张晓平

武汉大学数学与统计学院

Homepage: xpzhang.me

C 语言程序设计

存储类、链接和内存管理

存储类

- 存储期 (storage duration)
- 作用域 (scope)
- 链接 (linkage)

- 存储期：变量在内存中保留的时间
- 变量的作用域与链接一起表明程序的哪些部分可以通过变量名来使用它。

作用域

作用域描述了程序中可以访问一个标识符的一个或多个区域。

一个 C 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
- 一个函数内的任一复合语句

作用域

代码块中定义的变量具有代码块作用域 (block scope)，从该变量被定义的地方到包含该定义的代码块的末尾该变量均可见。

注

函数的形参尽管在函数的开始花括号前被定义，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

如下代码中， x 和 y 都有直到结束花括号的代码块作用域。

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

作用域

在一个内部代码块中声明的变量，其作用域局限于该代码块。

作用域

```
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;
}
```

作用域

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);
```

这里，i 的作用域仅限于 for 循环，离开 for 循环后就看不到变量 i 了。

作用域

函数原型作用域 (function prototype scope) 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

一个在所有函数之外定义的变量具有文件作用域 (file scope)。具有文件作用域的变量从它定义处到包含该定义的文件结尾处都是可见的。

作用域

```
#include <stdio.h>
int units = 0;
void critic(void);

int main(void) { ... }

void critic(void) {  ... }
```

这里， units 具有文件作用域，在 main() 和 critic() 中都可以使用它。因它们可以在不止一个函数中使用，故文件作用域变量也被称为全局变量 (global variable)。

链接

一个 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

- 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。
- 全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

要区分一个全局变量是具有内部链接还是外部链接，可以看看在外部定义中是否使用了关键字 static。

```
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

- 静态存储时期 (static storage duration)

  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。

  例如，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。

- 自动存储时期 (auto storage duration)

  具有代码块作用域的变量一般具有自动存储时期。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。

存储类

如下代码中，变量 number 和 index 在每次调用 bore() 时生成，每次结束函数调用时消失：

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

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

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。

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

自动变量

- 代码块作用域与空链接意味着只有变量定义所在的代码块才能通过名字访问该变量，其他函数中与其同名的变量与它无关。
- 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。

自动变量

观察一下嵌套代码：

```
int loop(int n)
{
  int m;                  // m 的作用域
  scanf("%d'', &m);
  {
    int i;                // m 和 i 的作用域
    for (i = m ; i < n; i++)
      puts("i is local to a sub-block\n'');
  }
  return m;               // m 的作用域, i 已经消失
}
```

自动变量

该代码中，变量 i 仅在内层花括号中可见，变量 n 和 m 分别在函数头和外层代码块中定义，在整个函数中可用，并且一直存在函数终止。

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

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 I

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int x = 30;
5    printf("x in outer block: %d\n", x);
6    {
7      int x = 77;
8      printf("x in inner block: %d\n", x);
9    }
10   printf("x in outer block: %d\n", x);
```

自动变量 II

```
11    while (x++ < 33) {
12      int x = 100;
13      x++;
14      printf("x in while loop: %d\n", x);
15    }
16    printf("x in outer block: %d\n", x);
17    return 0;
18  }
```

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 I

```
1  #include<stdio.h>
2  int main(void)
3  {
4    int n = 10;
5
6    printf("Initially, n = %d\n", n);
7    for (int n = 1; n < 3; n++)
8      printf("loop 1: n = %d\n", n);
9    printf("After loop 1, n = %d\n", n);
10   for (int n = 1; n < 3; n++) {
```

自动变量 II

```
11      printf("loop 2: index n = %d\n", n);
12      int n = 30;
13      printf("loop 2: n = %d\n", n);
14      n++;
15    }
16    printf("After loop 2, n = %d\n", n);
17  }
```

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。

寄存器变量

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

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

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)
{
  ...
}
```

具有代码块作用域的静态变量

对于静态变量 (static variable)，“静态”是指变量的位置固定不变。

- 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。

  这些变量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。

  也就是说，这些变量具有代码块作用域、空链接，却有静态存储时期。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。

  这样的变量使用关键字 static 在代码块中声明创建。

具有代码块作用域的静态变量 I

```
1  #include <stdio.h>
2  void trystat(void);
3  int main(void)
4  {
5    int count;
6    for (count = 1; count <= 3; count++) {
7      printf("iteration %d: \n", count);
8      trystat();
9    }
10   return 0;
```

