2008

高级 C 语言

杨福林

C 语言编程要点

杨福林
ZTE-S
2008-11-28

目录

1. C 语言中的指针和内存泄漏

在使用 C 语言时，您是否对花时间调试指针和内存泄漏问题感到厌倦？如果是这样，那么本文就适合您。您将了解可能导致内存破坏的指针操作类型，您还将研究一些场景，了解要在使用动态内存分配时考虑什么问题。

引言

对于任何使用 C 语言的人，如果问他们 C 语言的最大烦恼是什么，其中许多人可能会回答说是指针和内存泄漏。这些的确是消耗了开发人员大多数调试时间的事项。指针和内存泄漏对某些开发人员来说似乎令人畏惧，但是一旦您了解了指针及其关联内存操作的基础，它们就是您在 C 语言中拥有的最强大工具。

本文将与您分享开发人员在开始使用指针来编程前应该知道的秘密。本文内容包括：

1. 导致内存破坏的指针操作类型
2. 在使用动态内存分配时必须考虑的检查点
3. 导致内存泄漏的场景

如果您预先知道什么地方可能出错，那么您就能够小心避免陷阱，并消除大多数与指针和内存相关的问题。

- **什么地方可能出错？**

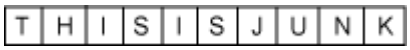
有几种问题场景可能会出现，从而可能在完成生成后导致问题。在处理指针时，您可以使用本文中的信息来避免许多问题。

1. 未初始化的内存

在本例中，p 已被分配了 10 个字节。这 10 个字节可能包含垃圾数据，如图 1 所示。

```
char *p = malloc ( 10 );
```

图 1. 垃圾数据



如果在对这个 p 赋值前，某个代码段尝试访问它，则可能会获得垃圾值，您的程序可能具有不可预测的行为。p 可能具有您的程序从未曾预料到的值。

良好的实践是始终结合使用 memset 和 malloc，或者使用 calloc。

```
char *p = malloc (10);memset(p,'\0',10);
```

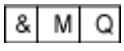
现在，即使同一个代码段尝试在对 p 赋值前访问它，该代码段也能正确处理 Null 值（在理想情况下应具有的值），然后将具有正确的行为。

2. 内存覆盖

由于 p 已被分配了 10 个字节，如果某个代码片段尝试向 p 写入一个 11 字节的值，则该操作将在不告诉您的情况下自动从其他某个位置“吃掉”一个字节。让我们假设指针 q 表示该内存。

图 2. 原始 q 内容

图 3. 覆盖后的 q 内容



结果，指针 q 将具有从未预料到的内容。即使您的模块编码得足够好，也可能由于某个共存模块

执行某些内存操作而具有不正确的行为。下面的示例代码片段也可以说明这种场景。

Char *name = (char *) malloc(11); // Assign some value to namememcpy (p,name,11); // Problem begins here

在本例中，memcpy 操作尝试将 11 个字节写到 p，而后者仅被分配了 10 个字节。

作为良好的实践，每当向指针写入值时，都要确保对可用字节数和所写入的字节数进行交叉核对。一般情况下，memcpy 函数将是用于此目的的检查点。

内存读取越界

内存读取越界 (overread) 是指所读取的字节数多于它们应有的字节数。这个问题并不太严重，在此就不再详述了。下面的代码提供了一个示例。

char *ptr = (char *)malloc(10);char name[20] ;memcpy (name,ptr,20); // Problem begins here

在本例中，memcpy 操作尝试从 ptr 读取 20 个字节，但是后者仅被分配了 10 个字节。这还会导致不希望的输出。

内存泄漏

内存泄漏可能真正令人讨厌。下面的列表描述了一些导致内存泄漏的场景。

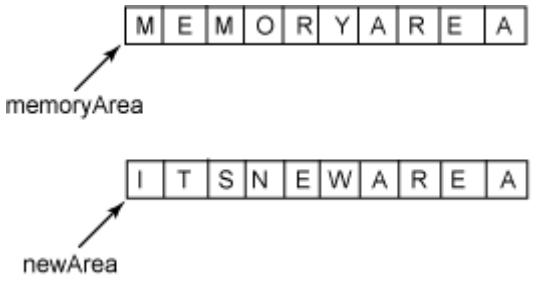
重新赋值

我将使用一个示例来说明重新赋值问题。

char *memoryArea = malloc(10);char *newArea = malloc(10);

