目录

[关于函数名： 2](#_Toc427568252)

[关于变量： 2](#_Toc427568253)

[补充：static与extern： 4](#_Toc427568254)

[关于函数的返回： 4](#_Toc427568255)

[关于如何生成一个可执行文件: 5](#_Toc427568256)

[关于指针： 7](#_Toc427568257)

[关于typedef: 11](#_Toc427568258)

[关于结构体： 16](#_Toc427568259)

[关于指针与数组： 19](#_Toc427568260)

[关于不定参数: 29](#_Toc427568261)

[关于函数指针： 30](#_Toc427568262)

[关于内存中的数据与指令 31](#_Toc427568263)

[关于内存空间的几个段: 31](#_Toc427568264)

[关于函数封装： 34](#_Toc427568265)

[关于C语言中的文件结构： 34](#_Toc427568266)

[关于结构体实现对象的封装： 36](#_Toc427568267)

[关于DOS向main函数传递参数： 37](#_Toc427568268)

[关于System函数: 40](#_Toc427568269)

[关于单链表： 40](#_Toc427568270)

[关于面向对象： 41](#_Toc427568271)

# 关于函数名：

函数名代表了函数的入口地址信息。我们可以通过函数名获得函数的入口地址。Tc下可以通过函数名获得函数的入口地址信息，包括段地址和偏移地址，不过它可能会将段地址信息隐藏起来，在获取段地址时需要我们采取某种方法。（如下小模式下， 定义了一个函数func, j = func只能获得偏移地址， 但是j=(long)func即能获得偏移地址又能获得段地址）。 函数名可以看成为一个函数指针符号；但不能对函数名进行运算，因为不知道它指向内容的长度。

补充：有一个函数void func(void); 那么 func与&func在数值上是相等的。（实际上&func中的&是多余的）。

# 关于变量：

全局变量：存储空间是在编译期分配的、在数据段中 （不是在栈上），它的生存周期是全程的，作用域是全局的（如用stitac修饰则为静态全局变量作用域被限制在本文件），它是编译期初始化的（如果没有显示的初始化编译器会将他们初始化为0，无对应的初始化机器指令）。

静态局部变量：存储空间是在编译期分配的、在数据段中 （不是在栈上），它的生存周期是全程的，作用域是局部的（在定义它的函数内有效），它是编译期初始化的（如果没有显示的初始化编译器会将他们初始化为0，无对应的初始化机器指令）。

非静态局部变量：存储空间是在运行期分配的、在栈段中，它的生存周期是局部的，作用域是局部的（在定义它的函数内有效），编译器不会对他们进行初始化。在定义变量时的初始化实际上是运行时完成的（有对应的初始化机器指令）。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | 非静态、初始化 | 非静态、未初始化 | 静态、未初始化 | 静态、初始化 |
| 作用  域 | | 全局  变量 | 从定义处开始到程序结束 | 从定义处开始到程序结束 | 从定义处开始到本文件结束 | 从定义处开始到本文件结束 |
|  |  |  |  |  |
| 局部  变量 | 定义该局部变量的函数内 | 定义该局部变量的函数内 | 定义该局部变量的函数内 | 定义该局部变量的函数内 |
| 生存  周期 | | 全局  变量 | 全局 | 全局 | 全局 | 全局 |
|  |  |  |  |  |
| 局部  变量 | 定义处至定义它的函数返回 | 定义处至定义它的函数返回 | 全局 | 全局 |
| 内存  分配 | | 全局  变量 | 编译期  （静态数据区） | 编译期  （静态数据区） | 编译期  （静态数据区） | 编译期  （静态数据区） |
|  | 运行时 | 运行时 | 编译期 | 编译期 |
| 局部变量 | （栈中） | （栈中） | （静态数据区） | （静态数据区） |

# 补充：static与extern：

Static: 的作用有三：一隐藏，二将数据放到静态存储区，三初始化为0.

对函数：改变函数的作用域，将函数的作用域限制在定义函数的文件中。

对变量：

全局变量：改变全局变量的作用域，将全局变量的作用域限制在定义它的文件中。

局部变量：将局部变量存放在静态存储区，所以扩展了局部变量的生存周期。

Extern：extern可置于变量或者函数前，以表示变量或者函数的定义在别的文件中，提示[编译器](http://baike.baidu.com/view/487018.htm)遇到此变量或函数时，在其它模块中寻找其定义。另外，extern也可用来进行链接指定。

函数默认是Extern的，变量前加上Extern构成了变量的声明。函数的声明由函数的头部直接加’;’构成。 Extern “C”、 C++语言在编译的时候为了解决函数的多态问题，会将函数名和参数联合起来生成一个中间的函数名称，而[C语言](http://baike.baidu.com/view/1219.htm)则不会，因此会造成链接时找不到对应函数的情况，此时C函数就需要用extern “C”进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。这样就能在C++程序中使用C的函数。

# 关于函数的返回：

返回值类型是int 、 char等类型用寄存器（多用AX,DX）返回；浮点数的返回比较复杂（对应的汇编用int 39指令）；结构体作返回值时，先将要返回的结构体的内容拷贝到外部空间（不是栈中）中，再用AX返回该外部空间的地址。数组不能作函数返回值。

关于函数的参数传递：

函数参数传递只是简单的值拷贝；形参与实参是完全独立的，函数参数可以作为输入，也可以作为输出。我想说的是当用函数参数作为输出时，必须能对它进行间接访问操作。如:若想用data接收函数的输出，那么传递给函数的必须是data的地址。 如果直接传递data，因为形参与实参是独立的，改变形参无法达到改变实参的目的。当以&data为实参时，此时形参也应为一个指针，此时虽然改变形参也不会改变&data和data（形参与实参是独立的），但是因为形参是一个指针，此时我们可以进行间接访问操作，所以我们可以通过间接访问操作修改形参所指向的空间的内容。

简言之：函数参数传递只是简单的值拷贝；形参与实参是完全独立的，它们各自占有不同的内存，要想通过函数参数使函数的运行能对外部（相对于函数而言）产生影响那么这个函数参数一定能进行间接访问操作。

数组名作为函数参数时自动退化为一个指针。

如：int func(char a[]) 🡸等价于🡺 int func(char \*a)

int func(char a[][N]) 🡸等价于🡺 int func(char (\*a)[])

# 关于如何生成一个可执行文件:

预处理 🡺 编译 🡺 链接

预处理：对程序中的预编译指令进行处理。如将#include的文件包含进来。生成预处理文件。

编译：以单个的源文件（预处理生成的预处理文件）为单位进行编译，生成目标文件。编译时就将这个文件中的所有变量，函数分配空间，将各个函数编译成二进制码，按照特定目标文件格式生成目标文件，在这种格式的目标文件中进行各个全局变量，函数的符号描述，将这些二进制码按照一定的标准组织成一个目标文件。（编译器的目光比较短，对于它编译的文件中的变量或函数，可以只给出声明，不给出定义；这时编译器会假设这些变量或函数是在其它文件中定义的， 即使在其它文件中没有定义，它也不会报错。又因为它的编译是以单个文件为单位的，所以它只能发现单个文件内出现的重定义等错误，而无法发现不同文件中的重定义错误。如一个工程中有两个文件，a文件和b文件：若a中定义了两个x,编译时能发现a文件出现重定义x;但a文件中定义了y，b文件中也定义了y，编译时无法发现工程中的重定义）

