1. 小方带你进阶嵌入式C语言

1.1 C/C++

1.1.1 关键字

1.1.1.1 C语言宏中"#“和”##"的用法

1. **(#)字符串化操作符**

作用：将宏定义中的传入参数名转换成用一对双引号括起来参数名字符串。其只能用于有传入参数的宏定义中，且必须置于宏定义体中的参数名前。

如：

```
#define example( instr ) printf("the input string is:\t%s\n",#instr )
#define example1( instr )#instr
```

当使用该宏定义时：

```
example( abc ); //在编译时将会展开成：printf("the input string is:\t%s\n","abc")
string str = example1( abc );//将会展成：string str="abc"
```

2. **(##)符号连接操作符**

作用：将宏定义的多个形参转换成一个实际参数名。

如：

```
#define exampleNum( n ) num##n
```

使用：

```
int num9 = 9;
int num = exampleNum(9 ); //将会扩展成 int num = num9
```

注意：

a.当用##连接形参时，##前后的空格可有可无。

如：

```
#define exampleNum( n ) num ## n
//相当于#define exampleNum( n ) num##n
```

b.连接后的实际参数名，必须为实际存在的参数名或是编译器已知的宏定义。

c.如果##后的参数本身也是一个宏的话，##会阻止这个宏的展开。

```
#include <stdio.h>
#include <string.h>
```

```
#define STRCPY(a, b) strcpy(a ## _p,#b)
int main()
{
  char var1_p[20];
  char var2_p[30];
  strcpy(var1_p,"aaaa");
  strcpy(var2_p,"bbbb");
  STRCPY(var1, var2);
  STRCPY(var2, var1);
  printf("var1 = %s\n", var1_p);
  printf("var2 = %s\n", var2_p);
  //STRCPY(STRCPY(var1,var2),var2);
  //这里是否会展开为： strcpy(strcpy(var1_p,"var2")_p,"var2“)？答案是否定的：
  //展开结果将是： strcpy(STRCPY(var1,var2)_p,"var2")
  //##阻止了参数的宏展开!如果宏定义里没有用到#和##,宏将会完全展开
  //把注释打开的话，会报错:implicit declaration of function 'STRCPY'
  return 0;
}
```

结果：

```
var1 = var2
var2 = var1
```

1.1.1.2 关键字volatile有什么含意？并举出三个不同的例子？

1. **并行设备的硬件寄存器。**存储器映射的硬件寄存器通常加volatile，因为寄存器随时可以被外设硬件修改。当声明指向设备寄存器的指针时一定要用volatile，它会告诉编译器不要对存储在这个地址的数据进行假设。
2. **一个中断服务程序中修改的供其他程序检测的变量。**volatile提醒编译器，它后面所定义的变量随时都有可能改变。因此编译后的程序每次需要存储或读取这个变量的时候，**都会直接从变量地址中读取数据。**如果没有volatile关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。
3. **多线程应用中被几个任务共享的变量。**单地说就是防止编译器对代码进行优化.比如如下程序：

```
XBYTE[2]=0x55;
XBYTE[2]=0x56;
XBYTE[2]=0x57;
XBYTE[2]=0x58;
```

对外部硬件而言，上述四条语句分别表示不同的操作，会产生四种不同的动作，但是编译器却会对上述四条语句进行优化，**认为只有XBYTE[2]=0x58(即忽略前三条语句，只产生一条机器代码)。**如果键入volatile，编译器会逐一的进行编译并产生相应的机器代码(产生四条代码)。

1.1.1.3 关键字static的作用是什么?

1. 在函数体，**只会被初始化一次**，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。
2. 在模块内(但**在函数体外**)，一个被声明为**静态的变量**可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个**本地的全局变量**(只能被当前文件使用)。
3. 在模块内，一个被声明为**静态的函数**只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用(**只能被当前文件使用**)。

在C语言中，为什么 static变量只初始化一次?

对于所有的对象(不仅仅是静态对象)，**初始化都只有一次**，而由于静态变量具有“记忆”功能，初始化后，一直都没有被销毁，都会**保存在内存区域中**，所以不会再次初始化。存放在静态区的变量的生命周期一般比较长，它与整个程序“同生死、共存亡”，所以它只需初始化一次。而auto变量，即自动变量，由于它**存放在栈区**，一旦函数调用结束，就会**立刻被销毁**。

1.1.1.4 extern”C”的作用是什么?

extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会**指示编译器这部分代码按C语言的进行编译**，而不是C++的。

1.1.1.5 const有什么作用?

1. 定义变量(局部变量或全局变量)为常量，例如：

```
const int N=100;//定义一个常量N
N=50; //错误，常量的值不能被修改
const int n; //错误，常量在定义的时候必须初始化
```

2. 修饰函数的参数，表示在函数体内不能修改这个参数的值。
3. 修饰函数的返回值。
    - 如果给用const修饰**返回值的类型为指针**，那么函数返回值(即指针)的内容是**不能被修改**的，而且这个返回值只能赋给被 const修饰的指针。例如：

```
const char *GetString()//定义一个函数
char *str= GetString()//错误，因为str没有被 const修饰
const char *str=GetString()//正确
```

    - 如果用 const修饰**普通的返回值**，如返回int变量，由于这个返回值是一个临时变量，在函数调用结束后这个临时变量的生命周期也就结束了，因此把这些**返回值修饰为 const是没有意义**的。
4. 节省空间，避免不必要的内存分配。例如：

```
#define PI 3.14159//该宏用来定义常量
const doulbe Pi=3.14159//此时并未将P放入只读存储器中
double i=Pi//此时为Pi分配内存，以后不再分配
double I=PI//编译期间进行宏替换，分配内存
double j=Pi//没有内存分配再次进行宏替换，又一次分配内存
```

1.1.1.6 什么情况下使用const关键字?