具有代码块作用域的静态变量 II

```
11 }
12 void trystat(void)
13 {
14   int fade = 1;
15   static int stay = 1;
16   printf("fade = %d and stay = %d\n", fade++, stay++)
     ;
17 }
```

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

对函数形参不能使用 static：

```
int wontwork(static int flu);   // 不允许
```

具有外部链接的静态变量

具有外部链接的静态变量具有文件作用域、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)。

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。

具有外部链接的静态变量

- 为了使程序更加清晰，可以在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;              //外部定义的变量
double arr[100];      //外部定义的数组
extern char ch;       //必须的声明，因 ch 在其他文件中定义
void next(void);
int main(void)
{
  extern int num;       //可选的声明
  extern double arr[]; //可选的声明
  ...
}
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，因为它们指向同一变量。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

- 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 I

```
1  // global.c :
2  #include<stdio.h>
3  int units = 0;
4  void critic(void);
5
6  int main(void)
7  {
8    extern int units;
9
10   printf("How many pounds to a firkin of butter?\n");
```

外部变量的使用 II

```
11    scanf("%d", &units);
12    while (units != 56)
13      critic();
14    printf("You must have looked it up!\n");
15    return 0;
16  }
17
18  void critic(void)
19  {
20    printf("No luck. Try again.\n");
21    scanf("%d", &units);
```

外部变量的使用 III

```
22 }
```

外部变量的使用 I

```
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

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。

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) { ... }
```

具有内部链接的静态变量

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

```
int traveler = 1;           // 外部链接
static int stayhome = 1;    // 内部链接

int main(void)
{
  extern int traveler;      //使用全局变量travelre
  extern int stayhome;      //使用全局变量stayhome
}
```

对这个文件来说， 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。

具有内部链接的静态变量

```
int traveler = 1;           // 外部链接
static int stayhome = 1;    // 内部链接

int main(void)
{
  extern int traveler;      //使用全局变量travelre
  extern int stayhome;      //使用全局变量stayhome
}
```

对这个文件来说， 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。

多文件

只有在使用多文件程序时，内部链接与外部链接的区别才显得重要。

多文件

复杂的 C 程序往往使用多个独立的代码文件。有时，这些文件可能需要共享一个外部变量。ANSI C 通过在一个文件中定义变量，在其他文件中引用声明这个变量来实现共享。

除了一个声明（定义声明）外，其他所有声明都必须使用关键字 extern，并且只有在定义声明中才可以对该变量进行初始化。

多文件

除非在第二个文件中也声明了该变量（使用 extern），否则在一个文件中定义的外部变量不可以用于第二个文件。

多文件

关键字 static 与 extern 的含义随上下文而不同。 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。

多文件

注

不可以在一个声明中使用一个以上存储类说明符，这意味着不能将其它任一存储类说明符作为 *typedef* 的一部分。

多文件

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。

多文件

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。

多文件

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。

多文件

extern 表明你在声明一个已经在别处定义的了变量，

- 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。

secname

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

gamma() 与 delta() 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。

secname

- 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。

secname：小结

C 有 3 种链接属性:

- 外部链接：使用 extern 关键字
- 内部链接：使用 static 关键字
- 无链接

secname：小结

- extern 修饰的变量必须在某文件中已经定义；
- 使用 extern 可以实现在不同文件中使用相同变量；
- static 修饰的变量仅能在当前文件中使用，其他文件不能访问。

secname：测试 1

```
// unit1.c
#include<stdio.h>
int a = 10;
void print1(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 1

```
// unit2.c
#include<stdio.h>
int a = 20;
void print2(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 1

```
// main1.c
#include<stdio.h>
void print1(void);
void print2(void);
int main(void)
{
  print1();
  print2();
}
```

secname：测试 1

```
$ gcc main1.c unit1_1.c unit2_1.c
/tmp/ccFGkVaL.o:(.data+0x0): 'a'被多次定义
/tmp/cciTxJFq.o:(.data+0x0): 第一次在此定义
collect2: error: ld returned 1 exit status
```

secname：测试 2

```
// unit1_2.c
#include<stdio.h>
int a = 20;
void print1(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 2

```
// unit2_2.c
#include<stdio.h>
extern int a;
void print2(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 2

```
// main2.c
#include<stdio.h>
void print1(void);
void print2(void);

int main(void)
{
  extern int a;
  a = 20;
  print1();
  print2();
```

secname：测试 2

```
$ gcc main2.c unit1_2.c unit2_2.c
At unit1_2.c, a = 20
At unit2_2.c, a = 20
At unit1_2.c, a = 30
At unit2_2.c, a = 30
```

secname：测试 3

```
// unit1_3.c
#include<stdio.h>
static int a = 20;
void print1(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 3

