信手相连——面向过程的程序设计 (C 语言)

3. 指针与自定义类型

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
(11.28, 10:00-11:30, 教二 2118)

目录

进度

上次完成了数组和函数的讲解，今天开启新的内容——指针与自定义类型。

- ✓ 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）。

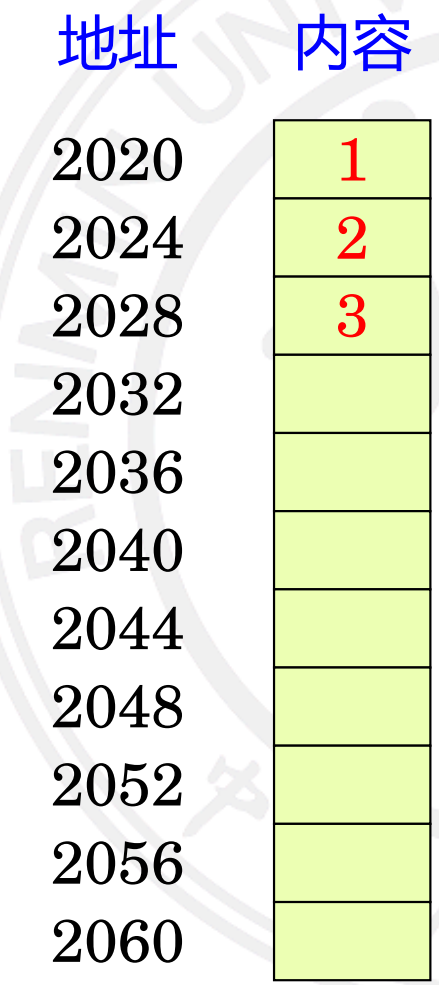
回顾——内存模型

C 语言中的指针与内存地址息息相关，想要搞清指针的知识需要大致了解一下内存模型，所以先向大家简单介绍一下变量在内存中的存储。

假若有下列代码：

```
1 int arr[ ] = {1, 2, 3};
```

如果数组 a 的起始地址为 2020，以及 sizeof(int) == 4，则该数组在内存中的安放如右图：



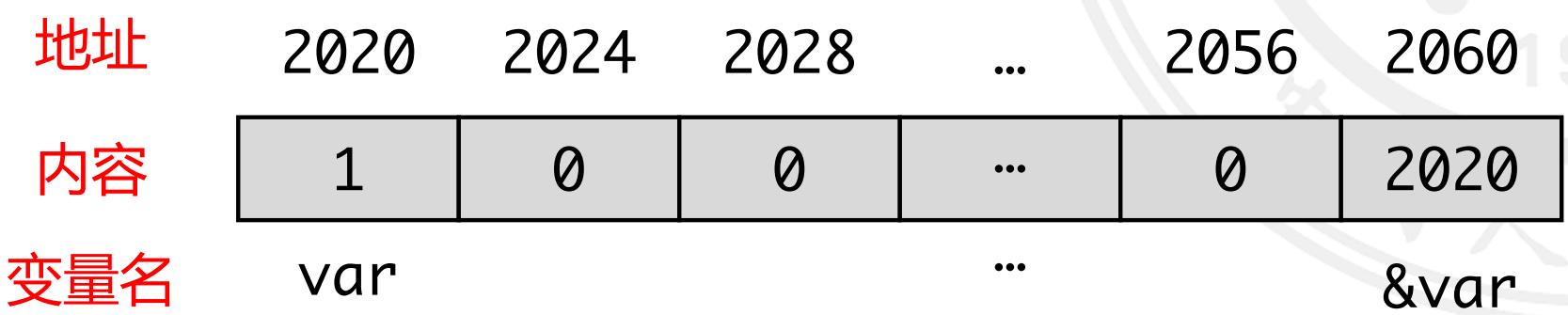
指针的概念

就如真实的世界中有地址一样，内存中的每个单元也有一串数字对应作为它的地址。某个变量所处的内存地址称为该变量的指针。

学习指针的相关知识时，请时刻记住一件事：

指针的本质是内存地址。

或者更本质的，是一个数字。



指针变量

既然变量的指针是一个值，它也可以存储到某个变量中，存储某个变量的指针的变量称为指针变量。

定义一个指针变量的一般格式是：

```
typename * pointerName [= address];
```

如果要获得一个变量的地址，可以在变量名前面加 &，如 &var。

关于指针，请注意以下几点：

1. * 表示这是一个指针变量，变量名中并不包括 *，如上面的例子中 pointerName 是变量名；
2. 指针变量如果在定义时初始化，应当与后面取地址的变量的类型保持一致；

```
1 float a;
2 int * p = &a;   // 错误。
```

