# Typedef

## （一）简单用法

1.使用typedef为现有类型创建别名，定义易于记忆的类型名

typedef　int　size; size　array[4];

2.typedef 还可以掩饰[复合](https://baike.baidu.com/item/%E5%A4%8D%E5%90%88/2437031" \t "https://baike.baidu.com/item/_blank)类型，如[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "https://baike.baidu.com/item/_blank)和[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84" \t "https://baike.baidu.com/item/_blank)

* 只需这样定义，Line类型即代表了具有81个元素的[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6" \t "https://baike.baidu.com/item/_blank)数组，使用方法如下：

char　line[81];

typedef char Line[81];

Line　text,line;

* 可以像下面这样隐藏[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "https://baike.baidu.com/item/_blank)语法：

|  |
| --- |
| **typedef**　**char**\*　pstr; |

|  |  |
| --- | --- |
| 1 | **int**　mystrcmp(**const**　pstr　p1,**const**　pstr　p3); |

const char\*和char\* const表达的并非同一意思，const char\*的意思是创建一个指向char类型的指针且**不能更改指向地址上的值**，而char\* const则是**不能更改指向的地址**；

char\* const p ： 定义一个指向字符的指针常数，即const指针，常量指针。

const char\* p ：定义一个指向字符型常量的指针。

### 1、typedef的最简单使用

|  |  |
| --- | --- |
| 1 | **typedef**　**long**　byte\_4; |

给已知数据类型long起个新名字，叫byte\_4。

### 2、 typedef与结构结合使用

|  |  |
| --- | --- |
| 1  2  3  4  5 | **typedef**　**struct**　tagMyStruct  {  **int**　iNum;  **long**　lLength;  }MyStruct; |

这语句实际上完成两个操作：

1) 定义一个新的结构类型

|  |  |
| --- | --- |
| 1  2  3  4  5 | **struct**　tagMyStruct  {  **int**　iNum;  **long**　lLength;  }; |

分析：tagMyStruct称为“tag”，即“标签”，实际上是一个临时名字，struct[关键字](https://baike.baidu.com/item/%E5%85%B3%E9%94%AE%E5%AD%97" \t "https://baike.baidu.com/item/_blank)和tagMyStruct一起，构成了这个结构类型，不论是否有typedef，这个结构都存在。

我们可以用struct tagMyStruct varName来定义变量，但要注意，使用tagMyStruct varName来定义变量是不对的，因为struct 和tagMyStruct合在一起才能表示一个结构类型。

2) typedef为这个新的结构起了一个名字，叫MyStruct。

typedef struct tagMyStruct MyStruct;

因此，MyStruct实际上相当于struct tagMyStruct，我们可以使用MyStruct varName来定义变量。

1)、

|  |  |
| --- | --- |
| 1  2  3  4  5 | **typedef**　**struct**　tagNode  {  **char**\*　pItem;  **struct**　tagNode\*　pNext;  }\*pNode; |

2)、

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **typedef**　**struct**　tagNode\*　pNode;  **struct**　tagNode  {  **char**\*　pItem;  pNode　pNext;//这边不用pNode\* ，pNode 已经表示了struct tagNode\*  }; |

注意：在这个例子中，你用typedef给一个还未完全声明的类型起新名字。C语言[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "https://baike.baidu.com/item/_blank)支持这种做法。

3)、规范做法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **struct**　tagNode  {  **char**\*　pItem;  **struct**　tagNode\*　pNext;  };  **typedef**　**struct**　tagNode\*　pNode; |

### 3. typedef & #define的问题

有下面两种定义pStr数据类型的方法，两者有什么不同？哪一种更好一点？

|  |  |
| --- | --- |
| 1  2 | **typedef**　**char**\*　pStr;  #define　pStr　char\* |