```
// unit2_3.c
#include<stdio.h>
extern int a;
void print2(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 3

```
// main3.c
#include<stdio.h>
void print1(void);
void print2(void);

int main(void)
{
  print1();
  print2();
}
```

secname：测试 3

```
$ gcc main3.c unit1_3.c unit2_3.c
/tmp/ccUR88Tz.o：在函数‘print2’中：
unit2_3.c:(.text+0x6)：对‘a’未定义的引用
collect2: error: ld returned 1 exit status
```

secname：测试 4

```
// unit1_4.c
#include<stdio.h>
static int a = 20;
void print1(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 4

```
// unit2_4.c
#include<stdio.h>
int a = 10;
void print2(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 4

```
// main3.c
#include<stdio.h>
void print1(void);
void print2(void);
int main(void)
{
  extern int a;
  a = 20;  print1(); print2();
  a = 30;  print1(); print2();
}
```

secname：测试 4

```
$ gcc main4.c unit1_4.c unit2_4.c
At unit1_4.c, a = 20
At unit2_4.c, a = 20
At unit1_4.c, a = 20
At unit2_4.c, a = 30
```

secname：测试 5

```
// unit1_5.c
#include<stdio.h>
extern int a;
void print1(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 5

```
// unit2_5.c
#include<stdio.h>
extern int a;
void print2(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 5

```
// main5.c
#include<stdio.h>
void print1(void);
void print2(void);
int main(void)
{
  extern int a;
  a = 20;  print1();  print2();
  a = 30;  print1();  print2();
}
```

secname：测试 5

```
$ gcc main5.c unit1_5.c unit2_5.c
/tmp/cc0aTHQd.o：在函数‘main’中：
main5.c:(.text+0x6)：对‘a’未定义的引用
main5.c:(.text+0x1a)：对‘a’未定义的引用
/tmp/ccBk0v4z.o：在函数‘print1’中：
unit1_5.c:(.text+0x6)：对‘a’未定义的引用
/tmp/ccyf0TlW.o：在函数‘print2’中：
unit2_5.c:(.text+0x6)：对‘a’未定义的引用
collect2: error: ld returned 1 exit status
```

secname：测试 6

```
// unit1_6.c
#include<stdio.h>
int a = 10;
static void print1(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 6

```
// unit2_6.c
#include<stdio.h>
extern int a;
void print2(void)
{
  printf("At %s, a = %d\n", __FILE__, a);
}
```

secname：测试 6

```
// main6.c
#include<stdio.h>
void print1(void);
void print2(void);
int main(void)
{
  extern int a;
  a = 20;  print1();  print2();
  a = 30;  print1();  print2();
}
```

secname：测试 6

```
$ gcc main6.c unit1_6.c unit2_6.c
/tmp/ccdJYjYo.o：在函数‘main’中：
main6.c:(.text+0xf)：对‘print1’未定义的引用
main6.c:(.text+0x23)：对‘print1’未定义的引用
collect2: error: ld returned 1 exit status
```

随机数函数与静态变量

C 提供了 rand() 来产生随机数，它使用了一个具有内部链接的静态变量。rand() 是一个“伪随机数发生器”，这意味着可以预测数字的实际顺序，但这些数字在可能的取值范围内均匀地分布。

随机数函数与静态变量

```
// rand0.c: 产生随机数
static unsigned long int next = 1;
int rand0(void)
{
  next = next * 1103515245 + 12345;
  return (unsigned int) (next/65536) % 32768;
}
```

随机数函数与静态变量

```
// r_derive0.c: 测试 rand0()
#include <stdio.h>
extern int rand0(void);

int main(void)
{
  int count;

  for (count = 0; count < 5; count++)
  printf("%hd\n", rand0());

  return 0;
```

随机数函数与静态变量

先运行一次：

```
16838
5758
10113
17515
31051
```

再运行一次：

```
16838
5758
10113
17515
```

随机数函数与静态变量

```
1  // s_and_r.c: 包含 rand1() 和 srand1() 的文件
2  static unsigned long int next = 1;
3  int rand1(void)
4  {
5    next = next * 1103515245 + 12345;
6    return (unsigned int) (next/65536) % 32768;
7  }
8
9  void srand1(unsigned int seed)
10 {
11   next = seed;
12 }
```

随机数函数与静态变量

注意 next 是一个具有内部链接的文件作用域变量，这意味着它可以同时被 rand1() 和 srand1() 使用，但不可以被其他文件中的函数使用。

随机数函数与静态变量 I