1. 修饰一般常量。一般常量是指简单类型的常量。这种常量在定义时，修饰符const可以用在类型说明符前，也可以用在类型说明符后。例如：

```
int const x=2; const int x=2
```

2. 修饰常数组。定义或说明一个常数组可以采用如下格式：

```
int const a[8]={1,2,3,4,5,6,7,8}
const int a[8]={1,2,3,4,5,6,7,8}
```

3. 修饰常对象。常对象是指对象常量，定义格式如下：

```
class A;
const A a;
A const a;
```

定义常对象时，同样要进行初始化，并且该对象不能再被更新。修饰符const可以放在类名后面，也可以放在类名前面。

4. 修饰常指针

```
const int*p; //常量指针，指向常量的指针。即p指向的内存可以变，p指向的数值内容不可变
int const*p; //同上
int*const p; //指针常量，本质是一个常量，而用指针修饰它。即p指向的内存不可以变，但是p内存位置的数值可以变
const int* const p; //指向常量的常量指针。即p指向的内存和数值都不可变
```

5. 修饰常引用。被const修饰的引用变量为常引用，一旦被初始化，就不能再指向其他对象了。
6. 修饰函数的常参数。 const修饰符也可以修饰函数的传递参数，格式如下：

```
void Fun(const int Var)
```

告诉编译器Var在函数体中不能被改变，从而防止了使用者一些无意的或错误的修改。

7. 修饰函数的返回值。 const修饰符也可以修饰函数的返回值，表明该返回值不可被改变，格式如下：

```
const int FunI();
const MyClass Fun2();
```

8. 在另一连接文件中引用 const常量。使用方式有

```
extern const int 1;
extern const int j=10;
```

1.1.1.7 new/delete与malloc/free的区别是什么?

1. new、delete是C++中的操作符，而malloc和free是标准库函数。
2. 对于非内部数据对象来说，只使用malloc是无法完成动态对象要求的，一般在创建对象时需要调用构造函数，对象消亡时，自动的调用析构函数。而malloc free是库函数而不是运算符，不在编译器控制范围之内，不能够自动调用构造函数和析构函数。而NEW在为对象申请分配内存空间时，可以自动调用构造函数，同时也可以完成对对象的初始化。同理，delete也可以自动调用析构函数。而 mallloc只是做一件事，只是为变量分配了内存，同理，free也只是释放变量的内存。
3. new返回的是指定类型的指针，并且可以自动计算所申请内存的大小。而 malloc需要我们计算申请内存的大小，并且在返回时强行转换为实际类型的指针。

1.1.1.8 strlen("\0")=? sizeof("\0")=?

```
strlen("\0")=0;
sizeof("\0")=2;
```

strlen用来计算字符串的长度(在C/C++中，字符串是**以"\0"作为结束符的**)，它从内存的某个位置(可以是字符串开头，中间某个位置，甚至是某个不确定的内存区域)开始扫描直到碰到第一个字符串结束 符\0为止，然后返回计数器值sizeof是C语言的关键字，它以**字节的形式**给出了其操作数的**存储大小**，操作数可以是一个表达式或括在括号内的类型名，操作数的存储大小由操作数的类型决定。

1.1.1.9 sizeof和strlen有什么区别?

strlen与sizeof的差别表现在以下5个方面。

1. sizeof是运算符(是不是被弄糊涂了？事实上， sizeof既是关键字，也是运算符，但不是函数)， 而strlen是函数。 sizeof后如果是类型，则必须加括弧，如果是变量名，则可以不加括弧。
2. sizeof运算符的结果类型是 size_t，它在头文件中 typedef为 unsigned int类型。该类型保证能够容纳实现所建立的最大对象的字节大小
3. sizeof可以用类型作为参数， strlen只能用char*作参数，而且必须是以“0结尾的。 sizeof还可以以函数作为参数，如int g()，则 sizeof(g())的值等于 sizeof(int的值，在32 位计算机下，该 值为4 。
4. 大部分编译程序的 sizeof都是在**编译**的时候计算的，所以可以通过 sizeof(x)来定义数组维数。 而strlen则是在**运行期**计算的，用来计算字符串的实际长度，不是类型占内存的大小。例如， char str[20]= "0123456789”，字符数组str是**编译期**大小已经固定的数组，在32 位机器下，为sizeof(char)*20=20，而其 strlen大小则是在**运行期**确定的，所以其值为字符串的实际长度10 。
5. 当数组作为参数传给函数时，传递的是指针，而不是数组，即传递的是数组的首地址。

1.1.1.10 不使用 sizeof，如何求int占用的字节数?

```
#include <stdio.h>
#define MySizeof(Value)  (char *)(&value+1)-(char*)&value
int main()
{
  int i ;
  double f;
  double *q;
  printf("%d\r\n",MySizeof(i));
  printf("%d\r\n",MySizeof(f));
  printf("%d\r\n",MySizeof(a));
  printf("%d\r\n",MySizeof(q));
  return 0;
}
```

输出为:

```
4 8 32 4
```

上例中，(char *) & Value返回 Value的地址的第一个字节，(char *)(& Value+1)返回 value的地址的下一个地址的第一个字节，所以它们之差为它所占的字节数。

1.1.1.11 C语言中 struct与 union的区别是什么?

struct(结构体)与union(联合体)是C语言中两种不同的数据结构，两者都是常见的复合结构，其区别主要表现在以下两个方面。

1. 结构体与联合体虽然都是由多个不同的数据类型成员组成的，但不同之处在于联合体中所有成员**共用一块地址空间**，即联合体只存放了一个被选中的成员，而结构体中所有成员占用空间是累加的，其所有成员都存在，不同成员会存放在不同的地址。在计算一个结构型变量的总长度时，其内存空间大小等于所有成员长度之和(需要考虑字节对齐)，而在联合体中，所有成员不能同时占用内存 空间，它们不能同时存在，所以**一个联合型变量的长度等于其最长的成员的长度。**
2. 对于联合体的不同成员赋值，**将会对它的其他成员重写**，原来成员的值就不存在了，而对结构体的不同成员赋值是互不影响的。

举个例子。下列代码执行结果是多少?

```
typedef union {double i; int k[5]; char c;}DATE;
typedef struct data( int cat; DATE cow;double dog;)too;
DATE max;
printf ("%d", sizeof(too)+sizeof(max));
```

假设为32 位机器，int型占4 个字节， double型占8 个字节，char型占1 个字节，而DATE是一个联合型变 量，联合型变量共用空间，uion里面最大的变量类型是int[5]，**所以占用20 个字节，它的大小是20**，而 由于 union中 double占了8 个字节，因此 union是要8 个字节对齐，**所占内存空间为8 的倍数。**为了实现 8 个字节对齐，**所占空间为24**.而data是一个结构体变量，每个变量分开占用空间，依次为sizeof(int) + sizeof(DATE)+ sizeof(double)=4+24+8=36按照8 字节对齐，占用空间为40 ，所以结果为40+24=64。

1.1.1.12 左值和右值是什么?