答案与分析：

通常讲，typedef要比#define要好，特别是在有[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "https://baike.baidu.com/item/_blank)的场合。请看例子：

|  |  |
| --- | --- |
| 1  2  3  4 | **typedef**　**char**\*　pStr1;  #define　pStr2　char\*  pStr1　s1,s2;   pStr2　s3,s4; |

在上述的变量定义中，s1、s2、s3都被定义为char \*，而s4则定义成了char，不是我们所预期的[指针变量](https://baike.baidu.com/item/%E6%8C%87%E9%92%88%E5%8F%98%E9%87%8F" \t "https://baike.baidu.com/item/_blank)，根本原因就在于#define只是简单的字符串替换而typedef则是为一个类型起新名字。

上例中define语句必须写成 pStr2 s3, \*s4; 这样才能正常执行。

### const 与 指针

下面的代码中[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "https://baike.baidu.com/item/_blank)会报一个错误，你知道是哪个语句错了吗？

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **typedef** **char** \*pStr;  **char** string[4]="abc";  **const** **char** \*p1=string;  **const** pStr p2=string;  p1++;  p2++; |

答案与分析：

是p2++出错了。这个问题再一次提醒我们：typedef和#define不同，它不是简单的[文本替换](https://baike.baidu.com/item/%E6%96%87%E6%9C%AC%E6%9B%BF%E6%8D%A2" \t "https://baike.baidu.com/item/_blank)。上述代码中const pStr p2并不等于const char \* p2。const pStr p2和pStr const p2本质上没有区别，都是对变量进行只读限制，只不过此处变量p2的数据类型是我们自己定义的而不是系统固有类型而已。因此，const pStr p2的含义是：限定数据类型为char \*的变量p2为只读，因此p2++错误。

解释：const char \*p1 = string; 你可以这样理解：(const char) \*p1 = string, p1是一个指针，指向const char的东西，这个东西就是string(string是一个字符数组的首地址，它的地址声明后肯定是const的，除非该数组销毁)，但是p1是一个指针变量，它是可以递增的，即你看到的p1++，它可以完成从数组的来遍历数组的目的。  
　　而const pStr p2 = string;是这样的：由于p2不是指针，const直接修饰到了p2,即现在的p2是常量了（所以p2++就出错了，因为常量不能改），它的类型是pStr(我们自己定义的类型)，相当于 const int p2, const long p2等等，const都是直接修饰p2的，只不过int，long是系统类型，而pStr是我们定义的类型。为什么会出现这种效果了，就是因为 typedef,它把**char \***定义成一个复合的类型，要从整体上来理解语义，而不是字符替换后来理解语义。

#### **const在前面**

　　const int nValue； //nValue是const  
　　const char \*pContent; //\*pContent是const, pContent可变  
　　const (char \*) pContent;//pContent是const,\*pContent可变  
　　char\* const pContent; //pContent是const,\*pContent可变  
　　const char\* const pContent; //pContent和\*pContent都是const

#### **const在后面**

int const nValue； // nValue是const  
　　char const \* pContent;// \*pContent是const, pContent可变  
　　(char \*) const pContent;//pContent是const,\*pContent可变  
　　char\* const pContent;// pContent是const,\*pContent可变  
　　char const\* const pContent;// pContent和\*pContent都是const

#### 判断原则

const和指针一起使用是C语言中一个很常见的困惑之处，在实际开发中，特别是在看别人代码的时候，常常会因为这样而不好判断作者的意图，下面讲一下我的判断原则：  
　　沿着\*号划一条线，如果const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；如果const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。你可以根据这个规则来看上面声明的实际意义，相信定会一目了然。  
　　另外，需要注意：对于const (char \*) ; 因为char \*是一个整体，相当于一个类型(如 char)，因此，这是限定指针是const。

#### 助记

Bjarne在他的The C++ Programming Language里面给出过一个助记的方法：  
把一个声明从右向左读。  
  
char \* const cp; ( \* 读成 pointer to ) cp is a const pointer to char  
const char \* p; p is a pointer to const char;  
char const \* p; 同上因为C++里面没有const\*的运算符，所以const只能属于前面的类型。