指针变量

❸ 正常的指针应该指向存在的变量等实体。虽然指针的本质是一串数字，编译器也允许用户将整型数字转类型后赋值给指针变量，但是由于乱赋值而访问不存在的变量的行为会引发运行时错误（runtime error，RE），如：

```
1 int a = 0;
2 int * p = &a;  // 正确。
3 p = (int *) 1;  // 有隐患，该地址不一定有效。
4 * p = 2020;  // 很可能会发生错误。
```

C 语言的指针乱飞是一个令人头痛的问题。

回顾——常量

常量是指在运行时不可改变的量，主要有以下几类：

1. 字面量：比如 1.0f (float)，1024 (int)，'a' (char)，"C++" (const char *)。
2. 宏定义的符号常量：比如 #define PRICE 30。因为宏定义本质上是预处理阶段由预处理器扫描源代码进行替换，所以在后续使用中不能对其赋值。
3. const 关键字定义的常变量：如 const int a = 1023。此处可能难以理解，a 是变量，但是 const 关键字会提示编译器检查源代码，不允许直接显式地修改该变量的值。

```
1 const int a = 0;
2 //a = 2;  // 报错!
3 int * p = (int *)&a;
4 *p = 2;  // 允许，但不提倡!
```

回顾——容易让人晕头转向的 const 关键字详解

当 typename 是 int *、char * 等指针类型[1]时，两种写法功能不等价，如：

```
1 const int * pa;  // 指针值可变，指向的值不可变
2 int const * pb;  // 同上
3 int * const pc;  // 指针值不可变，指向的值可变
4 const int * const pd;  //  二者都不可变
5 int const * const pe;  //  同上
```

简单记忆 / 理解方法：const 仅修饰后面离的最近的部分。

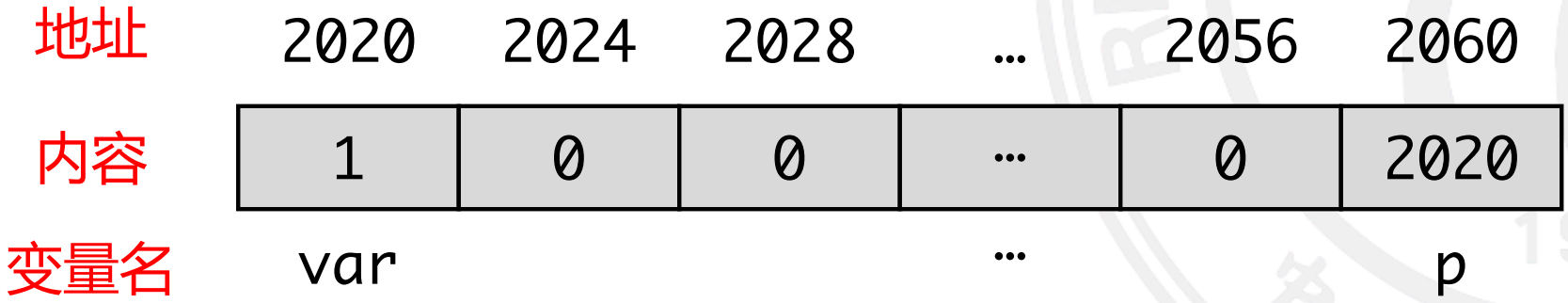
[1]按进度来说超前了，大家可以暂时不理解，等讲到指针时会再提。

深入理解 & 和 *

指针有关的运算符最基本的是 & 和 *，下面讲一下如何简单理解二者的功能：

& 符号的功能是“取地址”，在一个变量前加一 &，表示取得它在内存中的地址。

* 符号的功能是“顺着地址找变量”，可以理解成取出该变量存的地址，然后把当前操作的目标定位到该地址上。



在上图中，&p == 2060，*p = 1010 则会将地址 2020 处 var 的值修改为 1010。

数组与指针

在上一节中，我们知道数组在内存中是连续存放的，数组和数组元素也具有地址，因而指针的概念对它们也适用。

对于数组的元素而言，可以把它们看作一个相对独立的变量，他们的指针和普通变量的指针类似，如下：

```
1 int a[10] = {0};
2 int * p = &a[3];  // 指向数组 a 的第四个元素。
```