链接：将编译生成的各个目标文件与其他的一些库文件，根据一些参数，连接生成最终的可执行文件，主要的工作就是重定位各个目标文件的函数，变量等，相当于将个目标文件中的二进制码按一定的规范合到一个文件中。（编译器的目光比编译器远，它能发现不同文件中出现的重定义等问题）

补充：#include “filename”预处理命令实现的是文件的包含，它将整个filename的内容搬到#include “filename”处。

代码块以文件（.obj）为单位加入到可执行文件中，如果程序中调用了filename.obj中的一个函数，那么会将整个filename.obj的内容全部加入到可执行文件中。

# 关于指针：

指针即是地址，地址即是指针。

指针变量是一个变量，但这个变量存放的值是一个地址。作为变量它有自己的一块内存，作为指针变量它的值是被当作为一个地址（即指针变量所指向的内存的位置）。

一个变量具有两个重要属性：一是变量自身的地址，二是变量的类型。

一个指针变量有四个重要属性：一是它自身的类型，二是它指向的类型，三是它的值（即它所指向的内存区），四是它自身占的内存空间。

指针变量的类型：只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。

例：

(1)int \*ptr; //指针的类型是int \*   
(2)char \*ptr; //指针的类型是char \*   
(3)int \*\*ptr; //指针的类型是 int \*\*   
(4)int (\*ptr)[3]; //指针的类型是 int(\*)[3]   
(5)int \*(\*ptr)[4]; //指针的类型是 int \*(\*)[4] 

指针变量指向的类型：当你通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。从语法上看，你只须把指针声明语句中的指针名字和名字左边的指针声明符 \* 去掉，剩下的就是指针所指向的类型。指针变量的加n运算(p +=n;)表示指针往后移n个单元（以指针指向的类型为单元）；如p++,表示指针指向下一个单元。 指针变量指向的类型决定了指针进行运算时的单位量。如： 定义一个指向type类型的指向p, type \*p;那么 p++🡸等价于🡺p的值加上sizeof(type)。

例如：   
(1)int \*ptr; //指针所指向的类型是int   
(2)char \*ptr; //指针所指向的的类型是char   
(3)int \*\*ptr; //指针所指向的的类型是 int \*   
(4)int (\*ptr)[3]; //指针所指向的的类型是 int()[3]   
(5)int \*(\*ptr)[4]; //指针所指向的的类型是 int \*()[4] 

指针变量的值（指针变量所指向的内存区）：指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。指针所指向的内存区就是从指针的值所代表的那个内存地址开始，长度为sizeof(指针所指向的类型)的一片内存区。以后，我们说一个指针的值是XX，就相当于说该指针指向了以XX为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。

指针变量自身占的内存区：在同一个平台在所有指针变量所占据的长度是一样的。它的长度只与平台有关。指针本身占据的内存这个概念在判断一个指针表达式是否是左值时很有用。

指针所指向的内存区和指针所指向的类型是两个完全不同的概念。指针所指向的类型在定义指针时就已经确定了，但指针所指向的内存由定义是无法确定的。如int \*ptr; 指针所指向的类型已经确定了，它指向int类型，但它却没有指定它所指向的内存（或者说它所指向的内存是无意义的）。以后，每遇到一个指针，都应该问问：这个指针的类型是什么？指针指向的类型是什么？该指针指向了哪里？

对指针变量的间接访问运算“\*”：指针变量的意义就在于可以进行间接访问运算。对指针变量的间接访问运算使得指针功能强大，同时也使得指针看起来更加复杂。在需要通过函数参数来修改函数外部的一些变量时，指针变量的间接访问操作便发挥了不可代替的作用。

对变量取地址运算“&”与指针变量的比较：对变量进行取地址运算将得到一个指针（地址），但它是一个指针常量，这个指针常量不占据存储空间，它不能作为左值。这个指针常量所指向的类型是取地址的变量的类型。而指针变量是占据存储空间的，它可以作为一个左值。

如：int a; int \*b; b = &a; a是一个int 型变量。b是一个int \*型的指针变量，它指向的类型是int型。&a是一个指针常量，它的类型是int \*，它所指向的类型是int。因为b是一个指针变量所以它可以作为左值：如b = b + 2是正确的，因为&a是一个指针常量，所以它不能作为左值，如&a = &a +2是错误的，不能对它进行赋值。

指针的运算：

指针的运算是以它所指向的类型长度为单位的。若有Type \*p，则p+n也是一个指针，它的类型与p相同，它指向的类型也与p所指向的类型相同。P+n指向的位置为p指向位置的后n个单元（以指针指向的类型type为单元）。p+n🡸等价于🡺p的值 + n \* sizeof(它所指向的类型type)。

[ ]运算符：

若p是一个指针，p[n]的意义等同于\*(p + n)。

可以这么看[]运算符：[ ]的作用是取出指针p所指向的位置偏移n个单元处的内容。P[n]将指针p指向的位置后移n个单位，再取出它所指向的内容。值得注意的是：指针p还是指向原来的位置，这里说的移动指针是通过一个中间量实现的，并没有改变p。

# 关于typedef:

typedef为C语言的关键字，作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（int,char等）和自定义的数据类型（struct等）。

我的理解：在定义变量的语句前加上typedef关键字，将变量抽象为它对应的类型。

如: int INT;对应了一个名为INT的int 型变量，

typedef int INT，声明了一个名为INT的类型，它与*int*是同一类型。

再如：用typedef 为复杂结构取别名时的理解

typedef struct LNode

{

int data;

struct LNode \*next;

}LNODE,\*LinkList;

没有typedef 时，表示创建了

一个struct Lnode类型的结构体，

并定义了一个该结构体类型的变量LNODE,

和一个指向该结构体类型的指针LinkList，LinkList的类型为struct LNode \*。

因为加了typedef 所以LNODE被抽象为它对应的类型即struct Lnode结构体类型。LinkList被抽象为它所对应的类型即struct LNode \*。此时LNODE、LinkList表示的是一种类型也不再是变量。

Struct LNode

{

int data;

struct LNode \*next;

};

typedef LNODE,\*LinkList;