```
1  // r_derive1.c:
2  #include <stdio.h>
3  extern void srand1(unsigned int x);
4  extern int rand1(void);
5
6  int main(void)
7  {
8    int count;
9    unsigned seed;
10
```

随机数函数与静态变量 II

```
11    printf("Please enter your choice for seed.\n");
12    while (scanf("%u", &seed) == 1) {
13      srand1(seed);
14      for (count = 0; count < 5; count++)
15        printf("%hd\n", rand1());
16      printf("Please enter next seed (q to quit)\n");
17    }
18    printf("Done\n");
19
20    return 0;
21  }
```

随机数函数与静态变量 III

随机数函数与静态变量 I

```
Please enter your choice for seed.
1
16838
5758
10113
17515
31051
Please enter next seed (q to quit)
2
908
```

随机数函数与静态变量 II

```
22817
10239
12914
25837
Please enter next seed (q to quit)
q
Done
```

内存的动态分配

本节主要介绍四个函数

- malloc()
- calloc()
- realloc()
- free()

内存的动态分配

回顾一下，对于如下声明

```
float x;
char place[]="Beijing";
```

此时系统将留出足够存储 float 或字符串的内存空间。而声明

```
char plates[100];
```

将留出 100 个内存位置，每个位置可存储一个 int 值。

内存的动态分配

C 的功能远不止如此，可以在程序运行时分配更多的内存。主要工具是 malloc()，

- 它接受一个参数：所需内存字节数。
- 然后 malloc() 找到可用内存中一个大小合适的块。
- 内存是匿名的，也就是说， malloc() 分配了内存，但没有为它指定名字。但它可以返回那块内存中第一个字节的地址。
- 于是，你可以把该地址赋值给一个指针变量，并使用该指针来访问那块内存。

内存的动态分配

- 因 char 代表一个字节，故传统上曾将 malloc() 定义为指向 char 的指针类型。
- ANSI C 标准使用了一个新类型：指向 void 的指针，并允许将 void 指针赋值给其他类型的指针。
- 若 malloc() 找不到所需的空间，则返回空指针。

内存的动态分配

以下展示如何用 malloc() 创建一个数组。

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

这段代码请求 30 个 double 值的空间，并把 ptd 指向该空间所在位置。

内存的动态分配

以下展示如何用 malloc() 创建一个数组。

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

这段代码请求 30 个 double 值的空间，并把 ptd 指向该空间所在位置。

内存的动态分配

需要注意的是， ptd 是作为指向一个 double 值的指针声明的，而不是指向 30 个 double 值的数据块的指针。

而数组的名字是其第一个元素的地址，因此，如果令 ptd 指向一个内存块的第一个元素，就可以像使用数组名一样使用它。

内存的动态分配

创建数组的三种方法：

- 声明一个数组，声明时用常量表达式指定数组维数，然后用数组名访问各元素；
- 声明一个变长数组，声明时用变量表达式指定数组维数，然后用数组名访问各元素；
- 声明一个指针，调用 malloc()，然后使用该指针来访问数组元素。

使用第二种或第三种方法可以创建一个动态数组。

内存的动态分配

创建数组的三种方法：

- 声明一个数组，声明时用常量表达式指定数组维数，然后用数组名访问各元素；
- 声明一个变长数组，声明时用变量表达式指定数组维数，然后用数组名访问各元素；
- 声明一个指针，调用 malloc()，然后使用该指针来访问数组元素。

使用第二种或第三种方法可以创建一个动态数组。

内存的动态分配

在 C99 之前，不允许这么做：

```
double item[n];
```

然而，即使在 C99 之前，也可以这么做

```
ptd = (double *) malloc(n * sizeof(double));
```

内存的动态分配

一般地，分配了内存，就应该释放内存。释放内存的主要工具是 free()。

- free() 的参数是 malloc() 返回的地址，它释放先前分配的地址。
- 所分配内存的持续时间从调用 malloc() 分配内存开始，到调用 free() 释放内存结束。

内存的动态分配

设想 malloc() 函数与 free() 管理一个内存池。每次调用 malloc() 函数分配内存供程序使用，每次调用 free() 将内存归还到池中。

内存的动态分配 I

```
1  // dyn_arr.c:
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7    double * ptd;
8    int max;
9    int number;
10   int i = 0;
```

内存的动态分配 II

