Table of Contents

《深入理解C指针》笔记

指针可以说是C语言的皇冠一样，让C语言大放光彩的也是它，让C语言充满问题的也是它。不知道多少人在学习C语言时，被这座大山挡住了。

本人一直以来对指针都处于半懂的状态，架不住C指针的内容太过复杂。后来因为项目的原因，决心一定要抽个时间好好把它弄清楚。于是就选择了这本《深入理解C指针》，好好的研读了一番，并在此做了详细的笔记。

由于是本人的个人笔记，其主要是为了本人理解指针概念所写的，所以有些内容在外人看来不能理解。如果有什么疑问和错误，欢迎大家一起交流。

本人还将书中的代码整理了出来，放到了Github上，有兴趣的读者可以去看看。

第一章 认识指针

C程序的内存使用形式

内存方式	作用域	生命周期
全局内存	整个文件	应用程序生命周期
静态内存	声明函数内部	应用程序生命周期
自动内存	声明函数内部	函数执行生命周期
动态内存	由引用该内存的指针决定	内存释放（人工）

> 全局内存：常见的全局变量 静态内存：静态变量，例如`const int var` 自动内存：常见的就是生命在函数体内的变量了，这是最常见的 动态内存：通过指针创建的堆区域

常量与指针

指针类型	指针是否可修改	指针指向数据是否可修改
指向非常量的 指针(int * ptr)	是	是
指向 常量的 指针(const int * ptr)	是	否
指向非常量的常量指针(int * const ptr)	否	是
指向 常量的常量指针(const int * const ptr)	否	否

> 阅读指针声明的时候，采取从右往左读的方式。
>
> `* pci` - pci是个指针
>
> `int * pci` - pci是个指向整数的指针
>
> `const int * pci` - pci是个指向整数常量的指针，（指针可变，指针指向区域的整型数据不可变）

- `int const * p` 与 `const int * p` 是等价的。可以按上面那种方式来读取。`const * p` 是指向常量的指针, `int const * p` 是指向整型常量的指针。
- `int * const p` 是与上面不同的，`const p` 意味着是常量，`* const p` 则是常量指针，`int * const p` 是指向整型的常量指针（指针不可变，指针指向区域的整型数据可变）
- 结合上面两种的形式是`const int * const * p` 就是指针不可变，指针指向区域的数据也不可变

指针多层引用

- 多层指针通常与多维数组，指针数组等有关联，详细的放到第四章 数组与指针中进行详细记录

其他

- 指针的语法和语义规范
- 指针的声明：声明指针时，例如 `int * p` ， `*` 的位置是无所谓的，看个人习惯。
- `&` 用来取值， `%p` 来显示地址。
- 各平台显示地址方式不一定会一致。通过将要显示的指针转换为void指针来兼容各个平台，例如 `print("p%",(void *) p)`
- 通过程序显示的地址均为在操作系统虚拟内存映射下的地址，并非真正的物理地址
- 在取地址和解地址的方面， `*` 和 `&` 是互反的操作
- 在使用指针时， `NULL` 和 `0` 都可以，但是 `NULL` 更好些，可以告诉开发者在使用指针。
- void指针是通用指针。void指针可以转换为除函数指针外的任何指针。
- `intprt_t` 和 `uintptr_t` 是C99标准为了适应64位环境所新增的。暂时可以不用了解
- 指针相加的实质是按声明类型字节大小进行步增
- 指针相减的实质是两个指针地址相减后，按声明类型字节大小整除后得到的单位值，并通过符号老判断地址前后顺序

第二章 C语言动态内存管理

动态内存的分配

- malloc与free一起必须成对出现
- 避免内存泄漏
  - 丢失了内存的地址
  - 隐式泄漏：没有对使用完的空间进行释放。例如定义了动态成员变量的结构体，只释放了结构体指针，而没有释放成员指针

动态内存释放函数

函数名	作用
`malloc`	分配内存空间，使用最多，函数返回内存区域第一个字节的地址
`realloc`	重新分配内存空间，用于扩展和缩小空间
`calloc`	分配空间并清空，等同于 `malloc` + `memset`
`memset`	清空内存区域
`free`	释放内存区域

- 使用 `malloc` 可能会返回空指针，所以可以加一步判断

```
int *pi = (int* ) malloc (sizeof(int));
if(pi != NULL) {
// Pointer should be good
} else {
// Bad pointer
}
```

- `realloc` 函数的特殊性可以让其有多种功能

第一个参数	第二个参数	行为
空	无	同malloc
非空	0	原内存块被释放
非空	比原内存块小	利用当前位置分配更小的块
非空	比元内存块大	在当前位置或其他位置分配更大的块

迷途指针

如果原内存被释放，但是原指针确仍然指向原来区域，那么该指针则被称为迷途指针。迷途指针最好的解决办法就是将释放后的指针置为`NULL`

动态分配内存的监控

正由于动态分配的内存存在大量的陷阱，因此产生了很多工具来监视和检测迷途指针和内存泄漏等。以下是书中介绍的工具和技术，具体可以查看相应网站

- Valgrind
- Exceptions in C
- Resource Acquisition Is Initialization
- Dmalloc
- Finding Memory Leaks Using the CRT Library

第三章 函数与指针

程序中的栈和堆

- 程序帧是支持函数执行的内存区域，与堆共享内存区域
- 栈在程序帧下部，由下往上增长；堆在上部，由上往下增长
- 栈存放函数参数和局部变量，堆管理动态内存
- 栈帧保存以下元素
  - 返回地址
  - 局部数据存储
  - 参数存储
  - 栈指针和基指针

函数参数指针传递

- 函数参数传递时，实参传递给形参的是值传递，这会复制实参，不够高效，尤其是传递结构体时，我们不希望再一次复制完全的结构体。而使用指针传递，不但可以改变实参，还能更高效的传递参数。如传递结构体时，传递结构体指针即可，无需其他消耗。
- 如果传递时，使用指向常量的指针，则可以更高效的获取实参内容，而且不会改变实参内容。