其实数组和指针还有更深一层的联系。

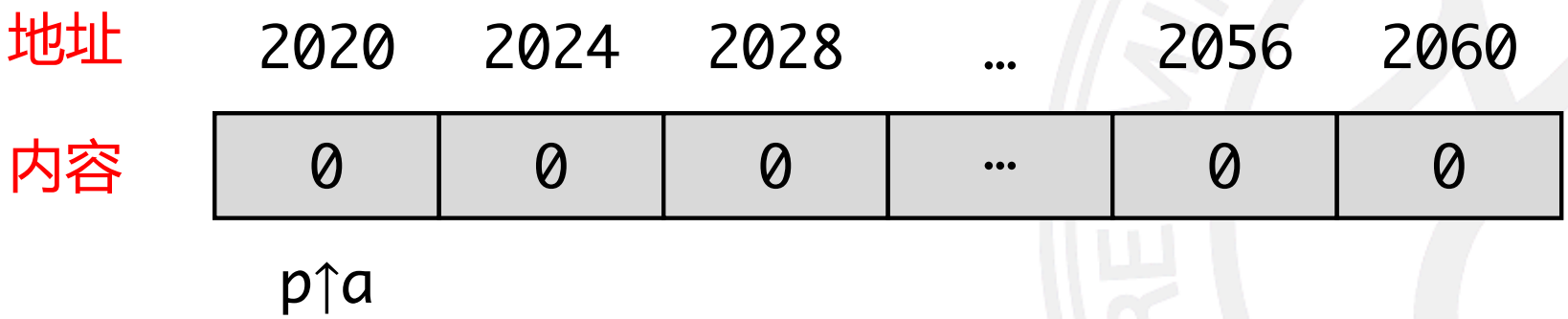
第一点，数组名也是一个地址，故而可以作为指针的值赋给元素[2]。以一维数组为例：

```
1 int a[10] = {0};
2 int * p = a;  // 指向 a 的开头，即第一个元素。
```

[2]注意：编译器在编译过程中自动给数组分配固定的地址，所以数组名代表的地址是常量，不能给它赋新值。

数组与指针

指针变量既然是一种变量，底层看还是一种数字，那么自然有加减等运算。这种特性经常与数组和循环相结合。



上图是定义 `int a[10] = {0};` 和 `int * p = a;` 后数组的一种可能情况。

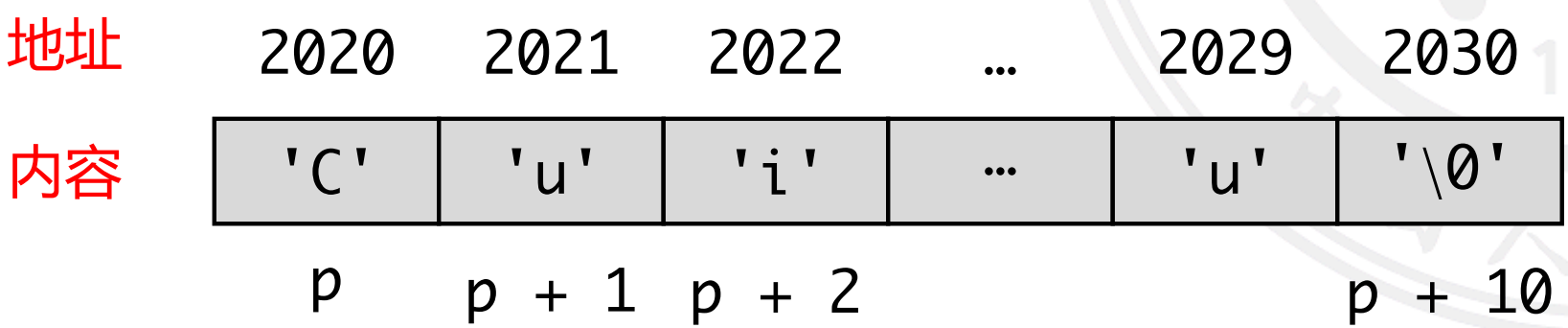
那么 `p + 1` 代表什么呢？是将 `p` 的地址直接加一吗？不是的。如果是这样的，以上面的数组为例，如果 `p + 1 == 2021` 的话，就无法对齐任何变量了。