```
11
12    puts("What is the maximum number of type double
      entries?");
13    scanf("%d", &max);
14    ptd = (double *) malloc(max * sizeof(double));
15    if (ptd == NULL){
16      puts("Memory allocation failed. Goodbye.");
17      exit(EXIT_FAILURE);
18    }
19    puts("Enter the values (q to quit): ");
20    while (i < max && scanf("%lf", &ptd[i]) == 1)
```

内存的动态分配 III

```
21      ++i;
22    printf("Here are your %d entries:\n", number = i);
23    for (i = 0; i < number; i++){
24      printf("%7.2f ", ptd[i]);
25      if (i % 7 == 6)
26        putchar('\n');
27    }
28    if (i % 7 != 0)
29      putchar('\n');
30    puts("Done.");
31    free(ptd);
```

内存的动态分配 IV

```
32
33    return 0;
34 }
```

内存的动态分配

```
What is the maximum number of type double entries?
8
Enter the values (q to quit):
1 2 3 4 5 6  7 8
Here are your 8 entries:
   1.00     2.00     3.00     4.00     5.00     6.00
   7.00
   8.00
Done.
```

内存的动态分配

注意在程序末尾的 free()，用于释放 malloc() 函数分配的内存。 free() 只释放其参数所指向的内存块。

该例中， free() 不是必须的，因为程序终止后所有已分配的内存都将被自动释放。

内存的动态分配

为什么会使用动态数组？主要是获得了程序的灵活性。

假定某程序在大多数时候需要的数组元素不超过 100 个；而在某些情况下，却需要 10000 个元素。在声明数组时，不得不考虑最坏情形并声明一个长度为 10000 的数组。这样一来，在大多数情况下，程序将浪费内存。

内存的动态分配： free() 的重要性 I

观察以下代码：

```
1 int main(void)
2 {
3   double glad[2000];
4   int i;
5   ...
6   for (i = 0; i < 1000; i++)
7     gobble(glad, 2000);
8   ...
9 }
```

内存的动态分配： free() 的重要性 II

```
10
11 void gobble(double ar[], int n)
12 {
13   double * tmp = (double *) malloc(n * sizeof(double)
     );
14   ...
15   // free(tmp);
16 }
```

内存的动态分配： free() 的重要性

第一次调用 gobble() 时，它创建了指针 tmp，并使用 malloc() 为之分配了 16000 字节的内存。

若没有调用 free()，当函数终止时，指针 tmp 作为一个自动变量消失了，但它所指向的 16000 字节的内存依然存在。我们无法访问这些内存，因为地址不见了。

内存的动态分配： free() 的重要性

第二次调用 gobble() 时，它又创建了指针 tmp，再次使用 malloc() 为之分配了 16000 字节的内存。
第一个 16000 字节的块已不可用，因此 malloc() 不得不再找一个 16000 字节的块。当函数终止时，这个内存块也无法访问，不可再用。

内存的动态分配： free() 的重要性

循环执行了 1000 次，当循环结束时，已经有 1600 万字节的内存从内存池中移走。事实上，在到达这一步之前，程序可能已经内存溢出了。这类问题被称为内存泄露 (memory leak)，可以在函数末尾处调用 free() 防止该问题出现。

内存的动态分配：calloc 函数

内存分配还可以使用 calloc()，典型应用如

```
long * newmem;
newmem = (long *) calloc(100, sizeof(long));
```

- calloc() 的第一个参数是所需内存单元的数量，第二个参数是每个单元的字节大小，返回类型同 malloc() 函数。
- calloc() 的另一个特性：它将块中的全部位置都设置为 0。
- free() 也可用来释放由 calloc() 分配的内存。

内存的动态分配：calloc 函数

内存分配还可以使用 calloc()，典型应用如

```
long * newmem;
newmem = (long *) calloc(100, sizeof(long));
```

- calloc() 的第一个参数是所需内存单元的数量，第二个参数是每个单元的字节大小，返回类型同 malloc() 函数。
- calloc() 的另一个特性：它将块中的全部位置都设置为 0。
- free() 也可用来释放由 calloc() 分配的内存。

内存的动态分配：realloc 函数

可以使用 realloc() 给一个已经分配了地址的指针重新分配空间，其函数原型声明为

```
void * realloc(void * ptr, unsigned newsize)
```

- 参数 ptr 为原有的空间地址；
- 参数 newsize 是重新申请的地址长度。

内存的动态分配：realloc 函数

可以使用 realloc() 给一个已经分配了地址的指针重新分配空间，其函数原型声明为