传递指针的指针

对于以下代码

```
void allocateArray(int *arr, int size, int value)
{
    arr=(int *)malloc(size * sizeof(int));
    for(i=0;i<size;++i)
        *(arr+i)=value;
}
int *vector = NULL;
allocateArray(&vector,10,1);
```

会产生问题

1. 首先函数内部虽然将地址传给了 `arr` ，但是紧接着 `arr` 的地址被重新通过 `malloc` 赋值，所以并没有使用 `vector` 的内容
2. 函数释放后，分配的空间地址丢失，导致内存泄漏

为了解决以上的问题，我们使用指向指针的指针。

想改变的实参类型	形参形参的形式
int	int *
int *	int **

> 1. 当函数改变的实参的值时，而不是形参创建的副本，那么函数传递就要传递实参的地址
> 2. 当函数要改变的是实参本身就是指针，那么函数就要传递指针的地址，也就是函数的形参要定义成指向指针的指针。

所以修改代码

```
void allocateArray(int **arr, int size, int value)
{
    *arr=(int *)malloc(size * sizeof(int));
    for(i=0;i<size;++i)
        *(*arr+i)=value;
}
int *vector = NULL;
allocateArray(&vector,10,1);
```

这样调用，函数内部分配的内存地址会直接传递给实参 `vector` ，不会导致地址的丢失等问题

函数返回指针

- 声明 `int * fcuntion()` 即意味着此函数返回的是一个整型指针。 `注意与int (*fcuntion)()的区别`

常用的函数返回指针技术

常用的返回指针技术有两种：

- 函数内部使用 `malloc` 分配内存空间。调用者负责释放内存

```
int * allocateArray(int size, int value)
{
    int *arr = malloc(size * sizeof(int));
    for(int i=0;i<size;++i)
        *(arr+i)=value;
    return arr;
}
```

然后使用以下方式调用

```
int *vector = allocateArray(10,1);
for(i=0;i<10;++i)
    printf("%p",*(vector+i));
```

> 最后要注意一定要释放内存，因为函数内部只负责了分配内存，要由调用者负责释放

```
free(vector);
```

- 函数内部只负责修改传递过来的指针并修改它，内存的分配和释放都有调用者负责。对于传递过来的指针，优先判断是否为空是个好习惯。如下例：

```
int * allocateArray(int *arr, int size, int value)
{   if(arr != NULL)
    {
        for(int i=0;i<size;++i)
            *(arr+i)=value;
    }
    return arr;
}
```

然后使用以下方式调用

```
int* vector=(int *)malloc(size * sizeof(int));
allocateArray(vector,10,1);
```

> 书中说此方法不推荐，目前不知道为什么
>
> 以上两种方式，函数内部使用的内存都是处于堆区域，所以即使函数结束，弹出栈，堆区域的内容仍然存在，只要保留地址即可。但是函数返回指针不要返回函数内部定义的变量，这样函数结束弹出栈帧以后，变量立即消失，这样就出现问题

函数返回指针注意的问题

函数返回指针要注意四个事项：

1. 返回未初始化的指针
2. 返回指向无效地址的指针
3. 返回局部变量指针
4. 返回指针但是外部调用没有释放

在处理函数返回指针时，一定要根据内存方式注意以上四项。

完善的free函数

内置 `free()` 函数存在以下问题：

1. 不会检查传入指针是否为空
2. 释放后不会将指针置为NULL 这时候可以创建自己的free函数

```
void safeFree(void **p){
    if(p != NULL && *P != NULL){
        free(*p);
        *p =NULL;
    }
}
```

其中 `void指针` 意味着可以传入任何指针类型，定义 `**p` 是因为释放的本身是个指针，需要使用传递指针的指针来真正操作指针。

更快的方式可以定义一个宏 `#define safeFree(p) safeFree(void ** (&p))` 配合 `safeFree` 函数使用。

函数指针

- 函数指针定义的方式就是 `int (*function)()` ，这时候 `function` 就作为一个指针指向了函数的地址。
- 函数指针对性能是有一定影响的，使用它处理器就无法配合流水分支预测
- 正常的函数声明中 `int func()` ，函数名 `func` 没有明确定义是指针，还是其他类型。但是对打印函数名 `func` 的地址，或者函数名取值后 `&func` 的地址都是同一个值。所以我们只要知道正常的函数声明中，函数名可以等效为该函数的地址。
- 对于函数指针，建议使用 `fptr` 作为前缀
- 使用了函数指针，使用该指针调用函数，程序将不检查参数传递的是否正确
- 为函数指针声明一个类型定义会比较方便 `typedef int (*fptrfunc)(int,int)`
- 函数指针可以实现在函数调用中动态的调用其他函数，实现了函数作为参数的传递，这样可以使用C语言进行函数式编程。个人感觉函数指针是实现C++的某种基础。例如以下代码

```
typedef int (* fptrOperation)(int ,int);
int sum(int a, int b){ return a+b}
int sub(int a, int b){ return a-b}
int computer(fptrOperation operaton, int num1, int num 2) { return operaton(num1, num2
) }

printf("%d\n",computer(sum,1,2));
printf("%d\n",computer(sub,5,3));
```

- 返回函数指针即用函数指针去声明一个函数，例 `fptrOperation switchcode(char opecode)`
- 当用函数指针去声明一个数组时，那么数组中的每个元素，都代表函数指针指向的一个

函数操作，大大增加了函数调用的丰富性。例如

```
fptrOperation operaton[10]={NULL};
```

第四章 数组与指针

数组名的含义

一维数组

对于

```
int vector[]={1,2,3,4,5}
```

数组名 `vector` 是数组首元素的地址，但是其代表的是整个数组，也就是说 `sizeof(vector)` 是 `5*sizeof(int)` 。

> 首元素有很多种形式，可能是整型或字符型等基本类型，也可能是数组，也即多维数组

二维数组

对于二维数组：

```
int matrix[2][5]={
        {1,2,3,4,5},
        {6,7,8,9,10}
    }
```

`matrix` 代表首元素 `matrix[0]` 的地址，而首元素 `matrix[0]` 是个数组，意味着 `matrix[0]` 代表内层数组首元素 `matrix[0][0]` 的地址，所以则有: `matrix = &matrix[0]` , `matrix[0] = &matrix[0][0]` 进行解引则有: `*matrix = matrix[0] = &matrix[0][0]` `**matrix = *matrix[0] = matrix[0][0]`