相当于

用途：

○1定义一种类型的别名，而不只是简单的宏替换。可以用作同时声明指针型的多个对象。比如：我们想定义两个指向字符变量的指针

#define CH char\*

CH pa, pb; //等价于char \*pa, pb不符合我们的意图，它只声明了一个指向字符变量的指针，和一个字符变量；

以下则可行：

typedef char\* PCHAR; // 一般用大写

PCHAR pa, pb; // 可行，同时声明了两个指向字符变量的指针

○2用在旧的C的代码中（具体多旧没有查），帮助struct。以前的代码中，声明struct新对象时，必须要带上struct，即形式为： struct 结构名 对象名，如：

struct tagPOINT1

{

int x;

int y;

};

定义一个struct tagPOINT1类型的结构体变量需要写成struct tagPOINT1 p1;

typedef struct tagPOINT

{

int x;

int y;

}POINT;

POINT p1; // 这样就比原来的方式少写了一个struct，比较省事，尤其在大量使用的时候

C++中即使没有用typedef取别名也可以不写struct。

○3用typedef来定义与平台无关的类型。

比如定义一个叫 REAL 的浮点类型，在目标平台一上，让它表示最高精度的类型为：typedef long double REAL; 在不支持 long double 的平台二上，改为：

typedef double REAL; 在连 double 都不支持的平台三上，改为：typedef float REAL; 也就是说，当跨平台时，只要改下 typedef 本身就行，不用对其他源码做任何修改。标准库就广泛使用了这个技巧，比如size\_t。另外，因为typedef是定义了一种类型的新别名，不是简单的字符串替换，所以它比宏来得稳健（虽然用宏有时也可以完成以上的用途）。

○4为复杂的声明定义一个新的简单的别名。方法是：在原来的声明里逐步用别名替换掉表达式中结合顺序靠后的部分（不必一步一步的替换，对于容易理解的地方，可以一次替换掉几次结合的部分），如此循环，把带变量名的部分留到最后替换，得到的就是原声明的最简化版。举例：

1. 原声明：int \*(\*a[5])(int, char\*);

变量名为a，直接用一个新别名pFun替换a就可以了：

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

原声明的最简化版：

pFun a[5];

2. 原声明：void (\*b[10]) (void (\*)(void));

变量b先与“[10]”结合成b[10]，说明b是一个有10个元素的数组，再与左边的“\*”结合成\*b[10]，说明数组b的元素是一个指针，再与void (。。。) (void (\*)(void))结合，说明元素是一个指向函数的指针。最后还有一个void (\*)(void)表示的是函数指针所指向的函数的参数又是一个函数指针。

变量名为b，先替换结合最晚的部分，pFunParam为别名一：

typedef void (\*pFunParam)(void);

再替换结合稍早一点的，pFunx为别名二：

typedef void (\*pFunx)(pFunParam);

此时原声明的可以写为：pFunx b[10];

也可以继续再写简单：

typedef pFunx\_Array pFunx[10];

此时原声明的可以写为：pFunx\_Array b;

3. 原声明：double (\*(\*e)[9]) (void);;

变量名为e，先替换最后结合的部分，pFuny为别名一：

typedef double(\*pFuny)(void);

再替换稍早一点结合的部分，pFunParamy为别名二

typedef pFuny (\*pFunParamy)[9];

原声明的最简化版：

pFunParamy e;

陷阱

记住，typedef是定义了一种类型的新别名，不同于宏，它不是简单的字符串替换。比如：  
先定义：  
typedef char\* PSTR;  
然后：  
int mystrcmp(const PSTR, const PSTR);

const PSTR实际上相当于const char\*吗？不是的，它实际上相当于char\* const。  
原因在于const给予了整个指针本身以常量性，也就是形成了常量指针char\* const。  
简单来说，记住当const和typedef一起出现时，typedef不会是简单的字符串替换就行。

typedef在语法上是一个[存储](http://www.storworld.com/)类的关键字（如auto、extern、mutable、static、register等一样），虽然它并不真正影响对象的[存储](http://www.storworld.com/)特性，如：  
typedef static int INT2; //不可行  
编译将失败，会提示“指定了一个以上的存储类”。

typedef 与 #define的区别

案例一：

通常讲，typedef要比#define要好，特别是在有指针的场合。请看例子：

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，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。

案例二：

下面的代码中编译器会报一个错误，你知道是哪个语句错了吗？

typedef char \* pStr;

char string[4] = "abc";

const char \*p1 = string;

const pStr p2 = string;

p1++;

p2++;

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。上述代码中const pStr p2并不等于const char \* p2。const pStr p2和const long x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。

# 关于结构体：

我们可以通过定义结构体类型来实现对描述某一对象的多个数据的封装。

结构体类型的声明与结构体变量的定义：

一：先声明结构体类型，再定义该类型的变量（或指向该类型的指针）。

struct 结构体名

{

类型1 成员变量名1;

类型2 成员变量名2;

类型3 成员变量名3;

......

......

......

};

用这种方式声明了一个

“struct 结构体名” 的结构体类型。

“struct 结构体名 ”是一个整体，不能

分开。

用这种方式声明的结构体：在定义该类型的结构体变量时，如下

struct 结构体名 变量名1, 变量名2;

struct不能省略，即不能写成

结构体名 变量名; -------注在C++中可以这么写

二：声明结构体类型的同时定义该类型的变量或指向该类型的指针。

struct 结构体名

{

类型1 成员变量名1;

类型2 成员变量名2;

类型3 成员变量名3;

......

......

......

}结构体变量列表;

用这种方式声明了一个

“struct 结构体名” 的结构体类型。

“struct 结构体名” 是一个整体，

不能分开。

在定义一个“struct 结构体名”类型的结构体时也定义了“结构体变量列表”中的结构体变量。

这里：结构体名可以省略，不过省略

后将不能在其他地方再定义该结构体

类型的变量。

没有省略结构体名时，可以在其他地址定义该类型的结构体。方式如下：（省略结构体名时将不能在其他地方进行定义）

struct 结构体名 变量名1, 变量名2;

struct不能省略，即不能写成

结构体名 变量名; -------注在C++中可以这么写

三：用typedef为结构体类型取一个别名。

结构体变量的初始化与引用：

定义该结构体类型的变量：

结构体类型名 变量名1，变量名2;

这里：还可以用如下方式来定义该结构体类型的变量：

struct 结构体名 变量名1, 变量名2;

但是：用Typede 为结构体类型取别名时结构体名也可以省略，不过如果省略结构体名将无法在其他地方用“struct 结构体名 变量名1, 变量名2;”的方式定义变量。

Typedef struct 结构体名

{

类型1 成员变量名1;

类型2 成员变量名2;

类型3 成员变量名3;

......

......

......