左值是指可以出现在等号左边的变量或表达式，它最重要的特点就是可写(可寻址)。也就是说，它的值**可以被修改**，如果**一个变量或表达式的值不能被修改**，那么它就不能作为左值。

右值是指只可以**出现在等号右边的变量或表达式。**它最重要的特点是**可读。**一般的使用场景都是把一个右值赋值给一个左值。

通常，左值可以作为右值，但是右值不一定是左值。

1.1.1.13 什么是短路求值?

```
#include <stdio.h>
int main()
{
  int i = 6;
  int j = 1;
  if(i>0||(j++)>0);
  printf("%D\r\n",j);
  return 0;
}
```

输出结果为1 。

输出为什么不是2 ，而是1 呢? 其实，这里就涉及一个短路计算的问题。由于i语句是个条件判断语句，里 面是有两个简单语句进行或运算组合的复合语句，因为或运算中，只要参与或运算的两个表达式的值都 为真，则整个运算结果为真，而由于变量i的值为6 ，已经大于0 了，而该语句已经为true，则不需要执行 后续的j+操作来判断真假，所以后续的j++操作不需要执行，j的值仍然为1 。

因为短路计算的问题，对于&&操作，由于在两个表达式的返回值中，如果有一个为假则整个表达式的值 都为假，如果前一个语句的返回值为 false，则无论后一个语句的返回值是真是假，整个条件判断都为 假，不用执行后一个语句，而a>b的返回值为 false，程序不执行表达式n=c>d，所以，n的值保持为初 值2 。

1.1.1.14 ++a和a++有什么区别? 两者是如何实现的?

a++的具体运算过程为

```
int temp = a;
a=a+1;
return temp;
```

++a的具体运算过程为

```
a=a+1;
return a;
```

后置自增运算符需要把原来变量的值复制到**一个临时的存储空间**，等运算结束后才会返回这个临时变量的值。所以前置自增运算符效率比后置自增要高。

1.1.2 内存

1.1.2.1 C语言中内存分配的方式有几种?

1. **静态存储区分配**

   内存分配在程序编译之前完成，且在程序的整个运行期间都存在，例如全局变量、静态变量等。

2. **栈上分配**

   在函数执行时，函数内的局部变量的存储单元在栈上创建，函数执行结束时这些存储单元自动释放。

3. **堆上分配**

1.1.2.2 堆与栈有什么区别?

1. **申请方式**

   栈的空间由操作系统自动分配/释放，堆上的空间手动分配/释放。

2. **申请大小的限制**

   栈空间有限。在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M(也有的说是1M，总之是一个编译时就确定的常数)，如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小 堆是很大的自由存储区。

   堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用 链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大 小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

3. **申请效率**

   栈由系统自动分配，速度较快。但程序员是无法控制的。堆是由malloc分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

1.1.2.3 栈在C语言中有什么作用?

1. C语言中栈用来存储临时变量，临时变量包括**函数参数和函数内部定义的临时变量。**函数调用中和函数调用相关的**函数返回地址**，函数中的临时变量，寄存器等均保存在栈中，函数调动返回后从栈中恢复寄存器和临时变量等函数运行场景。
2. 多线程编程的基础是栈，**栈是多线程编程的基石**，每一个线程都最少有一个自己专属的栈，用来存储本线程运行时各个函数的临时变量和维系函数调用和函数返回时的函数调用关系和函数运行场景。操作系统最基本的功能是支持多线程编程，支持中断和异常处理，每个线程都有专属的栈，中断和异常处理也具有专属的栈，栈是操作系统多线程管理的基石。

1.1.2.4 C语言函数参数压栈顺序是怎样的?

从右至左。

C语言参数入栈顺序的好处就是可以动态变化参数个数。自左向右的入栈方式，最前面的参数被压在栈底。除非知道参数个数，否则是无法通过栈指针的相对位移求得最左边的参数。这样就变成了左边参数的个数不确定，正好和动态参数个数的方向相反。因此，C语言函数参数采用自右向左的入栈顺序，主要原因是为了支持可变长参数形式。

1.1.2.5 C++如何处理返回值?

C++函数返回可以按值返回和按常量引用返回，偶尔也可以按引址返回。多数情况下不要使用引址返回。

1.1.2.6 C++中拷贝赋值函数的形参能否进行值传递?

不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数(aa = ex.aa; //此处调用拷贝构造函数)。如此循环，无法完成拷贝，栈也会满。

```
class Example
{
public:
  Example(int a):aa(a) {} //构造函数

    Example(Example &ex) //拷贝构造函数（引用传递参数）
  {
    aa = ex.aa; //此处调用拷贝构造函数
  }
private:
    int aa;
};

int main()
{
  Example e1(10);
  Example e2 = e1;

  return 0;
}
```

1.1.2.7 C++程序的内存管理是怎样的?

在C++中，虚拟内存分为**代码段、数据段、BSS段、堆区、文件映射区以及栈区**六部分。

代码段：包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

数据段：存储程序中已初始化的全局变量和静态变量

BSS 段：存储未初始化的全局变量和静态变量(局部+全局)，以及所有被初始化为0 的全局变量和静态变量。

堆区：调用new/malloc函数时在堆区动态分配内存，同时需要调用delete/free来手动释放申请的内存。

映射区:存储动态链接库以及调用mmap函数进行的文件映射

栈：使用栈空间存储函数的返回地址、参数、局部变量、返回值

1.1.2.8 什么是内存泄漏?