这向如下面的图 4 所示的内存位置赋值。

图 4. 内存位置

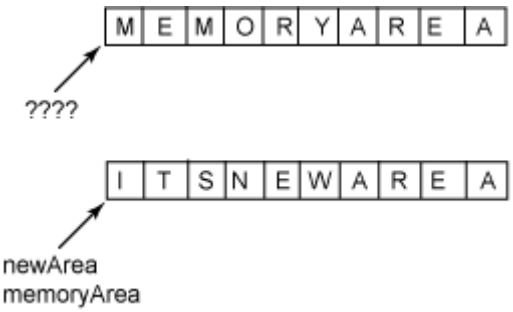


memoryArea 和 newArea 分别被分配了 10 个字节，它们各自的内容如图 4 所示。如果某人执行如下所示的语句（指针重新赋值）..... memoryArea = newArea;

则它肯定会在该模块开发的后续阶段给您带来麻烦。

在上面的代码语句中，开发人员将 memoryArea 指针赋值给 newArea 指针。结果，memoryArea 以前所指向的内存位置变成了孤立的，如下面的图 5 所示。它无法释放，因为没有指向该位置的引用。这会导致 10 个字节的内存泄漏。

图 5. 内存泄漏



在对指针赋值前，请确保内存位置不会变为孤立的。

首先释放父块

假设有一个指针 memoryArea，它指向一个 10 字节的内存位置。该内存位置的第三个字节又指向某个动态分配的 10 字节的内存位置，如图 6 所示。

图 6. 动态分配的内存

free(memoryArea)

如果通过调用 free 来释放了 memoryArea，则 newArea 指针也会因此而变得无效。newArea 以前所指向的内存位置无法释放，因为已经没有指向该位置的指针。换句话说，newArea 所指向的内存位置变为了孤立的，从而导致了内存泄漏。

每当释放结构化的元素，而该元素又包含指向动态分配的内存位置的指针时，应首先遍历子内存位置（在此例中为 newArea），并从那里开始释放，然后再遍历回父节点。

这里的正确实现应该为：

free(memoryArea->newArea);free(memoryArea);

返回值的不正确处理

有时，某些函数会返回对动态分配的内存的引用。跟踪该内存位置并正确地处理它就成为了 calling 函数的职责。

char *func ( ){ return malloc(20); // make sure to memset this location to '\0'...}

void callingFunc ( ){ func ( ); // Problem lies here}

在上面的示例中，callingFunc() 函数中对 func() 函数的调用未处理该内存位置的返回地址。结果，func() 函数所分配的 20 个字节的块就丢失了，并导致了内存泄漏。

归还您所获得的

在开发组件时，可能存在大量的动态内存分配。您可能会忘了跟踪所有指针（指向这些内存位置），并且某些内存段没有释放，还保持分配给该程序。

始终要跟踪所有内存分配，并在任何适当的时候释放它们。事实上，可以开发某种机制来跟踪这些分配，比如在链表节点本身中保留一个计数器（但您还必须考虑该机制的额外开销）。

访问空指针

访问空指针是非常危险的，因为它可能使您的程序崩溃。始终要确保您不是在访问空指针。

总结

本文讨论了几种在使用动态内存分配时可以避免的陷阱。要避免内存相关的问题，良好的实践是：

始终结合使用 memset 和 malloc，或始终使用 calloc。

每当向指针写入值时，都要确保对可用字节数和所写入的字节数进行交叉核对。

在对指针赋值前，要确保没有内存位置会变为孤立的。

每当释放结构化的元素（而该元素又包含指向动态分配的内存位置的指针）时，都应首先遍历子内存位置并从那里开始释放，然后再遍历回父节点。

始终正确处理返回动态分配的内存引用的函数返回值。

每个 malloc 都要有一个对应的 free。

确保您不是在访问空指针。

2. C 语言难点分析整理

这篇文章主要是介绍一些在复习 C 语言的过程中笔者个人认为比较重点的地方，较好的掌握这些重点会使对 C 的运用更加得心应手。此外会包括一些细节、易错的地方。涉及的主要内容包括：变量的作用域和存储类别、函数、数组、字符串、指针、文件、链表等。一些最基本的概念在此就不多作解释了，仅希望能有只言片语给同是 C 语言初学者的学习和上机过程提供一点点的帮助。

变量作用域和存储类别：

了解了基本的变量类型后，我们要进一步了解它的存储类别和变量作用域问题。

变量类别	子类别
局部变量	静态变量（离开函数，变量值仍保留）
	自动变量
	寄存器变量
全局变量	静态变量（只能在本文件中用）
	非静态变量（允许其他文件使用）