}结构体类型名;

用这种方式声明了一个

“struct 结构体名” 的结构体类型。

并为该类型取了一个别名“结构体

类型名”。在其他地方可以直接用

“结构体类型名 变量名” 的方式来定

义变量。也可以用“struct 结构体名

变量名”的方式来定义变量

在定义结构体变量时可以对它们进行初始化。初始化列表是用花括号括起来的用逗号分隔的一些常量，这些常量依次赋给结构体变量中的成员，未被指定初始化的：数值型成员被系统初始化为0，字符型成员被初始化为’\0’，指针型成员被系统初始化为NULL。

通过结构体变量名来引用结构体成员的方式： 结构体变量名.成员名。

通过结构体指针变量来引用结构体成员的方式：

结构体指针变量->成员名。

(\*结构体指针变量).成员名 --- 这种方式相当于通过结构体变量来引用结构体成员。

结构体的成员可以属于另一个结构体类型（即结构体可以嵌套声明），对结构体变量的成员可以像普通变量一样进行各种运算。结构体的成员名可以与程序中的变量名相同，因为结构体的成员名的作用域只在结构体中。

结构体变量名与数组变量名不同，数组变量名表示指向该数组的第0个元素的指针常量（数组变量名是一个值为&array[0]的指针常量），结构体变量名不表示指向结构体的指针。要得到结构体的地址需要对它进行取地址运算（&）。----简言之：数组变量名表示的是指向该数组的第一个元素的指针常量（大多数情况下是这样的），但结构体变量名表示的是结构体变量本身。

同类的结构体变量之间可以互相赋值。数组变量却不能。结构体变量可以作为参数返回值，数组却不能。结构体变量作函数参数时编译仍把它当作一个结构体变量，但数组变量作函数形式参数时，编译器把它当作一个指针处理，当数组变量作函数实参时，表示的是指向第0个元素的指针。（关于数组在以后再详细讲，这里先讲这么多）

结构体指针，结构体数组：

结构体指针（变量）：即是指向结构体的指针。

结构体数组：即是一个元素为结构体类型的数组。

# 关于指针与数组：

**一：指针与数组是不同的（本质不同<数组和指针属于两种不同的类型>）**

**数组对元素的访问是直接访问，而指针是间接访问。**

通过C与对应的汇编来分析对数组下标的引用，与指针对下标的引用

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

int \*p =a;

int c=0;

int main()

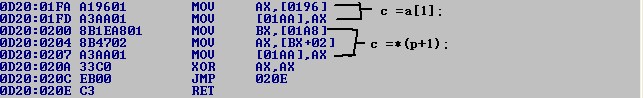
{

c =a[1];

c =\*(p+1);

return 0;

}



对汇编的分析可以看到数组对元素的访问是直接访问，而指针是间接访问

关于数组与指针的一些问题：

问题1. 数组与指针都可以用字符串常量来初始化，但是他们的底层实现一样吗？

先分析：对于数组、编译器为它分配的空间为它的基类型\*元素个数，数组的每个元素可以看作一个变量，它有自己的地址。所以char a[10]="c yu yan"，可以看作是a[0] =’c’,a[1]=’ ‘,…的简写。与变量初始化类似，所以其内容应是可以更改的。char a[10]="c yu yan"对应二个内存的分配，常量字符串"c yu yan"编译期分配空间于静态存储区，用"c yu yan"初始化的数组的空间位于栈上或数据区(从静态存储区拷贝"c yu yan"到数组空间)。

对于指针初始化语句char \*p ="hello word";编译器分配二个内存的空间，一个是指针本身占的内存，二个是"hello word"占的空间位于静态存储区，并使指针指向静态存储区。（只是将指针指向一个常量区，而没有重新开辟一个可操作的空间使指针指向）

用如下程序测试：

char a[10]="c yu yan";

char \*p ="hello word";

int main()

{

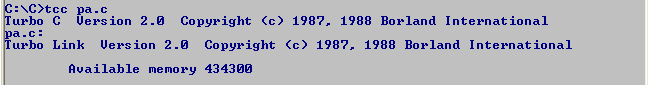
a[0] ='X';

\*p ='Y';

return 0;

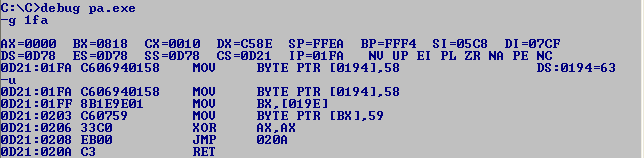
}

在TCC下：编译链接通过，运行也不显示错误。

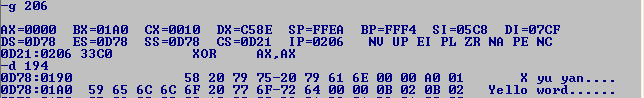




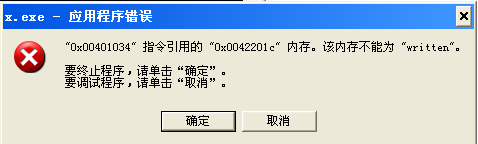
对应的汇编如下：



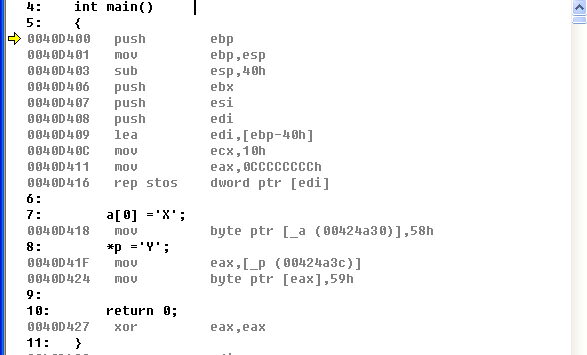




在vc6.0下再测试 编译链接均能通过，但运行时显示错误，如下



点击取消进行调试





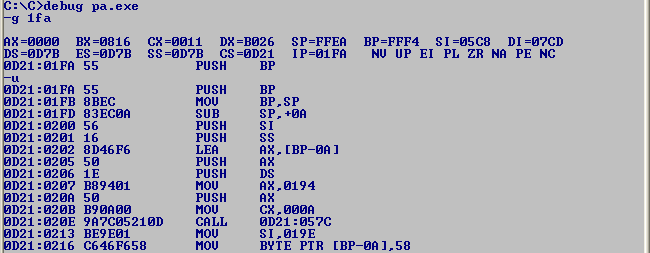
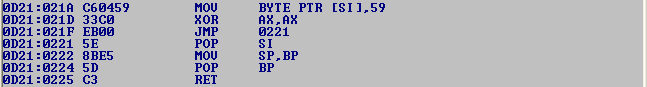
单步执行\*p ='Y';时出现如下错误



通过相应的汇编，我发现前面的分析中char a[10]="c yu yan"对应二次内存的分配，常量字符串"c yu yan"编译期分配空间于静态存储区，用"c yu yan"初始化的数组的空间位于栈上或数据区(从静态存储区拷贝"c yu yan"到数组空间)。与上面的验证不相符。----暂不总结它的对错。

将程序中的 数组与指针 定义为局部再次验证一个结果。

Tc下:

此时char a[10]="c yu yan"确实对应二次内存的分配。

字符串常量"c yu yan"



局部数组a[10]