- `matrix` 的本质是一个5个元素数组的数组指针，但其代表的是整个二维数组，也就是其大小为 `10*sizeof(int)`
- `matrix[0]` 的本质是一个5个元素数组**char**指针，但其代表的是内层一维数组，也就是起大小为 `5*sizeof(int)`

> 这里看到`matrix ，刚开始本人也以为二维指针与二维数组是等价的，后来才发现这个想法是错误的。这里的 **matrix**`是解引操作，跟二维指针没有任何关系。

多维数组与多维指针

> - 二维指针对应的是指针数组，二维数组对应的是数组指针
> - `char *pointerArray[]` 定义的是 `指针数组`
> - `char (*arrayPointer)[]` 定义的是 `数组指针`
> - 下面两个知识点针对的是静态分配情况。动态分配时，一维指针可以对应一维数组，而二维指针可以对应二维数组。

二维指针与指针数组

对于数组

```
char *titlesPointerArray[]={
    "A Tale of Two Cities",
    "Wuthering Heights",
    "Don Quixote",
    "Odyssey",
    "Moby-Dick",
    "Hamlet",
    "Gulliver's Travels"
}
```

这里 `titlesPointerArray` 是一个指针数组， `titlesPointerArray` 本身是一个指向数组首元素 `titlesPointerArray[0]` 的指针，即 `titlesPointerArray = &titlesPointerArray[0]` 。

而 `titlesPointerArray[0]` 又是一个指向 `char` 的指针，所以可以看到 `titlesPointerArray` 是一个指向字符型指针的指针，由此可以看到一个二维指针与指针数组是对应的。

```
char *titlesPointerArray[]
char **titlesPointerArray
```

是等价的。这也是为什么在 `main` 函数中 `char **argv` 和 `char *argv[]` 都是可以的。

> 二维指针对于处理字符串数组很有好处，动态性能好。可以不用去人为干涉元素数和各字符长度。

二维数组与数组指针

对于数组

```
char titlesDimensionArray[][40]={
    "The Art of Computer Programming ",
    "Python Programming Guide",
    "Programming Pearl",
    "Computer Network",
    "Modern Perl"
}
```

这里 `titlesDimensionArray` 是一个二维数组。 `titlesDimensionArray` 本身是一个指向数组首元素 `titlesDimensionArray[0]` 的指针，即 `titlesDimensionArray = &titlesDimensionArray[0]` 。

而与指针数组不同的是， `titlesDimensionArray[0]` 不再是指向char的指针，而是一个指向 `有40个元素数组` 的指针。同样， `titlesDimensionArray[0]` 代表这40个元素数组的首元素地址，即 `titlesDimensionArray[0] = &titlesDimensionArray[0][0]` 。所以可以看到 `titlesDimensionArray` 是一个 `指向数组的指针` 。由此可以看到一个二维数组与数组指针是对应的。

```
char titlesDimensionArray[][]
char (*titlesDimensionArray)[]
```

是等价的。

> 这里终于可以看懂二维数组的实质了。虽然最终引用时都使用解引 `**` 来获取首个元素内容，但这是不同的。还有就是这种形式跟上面相比要指定内层数组的维数。相对来说更浪费空间，动态性能不好。

动态创建数组

鉴于数组与指针相爱相杀，我们还可以使用指针来在堆动态的创建数组。最终使用的时候既可以使用指针，也可以使用数组下标方式。

> 动态创建的数组由于是在堆上，所以切记要在最后释放内存
>
> ### 一维数组
>
> - `int *pv = (int *)malloc(size * sizeof(int))` 即动态创建了一个数组pv[size]
> - 使用时既可以用pv[0]方式获取内容，也可以使用*(pv+0)来获取内容

使用 **`realloc`** 实现变长数组

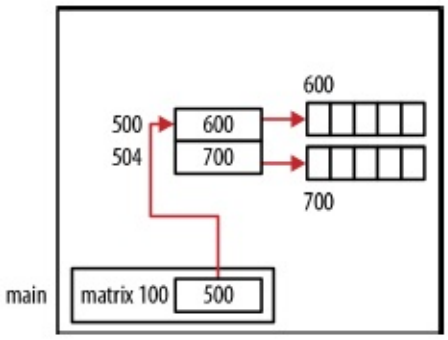
如果不使用C99标准，那么就不能使用变长数组了，我们只能用 `realloc函数` 来实现变长数组。

`realloc函数`的核心功能就是用新区域替换旧的区域，这样如果新区域大于旧区域，便可实现空间的扩大，同时返回区域指针。我们可以根据这个思路来实现变长数组。

二维数据

- 既然是二维，为了确定空间大小，我们使用`rows(外层)`和`columns(内层)`的概念来确定整体分配空间大小。总之总大小应该是`sizeof(rows * columns)`
- 使用二维数组定义时，空间是连续分配的，而使用动态分配时，内存可连续，也可能不连续，需要要人为来控制。
- 连续的空间存取速度快，但是空间利用率不高。而动态不连续的内存，空间利用率相对高些。这就是数据结构中静态链表和动态链表的优缺点的本质原因。
- 进行动态分配时，指针维数与数组维数是对应的。

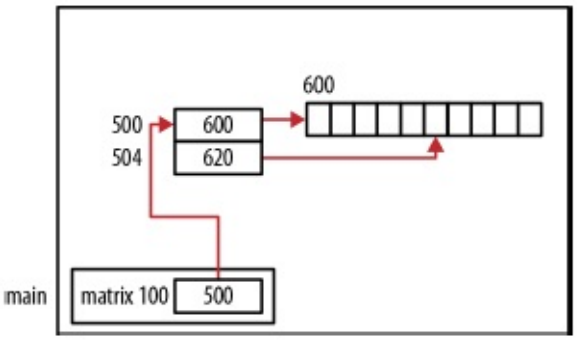
分配可能不连续的内存




```
//定义一个二维指针并分配外层(rows)空间
int **matrix = (int **)malloc(rows * sizeof(int *));
for(i=0; i<rows; ++i)
    *(matrix+i) = (int *)malloc(columns * sizeof(int));
```