简单地说就是申请了一块内存空间，使用完毕后没有释放掉。

它的一般表现方式是程序运行时间越长，占用内存越多，最终用尽全部内存，整个系统崩溃。由程序申请的一块内存，且没有任何一个指针指向它，那么这块内存就泄露了。

1.1.2.9 如何判断内存泄漏?

1. 良好的编码习惯，尽量在涉及内存的程序段，检测出内存泄露。当程式稳定之后，在来检测内存泄露时，无疑增加了排除的困难和复杂度。使用了内存分配的函数，一旦使用完毕,要记得要使用其

相应的函数释放掉。
2. 将分配的内存的指针以链表的形式自行管理，使用完毕之后从链表中删除，程序结束时可检查改链表。
3. Boost 中的smart pointer。
4. 一些常见的工具插件，如ccmalloc、Dmalloc、Leaky等等。

1.1.3 指针

1.1.3.1 数组指针和指针数组有什么区别?

数组指针就是指向数组的指针，它表示的是一个指针，这个指针指向的是一个数组，它的重点是指针。

例如，int(*pa)[8]声明了一个指针，该指针指向了一个有8 个int型元素的数组。下面给出一个数组 指针的示例。

```
#include <stdio. h>
#include <stdlib. h>
void main()
{
  int b[12]={1,2,3,4,5,6,7,8,9,10,11,12};
  int (*p)[4];
  p = b;
  printf("%d\n",  **(++p);
}
```

程序的输出结果为5 。

上例中，p是一个数组指针，它指向一个**包含有4 个int类型**数组的指针，刚开始p被初始化为指向数组b的首地址，++p相当于把p所指向的地址**向后移动4 个int所占用的空间**，此时p指向数组{5,6,7,8}，语句*(++p)；表示的是这个数组中**第一个元素**的地址(可以理解p为指向二维数组的指针，｛1,2,3,4｝，｛5,6,7,8｝，｛9,10,11,12｝。p指向的就是｛1,2,3,4｝的地址，*p就是指向元素，｛1,2,3,4｝，`**`p指向的就是1)，语句**(++p)会输出这个数组的第一个元素5 。

指针数组表示的是一个数组，而数组中的元素是指针。下面给出另外一个指针数组的示例

```
#include <stdio.h>
int main()
{
  int i;
  int *p[4];
  int a[4]={1,2,3,4};
  p[0] = &a[0];
  p[1] = &a[1];
  p[2] = &a[2];
  p[3] = &a[3];
  for(i=0;i<4;i++)
  printf("%d",*p[i]);
  printf("\n");
  return 0;
}
```

程序的输出结果为1234 。

1.1.3.2 函数指针和指针函数有什么区别?

1. **函数指针**

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**。

```
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)`。

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。

最后需要注意的是，指向函数的指针变量没有++和--运算。

```
# include <stdio.h>
int Max(int, int);//函数声明
int main(void)
{
  int(*p)(int, int);//定义一个函数指针
  int a, b, c;
  p = Max;//把函数Max赋给指针变量p,使p指向Max函数
  printf("please enter a and b:");
  scanf("%d%d",&a,&b);
  c = (*p)(a, b);//通过函数指针调用Max函数
  printf("a = %d\nb = %d\nmax = %d\n", a, b, c);
  return 0;
}
int Max(int x, int y)//定义Max函数
{
  int z;
  if (x > y)
  {
    z = x;
  }
  else
  {
    z = y;
  }
  return z;
}
```

2. **指针函数**

首先它是一个函数，只不过这个函数的返回值是一个地址值。函数返回值必须用同类型的指针变量来接受，也就是说，指针函数一定有“函数返回值”，而且，在主调函数中，函数返回值必须赋给同类型的指针变量。

`类型名*函数名(函数参数列表);`

其中，后缀运算符括号“()”表示这是一个函数，其前缀运算符星号“*”表示此函数为指针型函数，其函数值 为指针，即它带回来的值的类型为指针，当调用这个函数后，将得到一个“指向返回值为...的指针(地址)，“类型名”表示函数返回的指针指向的类型”。

“(函数参数列表)”中的括号为函数调用运算符，在调用语句中，即使函数不带参数，其参数表的一对括号 也不能省略。其示例如下：

```
int *pfun(int, int);
```

由于“*”的优先级低于“()”的优先级，因而pfun首先和后面的“()”结合，也就意味着，pfun是一个函数。即：

```
int *(pfun(int, int));
```

接着再和前面的“*”结合，说明这个函数的返回值是一个指针。由于前面还有一个int，也就是说，pfun是一个返回值为整型指针的函数。

```
#include <stdio.h>
float *find(float(*pionter)[4],int n);//函数声明
int main(void)
{
  static float score[][4]={{60,70,80,90},{56,89,34,45},{34,23,56,45}};
  float *p;
  int i,m;
  printf("Enter the number to be found:");
  scanf("%d",&m);
  printf("the score of NO.%d are:\n",m);
  p=find(score,m-1);
  for(i=0;i<4;i++)
    printf("%5.2f\t",*(p+i));
  return 0;
}

