C程序设计语言

第5章 指针与数组

孙志岗
sun@hit.edu.cn
http://sunner.cn

内存（Random Access Memory）
地址（Address）

- 计算机内的存储部件，活动中的所有指令和数据都保存在内存内
- 速度快，但是掉电即失
- 可以随机访问
  - 只要指名要访问的内存单元的*地址*，就可以立即访问到该单元
  - *地址*是一个无符号整数（通常用16进制数），其字长与主机相同
  - 内存中的每个字节都有唯一的一个地址

指针的故事

- “该程序执行了非法操作，即将关闭”
  - 这种错误几乎全是由指针和数组导致的
- 黑客攻击服务器利用的bug绝大部分都是指针和数组造成的
- 有些非计算机专业的人，尽量避免使用指针

指针的故事

- 铁杆C/C++程序员最挚爱的武器：指针
- 指针造就了C/C++的高效和强大
- 很多不可能的任务由指针完成
  - `main(){char*a="main(){char*a=%c%s%c;printf(a,34,a,34);}";printf(a,34,a,34);}`

关于指针的原则

- 学习原则
  - 一定要学会
  - 其实通常的应用很简单
    - 就是一个变量
  - 复杂的应用也不建议使用
- 使用原则
  - 永远要清楚每个指针指向了哪里
  - 永远要清楚指针指向的位置是什么

数组（Array）

- 若干类型相同的相关数据凑到一起，就是数组
- 定义
  - 类型 数组名[整型常数1][整型常数2] ……[整型常数n];
  - `int a[6][4];`
- 使用
  - a[0][0]、a[1][2]、a[5][3]
  - 每个元素都是一个普通变量
  - 下标可以是任意整型表达式
- 数组的各个元素在内存中分布在一起，分布规律是……array.c
- 思考一下一维和三维数组怎么分布呢？

从类型的角度理解数组

- `int a[10];`
  - 定义了一个有10个int类型元素的数组
  - a的类型可以看作int[10]（只是看作，语法并不允许这么定义：`int[10] a`）
- `int a[20][10];`
  - 定义了一个有20个int[10]类型元素数组
  - a[0]、a[1]……a[9]的类型是int[10]，所以a[0][0]、a[0][1]……a[19][9]的类型是int
- `int a[30][20][10];`
  - 这个呢？
- 这种特性决定了数组元素在内存的分布规律，也解释了数组的很多语法现象

数组初始化

- 数组定义后的初值仍然是随机数，一般需要我们来初始化
- `int a[5] = { 12, 34, 56 ,78 ,9 };`
- `int a[5] = { 0 };`
- `int a[] = { 11, 22, 33, 44, 55 };`
- 数组大小最好用宏来定义，以适应未来可能的变化

```
#define SIZE 10
int a[SIZE];
```

数组的使用

- 数组的下标都是从0开始
- 对数组每个元素的使用与普通变量无异
- 可以用任意表达式作为下标，动态决定访问哪个元素
  - `for (i=0; i<SIZE; i++)`
    `a[i] = 2 * i;`
- 下标越界是大忌！
  - 使用大于最大下标的下标，将访问数组以外的空间。那里的数据不是我们所想定的情况，可能带来严重后果
  - 有时，故意越界访问数组会起到特别效果，但一定要对自己在做什么了如指掌
- `sizeof`可以用来获得数组所占字节数
  - `sizeof(a)`
  - `sizeof(a[0])`

数组的用处与特点

- 保存大量同类型的相关数据
- 快速地随机访问
- 一旦定义，不能再改变大小
  - 在编译阶段就确定了数组的大小
- 数组名几乎就是一个指针

指针（Pointer）

- `int *p;`
  - 定义了一个指针变量p，简称指针p
    - p是变量，`int*`是类型
    - 变量都占用内存空间，p的大小是`sizeof(int*)`
  - p用来保存地址。此时这个地址是哪呢（p指向哪呢）？
  - `int i;`
    `p = &i;`
  - *p就像普通的变量一样使用，其值是p指向的内存的内容，类型是int（在上例和i等价）
  - p可以动态（任意）地指向不同内存，从而使*p代表不同的变量
    - `p = 0;`
    - `p = &a[0];`

指针

- 指针也是数据类型。指向不同数据类型的指针，分别为不同的数据类型
  - `int*`、`float*`、`char*`、`int**`、`int***`……
- 指针指向非其定义时声明的数据类型，将引起warning
- `void*`类型的指针可以指向任意类型的变量
- 指针在初始化时一般`int *p=NULL;`
  - `NULL`表示空指针，即无效指针
  - 但它只是逻辑上无效，并不是真正地无效
- 如果指针指向一个非你控制的内存空间，并对该空间进行访问，将可能造成危险

