信手相连——面向过程的程序设计 (C 语言)

2. 数组与函数

崔冠宇

2018202147

https://github.com/GuanyuCui/RUC-infohands-2020

信息学院
中国人民大学

2020-2021 学年秋季学期
(11.14, 14:00-15:30, 教二 2115)

目录

进度

上次完成了数据类型和控制结构的讲解，今天开启新的内容——数组和函数。

- ✓ 基础知识（计算机结构、代码风格与规范……）；
- ✓ 数据类型（长短整、单双精度浮点、字符……）；
- ✓ 控制结构（顺序、选择、循环）；
- → 数组；
- → 函数；
- 指针；
- 自定义类型（枚举、联合体、结构体）；
- 文件；
- 提高知识（递归、一点数据结构、大作业指导）。

一维数组的定义与初始化

首先，数组是用来存放一组相同数据类型[1]的有序结构。

定义一个一维数组的一般格式是：

```
typename arrayName[arrayLen] [= {initValues}];
```

定义时可以在大括号将数组的值初始化或部分初始化，但只有定义时能用大括号初始化。如果是对数组全部的值初始化，则数组的长度可以省略。

对于数组元素的引用应该使用中括号，注意数组元素的下标从 0 开始。

虽然 C99 标准中引入了变长数组[2]（variable-length array，VLA），允许使用整型变量作为数组长度，但还是推荐数组长度使用常量。

[1]可以是 C 自带的类型，也可以是用户定义的类型，如结构体。

[2]尽管叫变长数组，但根据英文名也可以知道是以变量作为数组长度，而不是运行时数组长度可变。

二维数组的定义与初始化

逻辑上，二维数组可以看成一张二维的表格，表格内部装有相同类型的数据。从另外一个角度，还可以把二维数组看作特殊的一维数组，其中每个元素都是一维数组。

a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3]	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