数组与指针

所以说指针的加减，打个比方，就是将内存按照指针基类型的字节数划分成大小相同的方格，然后移动指针，指向不同的方格。

p + i 的逻辑含义是将 p 向高地址移动 i 个对应类型的偏移量，物理含义是将 p 的地址增加 i * sizeof(p 的基类型)。因此，a[i] 和 *(a+i) 是等效的。

对于普通的整型 int 来说，它占用四个字节，p + 1 对应的地址就是 p 的值 +4；对于 char 来说，它占用一个字节，p + 1 对应的地址就是 p 的值 +1；以此类推。



常用字符/字符串操作函数

字符分类：<ctype.h>

- isalnum——检查一个字符是否是字母或数字
- isalpha——检查一个字符是否是英文字母
- islower——检查一个字符是否是小写字母
- isupper——检查一个字符是否是大写字母
- isdigit——检查字符是否为数字
- isspace——检查一个字符是否是空白字符
- isblank——检查一个字符是否是空格字符

字符操作：<ctype.h>

- tolower——将字符转换成小写
- toupper——将字符转换成大写

常用字符/字符串操作函数

转换成数值格式：<stdlib.h>

- atof——将字节字符串转换成浮点值
- strtof——同上
- atoi/atol/atoll——将字节字符串转换成整数
- strtol/strtoll——同上
- strtoul/strtoull——将字节字符串转换成无符号整数

字符串操作：<string.h>

- strcpy/strncpy——复制一个字符串给另一个
- strcat/strncat——连接两个字符串

常用字符/字符串操作函数

字符串检查：<string.h>

- strlen——字符串长度
- strcmp/strncmp——字符串比较
- strchr/strrchr——查找字符的首次/最后一次出现
- strspn/strcspn——返回由另一个字符串中（具有/不具有）的字符分割的最大起始段长
- strpbrk——查找一个字符串中的任意一个字符在另一个字符串中的首个位置
- strstr——查找子串字符的首次出现
- strtok——查找字节字符串中的下一个记号

函数指针

在汇编层面，函数也有其入口地址，既然有地址，就有指针。函数名本身就是地址。于是就引出了指向函数入口的指针——函数指针。

函数区分于变量有其特殊性，最主要的区别就是函数有参数。定义一个函数指针变量的格式如下：

```
return_type (* pointer_name)(param_type_list)
```

当一个函数的返回值类型、参数类型列表匹配时，就可以赋值给函数指针变量。

```
1 int max(int a, int b){return (a > b) ? a : b;}
2 int (* pFunc)(int, int) = max;
3 pFunc(3, 2);  // -> 3
```

void指针

void 指针是一种特殊的指针类型，它并不是指向void 类型的变量，而是说不区分它指向的变量的类型。它的存在就如同汇编语言一样，可以直接操作地址，它的加一操作就是真的在地址的值上加一。它主要是用于在一些函数中（如 qsort 的 cmp）泛泛地表示指针，用户可以根据自己的需要转成自己想要的指针类型。

函数指针

既然函数指针变量是一种变量，就可以作为参数传给其他函数。例如著名的 qsort 函数：

```
1 void qsort( void *ptr, size_t count, size_t
    size, int (*comp)(const void *, const void
    *) );
```

功能：对 ptr 所指向的数组以升序排序。数组包含 count 个长度为 size 字节的元素。用 comp 所指向的函数比较对象。

函数指针

参数：

- ptr——指向待排序的数组的指针
- count——数组的元素数目
- size——数组每个元素的字节大小
- comp——比较函数。若首个参数小于（先于）第二个，则返回负整数值，若首个参数大于（后于）第二个，则返回正整数值，若两参数等价，则返回零。比较函数的签名应等价于如下形式：

```
1 int cmp(const void *a, const void *b);
```