换一个角度

变量类别	子类别
静态存储变量	静态局部变量（函数）
	静态全局变量（本文件）
	非静态全局/外部变量（其他文件引用）
动态存储变量	自动变量
	寄存器变量
	形式参数

extern 型的存储变量在处理多文件问题时常能用到，在一个文件中定义 extern 型的变量即说明这个变量用的是其他文件的。顺便说一下，笔者在做课设时遇到 out of memory 的错误，于是改成做多文件，再把它 include 进来（注意自己写的*.h 要用“”不用<>），能起到一定的效用。static 型的在读程序写结果的试题中是个考点。多数时候整个程序会出现多个定义的变量在不同的函数中，考查在不同位置同一变量的值是多少。主要是遵循一个原则，只要本函数内没有定义的变量就用全局变量（而不是 main 里的），全局变量和局部变量重名时局部变量起作用，当然还要注意静态与自动变量的区别。

函数：

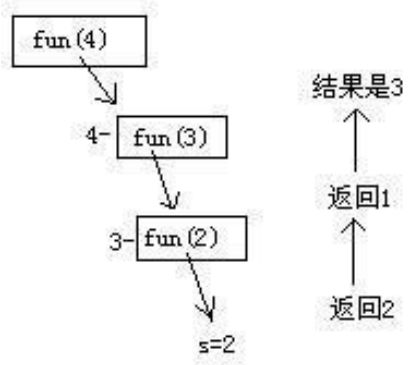
对于函数最基本的理解是从那个叫 main 的单词开始的，一开始总会觉得把语句一并写在 main 里不是挺好的么，为什么偏择出去。其实这是因为对函数还不够熟练，否则函数的运用会给我们编程带来极大的便利。我们要知道函数的返回值类型，参数的类型，以及调用函数时的形式。事先的函数说明也能起到一个提醒的好作用。所谓形参和实参，即在调用函数时写在括号里的就是实参，函数本身用的就是

形参，在画流程图时用平行四边形表示传参。

函数的另一个应用例子就是递归了，笔者开始比较头疼的问题，反应总是比较迟钝，按照老师的方法，把递归的过程耐心准确的逐级画出来，学习的效果还是比较好的，会觉得这种递归的运用是挺巧的，事实上，著名的八皇后、汉诺塔等问题都用到了递归。

例子：

```
longfun(intn)
{
longs;
if(n==1||n==2) s=2;
  elses=n-fun(n-1);
returns;
}
main()
{
printf("%ld",fun(4));
}
```



数组：

分为一维数组和多维数组，其存储方式画为表格的话就会一目了然，其实就是把相同类型的变量有序的放在一起。因此，在处理比较多的数据时（这也是大多数的情况）数组的应用范围是非常广的。

具体的实际应用不便举例，而且绝大多数是与指针相结合的，笔者个人认为学习数组在更大程度上是为学习指针做一个铺垫。作为基础的基础要明白几种基本操作：即数组赋值、打印、排序（冒泡排序法和选择排序法）、查找。这些都不可避免的用到循环，如果觉得反应不过来，可以先一点点的把循环展开，就会越来越熟悉，以后自己编写一个功能的时候就会先找出内在规律，较好的运用了。另外数组做参数时，一维的[]里可以是空的，二维的第一个[]里可以是空的但是第二个[]中必须规定大小。

冒泡法排序函数：

```
voidbubble(inta[],intn)
{
inti,j,k;
for(i=1,i<n;i++)
  for(j=0;j<n-i-1;j++)
```

```
  if(a[j]>a[j+1])
    {
     k=a[j];
     a[j]=a[j+1];
     a[j+1]=k;
     }
}
```

选择法排序函数：

```
voidsort(inta[],intn)
{
inti,j,k,t;
for(i=0,i<n-1;i++)
  {
  k=i;
  for(j=i+1;j<n;j++)
    if(a[k]<a[j])k=j;
    if(k!=i)
      {
      t=a[i];
      a[i]=a[k];
      a[k]=t;
      }
  }
}
```

折半查找函数（原数组有序）：

```
voidsearch(inta[],intn,intx)
{
intleft=0,right=n-1,mid,flag=0;
while((flag==0)&&(left<=right))
  {
  mid=(left+right)/2;
  if(x==a[mid])
    {
    printf("%d%d",x,mid);
    flag =1;
```

```
        }
        elseif(x<a[mid])right=mid-1;
                elseleft=mid+1;
      }
    }
```

相关常用的算法还有**判断回文，求阶乘，Fibanacci 数列，任意进制转换，杨辉三角形计算**等等。

字符串：

字符串其实就是一个数组（指针），在 scanf 的输入列中是不需要在前面加“&”符号的，因为字符数组名本身即代表地址。值得注意的是字符串末尾的”，如果没有的话，字符串很有可能会不正常的打印。另外就是字符串的定义和赋值问题了，笔者有一次的比较综合的上机作业就是字符串打印老是乱码，上上下下找了一圈问题，最后发现是因为

```
char*name;
```

