09 内存与变量生存期

局部 static 与动态内存

本讲学习目标

- 重点内容
  - 理解局部 static 如何在函数调用之间保留状态。
  - 读懂简单动态数组的申请、使用和释放过程。
- 概念性了解
  - 名称的作用域与对象的生存期。
  - 分配失败、释放责任和常见内存管理错误。

内存：从使用对象到管理存储

程序 = 数据对象 + 操作 + 控制 + 组合

- 数据对象具有类型和值，也具有一定的**生存期**。
- 程序可通过静态存储保留对象状态，也可动态申请存储并负责释放。

1 名称与局部 static

区分名称范围与对象存在时间，理解跨调用保留的状态

局部名称的作用域

```
1 void show(void) {
2     int x = 7;
3     printf("%d\n", x);
4 }
5 // 函数外不能直接使用这里的名称 x
```

名称的作用域

x 从声明处到这个块的末尾可直接使用。

这种名称可使用的范围称为**作用域**。

本例的边界

本例没有内层同名变量，只讨论块内名称。

普通局部对象的生存期

```
1 {
2     int x = 7;
3     int *p = &x;
4     printf("%d\n", *p);
5 } // 退出块，普通局部对象结束
```

对象存在的时间

生存期是执行期间为对象保留存储的时间段。

本例中，退出块后普通局部对象 x、p 的生存期结束。

作用域与生存期的区别

作用域看代码位置；生存期看执行过程。

保存地址不能延长普通局部对象的生存期。

预测：连续三次调用的输出

```c
#include <stdio.h>
void show(void) {
    int fresh = 1;
    static int saved = 1;
    printf("%d %d\n", fresh, saved);
    fresh++;
    saved++;
}
int main(void) {
    show();
    show();
    show();
    return 0;
}
```

连续调用 show 三次

写出每次输出的 fresh 和 saved。

哪个变量的值会在调用之间保留？

阅读条件

printf 在递增之前执行。

static 用于声明可跨调用保留的局部变量。

三次调用的输出核对

调用	fresh 输出值	saved 输出值	saved 递增后
第1次	1	1	2
第2次	1	2	3
第3次	1	3	4

- 输出依次为1 1、1 2、1 3。fresh 每次从1开始，saved 保留上次修改的值。
- saved 的名称仍只在块内可用，为什么下一次调用还能接续原值?

局部 static 的生存期与初始化

```c
void show(void) {
    int fresh = 1;
    static int saved = 1;
    printf("%d\n", saved);
    saved++;
}
```

对象的生存期

fresh 每次调用都有新的对象。

多次调用访问同一个 saved 对象。

saved 具有**静态存储期**，生存期覆盖整个程序执行。

名称 saved 仍只在块内可用。

初始化与修改

saved 在程序启动前初始化一次，下次调用不重置为1。

saved++ 仍可修改它的值。

静态整数的零初始化

```c
void count_calls(void) {
    static int count;
    count++;
    printf("%d\n", count);
}
```

count 的初始值

未显式初始化的静态整数初始化为0。

连续三次调用输出1、2、3。

使用范围

适合记录函数已经调用的次数。

名称 count 仅在块内可用，对象仍存在。

练习：两次调用的输出

```c
void next_value(void) {
    int temp = 10;
    static int total = 10;
    temp += 2;
    total += 2;
    printf("%d\n", temp);
    printf("%d\n", total);
}
// 连续调用 next_value 两次
```

两次调用分别输出什么?

每次先输出 temp，再输出 total。

用“是否重新初始化”解释两者的差别。

本例的初始化规则

temp 每次执行声明时初始化。

total 作为局部 static 初始化一次。

两次调用的输出核对

调用	temp 初始化后	total 进入调用时	两次递增后的输出
第1次	10	10	temp=12，total=12
第2次	10	12	temp=12，total=14