定义一个二维数组的一般格式是：

```
typename arrName[len1][len2] [= {initValues}];
```

其它高维数组可以按规律推广。

二维数组的定义与初始化

二维数组的初始化相较一维数组而言，方法更多。

1. 可以按行初始化（即从"嵌套数组"的角度看）：
```
int a[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```
2. 可以把上述初始化写在一个括号里，按照两个分量从小到大的顺序：
```
int a[2][4] = {1, 2, 3, 4, 5, 6, 7, 8};
```
3. 可以对部分元素初始化：
```
int a[2][4] = {{1}, {5, 6, 7}};
```
4. 在前两种情况下，可以省略第一维的长度：
```
int a[][4] = {1, 2, 3, 4, 5, 6, 7, 8};
```

总之，要让编译器能够完全确定数组的形状及内容。

字符数组、空字符结尾字符串

字符数组是存储字符的数组。它的定义、初始化的方法和普通（一维/二维）数组是一样的。

ASCII 码中有一个特殊字符——空字符 nul，它的值为 0，写作`'\0'`，在 C 语言中主要用于占字符数组后面的空位，同时标记字符串实际内容的结束。（设想一下，如果没有这个字符，在下面的情境中，我们将难以确定字符数组的实际内容在何处结束。）

大家要熟练使用字符串操作函数 puts、gets、strcat、strcpy、strcmp、strlen 等，了解它们的功能[3]。

[3] https://zh.cppreference.com/w/c/string/byte

字符变量、字符数组的输出

字符变量（char）类型的变量输出时，可以用 printf("%c", c) 实现，也可以用 std::cout << c 实现。

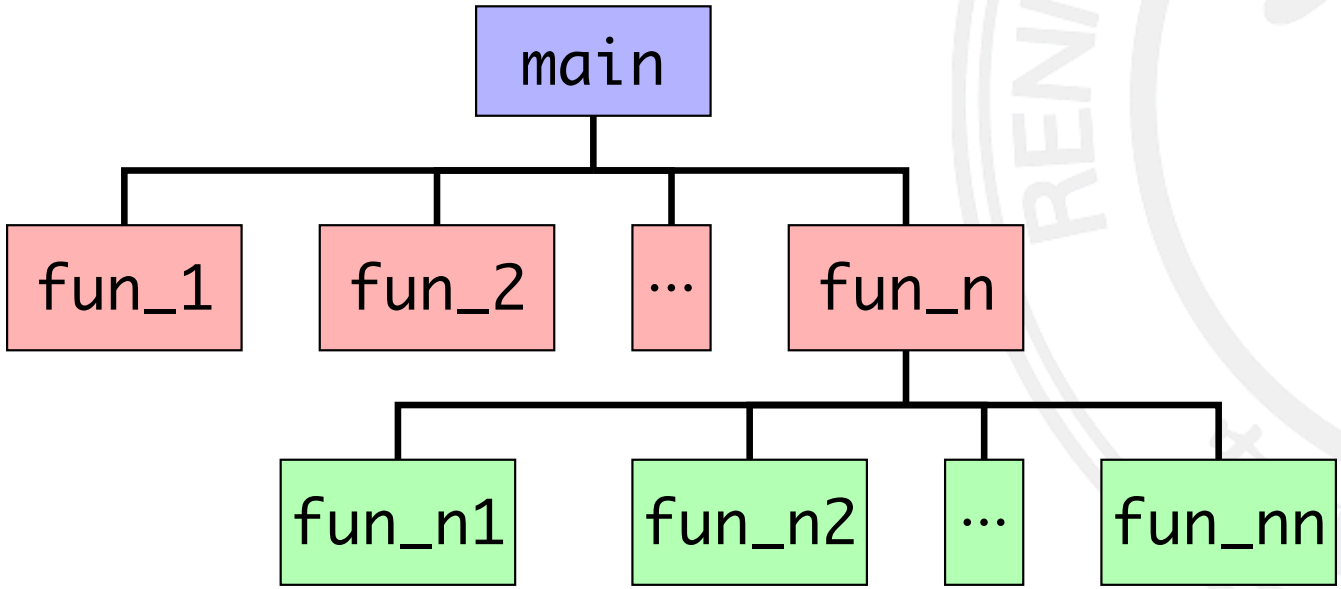
字符数组（char []，也可不严谨地称为“字符串”）类型的变量输出时，可以用 printf("%s", s) 实现，也可以用 std::cout << s 实现。前者的原理可以简单理解成循环输出 s[i] 直到遇到空字符为止。

这也是为什么要在字符数组后面补空字符，如果不补充的话，printf 将会无法区分字符数组在何处结束，从而把内存中相邻的其它变量信息输出了。

函数

函数的思想是将部分需要复用的代码封装起来，可以简化逻辑、减少代码重复、使程序结构更清晰。封装好的函数可以由其它函数调用。

理论上讲，C 程序从 main 函数开始执行[4]，main 函数调用其它函数，从而形成整个程序的树状调用结构。



[4]以 Linux 为例，实际上操作系统调用 execve 来加载程序，然后控制权交给程序内的 _start 函数，后面调用 __libc_start_main，然后才轮到 main。

函数定义、形参与实参

定义一个函数的方式如下：

```
1 typename functionName(paramList)
2 {
3   // 变量声明语句和执行语句;
4 }
```

在函数定义时参数列表里的参数叫形式参数，简称"形参"；调用函数时，真正传给函数的参数值称为实际参数，简称"实参"。

在定义函数时，要写清形参的类型；实参向形参传递值时，必须类型匹配；实参向形参传递值是单向的值传递，即只能由实参向形参传递，不能相反。而且形参只在本函数内有效。

函数返回值也要和定义的匹配，否则将尽可能进行转换；对于 void 型函数，可以写成"return;"或不出现 return。

函数调用

为了使用一个定义好了的函数，需要调用它。调用函数的方式如下：

```
functionName(paramList)
```

向函数传递实参列表时，需要与形参一一对应，用逗号隔开；此外还要注意，如果实参列表有表达式，C 语言标准对表达式的求值顺序并没有要求，所以要避免求值顺序对实参的值造成影响。

如果要调用一个函数，C 语言要求在调用之间这个函数已经"存在"；具体来说，就是要求编译器知晓函数名、函数返回值类型以及函数参数列表中各个参数的类型。这便引出了函数声明的概念。

函数声明

调用函数之前，必须保证编译器能够了解这个函数。

1. 如果这个函数是库函数或在其它文件中，则应该使用 #include 命令引入该文件和函数声明语句；
2. 如果这个函数是用户定义的，则在调用前应该定义函数，或是写出函数声明。

函数声明的一般形式为：