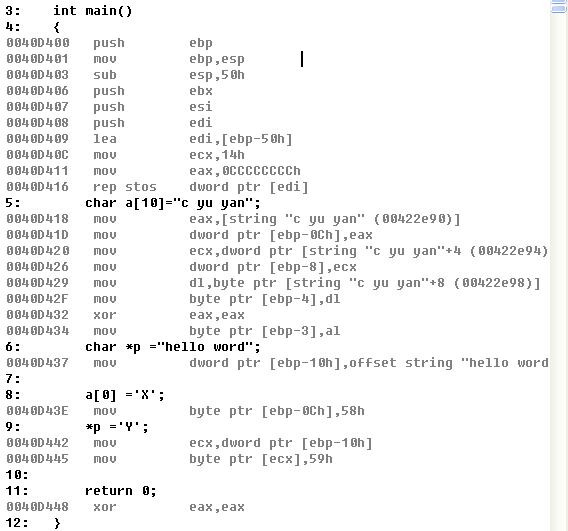


并用通过scopy函数（串传送）来实现初始化。初始化后



而对char \*p ="hello word"空间的分配和分析一致。

Vc6.0下



可以清楚的看出char a[10]="c yu yan";与char \*p ="hello word";在底层的区别。

调试到单步执行\*p ='Y';时出现下而所显的错误。



总结：对于数组编译器为它分配的空间为它的基类型\*元素个数，数组的内容是可以更改的。数组初始化语句char a[10]="c yu yan"对应二次内存的分配，常量字符串"c yu yan"编译期分配空间于静态存储区，用"c yu yan"初始化的数组的空间位于栈上或数据区(从静态存储区拷贝"c yu yan"到数组空间)。

对于指针初始化语句char \*p ="hello word";指针pr指向常量字符串"hello word"(位于静态存储区，内容为hello wor\0,<编译器为常量字符串提供只读内存区>)，常量字符串的内容是不可改变的。从语法上看，编译器并不觉得语句\*p ='Y';有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

补充：这个结论因编译器不同而有所差异。在上面tc下，char \*p ="hello word"，p指向的内容是可以修改的。（\*p ='Y';运行时没有出现错误）我的理解是tc比较古老，它可能不提供专门存放常量的只读内存区。

｛

拓展：

编写如下程序来验证关于只读内存区。

char const b[10]="hao";

char \*p;

int main()

{

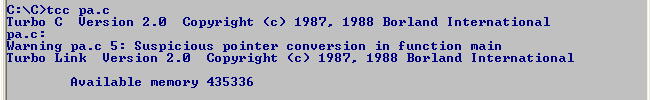
p =&b[0];

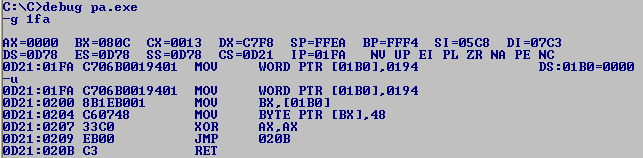
\*p ='H';

return 0;

}

在tc下，编译链接虽然出现警告但是能正常运行。

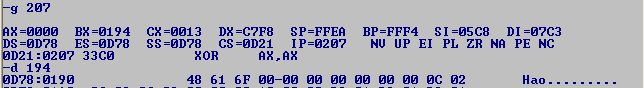




执行\*p ='H';前

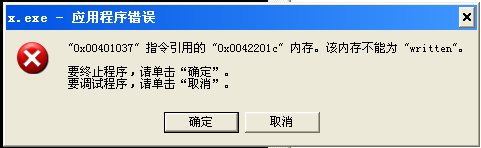


执行\*p ='H';后



内容被改变。（经验证若语句\*p ='H';改为b[0]=‘H’将无法通过编译）

在vc6.0下，编译出现如下错误



调试



（发现与char \*p ="hello word";语句中的"hello word"在同一内存区）

单步执行\*p ='H';出现如下错误：



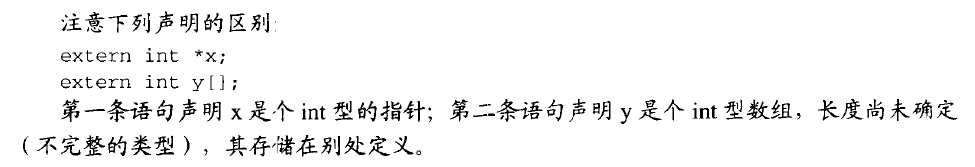
说明：tc下：虽然有const型全局变量，但没有只读内存区与它对应。

Vc6.0下：const型全局变量有对应的只读内存区。

｝

问题二：声明语句extern char p[]; 与extern char \*p;的区别。

经常听到一种说法就是“数组和指针是相同的”。这是非常危险的说法，并不完全正确。



在文件x-1.c中定义一个全局数组

char p[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

在文件x.c中声明为extern char \*p;并使用

extern char \*p;

char c =0;

int main()

{

c =\*p;

return 0;

}

猜想：执行c =\*p;后c的值为多少？

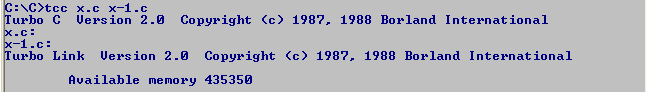
分析：若“数组和指针是相同的”这种说法正确，那么c的值为1。

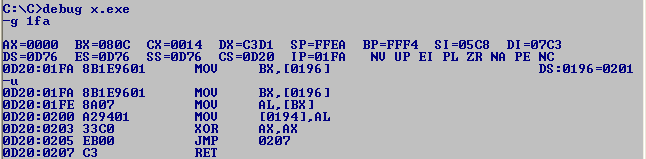
仔细分析，数组与指针的访问方式，（数组是直接访问的，而指针是间接访问的。）就会发现，结果不是2，而是以地址0201的单元的内容。原因如下：指针的访问方式是间接的：通过指针来访问数据的步骤为， 先从指针p<注1>的地址(为&p[0])中将其内容<注2>取出放在XX中，再取出地址XX处的内容放在YY中。<注意：这里涉及到了CPU的内存模式。大端模式还是小端模式，这里不作说明。。。> 。

注1：因为它被“错误的”声明为一个指针，所以编译器将它当作指针操作，而不是数组<虽然它“本应该“是数组>。

注2：内容被解释为一个地址，占2个字节<tc下>，对应原基类型为char的数组的第1和第0个元素，所以为0201。

TC下：





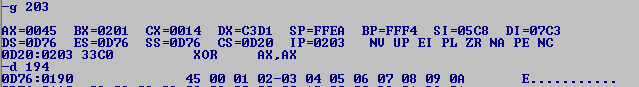
数组存放的位置及内容



地址0201处内容为45



执行c =\*p后，c值与地址201的值相同，为45



总结：对数组的声明与定义应当匹配。不要觉得“数组和指针是相同的”，它们“相同”是有条件的，无视它们“相同”的条件，将导致错误。（数组在作函数参数时于指针是等同的。。）