而不是

```
charname[10];
```

前者没有说明指向哪儿，更没有确定大小，导致了乱码的错误，印象挺深刻的。

另外，字符串的赋值也是需要注意的，如果是用字符指针的话，既可以定义的时候赋初值，即

```
char*a="Abcdefg";
```

也可以在赋值语句中赋值，即

```
char*a;
a="Abcdefg";
```

但如果是用字符数组的话，就只能在定义时整体赋初值，即 char a[5]={"abcd"};而不能在赋值语句中整体赋值。

常用字符串函数列表如下，要会自己实现：

函数作用	函数调用形式	备注
字符串拷贝函数	strcpy(char*,char *)	后者拷贝到前者
字符串追加函数	strcat(char*,char *)	后者追加到前者后，返回前者，因此前者空间要足够大
字符串比较函数	strcmp(char*,char *)	前者等于、小于、大于后者时，返回 0、正值、负值。注意，不是比较长度，是比较字符 ASCII 码的大小，可用于按姓名字母排序等。
字符串长度	strlen(char *)	返回字符串的长度，不包括".转义字符算一个字符。

字符串型->整型	atoi(char *)	
整型->字符串型	itoa(int,char *,int)	做课设时挺有用的
	sprintf(char *,格式化输入）	赋给字符串，而不打印出来。课设时用也比较方便

注：对字符串是不允许做==或！=的运算的，只能用字符串比较函数

指针：

指针可以说是 C 语言中最关键的地方了，其实这个“指针”的名字对于这个概念的理解是十分形象的。首先要知道，指针变量的值（即指针变量中存放的值）是指针（即地址）。指针变量定义形式中：基本类型 *指针变量名 中的“*”代表的是这是一个指向该基本类型的指针变量，而不是内容的意思。在以后使用的时候，如*ptr=a 时，“*”才表示 ptr 所指向的地址里放的内容是 a。

指针比较典型又简单的一应用例子是两数互换，看下面的程序，

```
swap(intc,intd)
{
intt;
t=c;
c=d;
d=t;
}
main()
{
inta=2,b=3;
swap(a,b);
printf(“%d,%d”,a,b);
}
```

这是不能实现 a 和 b 的数值互换的，实际上只是形参在这个函数中换来换去，对实参没什么影响。现在，用指针类型的数据做为参数的话，更改如下：

```
swap(#3333FF *p1,int*p2)
{
intt;
t=*p1;
*p1=*p2;
*p2=t;
}
main()
{
inta=2,b=3;
```

```
int*ptr1,*ptr2;
ptr1=&a;
ptr2=&b;
swap(prt1,ptr2);
printf(“%d,%d”,a,b);
}
```

这样在 swap 中就把 p1,p2 的内容给换了，即把 a，b 的值互换了。

指针可以执行**增、减运算**，结合++运算符的法则，我们可以看到:

*++s	取指针变量加 1 以后的内容
*s++	取指针变量所指内容后 s 再加 1
(*s)++	指针变量指的内容加 1

指针和数组实际上几乎是一样的，数组名可以看成是一个常量指针，一维数组中 ptr=&b[0]则下面的表示法是等价的：

a[3]等价于*(a+3)

ptr[3]等价于*(ptr+3)

下面看一个用指针来自己实现 atoi（字符串型->整型）函数：

```
intatoi(char*s)
{
intsign=1,m=0;
if(*s=='+'||*s=='-')/*判断是否有符号*/
sign=(*s++=='+')?1:-1;/*用到三目运算符*/
while(*s!='')/*对每一个字符进行操作*/
  {
  m=m*10+(*s-'0');
  s++;/*指向下一个字符*/
  }
returnm*sign;
}
```

指向多维数组的指针变量也是一个比较广泛的运用。例如数组 a[3][4]，a 代表的实际是整个二维数组的首地址，即第 0 行的首地址，也就是一个指针变量。而 a+1 就不是简单的在数值上加上 1 了，它代表的不是 a[0][1]，而是第 1 行的首地址，&a[1][0]。

指针变量常用的用途还有把指针作为参数传递给其他函数，即**指向函数的指针**。

看下面的几行代码：

```
    voidInput(ST *);
    voidOutput(ST *);
    voidBubble(ST *);
    voidFind(ST *);
    voidFailure(ST *);
    /*函数声明：这五个函数都是以一个指向 ST 型（事先定义过）结构的指针变量作为参数，无返回值。*/


    void(*process[5])(ST *)={Input,Output,Bubble,Find,Failure};
    /*process 被调用时提供 5 种功能不同的函数共选择(指向函数的指针数组）*/


    printf("\nChoose:\n?");
    scanf("%d",&choice);
    if(choice>=0&&choice<=4)
    (*process[choice])(a);/*调用相应的函数实现不同功能*;/
```