```
typename functionName(typename1 [varName1,
        typename2 varName2, ...]);
```

注意：函数定义与函数声明是不同的，前者要包含函数的具体实现，而后者只是告诉编译器存在这样一个函数（所以参数列表可以只写出数据类型，而不必写出参数的名字）。

函数嵌套与递归

函数内可以调用其它函数，称为函数的嵌套调用。

一个函数还可以调用它自己，称为函数的递归调用。

前者在逻辑上比较好理解，使用也较为普遍；而后者较难理解，使用相对较少，但也有用。

但是，递归是非常有用的，在许多算法中都有应用。如：快速排序、二叉树遍历、求解某些函数/数列、上楼梯、汉诺塔等问题。

当你发现某个问题可以经过一些处理，转变为规模更小的问题去解决的时候，就要考虑递归了。

大家上半学期的学习重点并不在递归等进阶问题上，所以更详细的知识我们留到后面讲。

函数的数组参数

函数定义 / 调用时可以传入数组作为参数，主要有下面几种情况：

1. 数组元素作为实参：此时函数的形参是数组的元素类型（不是整个数组类型），与普通简单参数的函数一致；
2. 整个（一维）数组作为参数：此时函数的形参应该写成数组类型 `typename arr[len]` [5]，也可省略长度写作 `typename arr[]`，在调用时实参写数组名即可；
3. 多维数组作为参数：此时函数形参也应该写成数组类型，但是要注意，最多可以省略第一维的长度，其它维度不能省略，在调用时实参写数组名即可。

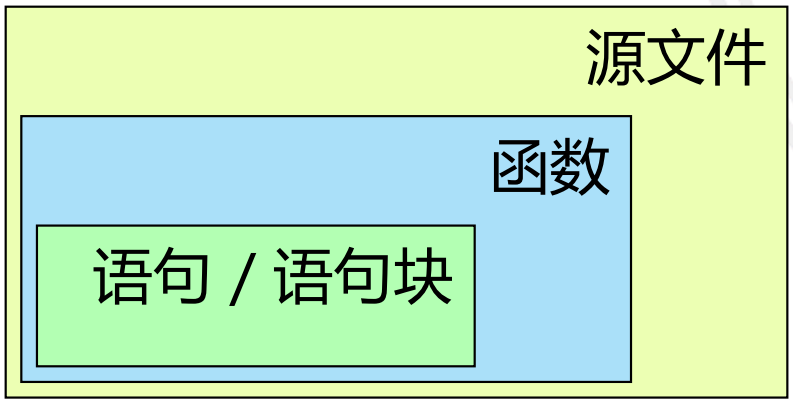
[5]也可写作指针类型 `typename * p`。就算写出数组长度，传递参数时长度也可以不匹配，因为编译器并不检查第一维长度。

变量的作用域——局部变量与全局变量

C 语言中的变量有其起作用的范围，称为变量的作用域。根据变量作用域的大小，可以将其分为局部变量和全局变量。

局部变量是指在某个函数、语句（如循环语句）或语句块（{} 包围的部分）定义的变量，它只能在定义它的范围内起作用（同一层范围内不能重名，不同的范围内有重名覆盖现象），因此被称为局部变量；

全局变量是指在所有函数之外定义的变量，它能在该编译单元（简单理解就是整个源文件）内起作用，因此被称为全局变量。



变量的作用域——局部变量与全局变量

一般不建议使用全局变量，理由如下：

1. 全局变量会在整个程序运行周期存在，而局部变量是动态存在的，所以全局变量稍微浪费内存；
2. 容易受外部干扰，可能出现同名冲突，降低了程序的逻辑清晰度；

变量的生命周期——动态变量与静态变量

在上面，我们从作用域角度讨论变量，下面再从生命周期来看变量：

动态变量是指调用函数时临时分配内存单元的变量。一般的局部变量是动态变量；

静态变量是指在程序运行的整个周期都存在的变量。全局变量和由 `static` 修饰的（局部）变量是静态变量。

`static` 修饰的静态局部变量需要说明以下几点：

1. 存放在静态存储区，整个运行期间不释放；
2. 编译时赋初值（默认为 0），每次调用时不再重新赋值；
3. 虽然每次调用不重新赋值，调用结束不释放空间，但是也只有对应函数可以访问（局部性）。

局部/全局、动态/静态变量是两个不同角度，二者没有必然联系。寄存器变量、外部变量等，由于不太常用，此处略。

数组的内存布局

数组在内存中是线性排列的，这对于一维数组很好理解，但是要注意，高维数组虽然逻辑上是一个“多维长方体”，但它在内存中也是线性排列的，而且是按行优先的顺序排列的，如下图：

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

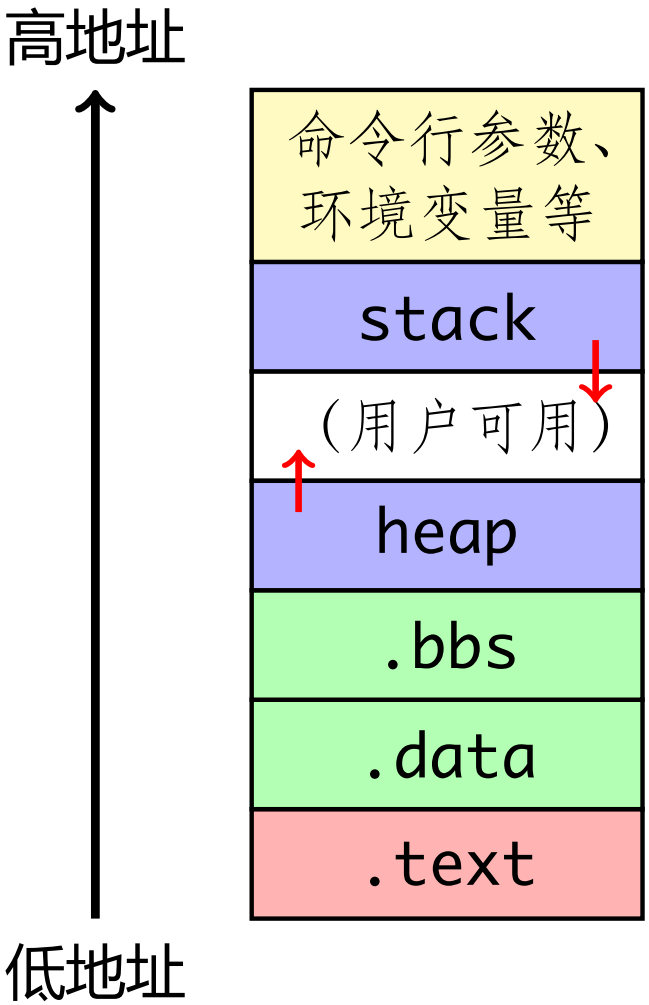