```
void * realloc(void * ptr, unsigned newsize)
```

- 参数 ptr 为原有的空间地址；
- 参数 newsize 是重新申请的地址长度。

内存的动态分配：realloc 函数

realloc() 可以对给定的指针所指的空间进行扩大或者缩小，无论是扩张或是缩小，原有内存的中内容将保持不变。当然，对于缩小，则被缩小的那一部分的内容会丢失。

realloc() 并不保证调整后的内存空间和原来的内存空间保持同一内存地址。相反， realloc() 返回的指针很可能指向一个新的地址。

内存的动态分配：realloc 函数

realloc() 是从堆上分配内存的。

当扩大一块内存空间时， realloc() 试图直接从堆上现存的数据后面的那些字节中获得附加的字节，

- 如果能够满足，自然天下太平；
- 如果数据后面的字节不够，问题就出来了，那么就使用堆上第一个有足够大小的自由块，现存的数据然后就被拷贝至新的位置，而老块则放回到堆上。

这句话传递的一个重要的信息就是数据可能被移动。

内存的动态分配：realloc 函数

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    char * p, * q;
7    p = (char *) malloc(10);
8    q = p;
9    p = (char *) realloc(p, 10);
10   printf("p = %p\n", p);
11   printf("q = %p\n", q);
12
```

内存的动态分配：realloc 函数

```
p = 0x110a010
q = 0x110a010
```

realloc() 后，内存地址没有发生改变。

内存的动态分配：realloc 函数

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    char * p, * q;
7    p = (char *) malloc(10);
8    q = p;
9    p = (char *) realloc(p, 1000000);
10   printf("p = %p\n", p);
11   printf("q = %p\n", q);
12
```

内存的动态分配：realloc 函数

```
p = 0x7fc2c9716010
q = 0x1574010
```

realloc() 后，内存地址发生了改变。

内存的动态分配：动态内存分配与变长数组

VLA 与 malloc() 在功能上有些一致，比如它们都可以用来创建一个大小在运行时决定的数组：

```
int vlamal()
{
  int n;
  int * pi;
  scanf(%d", &n);
  pi = (int *) malloc(n * sizeof(int));

  int ar[n];
  pi[2] = ar[2] = 5;
```

内存的动态分配：动态内存分配与变长数组

区别一：

- VLA 是自动存储的，其结果之一是 VLA 所用内存空间在运行后会自动释放，不必使用 free()。
- malloc() 创建的数组不必局限于在一个函数中。例如，函数可以创建一个数组并返回指针，供调用它的函数访问。然后，后者可以在它结束时调用 free()。

内存的动态分配：动态内存分配与变长数组

区别二：

- VLA 对多维数组来说更为方便。
- 也可以使用 malloc() 来定义一个二维数组，但语法较麻烦。

```
int n = 5;
int m = 6;
int ar1[n][m];    // VLA
int (* p1) [6];   // Before C99
int (* p2) [m];   // C99
```

内存的动态分配：动态内存分配与变长数组 I

```
1  // dyn_ar2d.c:
2  #include <stdio.h>
3  #include <stdlib.h>
4  int ** allocate_ar2d(int r, int c);
5  void free_ar2d(int r, int ** ar);
6
7  int main(void)
8  {
9    int i, j;
10   int r = 4, c = 5;
```

内存的动态分配：动态内存分配与变长数组 II

```
11  int ** ar;
12
13  ar = allocate_ar2d(r,c);
14  for (i = 0; i < r; i++) {
15    for (j = 0; j < c; j++){
16      ar[i][j] = i+j;
17      printf("%4d ", ar[i][j]);
18    }
19    putchar('\n');
20  }
21  free(ar);
```

内存的动态分配：动态内存分配与变长数组 III

```
22 }
23
24 int ** allocate_ar2d(int r, int c)
25 {
26   int i;
27   int ** ar = (int **) malloc(r * sizeof(int *));
28
29   for(i = 0; i < r; i++)
30     ar[i] = (int *) malloc(c * sizeof(int));
31
32   return ar;
```

内存的动态分配：动态内存分配与变长数组 IV

```
33 }
34
35
36 void free_ar2d(int r, int ** ar)
37 {
38   int i;
39
40   for(i = 0; i < r; i++)
41     free(ar[i]);
42   free(ar);
43 }
```

内存的动态分配：动态内存分配与变长数组 V

内存的动态分配

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```