#### 结论

char \* const cp     : 定义一个指向字符的指针常数，即const指针  
const char\* p       : 定义一个指向字符常数的指针  
char const\* p       : 等同于const char\* p  
const   char   \*\*是一个指向指针的指针，那个指针又指向一个字符串常量。     
char   \*\*也是一个指向指针的指针，那个指针又指向一个字符串变量。

#### Const修饰函数参数

const修饰函数参数是它最广泛的一种用途，它表示函数体中不能修改参数的值(包括参数本身的值或者参数其中包含的值)。它可以很好  
  
void function(const int Var); //传递过来的参数在函数内不可以改变(无意义，因为Var本身就是形参)  
  
void function(const char\* Var); //参数指针所指内容为常量不可变  
  
void function(char\* const Var); //参数指针本身为常量不可变(也无意义， 因为char\* Var也是形参)

## （二）typedef & 复杂的变量声明

理解复杂声明可用的“**右左法则**”：  
　　从变量名看起，先往右，再往左，碰到一个圆括号就调转阅读的方向；括号内分析完就跳出括号，还是按先右后左的顺序，如此循环，直到整个声明分析完。举例：  
　　int (\*func)(int \*p);  
　　首 先找到变量名func，外面有一对圆括号，而且左边是一个\*号，这说明func是一个指针；然后跳出这个圆括号，先看右边，又遇到圆括号（只有函数后面才跟形参圆括号），这说明 **(\*func)是一个函数**，所以**func是一个指向这类函数的**指针，即函数指针，这类函数具有int\*类型的形参，返回值类型是int，此处就是声明函数。

int (\*func[5])(int \*);  
　　func 右边是一个[]运算符，说明func是具有5个元素的数组；func的左边有一个\*，说明func的元素是指针（**注意这里的\*不是修饰func，而是修饰 func[5]的，原因是[]运算符优先级比\*高，func先跟[]结合**）。跳出这个括号，看右边，又遇到圆括号，说明func数组的元素是函数类型的指 针，它指向的函数具有int\*类型的形参，返回值类型为int。

### 2个模式：

type (\*)(....)函数指针

type (\*)[]数组指针

### 练习

>1：int \*(\*a[5])(int, char\*);

>2：void (\*b[10]) (void (\*)());

>3. double(\* (\*pa)[9])();

答案与分析：

对复杂变量建立一个类型别名的方法很简单，你只要在传统的变量声明[表达式](https://baike.baidu.com/item/%E8%A1%A8%E8%BE%BE%E5%BC%8F" \t "https://baike.baidu.com/item/_blank)里用[类型名](https://baike.baidu.com/item/%E7%B1%BB%E5%9E%8B%E5%90%8D" \t "https://baike.baidu.com/item/_blank)替代变量名，然后把[关键字](https://baike.baidu.com/item/%E5%85%B3%E9%94%AE%E5%AD%97" \t "https://baike.baidu.com/item/_blank)typedef加在该语句的开头就行了。

>1：int \*(\*a[5])(int, char\*);

//pFun是我们建的一个类型别名

typedef int \*(\*pFun)(int, char\*);

//使用定义的新类型来声明对象，等价于int\* (\*a[5])(int, char\*);

pFun a[5];

>2：void (\*b[10]) (**void (\*)()**);

//首先为上面[表达式](https://baike.baidu.com/item/%E8%A1%A8%E8%BE%BE%E5%BC%8F" \t "https://baike.baidu.com/item/_blank)加粗部分声明一个新类型

typedef void (\*pFunParam)();

//整体声明一个新类型

typedef void (\*pFun)(pFunParam);

//使用定义的新类型来声明对象，等价于void (\*b[10]) (void (\*)());

pFun b[10];

>3. double(\* [1]  (\*pa)[9])() [2]  ;

//首先为上面表达式蓝色部分声明一个新类型

typedef double(\*pFun)();

