# Void 指针

指针有两个属性：指向变量/对象的地址和长度，但是指针只存储地址,长度则取决于指针的类型；编译器根据指针的类型从指针指向的地址向后寻址，指针类型不同则寻址范围也不同,比如:int*从指定地址向后寻找 4 字节作为变量的存储单元 double*从指定地址向后寻找 8 字节作为变量的存储单元 void 即“无类型”，void *则为“无类型指针”，可以指向任何数据类型。

## 1 void 的作用

1) 对函数返回的限定

a) 当函数不需要返回值时，必须使用 void 限定。例如： voidfunc(int, int);

2) 对函数参数的限定

a) 当函数不允许接受参数时，必须使用 void 限定。例如： intfunc(void)。

## 2 void 指针使用规范

2.1 void 指针可以指向任意类型的数据，即可用任意数据类型的指针对 void 指针赋值。例如

```
int *pint;

void *pvoid; //它没有类型，或者说这个类型不能判断出指向对象的长度

pvoid = pint; //只获得变量/对象地址而不获得大小,但是不能 pint =pvoid;
```

2.2 如果要将 pvoid 赋给其他类型指针，则需要强制类型转换如：

```
pint = (int *)pvoid;   //转换类型也就是获得指向变量/对象大小
```

转:http://icoding.spaces.live.com/blog/cns!209684E38D520BA6!130.entry

2.3 void 指针不能复引用(即取内容的意思)

```
*pvoid                 //错误
```

要想复引用一个指针，或者使用“->”运算符复引用一部分，都要有对于指针指向的内存的解释规则。

例如，int *p;

那么，当你后面复印用 p 的时候，编译器就会把从 p 指向的地址开始的四个字节看作一个整数的补码。

因为 void 指针只知道指向变量/对象的起始地址，而不知道指向变量/对象的大小(占几个字节)所以无法正确引用

2.4　　void 指针类型运算

按照 ANSI(AmericanNationalStandardsInstitute)标准，不能对 void 指针进行算法操作，即下列操作都是不合法的：

```
void* pvoid;

pvoid++;                          //ANSI：错误

pvoid+=1;                         //ANSI：错误
```

ANSI 标准之所以这样认定，是因为它坚持：进行算法操作的指针必须是确定知道其指向数据类型大小的。

```
//例如：

int*pint;

pint++;                                 //ANSI：正确
```

pint++的结果是使其增大 sizeof(int)。

但是大名鼎鼎的 GNU(GNU'sNotUnix 的缩写)则不这么认定，它指定 void*的算法操作与 char*一致。

因此下列语句在 GNU 编译器中皆正确：

```
pvoid++;                          //GNU：正确

pvoid+=1;                         //GNU：正确
```

pvoid++的执行结果是其增大了 1。

在实际的程序设计中，为迎合 ANSI 标准，并提高程序的可移植性，我们可以这样编写实现同样功能的代码：

```
void*pvoid;

(char*)pvoid++;                           //ANSI：正确；GNU：正确

(char*)pvoid+=1;                          //ANSI：错误；GNU：正确
```

GNU 和 ANSI 还有一些区别，总体而言，GNU 较 ANSI 更“开放”，提供了对更多语法的支持。但是我们在真实设计时，还是应该尽可能地迎合 ANSI 标准。

2.5　　如果函数的参数可以是任意类型指针，那么应声明其参数为 void*。

典型的如内存操作函数 memcpy 和 memset 的函数原型分别为：

```
void*memcpy(void*dest,constvoid*src,size_tlen);

void*memset(void*buffer,intc,size_tnum);
```

这样，任何类型的指针都可以传入 memcpy 和 memset 中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么类型。如果 memcpy 和 memset 的参数类型不是 void*，而是 char*，那才叫真的奇怪了！这样的 memcpy 和 memset 明显不是一个“纯粹的，脱离低级趣味的”函数！

下面的代码执行正确：

```
//示例：memset 接受任意类型指针

int    intarray[100];

memset(intarray,0,100*sizeof(int));                    //将 intarray 清 0

//示例：memcpy 接受任意类型指针

int    intarray1[100],    intarray2[100];

memcpy(intarray1,intarray2,100*sizeof(int));//将 intarray2 拷贝给 intarray1
```

有趣的是，memcpy 和 memset 函数返回的也是 void*类型，标准库函数的编写者是多么地富有学问啊！

2.6　　void 不能代表一个真实的变量

下面代码都企图让 void 代表一个真实的变量，因此都是错误的代码：

```
void a;                                                //错误

function(void a);                                      //错误
```

void 体现了一种抽象，这个世界上的变量都是“有类型”的，譬如一个人不是男人就是女人（还有人妖？）。

void 的出现只是为了一种抽象的需要，如果你正确地理解了面向对象中“抽象基类”的概念，也很容易理解 void 数据类型。正如不能给抽象基类定义一个实例，我们也不能定义一个 void（让我们类比的称 void 为“抽象数据类型”）变量。

```
#include

#include

#include

using namespace std;

typedef struct tag_st

{

char id[10];
```

```
float fa[2];

}ST;

//我在程序里面这样使用的

int main()

{

ST * P=(ST *)malloc(sizeof(ST));

strcpy(P->id,"hello!");

P->fa[0]=1.1;

P->fa[1]=2.1;

ST * Q=(ST *)malloc(sizeof(ST));

strcpy(Q->id,"world!");

Q->fa[0]=3.1;

Q->fa[1]=4.1;

void ** plink=(void **)P;

*((ST *)(plink)) = * Q; //plink 要先强制转换一下,目的是为了让它先知道要覆盖的大小.

                                                    //P 的内容竟然给 Q 的内容覆盖掉了.

cout<id<<""<fa[0]<<" "<fa[1]<return 0;

}
```