三：内容的复制与比较

char a[10] ="hello";

char b[10];

char \*p;

int len;

int main()

{

scopy(b, a); /\* 不能用b =a; \*/

if(strcmp(b, a) == 0) /\* 不能用if(b ==a) \*/

{

b[0] ='X';

}

len =strlen(a);

p =(char \*)malloc(sizeof(char)\*(len +1));

strcpy(p, a),/\* 不要用 p= a \*/

if(strcmp(p, a) == 0) /\* 不要用if(p ==a) \*/

{

p[1] ='Y';

}

return 0;

}

不能对数组名进行直接的复制与比较。若想把数组a的内容复制给粗线条组b，不能用语句b=a，否则将产生编译错误。应该用标准库函数strcpy进行复制。同理，比较b和a的内容是否相同，不能用if(b==a)来判断应该用标准库函数strcnp进行比较。

语句p=a并不能把a的内容复制给指针p，而是把a的地赋给了p,要想复制a的内容，可以先用库函数malloc为申请一块容量为strlen(a)+1个字符的内存，再用strcpy进行字符串复制。同理if(p==a)比较的不是内容而是地址，应该用库函数strcmp来比较。

四：计算内存容量

int a[10]={1, 2, 3};

char \*p;

int main()

{

p =(char \*)malloc(20);

printf("a[10]-%d\n", sizeof(a));

printf("p-----%d\n", sizeof(p));

return 0;

}



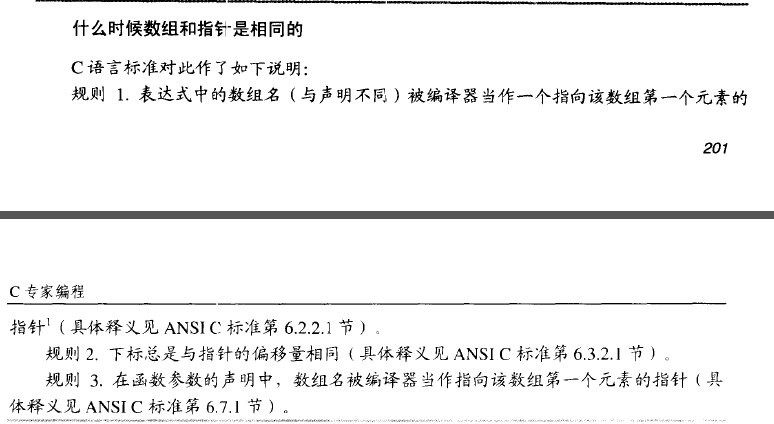
用sizeof(指针)只能得到指针本身占的空间大小，而无法得到它所指向的空间大小。

用sizeof(数组名)能得到整个数组的空间大小。

数组与指针的一些其他不同之处：

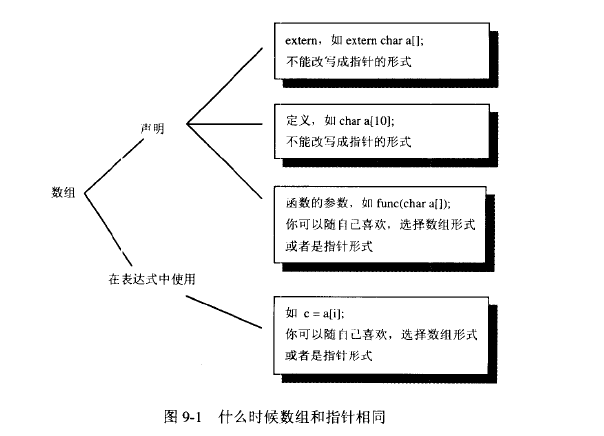
|  |  |
| --- | --- |
| 数组和指针的特点 | |
| 数组 | 指针 |
| 保存数据 | 保存地址 |
| 直接访问数据 | 间接访问数据，先取得指针的内容，然后以它为地址，取得数据 |
| 用于存储数目固定且类型相同的数据 | 通常用于动态数据结构 |
| 由编译器自动分配和删除 | 动态的分配和删除，相关函数为malloc()和free() |
| 自身即为数据名 | 通常指向隐式数据 |

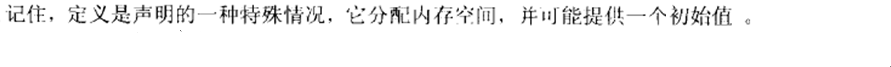
二：什么时候数组与指针可以通用（条件相同）



对于数组与指针相同的情况，我们使用的比较多，这里就不再验证了。

最后总结一下数组与指针什么时候相同，什么时候不同。





# 关于不定参数:

关于可变参数的几个宏：va\_list , va\_start , va\_arg, va\_end。#include <stdarg.h>

     在VC++6.0的include有一个stdarg.h头文件,有如下几个宏定义:

#define \_INTSIZEOF(n) ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )

/\*\_INTSIZEOF(n)宏是为了考虑那些内存地址需要对齐的系统，从宏的名字来应该是跟sizeof(int)对齐。一般的sizeof(int)=4，也就是参数在内存中的地址都为4的倍数。比如，如果sizeof(n)在1－4之间，那么\_INTSIZEOF(n)＝4；如果sizeof(n)在5－8之间，那么\_INTSIZEOF(n)=8。\*/

     #define va\_start(ap,v) ( ap = (va\_list)&v + \_INTSIZEOF(v) ) //ap指向变量v的下一个参数。 V相当于是一个基准，指针ap以参量v为基准来指向它的下一个参数 ; va\_list被定义成char\*型。

     #define va\_arg(ap,t) ( \*(t \*)((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) ) //使ap指向了下一个参数，但整个表达式对应的是当前参数的值（当前参数：ap未向后移时所指向的参数）（t为当前参数的类型）。

注意 (ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t) 的优先级。

#define va\_end(ap) ( ap = (va\_list)0 ) // 将指针置为无效

va\_arg(ap,type)宏 :使 ap指向了下一个参数，但整个宏表达式的类型为type，宏。

va\_arg(ap,type)宏的常用用法：

va\_list ptr; //定义一个va\_list 型变量 等价于char \*ptr;

type ty ; //定义一个type型变量ty。

ty = va\_arg(ptr, type);

其中ty 为type类型，ptr为一个va\_list型变量(或者char\*变量)。

作用： ty = va\_arg(ptr, type); 使指针ptr往后移\_INTSIZEOF(type)个字节（即使ptr指向下一个单元）, 并将ptr未移动前所指向的内容(为type型)赋给ty。

举个例子

编写一个函数showstr(char \* const str, …)打印多个

字符串，它可以接收不定数量的char\*型参数，最后一

个参数传入0,showst通过此识别何时停止打印。

调用showstr在屏幕上显示"This is a c program.";

"Welcome to c!";"Hello world!";

void showstr(char \*str, ...);

char \*p1 = "This is a c program.";

char \*p2 = "Welcome to c!";

char \*p3 = "Hello world!";

int main(void)

{

showstr(p1, p2, p3, 0);

return 0;

}

#include <stdarg.h>

void showstr(char \* const str, ...)