该函数必须不修改传递给它的对象，而且在调用比较相同对象时必须返回一致的结果，无关乎它们在数组中的位置。

数组与函数——数组/指针作为函数参数

其实数组作为函数参数时，本质也是传递的数组的首地址。由于数组的首地址可以看作是一个编译器指定的指针常量，因此，数组（尤其是一维数组）作为数组参数时，几乎可以完全用指针代替[3]。

```
1 void f(int a[], int n){} // int * a, 等价
2 int main(void)
3 {
4     int a[10] = {0};
5     f(a, 10);
6     return 0;
7 }
```

[3]注意：这里用词是"几乎"，严格意义上，两者类型不同。（CodeRunner 演示）

数组与函数——数组/指针作为函数参数

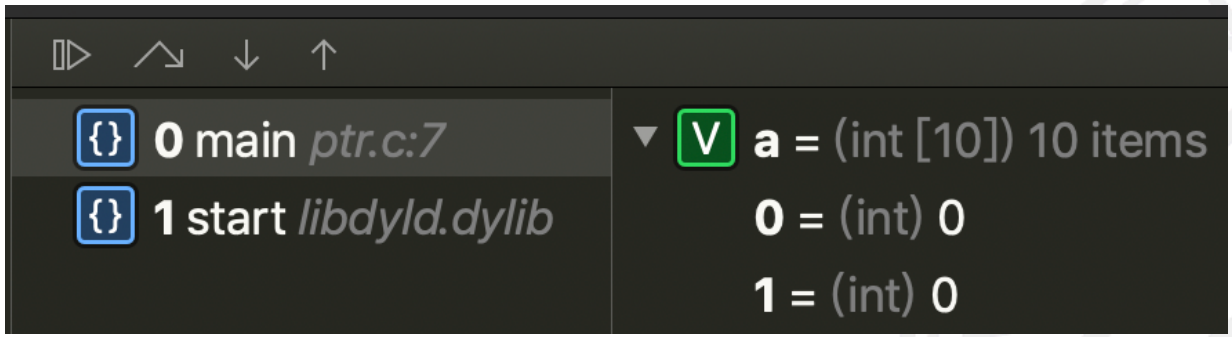


Figure: 在 main 内断点

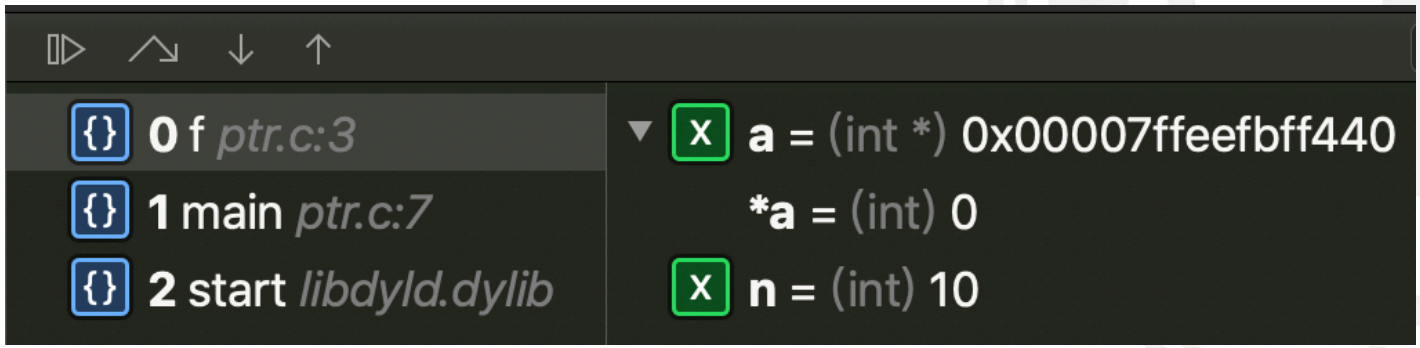


Figure: 在函数 f 内断点

多维数组与指针

之前提到，指针 +1 操作并不是直接将其地址加 1，而是按照指针的类型，增加 `sizeof(typeof(*p))` 个内存单元。

多维数组可以类似理解。如下图，a 是一个二维数组的首地址，类型是 `int **`[4]，它指向数组 a[0][]，"跨度"是一行，因此 a + 1 就会指向下一行 a[1][]。而 a[0] 虽然也是指针，但它"降了一级"，类型是 `int *`，"跨度"是一列，它指向的是行内的首地址，即它指向的是 a[0][0] 存放的位置。