//整体声明一个新类型

typedef pFun (\*pFunParam)[9];

//使用定义的新类型来声明对象，等价于double(\*(\*pa)[9])();

pFunParam pa;

pa是一个指针，指针指向一个数组，这个数组有9个元素，每一个元素都是“doube(\*)()”--也即一个指针，指向一个函数，函数参数为空，返回值是“double”。

# 指针

指针（Pointer）是编程语言中的一个对象，**利用地址**，它的值直接指向（points to）存在电脑存储器中另一个地方的值。由于**通过地址能找到所需的变量单元**，可以说，地址指向该变量单元。因此，**将地址形象化的称为“指针”**。意思是通过它能找到以它为地址的内存单元。指针一般指向**一个函数或一个变量**。在使用一个指针时，一个程序既可以直接使用这个指针所储存的内存地址，又可以使用这个地址里储存的函数的值。

使用指针来读取数据，在重复性操作的状况下，可以明显改善程序性能，例如在[遍历](https://baike.baidu.com/item/%E9%81%8D%E5%8E%86" \t "https://baike.baidu.com/item/%E6%8C%87%E9%92%88/_blank)字符串，查取表格，控制表格及树状结构上。对指针进行复制，之后再解引用指针以取出数据，无论在时间或空间上，都比直接复制及访问数据本身来的经济快速。

指针，是一个无符号整数（unsigned int），它是一个以当前系统寻址范围为取值范围的整数。32位系统下寻址能力（地址空间）是4G Bytes（0~2^32-1）二进制表示长度为32bits（也就是4Bytes）， unsigned int类型也正好如此取值。

每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。

## 什么是指针？

**指针**是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。**指针变量声明**的一般形式为：

type \*var-name;

以下是有效的指针声明：

int \*ip; /\* 一个整型的指针 \*/

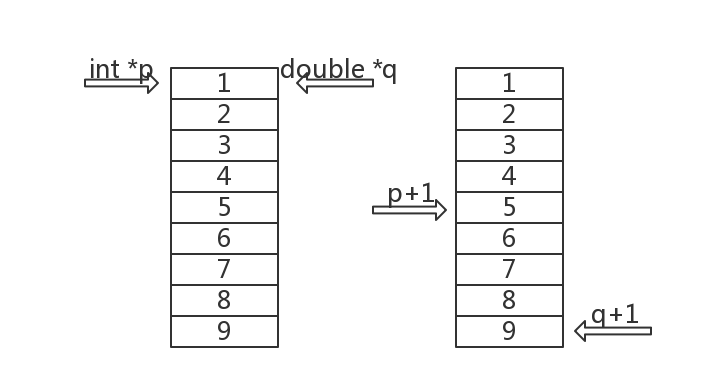
double \*dp; /\* 一个 double 型的指针\*/

float \*fp; /\* 一个浮点型的指针 \*/

char \*ch; /\* 一个字符型的指针 \*/

所有实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，对应**指针的值**的类型都是一样的，都是一个**代表内存地址**的长的十六进制数。

不同数据类型的指针之间**唯一的不同**是，指针所指向的**变量或常量**的**数据类型**不同。



## 如何使用指针？

使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 **\***来返回位于操作数所**指定地址的变量的值**。

## C 中的 NULL 指针

在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值是一个良好的编程习惯。赋为 NULL 值的指针被称为**空**指针。

NULL 指针是一个定义在标准库中的值为零的常量。

int \*ptr = NULL;

printf("ptr 的地址是 %p\n", ptr);

ptr 的地址是0x0

在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，它表明该指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

## C 指针的算术运算

* 指针的每一次递增，它其实会指向下一个元素的存储单元。
* 指针的每一次递减，它都会指向前一个元素的存储单元。
* 指针在递增和递减时跳跃的字节数取决于指针所指向变量数据类型长度，比如int就是 4 个字节。

