
{
   extern int num;
   ...
}
int magic()
{
   extern int num;
   ...
}
```

具有外部链接的静态变量

```
// example 2
int num;
int magic();
int main(void)
{
   extern int num;
   ...
}
int magic()
{
   ...
}
```

具有外部链接的静态变量

```
// example 3
int num;
int magic();
int main(void)
{
   int num;
   ...
}
int nnn;
int magic()
{
   auto int num;
   ...
}
```

外部变量的初始化

外部变量的初始化

- 和自动变量一样，外部变量可被显示地初始化；
- 不同于自动变量，若不对外部变量进行初始化，它们将被初始化为 0；
- 不同于自动变量，只可用常量表达式来初始化文件作用域变量。

该原则也适用于外部定义的数组。

外部变量的初始化

```
int x = 10;                    // OK
int y = 3 * 20;                // OK
size_t z = sizeof(int);        // OK
int x2 = 2 * x;                // INVALID
```

外部变量的使用

```
// global.c :
#include<stdio.h>
int units = 0;
void critic(void);
int main(void)
{
  extern int units;
  printf("How many pounds to a firkin of butter?\n");
  scanf("%d", &units);
  while (units != 56) critic();
  printf("You must have looked it up!\n");
  return 0;
}
void critic(void)
{
  printf("No luck. Try again.\n");
  scanf("%d", &units);
}
```

外部变量的使用

```
How many pounds to a firkin of butter?
14
No luck. Try again.
56
You must have looked it up!
```

外部变量的使用

```
How many pounds to a firkin of butter?
14
No luck. Try again.
56
You must have looked it up!
```

main() 与 critic() 都通过标识符 units 来访问同一变量。在 C 的术语中，称 units 具有文件作用域、外部链接及静态存储时期。

存储类

具有内部链接的静态变量

具有内部链接的静态变量

具有内部链接的静态变量

这种存储类的变量具有静态存储时期、文件作用域以及内部链接。
通常使用 `static` 在所有函数外部进行定义（同外部变量的定义）。

```
static int num = 1;
int main(void) { ... }
```

注

普通的外部变量可被程序中的任一文件中所包含的函数使用，而具有内部链接的静态变量只可以被同一文件中的函数使用。

具有内部链接的静态变量

可在函数中使用 `extern` 来再次声明任何具有文件作用域的变量，但这并不改变链接。

具有内部链接的静态变量

可在函数中使用 extern 来再次声明任何具有文件作用域的变量，但这并不改变链接。

```
int traveler = 1;           // external linkage
static int stayhome = 1;    // internal linkage

int main(void)
{
  extern int traveler;      // use global travelre
  extern int stayhome;      // use global stayhome
}
```

具有内部链接的静态变量

可在函数中使用 extern 来再次声明任何具有文件作用域的变量，但这并不改变链接。

```
int traveler = 1;           // external linkage
static int stayhome = 1;    // internal linkage

int main(void)
{
  extern int traveler;      // use global travelre
  extern int stayhome;      // use global stayhome
}
```

对该文件而言， traveler 和 stayhome 都是全局的，但只有 traveler 可被其他文件中的代码使用。使用 extern 的两个声明表明 main() 在使用两个全局变量，但 stayhome 仍具有内部链接。

存储类

存储类说明符

存储类说明符

存储类说明符

C 语言中有 5 个作为存储类说明符的关键字：

- `auto`
- `register`
- `static`
- `extern`
- `typedef`：它与内存存储无关，由于语法原因被归入此类。

存储类说明符

存储类说明符

C 语言中有 5 个作为存储类说明符的关键字：

- auto
- register
- static
- extern
- typedef：它与内存存储无关，由于语法原因被归入此类。

注

- 关键字 static 与 extern 的含义随上下文而不同。
- 不可以在一个声明中使用一个以上的存储类说明符，这意味着不能将其它任一存储类说明符作为 typedef 的一部分。

存储类说明符

- auto 表明一个变量具有自动存储时期，它只能用在具有代码块作用域的变量声明中。使用它仅用于明确指出意图，使程序更易读。
- register 也只能用在具有代码块作用域的变量声明中。它将一个变量归入寄存器存储类，这相当于请求将该变量存储在一个寄存器内，以更快地存取。register 的使用将导致不能获取变量的地址。

存储类说明符

- 对于 `static`，
  - 用于具有代码块作用域的变量声明时，使该变量具有静态存储时期，从而得以在程序运行期间存在并保留其值。此时，变量仍具有代码作用域和空链接。
  - 用于具有文件作用域的变量声明时，表明该变量具有内部链接。
- `extern` 表明你在声明一个已经在别处定义了的变量，
  - 若该声明具有文件作用域，所指向的变量必然具有外部链接；
  - 若该声明具有代码块作用域，所指向的变量可能具有外部链接也可能具有内部链接，这取决于该变量的定义声明。

总结

- 自动变量具有代码块作用域、空链接和自动存储时期。它们是局部的，为定义它们的代码所私有。
- 寄存器变量与自动变量具有相同的属性，但编译器可能使用速度更快的内存或寄存器来存储它们。无法获取一个寄存器变量的地址。

总结

具有静态存储时期的变量可能具有外部链接、内部链接或空链接。

- 当变量在文件的所有函数之外声明时，它是一个具有文件作用域的外部变量，具有外部链接和静态存储时期。
- 若在这样的声明中再加上 `static`，将获得一个具有静态存储时期、文件作用域和内部链接的变量。
- 若在一个函数内使用关键字 `static`声明变量，变量将具有静态存储时期、代码块作用域和空链接。

总结

- 当程序执行到包含变量声明的代码块时，给具有自动存储时期的变量分配内存，并在代码块结束时释放内存。如果没有初始化，该变量将是垃圾值。
- 在程序编译时给具有静态存储时期的变量分配内存，并在程序运行时一直保持。若没有初始化，将被设置为 0。

总结

- 具有代码块作用域的变量局部于包含变量声明的代码块。
- 具有文件作用域的变量对文件中在它声明之后的所有函数可见。
  - 若一个文件作用域变量具有外部链接，则它可被程序中的其他文件使用；
  - 若一个文件作用域变量具有内部链接，则它只能在声明它的文件中使用。

存储类与函数

存储类与函数

函数也有存储类。函数可以是外部的（默认情况下）或者静态的。

- 外部函数可被其他文件中的函数使用，而静态函数只可以在定义它的文件中使用。如

```
double alpha();
static double beta();
extern double gamma();
```

alpha() 与 gamma() 可被程序中其他文件中的函数使用，而 beta() 不行。因 beta() 被限定在同一文件内，故可在其他文件中使用同名的不同函数。

存储类与函数

- 使用 `static` 的原因之一就是创建一个特定模块所私有的函数，从而避免可能的名字冲突。
- 使用 `extern` 来声明在其他文件中定义的函数。这一习惯做法主要是为了使程序更清晰，因为除非函数声明中使用了关键字 `static`，否则就认为它是 `extern` 的。