float *find(float(*pionter)[4],int n)/*定义指针函数*/
{
  float *pt;
  pt=*(pionter+n);
  return(pt);
}
```

共有三个学生的成绩，函数find()被定义为指针函数，其形参pointer是指针指向包含4 个元素的一维数组的指针变量。pointer+n指向score的第n+1行。*(pointer+1)指向第一行的第0 个元素。pt是一个指针变量，它指向浮点型变量。main()函数中调用find()函数，将score数组的首地址传给pointer。

1.1.3.3 数组名和指针的区别与联系是什么?

1. **数据保存方面**

指针保存的是地址(保存目标数据地址，自身地址由编译器分配)，内存访问偏移量为4 个字节，无论其中保存的是何种数据均已地址类型进行解析。

数组保存的数据。数组名表示的是第一个元素的地址，内存偏移量是保存数据类型的内存偏移量；只有对数组名取地址(&数组名)时数组名才表示整个数组，**内存偏移量是整个数组的大小(sizeof(数组名))。**

2. **数据访问方面**

指针对数据的访问方式是间接访问，需要用到解引用符号(*数组名)。

数组对数据的访问则是直接访问，可通过下标访问或数组名+元素偏移量的方式

3. **使用环境**

指针多用于动态数据结构(如链表，等等)和动态内存开辟。

数组多用于存储固定个数且类型统一的数据结构(如线性表等等)和隐式分配。

1.1.3.4 指针进行强制类型转换后与地址进行加法运算，结果是什么?

假设在32 位机器上，在对齐为4 的情况下,sizeof(long)的结果为4 字节，sizeof(char*)的结果为4 字节，sizeof(short int)的结果与 sizeof(short)的结果都为2 字节， sizeof(char)的结果为1 字节， sizeof(int)的结果为4 字节，由于32 位机器上是4 字节对齐，以如下结构体为例：

```
struct BBB
{
  long num;
  char *name;
  short int data;
  char ha;
  short ba[5];
}*p;
```

`当p=0x100000;则p+0x200=? (ulong)p+0x200=? (char*)p+0x200=?`

其实,在32 位机器下,sizeof(struct BBB)=sizeof(*p)=4+4+2+2+1+3`/*补齐*/`+2_5+2`/*补齐*/`=24 *字节，而 p=0x100000,那么 p+0x200=0x1000000+0x200*24 指针加法，加出来的是指针所指类型的字节长度的整倍数，就是p偏移sizeof(p)*0x200。

`(ulong)p+0x200=0x10000010+0x200`经过ulong后,已经不再是指针加法，而变成一个数值加法了。

`(char*)p+0x200=0x1000000+0x200*sizeof(char)`结果类型是char*。

1.1.3.5 指针常量，常量指针，指向常量的常量指针有什么区别?

1. **指针常量**

```
int * const p
```

先看const再看*，p是一个常量类型的指针，**不能修改这个指针的指向**，但是这个指针所指向的地址上存储的**值可以修改。**

2. **常量指针**

```
const int *p
int const *p
```

先看*再看const，定义一个指针指向一个常量，不能通过指针来修改这个指针**指向的值**

3. **指向常量的常量指针**

```
const int *const p
```

对于“指向常量的常量指针”，就必须同时满足上述1和2中的内容，**既不可以修改指针的值，也不可以修改指针指向的值。**

1.1.3.6 指针和引用的异同是什么？如何相互转换?

相同

1. 都是地址的概念，指针指向某一内存、它的内容是所指内存的地址；引用则是某块内存的别名。

2. 从内存分配上看：两者都占内存，程序为指针会分配内存，一般是4 个字节；而引用的本质是指针常量，指向对象不能变，但指向对象的值可以变。两者都是地址概念，所以本身都会占用内存。

区别

1. 指针是实体，而引用是别名。
2. 指针和引用的自增(++)运算符意义不同，指针是对内存地址自增，而引用是对值的自增。
3. 引用使用时无需解引用(*)，指针需要解引用；(关于解引用大家可以看看这篇博客，传送门)。
4. 引用只能在定义时被初始化一次，之后不可变；指针可变。
5. 引用不能为空，指针可以为空。
6. “sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身的大小，在32位系统指针变量一般占用4 字节内存。

```
#include "stdio.h"
int main(){
  int x = 5;
  int *p = &x;
  int &q = x;
  printf("%d %d\n",*p,sizeof(p));
  printf("%d %d\n",q,sizeof(q));
}
```

```
//结果
5 8
5 4
```

由结果可知，引用使用时无需解引用(*)，指针需要解引用；我用的是64 位操作系统，“sizeof 指针”得到的是指针本身的大小，及8 个字节。而“sizeof 引用”得到的是的对象本身的大小及int的大小，4 个字节。

转换

1. **指针转引用**：把指针用*就可以转换成对象，可以用在引用参数当中。
2. **引用转指针**：把引用类型的对象用&取地址就获得指针了。

```
int a = 5;
int *p = &a;
void fun(int &x){}//此时调用fun可使用 ：  fun（*p）;
//p是指针，加个*号后可以转换成该指针指向的对象，此时fun的形参是一个引用值，
//p指针指向的对象会转换成引用x。
```

1.1.3.7 野指针是什么?

1. 野指针是指向不可用内存的指针，当指针被创建时，指针不可能自动指向NULL，这时，默认值是随机的，此时的指针成为野指针。
2. 当指针被free或delete释放掉时，如果没有把指针设置为NULL，则会产生野指针，因为释放掉的仅 仅是指针指向的内存，并没有把指针本身释放掉。
3. 第三个造成野指针的原因是指针操作超越了变量的作用范围。

1.1.3.8 如何避免野指针?

1. 对指针进行初始化。

```
//将指针初始化为NULL。
char * p = NULL;
//用malloc分配内存
char * p = (char * )malloc(sizeof(char));
//用已有合法的可访问的内存地址对指针初始化
char num[ 30] = {0};
char *p = num;
```

2. 指针用完后释放内存，将指针赋NULL。

```
delete(p);
p = NULL;
```

注：malloc函数分配完内存后需注意：

a.检查是否分配成功(若分配成功，返回内存的首地址；分配不成功，返回NULL。可以通过if语句来判断)

b.清空内存中的数据(malloc分配的空间里可能存在垃圾值，用memset或bzero 函数清空内存)