两层均使用动态进行空间分配，内存区域可能连续，也可能不连续。

分配可能连续的内存



```
//定义一个二维指针并分配外层(rows)空间
int **matrix = (int **)malloc(rows * sizeof(int *));
//在第二层分配时，先一次性规划处整个空间，然后再按行切分
*matrix = (int *)malloc(rows * columns * sizeof(int));
for(i=0; i<rows; ++i)
    *(matrix+i) = (*matrix)+(columns*i);
```

由于分配内层时使用了整体分配，所以空间是连续的，只需要按行切分即可

函数中数组的传递

一维数组的传递

在函数参数传递一维数组时，形参可以使用两种方式，以整型为例 `void function(int array[], ...)` ，或者 `void function(int *a, ...)` 这两种方式都是可以的，在函数体内部，既可以使用数组形式，也可以使用指针形式。

- 对于实参，必须是 `指向整型的指针` ,例：

  ```
  int vector=[]
  function(vector, ...)
  ```

二维数组的传递

- 对于二维数组的传递，要确定好形参的使用方式，是使用 `数组指针（二维数组）` ，还是 `指针数组（二维指针）`
- 传递二维数组时，一定要对应好形参和实参。
  - 如果形参使用 `数组指针（二维数组）` 。那么实参必须是一个 `指向数组的指针` 。
  - 如果形参使用 `指针数组（二维指针）` ，那么实参必须是一个 `指向指针的指针` 。

其他

- 对多维数组仅可省略最左侧一维的的大小。
- 指针类型与解引操作是不同的
- 二维以上的数组或指针相对难理解，尽量少使用
- 变长数组的支持和利用复合字面量*(composite literal)*实现不规则数组和指针是**C99**新增特性，目前不考虑使用
- 数组和指针还是有区别的。数据名是右值（常量），而使用指针定义的数据，指针是左值（变量）
- 求一维数组的大小可以使用 `index = sizeof(vector)/sizeof(int)`

第五章 字符串与指针

字符串基本

- 字符串有两种定义形式
  - 用字符数组来定义，如 `char string[]="Hello Wordl!"`
  - 用指针指向字符串常量来定义 `char *string="Hello Wordl!"`
- 不管是用上述那种方法，在字符串的最后都有 `\0` 以代表字符串结束。
- 计算字符串长度时是不包括 `\0` 字符的
- 注意 `NUL` 和 `NULL` 是不同的， `NUL` 代表 `\0` ， `NULL` 的定义是 `((void*)0)`
- 不能使用 `sizeof` 对字符串求取长度
- 字符串字面量不是位于堆区域，也不是栈区域。对于字符串字面量来讲，是没有作用域概念的。
- 在 `GCC` 中，字符串字面量可以更改，除非将其限制为字符串常量

字符串的初始化

目前初始化字符串常用的方法有四种

- 第一种，初学者最常用的，使用字符数组初始化字符串。 `char string[]="Hello World!"` 每个数组元素保存的就是字符串的每个字符

> 注意，这里字符数组的长度是13，而字符串长度是12.不要忘了最后的 `\0`

- 第二种，使用空数组配合 `strcpy` 函数将字符串字面量复制到数组

```
#include <string.h>
char string[13];
strcpy(string,"Hello World!");
```

> 注意，这里定义数组时不能定义成 `char string[]` ,这样程序无法为数组分配内存

- 第三种，使用字符指针，在堆中动态分配字符串

```
#include <string.h>
char *header = (char *)malloc(strlen("Hello World!")+1);
// 或者char *header = (char *)malloc(13);
strcpy(string,"Hello World!");
```

这种是在堆中创建了字面量的副本

- 第四种，使用字符指针，直接指向字符串字面量 `char *header="Hello World!"` 这是最简单的方式，推荐
- 第五种，通过标准输入获取字符串

```
char *command;
printf("Enter a Command: ");
scanf("%s",command);
```

字符串的内存分布

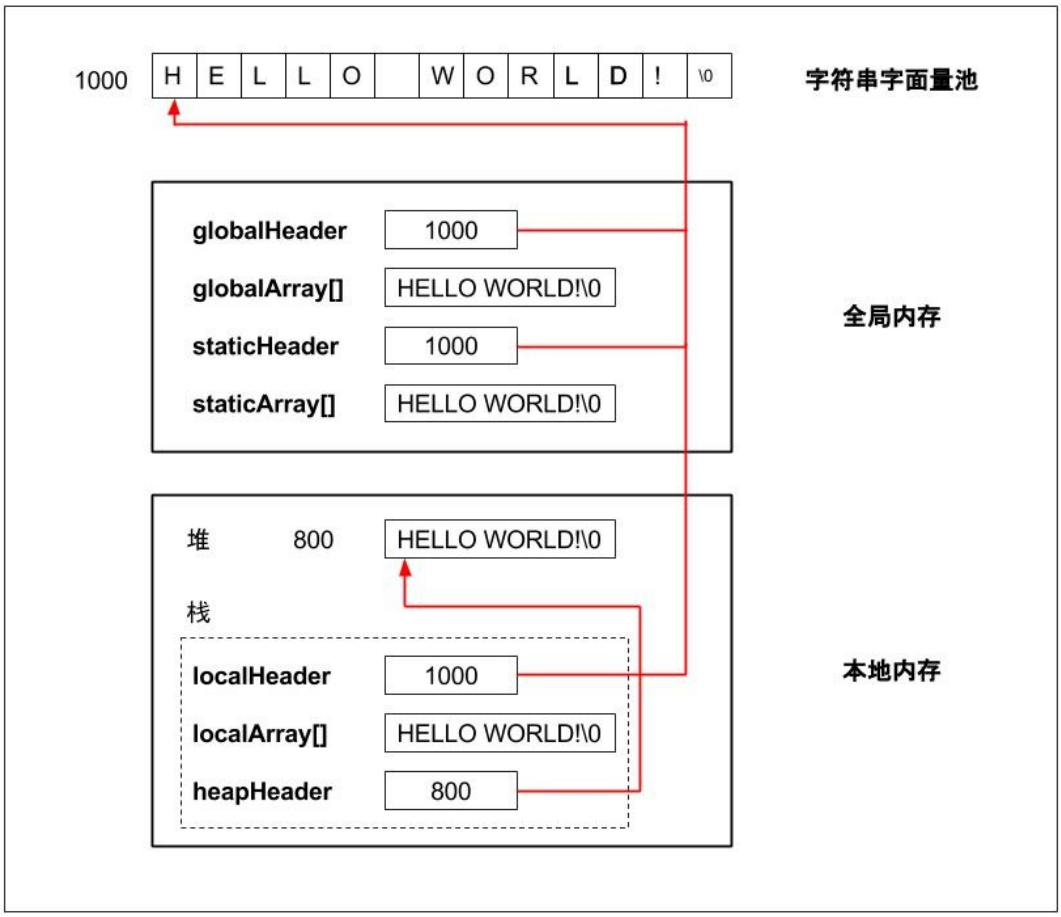
字符串很有意思的地方就是，它可以以很多种内存形式存在。以下的代码和图片能清晰的概括出字符串的内存分布情况