⇐ 逻辑结构

⇓ 物理结构

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

其中，数组名本身就是一个地址常量（指针常量），也可以做算术运算，详细情况将会在在指针部分讲解。

变量（在程序中）的内存布局



1. 栈区（stack）：自动分配释放，存储函数参数，局部变量等，操作方法类似数据结构中的栈；
2. 堆区（heap）：一般由程序员申请和释放，与数据结构中的堆没有关系，分配方式类似链表；
3. 未初始化静态 / 全局变量区（.bbs）：在程序编译时分配；
4. 已初始化静态 / 全局变量区（.data）：在程序编译时分配；
5. 程序代码区（.text）：存储函数的二进制代码。

排序 (sorting)

排序，是计算机科学中的重要而又意义深远的问题，也是大家上半学期接触到的第一类主要算法。

现阶段大家常见的普适排序算法（基于比较排序字段）有冒泡排序（原理和代码必会）、选择排序（原理和代码必会）、插入排序（了解）、归并排序（了解）、快速排序（原理了解，会调用库函数 qsort[6]即可）。除此之外，还有希尔排序、堆排序、基数排序、桶排序等方法。

有趣的是——"与名称无关，C 和 POSIX 标准都未要求此函数用快速排序实现，也未保证任何复杂度或稳定性。"

注意

与名称无关，C 和 POSIX 标准都未要求此函数用快速排序实现，也未保证任何复杂度或稳定性。

与其他边界检查函数不同， qsort_s 不将零大小数组视作运行时强制违规，而是不修改数组并成功返回（另一个接受零大小数组的函数是 bsearch_s ）。

在 qsort_s 之前，qsort 的用户通常用全局变量来将附加语境传递给比较函数。

[6] https://zh.cppreference.com/w/c/algorithm/qsort

排序 (sorting)

下面简单回顾常见排序的思想（假定按关键字升序排序）：

- 冒泡排序：算法维护数组有序区和无序区。每次循环遍历无序区，并进行两两比较，若有逆序，则交换。
- 选择排序：算法维护数组有序区和无序区。每次循环遍历无序区，选出无序区的最值，放入有序区。
- 插入排序：算法维护数组有序区和无序区。每次循环将无序区的第一个元素逐个比较，挪到有序区的合适位置。（类似打牌时摸牌的方法。）
- 归并排序：递归对数组两部分进行归并排序，然后将它们保持有序合并。
- 快速排序：算法每次选定一个元素作为主元，并以其为标准运行划分算法，将比主元小的放到左边，比主元大的放到右边，然后对两侧递归调用快速排序。

排序 (sorting)

下面给出各排序算法的时间复杂度、辅助空间和稳定性：

$$T(n) = O(f(n)) \Leftrightarrow \overline{\lim_{n\to+\infty}} \frac{T(n)}{f(n)} < +\infty$$

排序算法	最好情况	平均	最坏情况	辅助空间	稳定性
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	是
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	否
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	否
希尔排序	$O(n\log n)$	$O(n^{4/3})$	$O(n^{3/2})$	$O(1)$	否
猴子排序	$O(n)$	$O(n\times n!)$	∞	$O(1)$	否

奇妙的宏定义——以 MAX 为例子

因为宏定义 (#define) 本质是在预处理阶段的扫描替换，所以可能有潜在危险。

例

定义一个宏，求两数中较大的数。

- 合格版：