## 递增一个指针

我们喜欢在程序中使用指针代替数组，因为变量指针可以递增，而数组不能递增，数组可以看成一个指针常量。

## C 指针数组和数组指针

### 1.char类型的指针

char \*names[]={"zhangsan","lisi","wangwu"};//可以这样初始化

这样，每个指针指向对应的字符串

### 2.其他类型的指针

int \*nums[]={1,2,3,4,5};//不可以这样初始化

原因：指针的初始化或赋值可以使用0值、常量表达式、和类型匹配的对象的地址。//指针需要用地址初始化

如上，字符串字面值常量类型为const char \*，与指针类型匹配，可赋值。

整形字面值常量可以为int, long等类型。但给指针赋值应该是地址，即&i，VS2010下编译错误也会提示：不可将int 类型赋给int \*类型。

在许多 C 程序中，指针常被用于引用数组，或者作为数组的元素。指向数组的指针常被简称为数组指针（array pointer），而具有指针类型元素的数组则被称为指针数组（pointer array）。

### 3.数组指针

要声明指向数组类型的指针，必须使用括号，如下所示：

1. int(\* arrPtr)[10]= NULL;// 一个指针，它指向一个**有10个int元素的数组**

如果没有括号，则声明 int\*arrPtr[l0]；表示 arrPtr 是一个具有 10 个 int 类型指针的数组。

在该例中，指向有 10 个 int 元素的数组的指针会被初始化为 NULL。然而，如果把合适数组的地址分配给它，那么表达式 \*arrPtr 会获得数组，并且（\*arrPtr）[i] 会获得索引值为 i 的数组元素。根据下标运算符的规则，表达式（\*arrPtr）[i] 等同于 \*（（\*arrPtr）+i）。因此，\*\*arrPtr 获得数组的第一个元素，其索引值为 0。  
  
为了展示数组指针 arrPtr 的几个运算，下例使用它来定位一个二维数组的某些元素，也就是矩阵内的某些行：

1. int matrix[3][10];// 3行，10列的数组
2. // 数组名称是一个指向第一个元素的指针，也就是第一行的指针
3. arrPtr = matrix;// 使得arrPtr指向矩阵的第一行
4. (\*arrPtr)[0]=5;// 将5赋值给第一行的第一个元素
5. arrPtr[2][9]=6;// 将6赋值给最后一行的最后一个元素
6. ++arrPtr;// 将指针移动到下一行
7. (\*arrPtr)[0]=7;// 将7赋值给第二行的第一个元素

在初始化赋值后，arrPtr 指向矩阵的第一个行，正如矩阵名称 matrix 一样。在这种情况下，使用 arrPtr 获取元素的方式与使用 matrix 完全一样。例如，赋值运算（\*arrPtr）[0]=5 等效于 arrPtr[0][0]=5 和 matrix[0][0]=5。

然而，与数组名称 matrix 不同的是，指针名称 arrPtr 并不代表一个常量地址，如运算 ++arrPtr 所示，它进行了自增运算。这个自增运算会造成存储在数组指针的地址增加一个数组空间大小，在本例中，即增加矩阵一行的空间大小，也就是 10 乘以 int 元素在内存中所占字节数量。

### 4.指针数组

指针数组（也就是元素为指针类型的数组）常常作为二维数组的一种便捷替代方式。一般情况下，这种数组中的指针会指向动态分配的内存区域。

使用指针数组，让指针指向对象（在此处的对象就是字符串），然后只给实际存在的对象分配内存（未用到的数组元素则是空指针）。

1. #define ARRAY\_LEN 100
2. char \*myStrPtr[ARRAY\_LEN] = // char指针的数组
3. { // 墨菲定律的几条推论：
4. “会出错的事，总会出错。”
5. “世上没有绝对正确的事情。”
6. “每个解决办法都会衍生出新的问题。”
7. };