> 为了更好地理解书中的图片，本人自己重新绘制了图片

代码

```
//全局变量，位于全局内存
#include<string.h>
char    *globalHeader = "HELLO WORLD!";
char    globalArray[] = "HELLO WORLD!";
void displayString()
{
    //静态变量，位于全局内存
    static char     *staticHeader = "HELLO WORLD!";
    static char     staticArray[] = "HELLO WORLD!";
    //局部变量，位于自动内存
    char    *localHeader          = "HELLO WORLD!";
    char    localArray[]          = "HELLO WORLD!";
    //动态变量，位于动态内存（堆）
    char *heapHeader              = (char *)malloc(strlen("HELLO WORLD!")+1);
    strcpy(heapHeader,""HELLO WORLD!"")
}
```

说明图



标准字符串操作函数

在C标准库中，定义了对字符串基本操作的函数，而且这些函数经常被使用。这里不再赘述每个函数的意义，函数原型很简单。有什么疑问可以查看标准手册。具体的内容将在代码中体现。

strcmp()

- `memset()` 函数可以用来清空数组，第一个参数是指向数组的指针，第二个是打算使用的值，第三个参数是长度
- 比较字符不用包括 `\0`
- 在代码

```
char cmd[20];
scanf("%s", cmd);
if (!strcmp(cmd, "Quit"))
```

中，不能使用 `if(cmd == "Quit")` ，这是在比较 `cmd` 的地址！

strcpy()

- 拼接函数的原型是 `char *strcat(char *, const char *)` ，此函数返回拼接后字符串的地址。
- 与上面两个不同时的是，我们必须先为拼接后的字符串分配好内存，然后再进行拼接，否则会出现内存越界等错误。

传递字符串作为函数的参数

- 传递字符串时，最好以常量的形式进行传递，以防止对实参的修改
- 传递需要初始化的字符串，即想通过一函数来初始化字符串内容，那么需要将字符串指针传递给函数，并返回处理后缓冲区的指针，这里要注意三点：
  - 必须传递缓冲区的地址和长度
  - 调用者负责释放缓冲区
  - 函数通常返回缓冲区的指针
- 在C语言中，数字 `0` 与 `\0` ，或者说 `NUL` 是不同的.ASIC中眞的的0是NUL，而数字0是有值的。
- `snprintf()` 函数是 `printf()` 函数的变体，第一和第二个参数是指定一个分配好了的区域地址和长度，以后的参数与 `printf()` 就相同了

`sizeof` 与 `strlen` 的区别

- `sizeof` 是运算符， `strlen` 是函数
- `sizeof` 可以用类型做参数， `strlen` 只能用 `char*` 做参数，且必须是以 `'\0'` 结尾的。
- 数组做 `sizeof` 的参数不退化，传递给 `strlen` 就退化为指针了。
- 实际上，用 `sizeof` 来返回类型以及静态分配的对象、结构或数组所占的空间，返回值跟对象、结构、数组所存储的内容没有关系。具体而言，当参数分别如下时， `sizeof` 返回的值表示的含义如下：
  - 数组——编译时分配的数组空间大小；
  - 指针——存储该指针所用的空间大小（存储该指针的地址的长度，是长整型，应该为4）；
  - 类型——该类型所占的空间大小；
  - 对象——对象的实际占用空间大小；
  - 函数——函数的返回类型所占的空间大小。函数的返回类型不能是void。
- 对于静态声明 `char str[20]="0123456789"`
  - `int a=strlen(str); //a=10` ， `strlen` 计算字符串的长度，以结束符 `\0` 为字符串结束。
  - `int b=sizeof(str); //b=20;` ， `sizeof` 计算的则是分配的数组 `str[20]` 所占的内存空间的大小，不受里面存储的内容改变。
- 对于静态声明 `char str[20]`
  - `int a = strlen(str); //a=0` ， `strlen` 计算字符串的长度，以结束符 `\0` 为字符串

结束，然后没有字符串，所以为0。

- `int b=sizeof(str); //b=20;` ， `sizeof` 计算的则是分配的数组 `str[20]` 所占的内存空间的大小，不受里面存储的内容改变。

- 对声明 `char *str="Hello World!"`
  - `int a=strlen(str); //a=12` ， `strlen` 计算字符串的长度，以结束符 `\0` 为字符串结束。
  - `int b=sizeof(str); //b=4;` ， `sizeof` 这里计算的是 `str` 指针所占的空间，所以为4

请查看代码**diffStrlenSizeof.c**

返回字符串

返回字符串实际是返回字符串的地址，有三种方式可以返回字符串地址：

- 字面量
  - 字面量没有作用域的概念，所以在函数内将字符串分配于字符串字面量池内，字符串地址不会随着函数结束而变动。
- 动态分配的内存
  - 只要将动态分配在堆内的地址返回给调用者，那么字符串的地址也不会丢失。
- 本地字符串变量 - 不要使用！