```
1 #define MAX(x, y) x > y ? x : y
```

考虑 MAX(1 != 1,1 != 2)?(> 优先级高于 !=)

- 中级版：

```
1 #define MAX(x, y) (x) > (y) ? (x) : (y)
```

考虑 3 + MAX(1, 2)?

奇妙的宏定义——以 MAX 为例子

- 良好版：

```
1 #define MAX(x, y) ((x) > (y) ? (x) : (y))
```

考虑 MAX(i++, j++)?(i = 2, j = 6)

- 优秀版：

```
1 #define MAX(x, y)({     \
2     typeof(x) _x = (x); \
3     typeof(y) _y = (y); \
4     (void) (&_x == &_y);\
5     _x > _y ? _x : _y; })
```

考虑 MAX(x, _x)？

奇妙的宏定义——以 MAX 为例子

- Linux 无敌版：

```
1  #define __max(t1, t2, max1, max2, x, y)({\
2      t1 max1 = (x);                        \
3      t2 max2 = (y);                        \
4      (void) (&max1 == &max2);              \
5      max1 < max2 ? max1 : max2;})
6
7  #define ___PASTE(a,b) a##b
8  #define __PASTE(a,b) ___PASTE(a,b)
9
10 #define __UNIQUE_ID(prefix) __PASTE(__PASTE(__UNIQUE_ID_,
       prefix), __COUNTER__)
11
12 #define max(x, y)                           \
13     __max(typeof(x), typeof(y),             \
14     __UNIQUE_ID(max1_), __UNIQUE_ID(max2_), \
15     x, y))
```

补充：How do computers read code?

观看下面的视频[7]，了解计算机是如何理解代码的。

[7] Frame of Essence. Youtube.
https://www.youtube.com/watch?v=QXjU9qTsYCc

致谢

感谢大家到场支持，也欢迎大家积极提出意见和建议。大家也可以谈谈想要在信手相连的课程中听什么内容，我将酌情做出调整。