总之，指针的应用是非常灵活和广泛的，不是三言两语能说完的，上面几个小例子只是个引子，实际编程中，会逐渐发现运用指针所能带来的便利和高效率。

3. C 语言难点

这篇文章主要是介绍一些在复习 C 语言的过程中笔者个人认为比较重点的地方，较好的掌握这些重点会使对 C 的运用更加得 心应手。此外会包括一些细节、易错的地方。涉及的主要内容包括：变量的作用域和存储类别、函数、数组、字符串、指针、文件、链表等。一些最基本的概念在此 就不多作解释了，仅希望能有只言片语给同是 C 语言初学者的学习和上机过程提供一点点的帮助。

变量作用域和存储类别：

了解了基本的变量类型后，我们要进一步了解它的存储类别和变量作用域问题。

变量类别	子类别
局部变量	静态变量（离开函数，变量值仍保留）
	自动变量
	寄存器变量
全局变量	静态变量（只能在本文件中用）
	非静态变量（允许其他文件使用）

换一个角度

变量类别	子类别
静态存储变量	静态局部变量（函数）
	静态全局变量（本文件）
	非静态全局/外部变量（其他文件引用）
动态存储变量	自动变量
	寄存器变量
	形式参数

extern 型的存储变量在处理多文件问题时常能用到，在一个文件中定义 extern 型的变量即说明这个变量用的是其他文件的。顺便说一下，笔者在做课设时遇到 out of memory 的错误，于是改成做多文件，再把它 include 进来（注意自己写的*.h 要用“”不用<>），能起到一定的效用。static 型的在读程序写结果的试题中是个考点。多数时候整个程序会出现多个定义的变量在不同的函数中，考查在不同位置同一变量的值是多少。主要是遵循一个原则，只 要本函数内没有定义的变量就用全局变量（而不是 main 里的），全局变量和局部变量重名时局部变量起作用，当然还要注意静态与自动变量的区别。

函数：

对于函数最基本的理解是从那个叫 main 的单词开始的，一开始总会觉得把语句一并 写在 main 里不是挺好的么，为什么偏择出去。其实这是因为对函数还不够熟练，否则函数的运用会给我们编程带来极大的便利。我们要知道函数的返回值类型， 参数的类型，以及调用函数时的形式。事先的函数说明也能起到一个提醒的好作用。所谓形参和实参，即在调用函数时写在括号里的就是实参，函数本身用的就是形 参，在画流程图时用平行四边形表示传参。

函数的另一个应用例子就是递归了，笔者开始比较头疼的问题，反应总是比较迟钝，按照老师的方法，把递归的过程耐心准确的逐级画出来，学习的效果还是比较好的，会觉得这种递归的运用是挺巧的，事实上，著名的八皇后、汉诺塔等问题都用到了递归。

例子：

```
long fun(int n)
{
long s;
if(n==1||n==2) s=2;
  else s=n-fun(n-1);
return s;
}
main()
{
printf("%ld",fun(4));
}
```

数组：

分为一维数组和多维数组，其存储方式画为表格的话就会一目了然，其实就是把相同类型的变量有序的放在一起。因此，在处理比较多的数据时（这也是大多数的情况）数组的应用范围是非常广的。

具体的实际应用不便举例，而且绝大多数是与指针相结合的，笔者个人认为学习数组在 更大程度上是为学习指针做一个铺垫。作为基础的基础要明白几种基本操作：即数组赋值、打印、排序（冒泡排序法和选择排序法）、查找。这些都不可避免的用到 循环，如果觉得反应不过来，可以先一点点的把循环展开，就会越来越熟悉，以后自己编写一个功能的时候就会先找出内在规律，较好的运用了。另外数组做参数 时，一维的[]里可以是空的，二维的第一个[]里可以是空的但是第二个[]中必须规定大小。

冒泡法排序函数：

```
void bubble(int a[],int n)
{
int i,j,k;
for(i=1,i<n;i++)
  for(j=0;j<n-i;j++)
  if(a[j]>a[j+1])
    {
     k=a[j];
     a[j]=a[j+1];
     a[j+1]=k;
     }
}
```

选择法排序函数：

```
void sort(int a[],int n)
{
int i,j,k,t;
for(i=0,i<n-1;i++)
  {
  k=i;
  for(j=i+1;j<n;j++)
    if(a[k]<a[j]) k=j;
    if(k!=i)
      {
      t=a[i];
      a[i]=a[k];
      a[k]=t;
      }
  }
}
```

折半查找函数（原数组有序）：

```
void search(int a[],int n,int x)
{
int left=0,right=n-1,mid,flag=0;
while((flag==0)&&(left<=right))
  {
  mid=(left+right)/2;
  if(x==a[mid])
    {
    printf("%d%d",x,mid);
    flag =1;
    }
    else if(x<a[mid]) right=mid-1;
            else left=mid+1;
  }
}
```