{

va\_list ap;

char \*p;

va\_start(ap, str);

while (str != 0)

{

printf("%s\n", str);

p = va\_arg(ap, char \*);

}

va\_end(ap);

}

# 关于函数指针：

函数指针的定义： type1 (\*fun\_p)(type2, tpye3, …, typeN)。定义了一个名为fun\_p的函数指针，它指向的函数的返回值类型为type1,函数的参数列表对应的类型为type2, tpye3, … , typeN 。

或者使用typedef声明一个函数指针类型，再用声明的类型来定义该类型的指针。如下：

typedef type1 (\*TYPE\_ FUNC\_P)(type2, tpye3, …, typeN)声明了一个函数指针类型，该函数指针类型的叫TYPE\_ FUNC\_P, 该类型的函数指针指向的函数具有如下特征：函数的返回值的类型是type1，函数的参数列表对应的类型为type2, tpye3, … , typeN。

TYPE\_ FUNC\_P fun\_p; 定义了一个TYPE\_ FUNC\_P类型的变量。

函数指针可以作为函数参数也可以作为函数返回值。函数指针作用函数参数时可以实现回调（不是很了解，以后再补充）。

函数指针与结构体一起使用时可以实现数据的封装，通过函数指针来操作结构体的成员，从而对外部而言隐藏了结构体（对象）的内部细节。将函数指针与数据一起封装在结构体中（即将数据与对数据的操作封装起来）。C语言的结构体中不能定义函数，但可以定义指针（包括函数指针）。

# 关于内存中的数据与指令

在内存中数据与指令均是以二进制的形式存储的，本质上并没有什么区别。那么CPU怎么区分数据和指令呢（或者说CPU把哪些二进制的数据看作指令呢）？ 这里有一个相当于指令指示器的东西（如8086CPU中的CS:IP）这个指示器所指向的内容将被当作指令执行。

# 关于内存空间的几个段:

（还需进一步了解）

数据段：数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

BSS段：BSS段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称。BSS段属于静态内存分配。

代码段：代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。

这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读, 某些架构也允许代码段为可写，即允许修改程序。

在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。代码段是存放了程序代码的数据，

假如机器中有数个进程运行相同的一个程序，那么它们就可以使用同一个代码段。

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；

当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）

栈(stack)：栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

(1)内存分段和内存分页一样都是一种内存管理技术,分段：权限保护,分页：虚拟内存.

(2)分段后,程序员可以定义自己的段,各段有独立的地址空间,象进程的地址空间互相独立一样.

(3)同一个类的实例分配在一个段中,只有该类的方法可以访问,如果其他类的方法去访问,会因为段保护而出错.可以从硬件上实现类的数据保护和隐藏。

在进程被载入内存中时，基本上被分裂成许多小的节（section）。我们比较关注的是6个主要的节：

（1） .text 节

.text 节基本上相当于二进制可执行文件的.text部分，它包含了完成程序任务的机器指令。

该节标记为只读，如果发生写操作，会造成segmentation fault。在进程最初被加载到内存中开始，该节的大小就被固定。

（2）.data 节

.data节用来存储初始化过的变量，如：int a =0 ; 该节的大小在运行时固定的。

（3）.bss 节

栈下节（below stack section ,即.bss）用来存储为未初始化的变量，如：int a; 该节的大小在运行时固定的。

（4） 堆节

堆节（heap section）用来存储动态分配的变量，位置从内存的低地址向高地址增长。内存的分配和释放通过malloc() 和 free() 函数控制。

（5） 栈节

栈节（stack section）用来跟踪函数调用（可能是递归的），在大多数系统上从内存的高地址向低地址增长。同时，栈这种增长方式，导致了缓冲区溢出的可能性。

（6）环境/参数节

环境/参数节（environment/arguments section）用来存储系统环境变量的一份复制文件，进程在运行时可能需要。例如，运行中的进程，可以通过环境变量来访问路径、shell 名称、主机名等信息。

该节是可写的，因此在格式串（format string）和缓冲区溢出（buffer overflow）中都可以使用该节。

另外，命令行参数也保持在该区域中。

# 关于函数封装：

利用函数指针或函数指针数组对具有同一类接口的函数进行调用。

# 关于C语言中的文件结构：

每个C功能程序通常分为两个文件，一个文件用于保存程序的声明，称为头文件。另一个文件用于保存程序的实现，称为定义文件。

我们将某一类特定功能的函数或变量的定义和声明放在一对.c和.h文件中。

头文件的一些规则：

○1为了防止头文件被重复引用，应当用ifndef/define/endif结构产生预处理块。

○2用#include <filename.h>的格式来引用标准库的头文件(编译器将从标准库开始搜索)

○3用用#include “filename.h”的格式来引用非标准库的头文件(编译器将从用户的工作目录开始搜索)

对写头文件的建议：

（1）头文件中只存放“声明”而不存放“定义”。

（2）不提倡使用全局变量。

头文件的作用：

○1 通过头文件来调用库功能。在很多场合，源代码不便向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口是怎么实现的。编译器会从库中提取相应的代码。

○2 使用头文件能提高效率，减少重复包含大量声明的代码输入。

○3头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致时，编译器就会指出错误。

程序中的条件编译：

（1） 使用 #include “头文件” 会增加程序的大小（一个头文件中往往有多个声明，当我们只需要使用其中的某几个时，使用#include “头文件”会将整个文件加入进来，很可能将我们不需要的声明也加入其中），那么为什么还要使用它。

回答: 虽然使用#include “头文件”会加入一些不必要的代码，但是它能提高效率，减少重复包含大量声明时的代码编辑量。

（2） 条件编译 ifndef/define/endif 。。。的作用。

回答:

使用条件编译 可以防止重复包含头文件。

例如：a.c中包含了x.h和b.h头文件，而b.h中又包含了x.h头文件。这时如果x.h中没有使用条件编译，那么在a.c中就将会包含两份x.h头文件，从而增加了程序的大小。如果使用了条件编译就可以防止重复包含。

条件编译顾名思义是根据条件进行编译，所以的另一个重要作用是根据条件选择编译的内容。如：在不同的平台下，对一个功能的实现有所不同，那么可以在程序中定义不同的宏，再根据这些宏选择要编译的内容。

（3）为什么用了ifndef/define/endif 还要将 头文件与定义文件分开。

用了ifndef/define/endif可不可以将定义放到头文件中。

Answer: 因为用条件编译来防止重复包含头文件是在编译阶段进行的，它只能防止在一个文件中重复包含头文件，而不能防止在一个项目中重复包含头文件（一个项目可能包含多个文件，虽然每个文件不会重复包含某一头文件，但是在一个项目中可能会重复包含某一个头文件），这时如果将定义放在了头文件中，那么链接时就会出现重定义错误。