- temp 每次从10开始；total 从上次保留的12接续增加。
- 名称只在块内可用，并不意味着对象只能在一次调用中存在。

2 动态数组的申请、使用与释放

联系元素数量与申请量，读懂动态数组的完整正常过程

动态内存与元素数量

- 元素数与存储
  - int values[3] 为三个整数提供存储；已定义的这个数组不会因输入更多数据而自动扩大。
  - 若数量到运行时才确定，可按 n 申请存储；成功后按下标访问，存储不会自动扩容。
- 动态内存
  - 运行时显式申请、使用后释放的存储称为**动态内存**。
  - 通过指针访问申请得到的对象。
- 本讲示例
  - 固定 n 为3，便于追踪；申请量仍用 n 计算。
  - 依次申请并检查、给元素赋值、使用后释放。
  - 保存地址的指针与动态对象分别追踪。

malloc 的申请量与结果检查

```c
#include <stdio.h>
#include <stdlib.h>
// main 中的片段
int n = 3;
int *p = malloc(n * sizeof *p);
if (p == NULL) {
    puts("Allocation failed");
    return 1;
}
// 通过检查后才访问元素
```

申请量

malloc 按字节申请：n * sizeof *p。

n 是元素个数；sizeof *p 是一个 int 的大小，不读取 *p。

结果检查

成功返回起始地址，失败返回 NULL。

C 中可直接赋给 int *，无需强制转换。

失败分支输出提示，return 1 结束程序。

成功后的访问

通过检查后才访问申请范围内的元素。

malloc 不初始化元素，读取前先赋值。

本例 n 为3，申请量计算可表示。

指针与动态数组

- 图示时点：malloc 已成功，尚未释放。
- p 是一个指针对象；申请得到的三个元素是另一段存储。
- 分配后必须给元素赋值，才能读取预期内容。



预测：动态数组的值与下标

```
1 // 已成功申请，n 为3
2 for (int i = 0; i < n; i++)
3     p[i] = (i + 1) * 10;
4 printf("%d %d %d\n", p[0], p[1], p[2]);
```

根据循环计算

写出 p[0]、p[1]、p[2] 的值和输出。

这段循环访问哪些下标?

申请量与元素数

为3个 int 申请的字节数应怎样表示?

不假定每个 int 固定占4字节。

动态数组的值与下标核对

循环中的 i	写入位置	写入的值
0	p[0]	10
1	p[1]	20
2	p[2]	30

- 输出为10 20 30；有效下标是0、1、2，p[3] 越界。
- 申请量为3 * sizeof(int) 字节，三个元素均先写入后读取。

free 与动态对象的生存期

```
1 // 元素已经使用完毕
2 free(p);
3 p = NULL;
4 // 此后不再访问 p[i]
```

释放对象

free(p) 归还所申请的存储，结束动态对象的生存期。

传入尚未释放的分配起始地址；free(NULL) 不做操作。

修改指针

p = NULL 只改变 p 保存的值。

free 不自动把 p 或其他指针变为 NULL。

练习：动态数组的执行过程

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int n = 3;
    int *p = malloc(n * sizeof *p);
    if (p == NULL) {
        puts("Allocation failed");
        return 1;
    }
    for (int i = 0; i < n; i++)
        p[i] = (i + 1) * 10;
    printf("%d %d %d\n", p[0], p[1], p[2]);
    free(p);
    p = NULL;
    return 0;
}
```

申请与失败路径

申请几个元素？共需多少字节？

申请失败时，哪条语句结束程序？

成功路径

哪些语句保证三个元素先赋值再读取？

哪个调用结束动态对象？

p = NULL 修改了什么？

阅读条件

n 为3，申请量可表示；分别判断成功和失败路径。

动态数组的执行过程核对

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int n = 3;
    int *p = malloc(n * sizeof *p);
    if (p == NULL) {
        puts("Allocation failed");
        return 1;
    }
    for (int i = 0; i < n; i++)
        p[i] = (i + 1) * 10;
    printf("%d %d %d\n", p[0], p[1], p[2]);
    free(p);
    p = NULL;
    return 0;
}
```

分配与检查

申请3 * sizeof *p 字节。

失败时执行 return 1，不访问元素。

赋值与输出

循环先给三个元素赋值，printf 随后读取。

输出为10 20 30。

释放与结束

free(p) 结束动态对象。

p = NULL 只修改指针。

本例 n 为3，申请量可表示。

3 动态对象与释放责任

分别追踪指针和所指对象，认识责任交接与常见错误

判断：赋 NULL 后的对象状态

```c
// 假设申请成功
int *p = malloc(3 * sizeof *p);
p = NULL;
```

判断两个对象的状态

p 保存的值是什么?