字符串与函数指针

字符串与函数指针相结合可以实现强大的功能，详细请查看代码。

第五章 结构体与指针

- 以下划线来标识结构体
- 结构体的声明最好用 `typedef` 来重新定义，这样写代码时会很方便
- 直接使用结构体定义时，使用 `.` 来操作成员，如果使用动态定义方式，使用 `->` 来操作成员。
- 使用 `malloc` 为结构体分配空间时，空间大小大于等于各成员大小之和。因为结构体所占空间大小并不是简单的将各成员大小相加，其中包含字节对齐等额外空间。
- 结构体各成员变量需要手动初始化和释放。
- 成员变量如果是动态分配初始化的，那么在释放时不能直接释放整个结构体，这时候成员变量所指向的堆区域还是有值的，直接释放整个结构体会造成内存泄漏。所以要先释放成员变量，然后再释放整个结构体。
- 把以上过程组合起来，就形成了构造函数和析构函数的原型。在C中，这两个过程需要手动处理，而且高级的面向对象语言中，这两个过程被自动执行。

结构体池

- 在大量需要结构体操作的时候，反复创建和释放会导致程序性能下降，为了优化创建释放过程，我们可以使用先创建一个结构体池的，需要时从池中获取，不需要时返回到池中。
- 使用以下方式可以定义一个结构体池

  ```
  #define SIZE 10
  Person *structPool[SIZE];
  ```

  结构体池的大小 `SIZE` 是最需要考量的变量：值过大，那么池会占用大量的内存，浪费空间；值太小，程序很快就用完了池中资源，仍然会大量创建释放内存，效率仍然不理想。

- 以下代码可以从池中获取结构体

```
Person *getStructPool()
{
  int i;
  for (i = 0; i < SIZE; ++i)
  {
      if (structPool[i] != NULL)
      {
          Person *ptr = structPool[i];
          structPool[i] = NULL;
          return ptr;
      }
  }

  Person *newPerson = (Person *)malloc(sizeof(Person));
  return newPerson;
}
```

我们通过轮询池数组找到第一个不为空的成员，然后将其返回，并将此位置设为空，腾出位置。如果池中资源都为空，那么只能重新配分一块空间。

- 以下代码可以将结构图存入池中

```
Person *setStructPool(Person *person)
{
  int i;
  for (i = 0; i < SIZE; ++i)
  {
      if (structPool[i] == NULL)
      {
          structPool[i] = person;
          return person;
      }
  }

  deallocatePerson(person);
  free(person);
  return NULL;
}
```

我们通过轮询池数组找到第一个空的成员，也就意味着此位置是空的，然后将结构体地址赋给数组元素，紧接着返回指针即可。如果池中资源都不为空，也就意味着池是满的，我们只能按正常方式将其释放。

> 注意: 存入池中资源后，返回了原指针，但是在外层我们不能将该指针释放，因为如果释放，那么池中的资源也会被释放。但是我们可以将该指针指向`NULL`。即该指针不可用。

- 对于清空结构体池，可以使用一下方式

```
void clearStructPool()
{
  int i;
  for (i = 0; i < SIZE; ++i)
      if(structPool[i]!=NULL)
      {
          deallocatePerson(structPool[i]);
          free(structPool[i]);
          structPool[i] = NULL;
      }
}
```

> 注意:
>
> 1. 清空顺序，先释放每个结构体内存成员的空间，然后释放整个结构体的空间，再将结构体指针指向 `NULL` 。这时整个释放过程才是安全完整的
> 2. 可以使用工具 `[Valgrind](http://valgrind.org/)` 来检测程序是否存在内存泄漏问题。

第五章 安全问题与指针误用

由于C语言的涉及初衷，就是相信程序员，所以C语言被设计的很灵活。但这也就意味着C语言并不是一个安全的语言，很多问题都需要程序员自己来注意。在使用指针时更是如此，由于指针对内存操作的强大灵活性，稍有不慎便会导致内存的误操作，从而影响程序。对C语言的安全操作似乎成为了一个专门的研究方向。

- CERT 以上是一个很好的了解和获取C语言安全问题解决方案的来源。

本章从某种角度算是之前各章的一个安全问题总结，因此本章主要以TIP的形式给出使用C指针时需要注意的问题。

指针的声明和初始化

- `int *ptr1, ptr2` ，这里 `ptr2` 不是指针
- 使用 `类型定义` 会比 `宏定义` 要好， `类型定义` 可以允许编译器进行检验， `宏定义` 不一定会
- 代码 `typedef int* PINT; PINT ptr1,ptr2` 可以更好地声明指针。
- 指针声明后如果没有初始化，则是 `野指针` 。其指向的区域是未知的，这时候读写该指针都是危险的。所以声明指针后应当初始化。处理未初始化的指针有三种方式
  - 最简单的方式是赋值为 `NULL`
  - 使用 `<assert.h>` 中的 `assert` 函数
  - 使用编译器 `-Wall` 选项

指针的使用

- 使用指针前检测是否为 `NULL`
- 注意迷途指针的产生
- 时刻注意内存区域的边界以及数据的大小
- 注意 `sizeof` 与 `strlen` 的区别
- 不要使用结构体指针偏移的方式来访问结构体成员变量

内存的释放

- 不要重复释放内存

第八章 其他话题

指针应用的地方还有很多，这里列举了几个高级应用话题。

指针类型转换

访问特殊地址

例如 `int num=10; int *ptr = (int *)num` 就是将 `ptr` 指向地址10，这样做是很危险的，因为我们无法知道地址10到底可用不可用。除非我们所要指向的地址是明确的，否则不建议使用这种操作。

但是在嵌入式系统中，这么做确有明显意义。嵌入式系统中，我们经常要做某些地址访问。这里的地址就是明确的，所以我们可以使用指针的类型转换。例如

```
#define VIDEO_BASE 0xB8000
int *videoAddress = (int *)VIDEO_BASE;
*videoAddress=FFFFFF;
```

这里便可以直接使用 `videoAddress` 来读写地址上的数据了

访问端口

在嵌入式系统中，我们还可以使用指针的类型转换来访问特定的端口来方便方便操作。例如：

```
#define PORT 0x0000FFF0
volatile unsigned int *const port = (unsigned int *) PORT;
```