```
//s是需要置零的空间的起始地址； n是要置零的数据字节个数。
void bzero(void *s, int n);
//如果要清空空间的首地址为p，value为值，size为字节数。
void memset(void *start, int value, int size);
```

1.1.3.9 C++中的智能指针是什么?

智能指针是一个类，用来存储指针(指向动态分配对象的指针)。

C++程序设计中使用堆内存是非常频繁的操作，堆内存的申请和释放都由程序员自己管理。程序员自己管理堆内存可以提高了程序的效率，但是整体来说堆内存的管理是麻烦的，C++11中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露(忘记释放)，二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

1.1.3.10 智能指针的内存泄漏如何解决?

为了解决循环引用导致的内存泄漏，引入了弱指针weak_ptr，weak_ptr的构造函数不会修改引用计 数的值，从而不会对对象的内存进行管理，其类似一个普通指针，但是不会指向引用计数的共享内存， 但是可以检测到所管理的对象是否已经被释放，从而避免非法访问。

1.1.4 预处理

1.1.4.1 预处理器标识#error的目的是什么?

error预处理指令的作用是，编译程序时，只要遇到#error就会生成一个**编译错误提示消息**，**并停止编译**。其语法格式为：#error error-message。

下面举个例子：

程序中往往有很多的预处理指令

```
#ifdef XXX
...
#else
#endif
```

当程序比较大时，往往有些宏定义是在外部指定的(如makefile)，或是在系统头文件中指定的，当你 不太确定当前是否定义了 XXX 时，就可以改成如下这样进行编译：

```
#ifdef XXX
...
#error "XXX has been defined"
#else
#endif
```

这样,如果编译时出现错误,输出了XXX has been defined,表明宏XXX已经被定义了。

1.1.4.2 定义常量谁更好？# define还是 const?

尺有所短，寸有所长， define与 const都能定义常量，效果虽然一样，但是各有侧重。

define既可以替代常数值，又可以替代表达式，甚至是代码段，但是容易出错，而 const的引入可以增强程序的可读性，它使程序的维护与调试变得更加方便。具体而言，它们的差异主要表现在以下3 个方面。

1. define只是用来进行**单纯的文本替换**， define常量的**生命周期止于编译期**，**不分配内存空间**，它存在于程序的**代码段**，在实际程序中，它只是一个常数；而 const常量存在于程序的**数据段**，并在**堆栈中分配了空间**， const常量在程序中确确实实存在，并且可以被调用、传递
2. const常量有数据类型，而 define常量没有数据类型。编译器可以对 const常量进行类型安全检查，如类型、语句结构等，而 define不行。
3. 很多IDE **支持调试** const定义的常量，而不支持 define定义的常量由于 const修饰的变量可以排除程序之间的不安全性因素，保护程序中的常量不被修改，而且对数据类型也会进行相应的检查，极大地提高了程序的**健壮性**，所以一般**更加倾向于用const来定义常量类型。**

1.1.4.3 typedef和 define有什么区别?

typedef与 define都是**替一个对象取一个别名**，以此来增强程序的可读性，但是它们在使用和作用上也存在着以下4 个方面的不同。

1. **原理不同**

#define是C语言中定义的语法，它是预处理指令，在预处理时进行简单而机械的字符串替换，**不做正确性检查**，不管含义是否正确照样代入，只有在编译已被展开的源程序时，才会发现可能的错误并报错。

例如，`# define Pl3.1415926`，当程序执行`area=Pr*r`语句时，PI会被替换为3.1415926。于是该语句被替换为`area=3.1415926_r*r`。如果把# define语句中的数字9 写成了g，预处理也照样代入，而不去检查其是否合理、合法。

typedef是关键字，它在编译时处理，所以 typedef具有类型检查的功能。它在自己的作用域内给一个已经存在的类型一个别名，但是不能在一个函数定义里面使用标识符 typedef。例如， `typedef int INTEGER`，这以后就可用 INTEGER来代替int作整型变量的类型说明了，例如：`INTEGER a,b`;

用 typedef定义数组、指针、结构等类型将带来很大的方便，不仅使程序书写简单而且使意义更为明确，因而增强了可读性。例如：`typedef int a[10]`;

表示a是整型数组类型，数组长度为10 。然后就可用a说明变量，例如:语句a s1,s2；完全等效于语句int s1[10],s2[10].同理， typedef void(*p)(void)表示p是一种指向void型的指针类型。

2. **功能不同**

typedef用来定义类型的别名，这些类型不仅包含内部类型(int、char等)，还包括自定义类型(如struct)，可以起到使类型易于记忆的功能。

例如：`typedef int (*PF)(const char _, const char_)`

定义一个指向函数的指针的数据类型PF，其中函数返回值为int，参数为 const char*。typedef还有另外一个重要的用途，那就是定义机器无关的类型。例如，可以定义一个叫REAL的浮点类型，在目标机器上 它可以获得最高的精度：`typedef long double REAL`，在不支持 long double的机器上，该typedef 看起来会是下面这样：`typedef double real`，在 double都不支持的机器上，该 typedef看起来会是 这样：`typedef float REAL`。

define不只是可以为类型取别名，还可以定义常量、变量、编译开关等。

3. **作用域不同**

define没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用，而 typedef有自己的作用域。

程序示例如下：

```
void fun()
{
  #define A int
}
void gun()
{
  //这里也可以使用A，因为宏替换没有作用域，但如果上面用的是 typedef，那这里就不能用
  //A，不过，一般不在函数内使用 typedef
}
```

1. **对指针的操作不同**

两者修饰指针类型时，作用不同。

```
#define INTPTR1 int*
typedef int* INTPTR2;
INTPTR1pl, p2;
INTPTR2 p3,  p4;
```

`INTPTR1 pl, p2`和`INTPTR2 p3, p4`的效果截然不同。`INTPTR1 pl, p2`进行字符串替换后变成`int *p1,p2`，要表达的意义是声明一个指针变量p1和一个整型变量p2.而`INTPTR2 p3, p4`，由于INTPTR2是具有含义的，告诉我们是一个指向整型数据的指针，那么p3和p4都为指针变量，这句相当于`int*p1,*p2`.从这里可以看出，进行宏替换是不含任何意义的替换，仅仅为字符串替换；而用typedef 为一种数据类型起的别名是带有一定含义的。

程序示例如下