尚未使用的指针可以在运行时指向另一个字符串。所需的存储空间可以利用这种常见方法来动态地保留。当不再需要该内存时，可以释放。

### 函数指针

#### 定义

如果在程序中定义了一个函数，那么在编译时系统就会为这个函数代码分配一段存储空间，这段存储空间的首地址称为这个函数的地址。而且函数名表示的就是这个地址。既然是地址我们就可以定义一个指针变量来存放，这个指针变量就叫作函数指针变量，简称函数指针。

例如：

int(\*p)(int, int);

这个语句就定义了一个指向函数的指针变量 p。首先它是一个指针变量，所以要有一个“\*”，即（\*p）；其次前面的 int 表示这个指针变量可以指向返回值类型为 int 型的函数；后面括号中的两个 int 表示这个指针变量可以指向有两个参数且都是 int 型的函数。所以合起来这个语句的意思就是：定义了一个指针变量 p，该指针变量可以指向返回值类型为 int 型，且有两个整型参数的函数。**p 的类型为 int(\*)(int，int)**。  
  
所以函数指针的定义方式为：

函数返回值类型 (\* 指针变量名) (函数参数列表);

“函数返回值类型”表示该指针变量可以指向具有什么返回值类型的函数；“函数参数列表”表示该指针变量可以指向具有什么参数列表的函数。这个参数列表中**只需要写函数的参数类型即可。**  
我们看到，函数指针的定义就是**将**“函数声明”中的“**函数名**”**改成**“（**\*指针变量名**）”。但是这里需要注意的是：“（\*指针变量名）”***两端的括号不能省略***，括号改变了运算符的优先级。如果省略了括号，就不是定义函数指针而是一个函数声明了，即声明了一个返回值类型为指针型的函数。  
  
那么怎么判断一个指针变量是指向变量的指针变量还是指向函数的指针变量呢？首先看变量名前面有没有“\*”，如果有“\*”说明是指针变量；其次看变量名的后面有没有带有形参类型的圆括号，如果有就是指向函数的指针变量，即函数指针，如果没有就是指向变量的指针变量。  
  
最后需要注意的是，指向函数的指针变量没有 ++ 和 -- 运算。

#### 用函数指针调用函数

举个例子：

1. int Func(int x); /\*声明一个函数\*/
2. int (\*p) (int x); /\*定义一个函数指针\*/
3. p = Func; /\*将Func函数的首地址赋给指针变量p\*/

赋值时函数 Func 不带括号，也不带参数。由于函数名 Func 代表函数的首地址，因此经过赋值以后，指针变量 p 就指向函数 Func() 代码的首地址了。

#### 应用例程

1. # include <stdio.h>
2. int Max(int, int); //函数声明
3. int main(void)
4. {
5. int(\*p)(int, int); //定义一个函数指针
6. int a, b, c;
7. p = Max; //把函数Max赋给指针变量p, 使p指向Max函数
8. printf("please enter a and b:");
9. scanf("%d%d", &a, &b);
10. c = (\*p)(a, b); //通过函数指针调用Max函数
11. printf("a = %d\nb = %d\nmax = %d\n", a, b, c);
12. **return** 0;
13. }

# 三．C enum(枚举)

枚举是 C 语言中的一种基本数据类型，它可以让数据更简洁，更易读。

一个变量，只有几种选择的地方，比如性别设置，只有两个选择，就可以用枚举，在比如生日的月份，只有12个选择，也可以用枚举类型

枚举语法定义格式为：

enum　枚举名　{枚举元素1,枚举元素2,……};

**注意：**第一个枚举成员的默认值为整型的 0，后续枚举成员的值在前一个成员上加 1。我们在这个实例中把第一个枚举成员的值定义为 1，第二个就为 2，以此类推。

*可以在定义枚举类型时改变枚举元素的值：*

enum season {spring, summer=3, autumn, winter};