这里声明很长，有几点要注意

1. `volatile` 限制编译器优化。编译器在编译时会进行一定的优化操作，导致指针指向的地址并不是我们想要的，所以此字段阻止了编译器的优化操作。
2. `unsigned int * const` 定义了一个指向无符号整型的常量指针。也就意味着指针指向区域内容可写可读，但是指针指向哪个区域确不可改变。保证了地址安全。

判断大小端

可以使用如下代码来判断系统的字节序

```
int num = 0x12345678;
char *adr = (char *)&num;
int i;
for (i = 0; i < 4; ++i)
    printf("%p: %02x\n",adr+i,(unsigned char )*(adr+i));
```

通过前两行使用指针的类型转换，`adr`指向了`num`的地址。该区域一共有4个字节。通过循环，依次读出了每个字节中的值。

别名

- 当两个指针指向同一位置时，那么其中一个指针就是另一个指针的别名
- 编译器编译指针时，默认指针都有别名，这会有一定的性能下降问题。我们可以使用关键字`restrict`来告诉编译器，该指针没有别名，来提高编译器性能。
- 对于`restrict`关键字来说，指针不能有别名，否则操作结果是未定义的

第九章 高级应用

本章其实是原书第八章后半部分的一些内容，但是这些内容都是一些高级应用，而且目前没有深究的必要，所以单拿出来做一些注释。

指针与线程

- 线程的内容涉及UNIX系统编程内容，需要了解POSIX线程才能更好的理解该部分，所以这里只是简单的记录了一下 `互斥锁` 的使用，完整代码有待学习了《UNIX环境高级编程》中线程相关内容后再进行完善。
- 以下代码是利用多线程计算向量点积。其中对于和的操作，线程之间会产生冲突，这时便用到了POSIX线程中的 `互斥锁` 概念：当某一对象被锁定时，其他对象无法对其进行操作。

```
typedef struct _vectorInfo
{
    double *vector01;
    double *vector02;
    double sum;
    int length;
}VectorInfo;

typedef struct _product
{
    vectInfo *vectinfo;
    int beginIndex;
}Product;
```

首先定义好结构体，其中有几个变量解释一下：

- `vector01` 和 `vector02` 是承载实体的向量指针
- `sum` 点积的结果。就是因为多个线程都要操作此字段，所以才有互斥锁。
- `length` 指的是每个线程要处理的向量长度。假设向量是由16个元素组成的数组，那么每个线程点积运算时，都可以只运算4个长度，例如线程1计算 `[0]~[3]` ,线程2计算 `[4]~[7]` ，以此类推。 `length` 字段就是指定的这个长度
- `beginIndex` 跟 `length` 配合来确定每个线程计算元素的起始位置。

```
pthread_mutex_t sumLock;
void dotProduct(void *prd)
{
    int i;
    Product *product       = (Product *)prd;
    VectorInfo *vectorinfo = product->vectinfo;

    int beginIndex = product->beginIndex;
    int endIndex   = beginIndex + product->length;

    double total = 0;

    for (i = beginIndex; i < endIndex; ++i)
        total+=vectorinfo->vector01[i] *  vectorinfo->vector02[i];

    pthread_mutex_lock(&sumLock);
    vectorinfo->sum+=total;
    pthread_mutex_unlock(&sumLock);

    pthread_exit((void *)0);
}
```

这就是进行点积运算的核心，其中有几个变量解释一下：

- `pthread_mutex_t sumLock` 是POSIX线程的内容，通过 `pthread_mutex_t` 定义了一个互斥锁变量
- 通过循环，每个线程处理向量的一段，进行点积运算，然后将结果加到总和 `SUM` 上
- 锁定互斥锁，开始操作 `sum` 变量，然后解开锁，这样每个线程都可以不产生冲突的操作 `sum` 变量了。

指针与回调函数

> 由于本人目前对回调函数还不是很了解，所以这里暂时略过，等用到时再来学习

指针与面向对象技术 - 隐式指针

> 本章内容与数据结构结合比较密切，所以放到学习数据结构时进行补充

指针与面向对象技术 - 多态

多态的技术主要是运用了函数指针。正如之前在函数与指针那一章所提到的，函数指针正是实现面向对象技术多态的关键。通过定义函数指针，在建立结构体时，动态的指向不同函数而实现多态方式。具体代码可以参看 **ploy.c** 这里会对代码进行说明

首先创建结构体，这里的结构体就是模拟类的行为

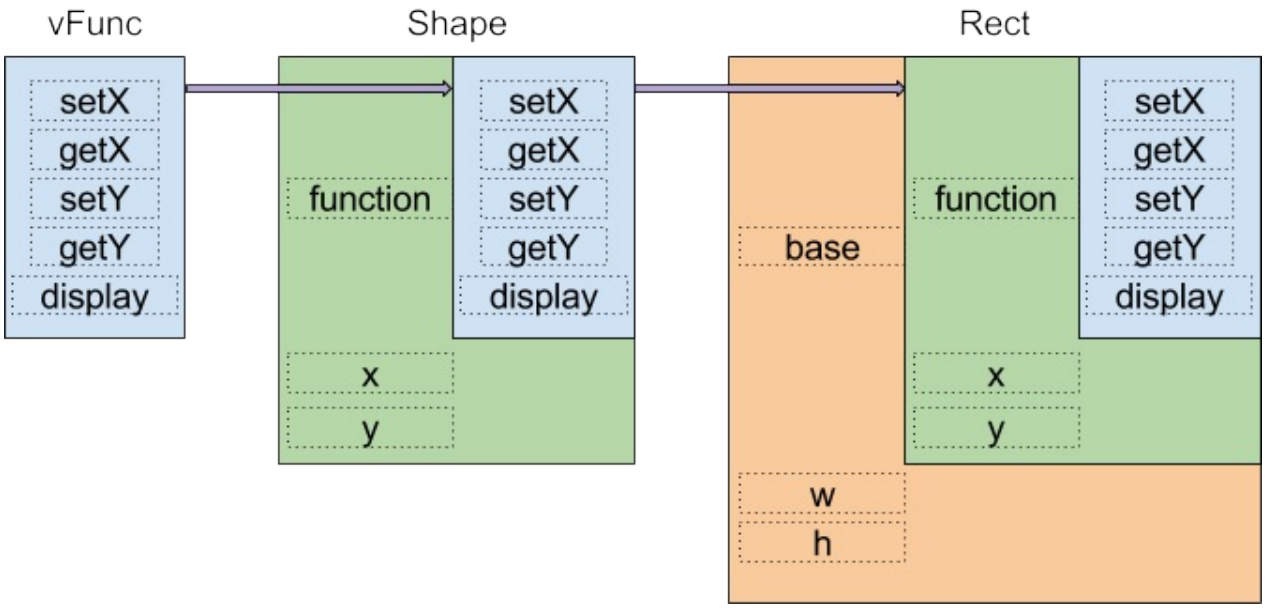
```
typedef void (*fptrSet)(void*, int);
typedef int  (*fptrGet)(void*);
typedef void (*fptrDisplay)();

typedef struct _vfunc
{
    fptrSet     setX;
    fptrGet     getX;
    fptrSet     setY;
    fptrGet     getY;
    fptrDisplay display;
} vFunc;

typedef struct _shape
{
    vFunc function;
    int x;
    int y;
} Shape;

typedef struct _rect
{
    Shape base;
    int w;
    int h;
} Rect;
```