```
#define INTPTR1 int*
typedef int* INTPTR2
int a=1;
int b=2;
int c=3;
const INTPTR1p1=&a;
const INTPTR2 p2=&b;
INTPTR2 const p3=&c;
```

上述代码中， const INTPTR1 p1表示p1是一个常量指针，即不可以通过p1去修改p1指向的内容，但是p1可以指向其他内容。而对于 const INTPTR2 p2，由于 INTPTR2表示的是个指针类型，因此用 const 去 限定，表示封锁了这个指针类型，因此p2是一个指针常量，不可使p2再指向其他内容，但可以通过p2修 改其当前指向的内容。 INTPTR2 const p3同样声明的是一个指针常量。

1.1.4.4 如何使用 define声明个常数，用以表明1 年中有多少秒(忽略闰年问题)

```
#define SECOND_PER_YEAR (60*60*24*365)UL
```

1.1.4.5 # include < filename. h>和# include " filename. h"有什么区别?

对于 include< filename. h>，编译器先从标准库路径开始搜索 filename.h，使得系统文件调用较快。而对于# include“ filename.h””，编译器先从用户的工作路径开始搜索 filename.h，然后去寻找系统路径，使得自定义文件较快。

1.1.4.6 头文件的作用有哪些?

头文件的作用主要表现为以下两个方面：

1. 通过头文件来调用库功能。出于对源代码保密的考虑，源代码不便(或不准)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. 头文件能加强类型安全检查。当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，编译器就会指出错误，大大减轻程序员调试、改错的负担。

1.1.4.7 在头文件中定义静态变量是否可行，为什么?

不可行，如果在头文件中定义静态变量，会造成资源浪费的问题，同时也可能引起程序错误。因为如果在使用了该头文件的每个C语言文件中定义静态变量，按照编译的步骤，在**每个头文件中都会单独存在一个静态变量**，从而会引起**空间浪费**或者**程序错误**所以，不推荐在头文件中定义任何变量，当然也包括静态变量。

1.1.4.8 不使用流程控制语句，如何打印出1~1000的整数?

宏定义多层嵌套(10 *10 *10)，printf多次输出。

```
#include <stdio. h>
#define B P,P,P,P,P,P,P,P,P,P
#define P L,L,L,L,L,L,L,L,L,L
#define L I,I,I,I,I,I,I,I,I,I,N
#define I printf("%3d", i++)
#define N printf("n")
int main()
{
  int i = 1;
  B;
  return 0;
}
```

简便写法,同样使用多层嵌套

```
#include<stdio.h>
#define A(x) x;x;x;x;x;x;x;x;x;

int main ()
{
  int n=1;
  A(A(A(printf("%d",n++);
  return 0;
}
```

1.1.5 变量

1.1.5.1 全局变量和静态变量的区别是什么?

1. 全局变量的作用域为程序块，而局部变量的作用域为当前函数。
2. 内存存储方式不同，全局变量(静态全局变量，静态局部变量)分配在全局数据区(静态存储空间)，后者分配在栈区。
3. 生命周期不同。全局变量随主程序创建而创建，随主程序销毁而销毁，局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在了。
4. 使用方式不同。通过声明为全局变量，程序的各个部分都可以用到，而局部变量只能在局部使用。

1.1.5.2 全局变量可不可以定义在可被多个.C文件包含的头文件中？为什么?

可以，在不同的C文件中以static形式来声明**同名全局变量**。

可以在不同的C文件中声明同名的全局变量，前提是其中**只能有一个C文件中对此变量赋初值**，此时连接不会出错。

1.1.5.3 局部变量能否和全局变量重名?

能，局部会屏蔽全局。

局部变量可以与全局变量同名，在函数内引用这个变量时，会用到同名的局部变量，而不会用到全局变量。

对于有些编译器而言，在同一个函数内可以定义多个同名的局部变量，比如在两个循环体内都定义一个同名的局部变量，而那个局部变量的作用域就在那个循环体内。

1.1.6 函数

1.1.6.1 为什么析构函数必须是虚函数?

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄漏。

1.1.6.2 为什么C++默认的析构函数不是虚函数?

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是只有当需要当作父类时，设置为虚函数。

1.1.6.3 C++中析构函数的作用?

如果构造函数打开了一个文件，最后不需要使用时文件就要被关闭。析构函数允许类自动完成类似清理工作，不必调用其他成员函数。

析构函数也是特殊的类成员函数。简单来说，析构函数与构造函数的作用正好相反，它用来完成对象被删除前的一些清理工作，也就是专门的扫尾工作。

1.1.6.4 静态函数和虚函数的区别?

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销。

1.1.6.5 重载和覆盖有什么区别?

1. 覆盖是子类和父类之间的关系，垂直关系；重载同一个类之间方法之间的关系，是水平关系。
2. 覆盖只能由一个方法或者只能由一对方法产生关系；重载是多个方法之间的关系。
3. 覆盖是根据对象类型(对象对应存储空间类型)来决定的；而重载关系是根据调用的实参表和形参表来选择方法体的。

1.1.6.6 虚函数表具体是怎样实现运行时多态的?

原理:

虚函数表是一个类的虚函数的地址表，每个对象在创建时，都会有一个指针指向该类虚函数表，每一个类的虚函数表，按照函数声明的顺序，会将函数地址存在虚函数表中，当子类对象重写父类的虚函数的时候，父类的虚函数表中对应的位置会被子类的虚函数地址覆盖。

作用:

在用父类的指针调用子类对象成员函数时，虚函数表会指明要调用的具体函数是哪个。

1.1.6.7 C语言是怎么进行函数调用的?

大多数CPU上的程序实现**使用栈来支持函数调用操作**，栈被用来传递函数参数、存储返回信息、临时保存寄存器原有的值以备恢复以及用来存储局部变量。

函数调用操作所使用的栈部分叫做**栈帧结构**，每个函数调用都有属于自己的栈帧结构，栈帧结构由两个指针指定，帧指针(指向起始)，栈指针(指向栈顶)，函数对大多数数据的访问都是基于帧指针。下面是结构图:

栈指针和帧指针一般都有专门的寄存器，通常使用ebp寄存器作为帧指针，使用esp寄存器做栈指针。

帧指针指向栈帧结构的头，存放着上一个栈帧的头部地址，栈指针指向栈顶。

1.1.6.8 请你说一说select

1. **select函数原型**