&与*运算符

- &运算的结果指向该变量的指针
  - `int i, *p;`
    `p = &i;`
  - `int *p, a[10];`
    `p = a;`
  - `int *p, a[10];`
    `p = &a[0];`
  - `int *p, a[10];`
    `p = &a[5];`
- *和指针的组合是一个变量，该变量的地址和类型分别是指针指向的地址和指针*定义的*指向的类型
  - `int i, *p;`
    `p = &i;`
    `*p = 0;`
  - `int *p, a[10];`
    `p = a;`
    `*p = 0;`
  - `int *p, a[10];`
    `p = &a[0];`
    `*p = 0;`
  - `int *p, a[10];`
    `p = &a[5];`
    `*p = 0;`

const和指针

- `const int * p;`
  - p是指向const int类型数据的指针
  - `const int * p;`
    `const int i;`
    `p = &i;`
- `int * const p;`
  - p是指向int类型的一个*常指针*
  - `int i;`
    `int * const p = &i;`

指针与数组

- 数组名可以看作一个指针
  - 只是不能修改这个指针的指向
    - 常指针
  - `int a[10];`
    - a的类型是`int[10]`
    - a的类型也是`int*`
- 指针可当作数组名使用，反之亦然
  - `int *p, a[10];`
    `p = a;`
    `p[1] = 0;`
    `*a = 0;`

指针运算

- `int* p=NULL;`
  `p++;`
  `/* p的值会是多少？ */`
- 指针的加减运算是以其指向的类型的字长为单位的
- `int *p, a[10];`
  `p = a;`
  - `*(p+3)` 等价于 `a[3]`
  - `p++;`
    `*p` 等价于 `a[1]`

指针运算

- `int *p, *q, a[10];`
  `p = a;`
  `q = &a[2];`
  - `q - p == ?`
  - `q = p + 3;`
- 运算法则
  - 只能进行加减和关系运算
  - 只能*同类型*指针之间或指针与整数之间运算

“类型”本不存在

- 存储器在保存数据时并不关心数据的类型
  - 完全以二进制方式工作
- 我们向计算机发出的指令说明了某块内存里数据的类型
  - 一块内存内保存着$(61\ 62\ 63\ 64)_{16}$
  - 以`char`类型看待每个字节：
    - `"abcd"`
  - 以`float`类型看待每个字节：
    - `1677799940808210400000.000000`
  - 以`int`类型看待每个字节：
    - `1684234849`

依天屠龙，强强联手

```
int main(void)
{
    int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
    int i;
    unsigned char *p;

    p = (unsigned char*)a;  /* 类型强转了 */

    for (i=0; i<sizeof(a); i++)
    {
        PrintHexChar(p[i]); /* 把指针当数组用 */
        putchar(' ');
    }
}
```

- 指针强转后，可以把一块内存当作另一种类型来处理
- 强强联手，我们可以随意控制任意内存

god.c

指针与函数

- 指针既然是数据类型，自然可以做函数的参数和返回值的类型
- 指针做参数的经典例子：

```
main()
{
    int x, y;
    swap(x, y);
}
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Not Work

指针做参数

```
main()
{
    int x, y;
    swap(&x, &y);
}
void swap(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

这里的函数调用过程还是“实参”的内容复制到“形参”，千万不要理解成什么“*传引用调用*”

指针做返回值

```
int* foo(void)
{
   int a = 100;
   return &a;
}
```

这是极端错误的！！！

- 永远不要把局部变量的地址作为返回值返回

指针做返回值

```
printf("%s", GetInput());
......
char* GetInput(void)
{
   char str[100];
   scanf("%s", str);
   return str;
}
```

```
char string[30];
printf("%s", GetInput(string));
......
char* GetInput(char* str)
{
   scanf("%s", str);
   return str;
}
```

三个月使用scanf目睹之怪现状

```
int i;
scanf("%d", i);
/* 这样会如何？*/
```

i的值被当作地址。例如，i的值如果是100，那么输入的整数就会从地址100开始写入内存

```
int i;
scanf("%f", &i);
/* 这样又会如何？*/
```