相关常用的算法还有**判断回文，求阶乘，Fibanacci 数列，任意进制转换，杨辉三角形计算**等等。

字符串：

字符串其实就是一个数组（指针），在 scanf 的输入列中是不需要在前面加 “&”符号的，因为字符

数组名本身即代表地址。值得注意的是字符串末尾的'\0'，如果没有的话，字符串很有可能会不正常的打印。另外就是字符串 的定义和赋值问题了，笔者有一次的比较综合的上机作业就是字符串打印老是乱码，上上下下找了一圈问题，最后发现是因为

```
char *name;
```

而不是

```
char name[10];
```

前者没有说明指向哪儿，更没有确定大小，导致了乱码的错误，印象挺深刻的。

另外，字符串的赋值也是需要注意的，如果是用字符指针的话，既可以定义的时候赋初值，即

```
char *a="Abcdefg";
```

也可以在赋值语句中赋值，即

```
char *a;
a="Abcdefg";
```

但如果是用字符数组的话，就只能在定义时整体赋初值，即 char a[5]={"abcd"};而不能在赋值语句中整体赋值。

常用字符串函数列表如下，要会自己实现：

函数作用	函数调用形式	备注
字符串拷贝函数	strcpy(char*,char *)	后者拷贝到前者
字符串追加函数	strcat(char*,char *)	后者追加到前者后，返回前者，因此前者空间要足够大
字符串比较函数	strcmp(char*,char *)	前者等于、小于、大于后者时，返回 0、正值、负值。注意，不是比较长度，是比较字符 ASCII 码的大小，可用于按姓名字母排序等。
字符串长度	strlen(char *)	返回字符串的长度，不包括'\0'.转义字符算一个字符。
字符串型->整型	atoi(char *)	
整型->字符串型	itoa(int,char *,int)	做课设时挺有用的
	sprintf(char *,格式化输入）	赋给字符串，而不打印出来。课设时用也比较方便

注：对字符串是不允许做==或！=的运算的，只能用字符串比较函数

指针：

指针可以说是 C 语言中最关键的地方了，其实这个“指针”的名字对于这个概念的理解 是十分形象的。首先要知道，指针变量的值（即指针变量中存放的值）是指针（即地址）。指针变量定义形式中：

基本类型 *指针变量名 中的“*”代表的是这是一个指向该基本类型的指针变量，而不是内容的意思。在以后使用的时候，如*ptr=a 时，“*”才表示 ptr 所指向的地址里放的内 容是 a。

指针比较典型又简单的一应用例子是两数互换，看下面的程序，

```
swap(int c,int d)
{
int t;
t=c;
c=d;
d=t;
}
main()
{
int a=2,b=3;
swap(a,b);
printf(“%d,%d”,a,b);
}
```

这是不能实现 a 和 b 的数值互换的，实际上只是形参在这个函数中换来换去，对实参没什么影响。现在，用指针类型的数据做为参数的话，更改如下：

```
swap(#3333FF *p1,int *p2)
{
int t;
t=*p1;
*p1=*p2;
*p2=t;
}
main()
{
int a=2,b=3;
int *ptr1,*ptr2;
ptr1=&a;
ptr2=&b;
swap(prt1,ptr2);
printf(“%d,%d”,a,b);
}
```

这样在 swap 中就把 p1,p2 的内容给换了，即把 a，b 的值互换了。

指针可以执行**增、减运算**，结合++运算符的法则，我们可以看到:

*++s	取指针变量加 1 以后的内容
*s++	取指针变量所指内容后 s 再加 1
(*s)++	指针变量指的内容加 1

指针和数组实际上几乎是一样的，数组名可以看成是一个常量指针，一维数组中 ptr=&b[0]则下面的表示法是等价的：

a[3]等价于*(a+3)

ptr[3]等价于*(ptr+3)

下面看一个用指针来自己实现 atoi（字符串型->整型）函数：

```
int atoi(char *s)
{
int sign=1,m=0;
if(*s=='+'||*s=='-') /*判断是否有符号*/
sign=(*s++=='+')?1:-1; /*用到三目运算符*/
while(*s!='\0') /*对每一个字符进行操作*/
  {
  m=m*10+(*s-'0');
  s++; /*指向下一个字符*/
  }
return m*sign;
}
```

指向多维数组的指针变量也是一个比较广泛的运用。例如数组 a[3][4]，a 代表的实际是整个二维数组的首地址，即第 0 行的首地址，也就是一个指针变量。而 a+1 就不是简单的在数值上加上 1 了，它代表的不是 a[0][1]，而是第 1 行的首地址，&a[1][0]。