```
int select(int maxfdp,fd_set *readfds,fd_set *writefds,fd_set
*errorfds,struct timeval *timeout);
```

2. **文件描述符的数量**

单个进程能够监视的文件描述符的数量存在最大限制，通常是1024 ，当然可以更改数量；(在linux内核头文件中定义：#define __FD_SETSIZE 1024)

3. **就绪fd采用轮询的方式扫描**

select返回的是int，可以理解为返回的是ready(准备好的)一个或者多个文件描述符，应用程序需要遍历整个文件描述符数组才能发现哪些fd句柄发生了事件，由于select采用轮询的方式扫描文件描述符(不 知道那个文件描述符读写数据，所以需要把所有的fd都遍历)，文件描述符数量越多，性能越差

4. **内核/用户空间内存拷贝**

select每次都会改变内核中的句柄数据结构集(fd集合)，因而每次调用select都需要从用户空间向内核空间复制所有的句柄数据结构(fd集合)，产生巨大的开销

5. **select的触发方式**

select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次调用select还是会将这些文件描述符通知进程。

6. **优点**

a. select的可移植性较好，可以跨平台；

b. select可设置的监听时间timeout精度更好，可精确到微秒，而poll为毫秒。

7. **缺点**:

a. select支持的文件描述符数量上限为1024 ，不能根据用户需求进行更改；

b. select每次调用时都要将文件描述符集合从用户态拷贝到内核态，开销较大；

c. select返回的就绪文件描述符集合，需要用户循环遍历所监听的所有文件描述符是否在该集合中，当监听描述符数量很大时效率较低。

1.1.6.9 请你说说fork,wait,exec函数

父进程产生子进程使用fork拷贝出来一个父进程的副本，此时只拷贝了父进程的页表，两个进程都读同一块内存，当有进程写的时候使用写实拷贝机制分配内存，exec函数可以加载一个elf文件去替换父进程，从此父进程和子进程就可以运行不同的程序了。fork从父进程返回子进程的pid，从子进程返回0.调用了wait的父进程将会发生阻塞，直到有子进程状态改变,执行成功返回0 ，错误返回-1。exec执行成功则子进程从新的程序开始运行，无返回值，执行失败返回-1。

1.1.7 数组

1.1.7.1 以下代码表示什么意思?

```
*(a[1]+1)、*(&a[1][1])、(*(a+1))[1]
```

第一个：因为a[1]是第2 行的地址，a[1]+1偏移一个单位(得到第2 行第2 列的地址)，然后解引用取值，得到`a[1][1]`；

第二个：[]优先级高，`a[1][1]`取地址再取值。

第三个：a+1相当于&a[1]，所以`*(a+1)=a[1]`，因此`*(a+1)[1]=a[1][1]`

1.1.7.2 数组下标可以为负数吗?

可以，因为下标只是给出了一个与**当前地址的偏移量**而已，只要根据这个偏移量能定位得到目标地址即可。下面给出一个下标为负数的示例：

数组下标取负值的情况：

```
#include <stdio.h>
int main()
{
  int i:
  int a[5]={0,1,2,3,4};
  int *p=&a[4]
    for(i=-4; i<=0; i++)
      printf("%d %x\n", p[i],&p[i]);
  return 0.
}
//输出结果为
//0 b3ecf480
//1 b3ecf484
//2 b3ecf488
//3 b3ecf48c
//4 b3ecf490
```

从上例可以发现，在C语言中，数组的下标并非不可以为负数，当数组下标为负数时，编译可以通过，而且也可以得到正确的结果，只是它表示的意思却是从当前地址**向前寻址**.

1.1.8 位操作

1.1.8.1 如何求解整型数的二进制表示中1 的个数?

程序代码如下：

```
#include <stdio.h>
int func(int x)
{
  int countx = 0;
  while(x)
  {
    countx++;
    x = x&(x-1);
  }
  return countx;
}
int main()
{
  printf("%d\n",func(9999));
  return 0;
}
```

程序输出的结果为8 。

在上例中，函数func()的功能是将x转化为二进制数，然后计算该二进制数中含有的1 的个数。首先以 9 为例来分析，9 的二进制表示为1001,8的二进制表示为1000 ，两者执行&操作之后结果为1000 ，此时 1000 再与0111 (7 的二进制位)执行&操作之后结果为0 。

为了理解这个算法的核心，需要理解以下两个操作：

1. 当一个数被减1 时，它最右边的那个值为1 的bit将变为0 ，同时其右边的所有的bit都会变成1 。
2. 每次执行x&(x-1)的作用是把x对应的二进制数中的最后一位1 去掉。因此，循环执行这个操作直 到x等于0 的时候，循环的次数就是x对应的二进制数中1 的个数。

1.1.8.2 如何求解二进制中0 的个数

图示分析(以25 为例)：

```
int CountZeroBit(int num)
{
  int count = 0;
  while (num + 1)
  {
    count++;
    num |= (num + 1); //算法转换
  }
  return count;
}
int main()
{
  int value = 25;
  int ret = CountZeroBit(value);
  printf("%d的二进制位中0的个数为%d\n",value, ret);
  system("pause");
  return 0;
}
```

1.1.8.3 交换两个变量的值，不使用第三个变量。即a=3,b=5,交换之后a=5,b=3;

有两种解法,一种用算术算法,一种用^(异或)。

```
a = a + b;
b = a - b;
a = a - b;
```

```
a = a^b;//只能对int,char..
b = a^b;
a = a^b;
or
a ^= b ^= a;
```

1.1.8.4 给定一个整型变量a，写两段代码，第一个设置a的bit 3，第二个清除a 的bit 3。在以上两个操作中，要保持其它位不变。

```
#define BIT3 (0x1<<3)
static int a;
void set_bit3(void)
{
  a |= BIT3;
}
void clear_bit3(void)
{
  a &= ~BIT3;
}
```