*没有指定值的枚举元素，其值为前一元素加 1。也就说 spring 的值为 0，summer 的值为 3，autumn 的值为 4，winter 的值为 5*

### **枚举变量的定义**

前面我们只是声明了枚举类型，接下来我们看看如何定义枚举变量。

我们可以通过以下三种方式来定义枚举变量

**1、先定义枚举类型，再定义枚举变量**

enum DAY{

MON=1, TUE, WED, THU, FRI, SAT, SUN};enum DAY day;

**2、定义枚举类型的同时定义枚举变量**

enum DAY{

MON=1, TUE, WED, THU, FRI, SAT, SUN} day;

**3、省略枚举名称，直接定义枚举变量**

enum{

MON=1, TUE, WED, THU, FRI, SAT, SUN} day;

在C 语言中，枚举类型是被当做 int 或者 unsigned int 类型来处理的，所以按照 C 语言规范是没有办法遍历枚举类型的。

不过在一些特殊的情况下，枚举类型必须连续是可以实现有条件的遍历。

### 枚举在 switch 中的使用：

实例

#include <stdio.h>

#include <stdlib.h>

int main()

{

enum color { red=1, green, blue };

enum color favorite\_color;

/\* ask user to choose color \*/

printf("请输入你喜欢的颜色: (1. red, 2. green, 3. blue): ");

scanf("%d", &favorite\_color);

/\* 输出结果 \*/

switch (favorite\_color)

{

case red:

printf("你喜欢的颜色是红色");

break;

case green:

printf("你喜欢的颜色是绿色");

break;

case blue:

printf("你喜欢的颜色是蓝色");

break;

default:

printf("你没有选择你喜欢的颜色");

}

return 0;

}

# 四．void \*

## 1.基本概念

void\* 这不叫空指针,这叫无确切类型指针.这个指针指向一块内存,却没有告诉程序该用何种方式来解释这片内存.所以这种类型的指针不能直接进行取内容的操作.必须先转成别的类型的指针才可以把内容解释出来.  
  
还有'\0',这也不是空指针所指的内容. '\0'是表示一个字符串的结尾而已,并不是NULL的意思.  
  
真正的空指针是说,这个指针没有指向一块有意义的内存,比如说:  
char\* k;  
这里这个k就叫空指针.我们并未让它指向任意地点.  
又或者  
char\* k = NULL;  
这里这个k也叫空指针,因为它指向NULL 也就是0,注意是整数0,不是'\0'  
一个空指针我们也无法对它进行取内容操作.  
空指针只有在真正指向了一块有意义的内存后,我们才能对它取内容.也就是说要这样  
k = "hello world!";  
这时k就不是空指针了.

## 函数参数

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, const void \*src, [size\_t](https://www.baidu.com/s?wd=size_t&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank) len);  
void \* memset ( void \* buffer, int c, [size\_t](https://www.baidu.com/s?wd=size_t&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank) num );  
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。空指针仅仅是申明了一个变量名，告诉编译器有个变量存在，当然你后面肯定会用到它（事实上你不用也没人怪你）。

## 3.void 指针的使用规则

1. void 指针可以指向**任意类型**的数据，就是说可以用任意类型的指针对 void 指针对 void 指针赋值。例如：

int \*a；void \*p；

p=a；

如果要将 void 指针 p 赋给其他类型的指针，则需要强制类型转换，就本例而言：**a=（int \*）p**。在内存的分配中我们可以见到 void 指针使用：内存分配函数 malloc 函数返回的指针就是 **void \*** 型，用户在使用这个指针的时候，要进行强制类型转换，也就是显式说明该指针指向的内存中是存放的什么类型的数据 **(int \*)malloc(1024)** 表示强制规定 malloc 返回的 void\* 指针指向的内存中存放的是一个个的 int 型数据。

2. 在 ANSI C 标准中，不允许对 void 指针进行一些算术运算如 p++ 或 p+=1 等，因为既然 void 是无类型，那么每次算术运算我们就不知道该操作几个字节，例如 char 型操作 sizeof(char) 字节，而 int 则要操作 sizeof(int) 字节。