指针变量常用的用途还有把指针作为参数传递给其他函数，即**指向函数的指针**。

看下面的几行代码：

```
void Input(ST *);
void Output(ST *);
void Bubble(ST *);
void Find(ST *);
void Failure(ST *);
/*函数声明：这五个函数都是以一个指向 ST 型（事先定义过）结构的指针变量作为参数，无返回值。*/
```

```
void (*process[5])(ST *)={Input,Output,Bubble,Find,Failure};
/*process 被调用时提供 5 种功能不同的函数共选择(指向函数的指针数组）*/

printf("\nChoose:\n?");
scanf("%d",&choice);
if(choice>=0&&choice<=4)
(*process[choice])(a); /*调用相应的函数实现不同功能*;/
```

总之，指针的应用是非常灵活和广泛的，不是三言两语能说完的，上面几个小例子只是个引子，实际编程中，会逐渐发现运用指针所能带来的便利和高效率。

文件：

函数调用形式	说明
fopen("路径","打开方式")	打开文件
fclose(FILE *)	防止之后被误用
fgetc(FILE *)	从文件中读取一个字符
fputc(ch,FILE *)	把 ch 代表的字符写入这个文件里
fgets(FILE *)	从文件中读取一行
fputs(FILE *)	把一行写入文件中
fprintf(FILE *,"格式字符串",输出表列）	把数据写入文件
fscanf(FILE *,"格式字符串",输入表列）	从文件中读取
fwrite（地址，sizeof（），n，FILE *）	把地址中 n 个 sizeof 大的数据写入文件里
fread（地址，sizeof（），n，FILE *）	把文件中 n 个 sizeof 大的数据读到地址里
rewind（FILE *）	把文件指针拨回到文件头
fseek（FILE *，x，0/1/2）	移动文件指针。第二个参数是位移量，0 代表从头移，1 代表从当前位置移，2 代表从文件尾移。
feof(FILE *)	判断是否到了文件末尾

文件打开方式	说明
r	打开只能读的文件

w	建立供写入的文件，如果已存在就抹去原有数据
a	打开或建立一个把数据追加到文件尾的文件
r+	打开用于更新数据的文件
w+	建立用于更新数据的文件，如果已存在就抹去原有数据
a+	打开或建立用于更新数据的文件，数据追加到文件尾

注：以上用于文本文件的操作，如果是二进制文件就在上述字母后加“b”。

我们用文件最大的目的就是能让数据保存下来。因此在要用文件中数据的时候，就是要 把数据读到一个结构（一般保存数据多用结构，便于管理）中去，再对结构进行操作即可。例如，文件 aa.data 中存储的是 30 个学生的成绩等信息，要遍历 这些信息，对其进行成绩输出、排序、查找等工作时，我们就把这些信息先读入到一个结构数组中，再对这个数组进行操作。如下例：

```
#include<stdio.h>
#include<stdlib.h>
#define N 30
typedef struct student /*定义储存学生成绩信息的数组*/
{
char *name;
int chinese;
int maths;
int phy;
int total;
}ST;
main()
{
ST a[N]; /*存储 N 个学生信息的数组*/
FILE *fp;
void (*process[3])(ST *)={Output,Bubble,Find}; /*实现相关功能的三个函数*/
int choice,i=0;
Show();
printf("\nChoose:\n?");
scanf("%d",&choice);
while(choice>=0&&choice<=2)
  {
  fp=fopen("aa.dat","rb");
  for(i=0;i<N;i++)
    fread(&a[i],sizeof(ST),1,fp); /*把文件中储存的信息逐个读到数组中去*/
```

```
  fclose(fp);
  (*process[choice])(a); /*前面提到的指向函数的指针，选择操作*/
  printf("\n");
  Show();
  printf("\n?");
  scanf("%d",&choice);
  }
}
void Show()
{
printf("\n****Choices:****\n0.Display the data form\n1.Bubble it according to the total score\n2.Search\n3.Quit!\n");
}
void Output(ST *a) /*将文件中存储的信息逐个输出*/
{
int i,t=0;
printf("Name Chinese Maths Physics Total\n");
for(i=0;i<N;i++)
  {
  t=a[i].chinese+a[i].maths+a[i].phy;
  a[i].total=t;
  printf("%4s%8d%8d%8d%8d\n",a[i].name,a[i].chinese,a[i].maths,a[i].phy,a[i].total);
  }
}
void Bubble(ST *a) /*对数组进行排序，并输出结果*/
{
int i,pass;
ST m;
for(pass=0;pass<N-1;pass++)
  for(i=0;i<N-1;i++)
    if(a[i].total<a[i+1].total)
      {
      m=a[i]; /*结构互换*/
      a[i]=a[i+1];
      a[i+1]=m;
      }
Output(a);
```

```
}
void Find(ST *a)
{
int i,t=1;
char m[20];
printf("\nEnter the name you want:");
scanf("%s",m);
for(i=0;i<N;i++)
  if(!strcmp(m,a[i].name)) /*根据姓名匹配情况输出查找结果*/
  {
  printf("\nThe result is:\n%s, Chinese:%d,
Maths:%d,    Physics:%d,Total:%d\n",m,a[i].chinese,a[i].maths,a[i].phy,a[i].total);
  t=0;
  }
if(t)
  printf("\nThe name is not in the list!\n");
}
```