- 首先是函数集合结构体 `vFunc` 。这里定义的函数都是类似于面向对象属性的读写操作。这里将使用函数指针。
- `Shape` 就是某种意义上的根类。其内部定义了两个私有变量，然后通过引用函数集合结构体，算作是类的方法
- `Rect` 结构体可以等同于 `Shape` 的派生类，其内部有两个私有变量，然后就是包含了整个 `Shape` 结构体。这样 `Rect` 也有了 `Shape` 的属性。而且函数集合是函数指针，在调用时可以动态的更改到所需要的函数。



接下来创建两个结构体 **Shape** 和 **Rect** 所需要的函数

```
void displayShape(){printf("Shape\n");}
void ShapeSetX(Shape *shape, int x){shape->x = x;}
void ShapeSetY(Shape *shape, int y){shape->y = y;}
int ShapeGetX(Shape *shape){return shape->x;}
int ShapeGetY(Shape *shape){return shape->y;}


void displayRect(){printf("Rect\n");}
void RectSetW(Rect *rect, int x){rect->w = x;}
void RectSetH(Rect *rect, int y){rect->h = y;}
int RectGetW(Rect *rect){return rect->w;}
int RectGetH(Rect *rect){return rect->h;}
```

创建建立实例的函数

```
Shape *newShape()
{
    Shape *shape = (Shape *)malloc(sizeof(Shape));
    shape->x = 10;
    shape->y = 10;
    shape->function.setX = ShapeSetX;
    shape->function.getX = ShapeGetX;
    shape->function.setY = ShapeSetY;
    shape->function.getY = ShapeGetY;
    shape->function.display = displayShape;
    return shape;
}

Rect *newRect()
{
    Rect *rect = (Rect *)malloc(sizeof(Rect));
    rect->w = 20;
    rect->h = 30;
    rect->base.function.setX = RectSetW;
    rect->base.function.getX = RectGetW;
    rect->base.function.setY = RectSetH;
    rect->base.function.getY = RectGetH;
    rect->base.function.display = displayRect;
    return rect;
}
```

正如刚才所说，这里通过函数集合结构体，分别指向需要的函数原型，从而实现了多态的效果。这两个建立实例的函数其实就很像**C++**中的构造函数了。那么既然有构造函数，就不要忘了在最后要释放内存，C语言是没有自动析构功能的！

> 这里要注意，由于函数指针fptrSet和fptrGet内部参数为**void ***。而直接调用时传入的是**Shape ***和**Rect ***。这里会导致编译器发出警告，至于怎么解决，目前还没有方法。需要以后深入研究。
>
> **2016年6月15日更新** 我将这个问题放到了stackoverflow上，最终得到了解答，更新后的代码如下。这样就不会产生警告了。

```
void displayShape(){printf("Shape\n");}
void ShapeSetX(void *voidShape, int x){Shape *shape = (Shape*)voidShape;shape->x = x;}
void ShapeSetY(void *voidShape, int y){Shape *shape = (Shape*)voidShape;shape->y = y;}
int ShapeGetX(void *voidShape){Shape *shape = (Shape*)voidShape;return shape->x;}
int ShapeGetY(void *voidShape){Shape *shape = (Shape*)voidShape;return shape->y;}

void displayRect(){printf("Rect\n");}
void RectSetW(void *voidRect, int x){Rect *rect = (Rect *)voidRect;rect->w = x;}
void RectSetH(void *voidRect, int y){Rect *rect = (Rect *)voidRect;rect->h = y;}
int RectGetW(void *voidRect){Rect *rect = (Rect *)voidRect;return rect->w;}
int RectGetH(void *voidRect){Rect *rect = (Rect *)voidRect;return rect->h;}
```

然后就是在主函数中使用之前定义的函数了。

```
    int i;

    Shape *sptr[2];
    sptr[0] = newShape();
    sptr[1] = newShape();
    sptr[0]->function.setX(sptr[0],111);
    sptr[1]->function.setX(sptr[1],112);

    for (i = 0; i < 2; ++i)
    {
        sptr[i]->function.display();
        printf("(%d, %d)\n", sptr[i]->function.getX(sptr[i]), sptr[i]->function.getY(s
ptr[i]));
    }

    Rect *rect = newRect();
    rect->base.function.display();
    printf("(%d, %d)\n", rect->base.function.getX(rect), rect->base.function.getY(rect
));

    free(sptr[0]);sptr[0]=NULL;
    free(sptr[1]);sptr[1]=NULL;
    free(rect);rect=NULL;
```

最后不要忘了释放内存！

以上的例子简单的实现了利用C语言中的指针实现多态技术。在C++中，实现多态的方式是虚表 （VTable） 。作用原理跟 vFunc 很相似。待到以后深入学习C++时再详细探讨。