众所周知，如果指针 p1 和 p2 的类型相同，那么我们可以直接在 p1 和 p2 间互相赋值；如果 p1 和 p2 指向不同的数据类型，则必须使用强制类型转换运算符把赋值运算符右边的指针类型转换为左边指针的类型。

float \*p1;int \*p2;

p1 = p2;//其中p1 = p2语句会编译出错，//提示“'=' : cannot convert from 'int \*' to 'float \*'”，必须改为：

p1 = (float \*)p2;

而 **void \*** 则不同，任何类型的指针都可以直接赋值给它，无需进行强制类型转换。

void \*p1;int \*p2;

p1 = p2;

但这并不意味着，**void \*** 也可以无需强制类型转换地赋给其它类型的指针。因为"无类型"可以包容"有类型"，而"有类型"则不能包容"无类型"。

# Union

## （一）定义

union 共用体名{

成员列表

};

共用体有时也被称为联合或者联合体，这也是 Union 这个单词的本意。

结构体和共用体的区别在于：结构体的各个成员会占用不同的内存，互相之间没有影响；而共用体的所有成员占用同一段内存，修改一个成员会影响其余所有成员。

**结构体**占用的内存大于等于所有成员占用的内存的总和（成员之间可能会存在缝隙），共用体占用的内存等于**最长的成员**占用的内存。共用体使用了**内存覆盖技术**，同一时刻只能保存一个成员的值，如果对新的成员赋值，就会把原来成员的值覆盖掉。

共用体也是一种自定义类型，可以通过它来创建变量，例如：

union data{

int n;

char ch;

double f;

};

union data a, b, c;

上面是先定义共用体，再创建变量，也可以在定义共用体的同时创建变量：

union data{

int n;

char ch;

double f;

} a, b, c;

如果不再定义新的变量，也可以将共用体的名字省略：

union{

int n;

char ch;

double f;

} a, b, c;

共用体 data 中，成员 f 占用的内存最多，为 8 个字节，所以 data 类型的变量（也就是 a、b、c）也占用 8 个字节的内存

## （二）例子

#include <stdio.h>

union data{

int n;

char ch;

short m;

};

int main(){

union data a;

printf("%d, %d\n", sizeof(a), sizeof(union data) );

a.n = 0x40;

printf("%X, %c, %hX\n", a.n, a.ch, a.m);

a.ch = '9';

printf("%X, %c, %hX\n", a.n, a.ch, a.m);

a.m = 0x2059;

printf("%X, %c, %hX\n", a.n, a.ch, a.m);

a.n = 0x3E25AD54;

printf("%X, %c, %hX\n", a.n, a.ch, a.m);

return 0;

}

运行结果：

4, 4

40, @, 40

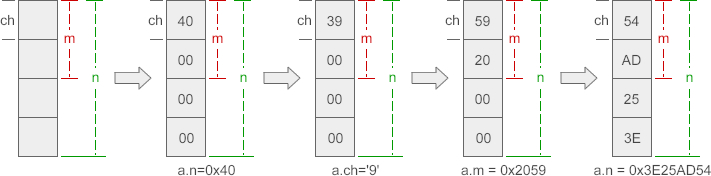
39, 9, 39

2059, Y, 2059

3E25AD54, T, AD54

这段代码不但验证了共用体的长度，还说明共用体成员之间会相互影响，修改一个成员的值会影响其他成员。

要想理解上面的输出结果，弄清成员之间究竟是如何相互影响的，就得了解各个成员在内存中的分布。以上面的 data 为例，各个成员在内存中的分布如下：



成员 n、ch、m 在内存中“对齐”到一头，对 ch 赋值修改的是前一个字节，对 m 赋值修改的是前两个字节，对 n 赋值修改的是全部字节。也就是说，ch、m 会影响到 n 的一部分数据，而 n 会影响到 ch、m 的全部数据。