原申请的动态对象是否仍存在?

这段代码是否完成了释放?

题目条件

没有其他指针保存原申请空间的地址。

只分析状态，不执行这段示例。

赋 NULL 与内存泄漏

- 指针对象 p
  - p 仍存在，保存的值变为 NULL，已失去原地址。
- 原动态对象
  - 没有调用 free，因此对象仍存在。
  - 没有其他指针保存原地址，已失去释放途径。
- 内存泄漏
  - 不再需要的已分配存储未释放，且失去释放途径，称为**内存泄漏**。
  - 正常顺序是先 free(p)，再按需要将 p 置为 NULL。

判断：函数返回后的对象状态

```c
int *make_pair(void) {
    int *p = malloc(2 * sizeof *p);
    if (p != NULL) {
        p[0] = 7;
        p[1] = 9;
    }
    return p;
}
```

调用与时点

调用者执行 int *data = make_pair();

申请成功、函数已返回，尚未调用 free。

判断两个对象

p 的生存期是否已结束?

动态数组是否仍存在?

data 保存了什么？为什么可用它访问元素?

本例约定

由调用者在使用完毕后负责释放。

根据该约定，释放应放在调用者的什么位置?

函数返回后的地址与动态对象

- 时点：make_pair 成功返回，尚未执行 free。
- 普通局部指针 p 的生存期已结束。
- return p 复制地址值，data 保存该副本。
- 动态数组尚未释放，data 仍可用于访问元素。
- 本例由调用者在最后一次使用后 free(data)。



动态对象的返回与释放

```c
#include <stdio.h>
#include <stdlib.h>
int *make_pair(void) {
    int *p = malloc(2 * sizeof *p);
    if (p != NULL) {
        p[0] = 7;
        p[1] = 9;
    }
    return p;
}
int main(void) {
    int *data = make_pair();
    if (data == NULL) return 1;
    printf("%d\n", data[0] + data[1]);
    free(data);
    return 0;
}
```

创建与返回

make_pair 申请两个整数的存储，返回地址或 NULL。

调用者的责任

main 检查返回值，成功后求和并输出16。

main 使用完毕后调用 free(data) 一次。

所指对象的生存期

普通局部整数随函数返回结束。

动态数组保留到显式释放，可返回其地址。

常见内存管理错误

问题	对象状态	基本含义
内存泄漏	对象仍存在	不再需要的存储未释放，失去释放途径
重复释放	对象已经释放	再次释放同一段已归还的存储
释放后访问	对象生存期结束	继续读取或写入原对象

- 重复释放和释放后访问属于未定义行为，不能预测固定输出。
- free 的非空参数须为未释放的分配起始地址，不能用局部或偏移地址。

多个指针与同一动态对象

```c
// 假设申请成功
int *p = malloc(sizeof *p);
int *q = p;
*p = 7;
free(p);
p = NULL;
// 不再读取 q，也不访问 *q 或 free(q)
```

释放之前

p 与 q 保存同一地址，可访问同一对象。

释放之后

动态对象的生存期已结束。

q 不能再用于访问或释放该对象。

p = NULL 只修改 p，不会同时修改 q。

三类对象的生存期

对象	名称或访问方式	生存期的结束
普通局部对象	块内名称或有效地址	所属块执行结束
局部 static 对象	名称仍局限于块内	程序执行结束
动态分配对象	通过有效指针访问	显式释放

- 局部 static 保留跨调用状态；动态对象按申请与释放管理。
- 名称是否可见、对象是否存在、指针是否仍有效，需要分别判断。

教材阅读要求

阅读要求	《C Primer Plus》第12章	本讲重点
重点阅读	§12.1.6	局部 static、初始化和跨调用状态
重点阅读	§12.4、12.4.1	malloc/free、动态数组及释放责任
概念性查阅	§12.1.1、12.1.3 ~ 12.1.4	作用域、存储期和普通局部对象

- 完整存储类别、链接、多文件组织及其他分配接口不作为本讲重点。
- 复习 static 与动态数组的完整程序，说明对象生存期、值变化和释放责任。