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]

[4]更严格地讲，它的类型应该是 `int (*)[5]`。

多维数组与指针——指向数组的指针

在多维数组的情景中，经常需要表达一次处理一行的要求。比如有一堆字符串，每个字符串是数组，它们构成了二维数组，这时候就要有指向数组的指针。（可以把指向数组的指针理解成一个框。）

指向数组的指针定义方式是 `type (* ptrName)[len]`，此时加一操作就不再是移动 `sizeof(type)` 了。

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]

指针作为返回值、指针数组、指针的指针

指针也可以作为函数的返回值。但是要注意，返回局部变量的地址是无效的，因为函数执行后局部变量将会被回收，将成为“野指针”。

指针数组就是一个数组里面的元素都是指针，如二维数组可以这么理解。还要注意与指向数组的指针区分。前者是多个指针构成数组，后者是一个指针，指向一个数组。

一个指针变量在内存中也有其地址，于是也可以有指针指向它，因此就有了指向指针的指针。

枚举类型

当一个变量的值出自于同一个域中时（比如星期几、性别等），可以将其定义为枚举。有些类似于批量宏定义。

```
1 enum Gender {MALE, FEMALE};
```

编译器会自动地给各枚举分配一个值（通常是 0、1、.....），之后可以直接使用标签，如：

```
1 Gender a = MALE;
```

结构体类型

生活中常见的类型（如学生），都会有许多属性（学号、姓名、成绩.....），当需要把它们定义成一种类型时，就需要结构体了。声明结构体类型时要用 `struct` 关键字，然后把各个变量的类型和名称列出。定义一个结构体变量有多种方式：

1. 先声明结构体类型，再定义

```
1 struct structType varName;
```

2. 声明结构体类型的同时定义

```
1 struct structName {...} varName;
```

结构体变量是一个具有各属性的整体，可以用 . 引用结构体变量的属性，用 -> 引用结构体变量指针指向的变量的属性。结构体变量可以构成数组。

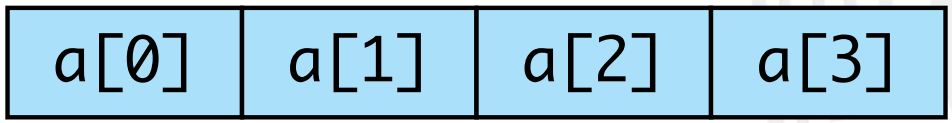
typedef关键字

当需要给某种类型起一个别名（通常是给长类型名一个短类型名，方便记忆）时，可以用 typedef 关键字。

使用方法是typename 原有类型名 新类型名;。以大家现在的了解程度，只有结构体类型这么做有意义。

一点数据结构——顺序表

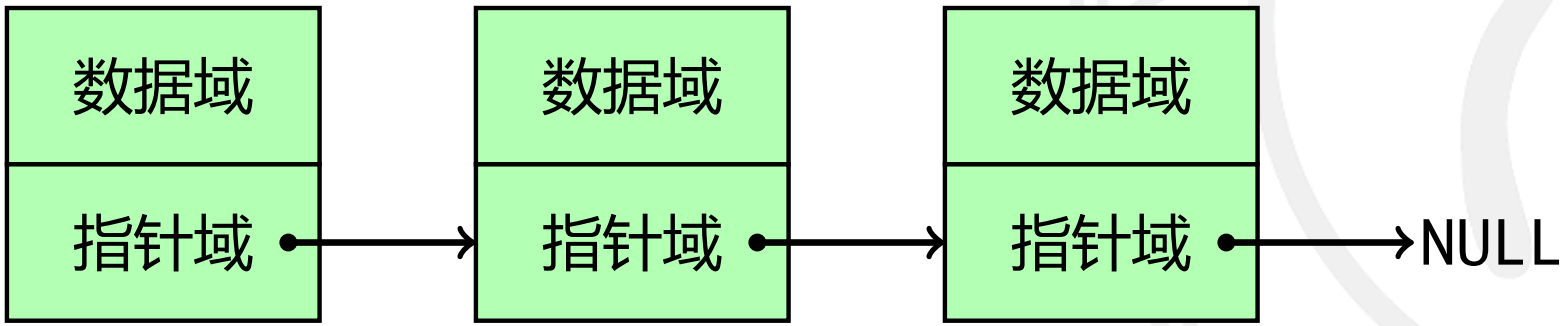
顾名思义，就是按顺序组织起来的一组相同类型的数据，支持增（$O(n)$）删（$O(n)$）改（$O(1)$）查（$O(n)$）操作，可以用数组实现。