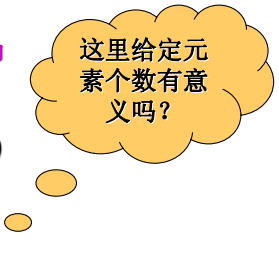
输入被当作float，以float的二进制形式写到i所在的内存空间

```
char c;
scanf("%d", &c);
/* 这样呢？*/
```

输入以int的二进制形式写到c所在的内存空间。c所占内存不足以放下一个int，其后的空间也被覆盖

数组做参数

- 其实就是指针做参数
  - 不是把整个数组的内容复制到函数内
- `int a[10];`
  `ProcessArray(a);`
- `void ProcessArray(int* p)`
  `{`
  `  ......`
  `}`
- `void ProcessArray(int a[])`
  `{`
  `  ......`
  `}`



数组做参数

- 我们应该让函数知道数组的大小
- `int a[10];`
  `ProcessArray(a, 10);`
- `void ProcessArray(int a[], int n)`
  `{`
  `  ......`
  `}`

数组做参数示例

- `main()`
  `{`
  `  int a[10];`
  `  RandFill(a, 10);`
  `}`
- `void RandFill(int a[], int size)`
  `{`
  `  int i;`
  `  for (i=0; i<size; i++)`
  `  a[i] = rand();`
  `}`

二维数组做参数

- `int a[10][20];`
  `ProcessArray(a, 10);`
- `void ProcessArray(int a[][20],`
  `int m)`
  `{`
  `  ......`
  `}`

数组做返回值？

- ???? ProcessArray(int a[], int n)
  {
  ......
  return a;
  }
- 上面语句返回的是什么类型？
  - int* const
- 函数没有返回数组内容的可能，只能返回指针

指针和数组做参数

- 通过参数，把数据传给调用者
- 通过一个参数把大量的数据送到函数内
  - 如果只是向内传送数据，就要把参数定义为const，防止意外修改数据，也让函数的功能更明了
- void PrintArray(const int * p, int n)
  {
  ......
  }
- void PrintArray(const int a[], int n)
  {
  ......
  }

字符串(String)与字符数组、字符指针

- 字符串
  - 一串以'\0'结尾的字符在C语言中被看作字符串
  - 用双引号括起的一串字符是字符串常量，C语言自动为其添加'\0'终结符
    - "Hello world!"
    - 把字符串常数作为表达式直接使用，值是该常数的地址，类型为const char*
  - C语言没有提供字符串类型，完全用字符数组和字符指针来处理
- 字符数组
  - 每个元素都是字符类型的数组
    - char string[100];
- 字符指针
  - 指向字符类型的指针
    - char* p;

字符串“赋值”还是“复制”？

- char* string;
  string="Free your mind";
  - 指针赋值
- char string[]="Free your mind";
  - 内容复制
- char string[20];
  string = "Free your mind";
  - 非法！
- char string[20];
  strcpy(string, "Free your mind");
  - String copy!

字符串处理函数

- 在<string.h>中定义了若干专门的字符串处理函数
- strcpy: string copy
  - char *strcpy(char *dest, const char *src);
- strlen: string length
  - size_t strlen(const char *s);
- strcat: string catenate
  - char *strcat(char *dest, const char *src);
- strcmp: string comparison
  - int strcmp(const char *s1, const char *s2);
- stricmp: string comparison ignoring case
  - int stricmp(const char *s1, const char *s2);

'\0'作为字符串终结符的天生缺陷

- 假若交给这些字符串处理函数的字符串没有'\0'会如何？
- '\0'很关键，如果没有，那么这些处理函数会一直进行处理直到遇到一个'\0'为止。此时可能已经把内存弄得乱七八糟
- ANSI C定义了一些“n族”字符串处理函数，包括strncpy、strncat、strncmp、strnicmp等，通过增加一个参数来限制处理的最大长度
- 确认：要写入字符串的地方存储空间是否足够放下所有字符和'\0'

命令行参数

- GUI界面之前，计算机的操作界面都是字符式的命令行界面（DOS、UNIX、Linux）
- 通过设置命令行参数，用户可以决定程序干什么、怎么干
- int main(int argc, char* argv[])
  - 当你把main函数写成这样时
  - argc的值为参数的数目+1
  - argv[0]为指向命令名的字符指针
  - argv[X](x>1)为指向每个参数的字符指针