# 关于结构体实现对象的封装：

函数指针与结构体一起使用时可以实现数据的封装，通过函数指针来操作结构体的成员，从而对外部而言隐藏了结构体（对象）的内部细节。将函数指针与数据一起封装在结构体中（即将数据与对数据的操作封装起来）。

C语言中结构体中不能定义函数成员，但是可以定义函数指针成员。所以可以通过函数指针成员来实现对数据的操作，从而隐藏结构体数据成员。

# 关于DOS向main函数传递参数：

http://blog.chinaunix.net/uid-27164526-id-3274872.html

每当你运行一个DOS或Windows程序时，都会生成一个程序段前缀(Program SegmentPrefix，简称PSP)。当DOS程序的装入程序把程序复制到RAM中来执行时，它先把256个字节分配给PSP，然后把可执行代码复制到紧接着PSP的内存区域中。PSP中包含了DOS为了执行一个程序所需要的各种各样的信息，其中的一部分数据就是命令行。PSP中偏移量为128的那个字节中存放着命令行中的字符个数，接下来的127个字节中存放着命令行本身。这也正是DOS把你能在其提示行中输入的字符个数限制在127个之内的原因——因为它为命令行分配的存储空间只有那么多。遗憾的是，PSP的命令行缓冲区中并没有存放可执行程序的名字——而只存放着在可执行程序名后键入的字符(包括空格符)。例如，如果你在DOS提示行中键入以下命令：  
    XCOPY AUTOEXEC.BAT  AUTOEXEC.BAK

假设XCOPY．EXE存放在c驱动器的DOS目录下，则XCOPY．EXE的PSP命令行缓冲区中将包含以下信息：

    AUTOEXEC.BAT  AUTOEXEC.BAK

注意，命令行中紧接着"XCOPY"的空格符也被复制到PSP的缓冲区中。

除了不能在PSP中找到可执行程序名外，PSP还有一个不足之处——在命令行中能看到的对于输出或输入的重定向，在PSP的命令行缓冲区中是无法看到的，也就是说，你无法从PSP中得知你的程序是否被重定向过。

到现在为止，你应该熟悉在C程序中可以通过argc和argv来获取一些有关信息，（main函数一头部一般写成main(int argc, char \*argv[]),mian参数的第一个参数argc得到dos向程序传递的参数个数，main函数的第二个参数指向的空间存放了输入的参数信息（一份拷贝的参数），程序名和程序的路径作为第0个参数，argv[0]指向的是第0个参数即程序名和路径)但是，这些信息是怎样从DOS的装入程序传给argv指针的呢?这是由程序的启动代码来完成的。启动代码在main()函数的第一行代码之前被执行，在其执行期间，它调用一个名为\_\_setargv()的函数，把程序名和命令行从PSP和DOS环境中复制到main()函数的argv指针所指向的缓冲区中。你可以在xLIBCE．LIB文件中找到\_setargv()函数，对于Small，Medium和Large这三种存储模式，这里的“x”分别为“S”，“M”和“L”。在生成可执行程序时，上述库文件会自动被连接进来。除了复制argv参数的内容外，c的启动代码还要完成其它一些工作。当启动代码执行完毕后，main()函数中的代码就开始执行了。

在DOS中的情况是这样的，那么在Windows中的情况又是怎样的呢?实际上，在Windows中的情况大致上和在DOS中的一样。当执行一个Windows程序时，与DOS的装入程序一样，Windows的装入程序也会建立一个PSP，其中包含了与DOS的PSP中相同的信息。主要的区别是此时命令行被复制到lpszCmdLine参数中，它是WinMain()函数的参数表中的第三个(也是倒数第二个)参数。在Windows C的xLIBCEW．LIB库文件中包含了启动函数setargv()，它负责把命令行信息复制到lpszCmdLine缓冲区中。同样，这里的“x”也表示程序所使用的存储模式。在Quick c中，函数\_setargv()包含在库文件xLIBCEWQ．LIB中。

尽管DOS程序和Windows程序的命令行信息的管理方式基本相同，但是传给你的C程序的命令行的格式在安排上稍有不同。在DOS中，启动代码获得以空格符为分隔符的命令行后，就把每个参数转换为其自身的以NULL为终止符的字符串。因此，你可把argv原型化为一个指针数组(char\* argv[])，并通过从O到n这些下标值来访问每个参数，其中n等于命令行中的参数个数减去1。你也可以把argv原型化为一个指向指针的指针(char \*\*argv)，并通过增减argv的值来访问每一个参数。

在Windows中，传给c程序的命令行是一个LPSTR类型或char\_far\*类型，其中的每一个参数都用空格符隔开，就象你在DOS提示行中键入这些字符后所看到的那样(实际上，在Windows中不可能真正键入这些字符，而是通过双击应用程序图标这样的方式来启动一个程序)。为了访问Windows命令行中的各个参数，你必须人工地访问lpszCmdLine所指向的存储区，并分隔存放在该处的参数，或者使用strtok()这样的函数，每次处理一个参数。

需要注意的是，在Small存储模式下，或者在只有一个代码段的汇编程序中，由DOS返回到BX寄存器中的值就是程序代码段的地址值；在Large模式的c程序中，或者在多个代码段的汇编程序中，所返回的值是程序中包含PSP的那个代码段的段地址值。如果你已经建立了一个指向这个数据的指针，你就可以在程序中使用这个数据了。

# 关于System函数:

windows操作系统下system () 函数详解（主要是在C语言中的应用）　（此处暂不讨论linux下的system函数）

功 能： 发出一个DOS命令

用 法： int system(char \*command);

参数： char \*command：DOS命令

说明：任何命令行下能执行的DOS命令都可以通过system来实现。

# 关于单链表：

单链表的结点包括数据域和指针域。

除第0个结点外、每个结点都有前驱，除最后一个结点外、每个结点都有后继。

设置一个头结点，头结点中一般一用于存储内容，头结点的指针域指向第一个数据结点。

最后一个结点的指针域用NULL表示其后没有元素了。

头结点的编号为0，数据结点从1 开始编号。

单链表通过头指针(head)进行访问，头指针指向头结点。

对一个单链表需要提供的一些基本操作：

1. 创建线性表。
2. 撤消线性表(包括清空所有数据结点、头结点,并撤消头指针)。
3. 清除线性表的数据结点。
4. 追加数据结点。
5. 指定编号插入数据结点。
6. 删除指定编号的数据结点。
7. 取出指定编号结点的数据。
8. 取出指定编号的结点的的地址。
9. 遍历。
10. 创建一个存储结点，并初始化。

在操作结点时，比如插入或者删除一个结点，要先找到它的前驱。通过它的前驱来操作它。

在理解链表时要充分理解前驱与后继的作用。

# 关于面向对象：

（还需进一步了解）