不是很重要，略。

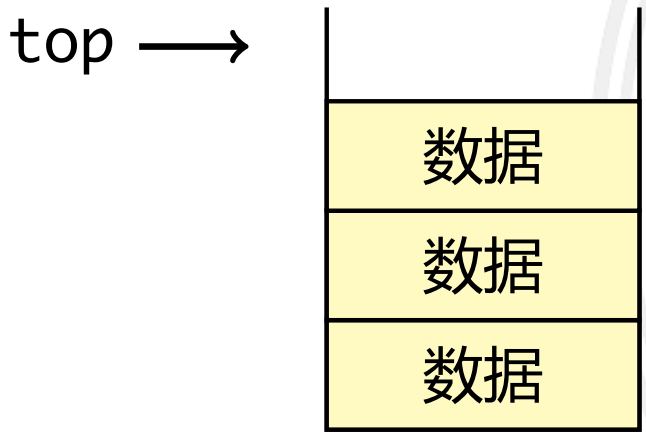
一点数据结构——链表

由一个个不连续存放的节点通过指针串接而成。每个节点有数据域和指针域，前者存放数据，后者指向下一个节点。支持增（$O(1)$）删（$O(1)$）改（$O(n)$）查（$O(n)$）操作。



一点数据结构——栈

通俗的讲，像一个水桶，就是一摞同型数据叠在一起，具有后进先出的特性。支持入栈 push（$O(1)$），出栈 pop（$O(1)$）和查看栈顶元素 top（$O(1)$）等操作，可以用数组实现。



例题（括号匹配）

现有一组小括号串，请你检查它们中的括号是否正确匹配。

一点数据结构——栈

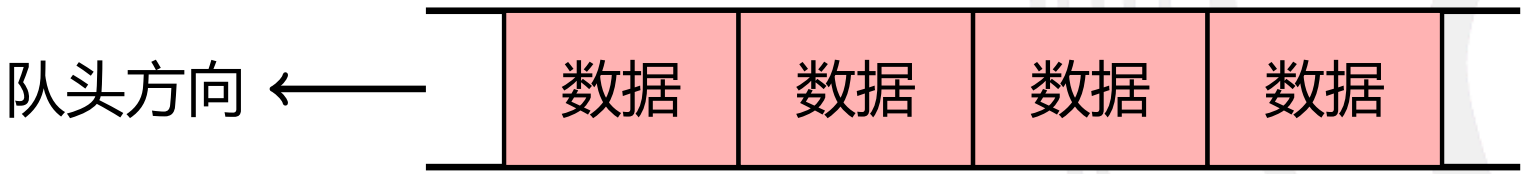
```
1  typedef int DataType;
2
3  typedef struct _Stack
4  {
5      DataType data[N];
6      unsigned long long top;
7  }Stack;
8
9  void push(Stack * s, DataType d)
10 {
11     assert((s -> top) < N);
12     (s -> data)[top] = d;
13     (s -> top)++;
14 }
```

一点数据结构——栈

```
15 DataType top(Stack * s)
16 {
17     assert((s -> top) > 0);
18     return (s -> data)[(s -> top) - 1];
19 }
20
21 void pop(Stack * s, DataType d)
22 {
23     assert((s -> top) > 0);
24     (s -> top)--;
25 }
```

一点数据结构——队列

顾名思义，像一根管子，就是“排队”，具有先进先出的特点。支持入队 enqueue（$O(1)$），出队 dequeue（$O(1)$）和查看队首元素 front（$O(1)$）等操作，可以用数组实现。



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。