命令行参数

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int i;

   printf("The command line has %d
arguments:\n", argc - 1);

   for (i = 1; i < argc; i++)
      printf("%d: %s\n", i, argv[i]);

   return 0;
}
```

arg.c

指针、数组以及其它的类型组合

- 基本数据类型和修饰符
  - int、char、float、double
  - long、short、const、signed、unsigned
- 指针是一种数据类型
  - 是从其它类型派生的类型
    - XX类型的指针
- 数组也是一种数据类型
  - 是从其它类型派生的类型
    - 元素是XX类型
- 任何类型都可以做指针或者数组的基础类型
  - 它们自己也可以做彼此或自己的基础类型

用typedef简化类型的形式

- 为一个类型起个名字
- ```
  typedef unsigned char * string;
  string str="Free you mind\n";
  printf(str);
  ```
- ```
  typedef int array[10][20];
  void func(array a);
  array a;
  a[0][0]=1;
  func(a);
  ```
- string和array用起来更直观一些

动态分配内存

- 在<stdlib.h>中定义了下面的函数
- void* malloc(size_t size);
  - size_t是ANSI C定义的数据类型，一般就是unsigned int，但使用时要用size_t
  - 向系统申请大小为size的内存块，把指向首地址的指针返回。如果申请不成功，返回NULL（一定要检查返回值）
- void free(void* block);
  - 释放由malloc()申请的内存块。block是指向此块首地址的指针（malloc()的返回值）
- malloc申请的内存
  - 在“堆（heap）”中分配，内容随机
  - 被free之前，永久有效
  - 在被free之后，该块内存不再属于你

动态分配内存

- malloc()申请的内存不再使用时就要及时free()，否则将产生内存泄露(Memory Leak)
  - “内存泄露”一词类似“原料泄露”。泄露出去的原料不能被利用，导致生产过程中原料不足
  - malloc()时，系统找到一块未占用的内存，将其标记为已占用，然后把地址返回，并标记此程序占用此块内存，其它程序不能再用它
  - free()时，系统标记此块内存为未占用，可以被重新分配
  - 如果申请来的内存不用，别的程序也不能用，就好像这块内存泄露出去一样，造成浪费
  - 但不要频繁申请/释放，消耗时间，造成内存碎片

防止内存泄露之道

- 在需要的时候才malloc，并尽量减少malloc的次数
  - 能用自动变量解决的问题，就不要用malloc来解决
  - malloc一般在大块内存分配和动态内存分配时使用
  - malloc本身的执行效率就不高，所以过多的malloc会使程序性能下降
- 重复使用malloc申请到的内存
- 尽量让malloc和与之配套的free在一个函数或模块内
  - 尽量把malloc集中在函数的入口处，free集中在函数的出口处
- 以上做法只能尽量降低产生泄露的概率。完全杜绝内存泄露，关键要靠程序员的细心与责任感

指向函数的指针

- `int (*pfunc) (int, int)`
  - 定义了一个函数指针pfunc，它指向的函数必须是：返回值为int类型，有两个int类型的参数
-
```
int f1(int a, int b);
int f2(int x, int y);
```
  -
```
pfunc = f1;
(*pfunc)(1, 2);
```
  -
```
pfunc = f2;
(*pfunc)(1, 2);
```
- 函数指针也是一种类型
  -
```
typedef void (*pFoo) (int*, int*);
pFoo p;
```


qsort

-
```
void qsort(void *base,
           size_t nelem,
           size_t width,
           int(*fcmp)(const void *,
                      const void *)
           );
```
  - 定义在<stdlib.h>中
  - 用“快速排序”算法对base所指向的有nelem个元素的数组进行排序。每个数组元素占width个字节
  - fcmp是指向用户自己定义的函数的指针
    - 该函数通过两个参数得到两个值，如果第一个值大于第二个，返回大于0的数；如果两个值相等，返回0；否则，返回小于0的数
  - qsort会调用fcmp进行元素的比较，并最终排序
- 这种方法在程序设计学中被称为“函数回调（Callback）”。被“回调”的函数称为“回调函数”

qsort.c

使用指针与数组的万灵丹

- 指针变量是变量
  - 特别1：可以对此变量进行*操作
  - 特别2：此变量的加减法有些特殊
- 数组成员是变量
- 指针是一种数据类型
- 数组是常量指针
- 指针也是数组名
- 对它们，只要按照变量和类型的一般原则使用就可以，没有多少特殊化的地方

避免出错

- 数组
  - 永远清楚每个数组有多大
  - 永远让下标不会越界
- 字符数组
  - 永远留意'\0'
- 指针
  - 永远要清楚每个指针指向了哪里
  - 永远要清楚指针指向的位置是什么
- 总纲
  - 永远清楚你正在操作哪块内存
  - 永远清楚这种操作是否合理、合法