链表：

链表是 C 语言中另外一个难点。牵扯到结点、动态分配空间等等。用结构作为链表的结点是非常适合的，例如：

```
struct node
{
int data;
struct node *next;
};
```

其中 next 是指向自身所在结构类型的指针，这样就可以把一个个结点相连，构成链表。

链表结构的一大优势就是动态分配存储，不会像数组一样必须在定义时确定大小，造成不必要的浪费。用 malloc 和 free 函数即可实现开辟和释放存储单元。其中，malloc 的参数多用 sizeof 运算符计算得到。

链表的基本操作有：**正、反向建立链表；输出链表；删除链表中结点；在链表中插入结点**等等，都是要熟练掌握的，初学者通过**画图**的方式能比较形象地理解建立、插入等实现的过程。

```
typedef struct node
{
char data;
struct node *next;
}NODE; /*结点*/
```

正向建立链表：

```
NODE *create()
{
char ch='a';
NODE *p,*h=NULL,*q=NULL;
while(ch<'z')
  {
  p=(NODE *)malloc(sizeof(NODE)); /*强制类型转换为指针*/
  p->data=ch;
  if(h==NULL) h=p;
    else q->next=p;
  ch++;
  q=p;
  }
q->next=NULL; /*链表结束*/
return h;
}
```

逆向建立：

```
NODE *create()
{
char ch='a';
NODE *p,*h=NULL;
while(ch<='z')
  {
  p=(NODE *)malloc(sizeof(NODE));
  p->data=ch;
  p->next=h; /*不断地把 head 往前挪*/
  h=p;
  ch++;
  }
return h;
}
```

用递归实现链表逆序输出：

```
void output(NODE *h)
{
if(h!=NULL)
  {
  output(h->next);
  printf("%c",h->data);
  }
}
```

插入结点（已有升序的链表）：

```
NODE *insert(NODE *h,int x)
{
NODE *new,*front,*current=h;
while(current!=NULL&&(current->data<x)) /*查找插入的位置*/
  {
  front=current;
  current=current->next;
  }
new=(NODE *)malloc(sizeof(NODE));
new->data=x;
new->next=current;
if(current==h) /*判断是否是要插在表头*/
  h=new;
else front->next=new;
return h;
}
```

删除结点：

```
NODE *delete(NODE *h,int x)
{
NODE *q,*p=h;
while(p!=NULL&&(p->data!=x))
  {
  q=p;
  p=p->next;
  }
if(p->data==x) /*找到了要删的结点*/
```

```
    {
    if(p==h) /*判断是否要删表头*/
    h=h->next;
      else q->next=p->next;
    free(p); /*释放掉已删掉的结点*/
    }
  return h;
  }
```

4. C/C++实现冒泡排序算法

最简单的排序方法是冒泡排序方法。这种方法的基本思想是，将待排序的元素看作是竖着排列的“气泡”，较小的元素比较轻，从而要往上浮。在冒泡排序算法中我们要对这个“气泡”序列处理若干遍。所谓一遍处理，就是自底向上检查一遍这个序列，并时刻注意两个相邻的元素的顺序是否正确。如果发现两个相邻元素的顺序不对，即“轻”的元素在下面，就交换它们的位置。显然，处理一遍之后，“最轻”的元素就浮到了最高位置；处理二遍之后，“次轻”的元素就浮到了次高位置。在作第二遍处理时，由于最高位置上的元素已是“最轻”元素，所以不必检查。一般地，第 i 遍处理时，不必检查第 i 高位置以上的元素，因为经过前面 i-1 遍的处理，它们已正确地排好序。这个算法可实现如下。

Bubble Sort 程序：

STL C++程序：(VC++6.0 通过)

```
#include "stdafx.h"
#include "iostream.h"

template<class T>
class doit{
private:
int x,y;
T temp;
public:
doit(T* in,int count)
{
for(y=0;y<count-1;y++)
{
for(x=1;x<count-y;x++)
{
if((*(in+x))>(*(in+x-1)))
{
temp=(*(in+x-1));
(*(in+x-1))=(*(in+x));
(*(in+x))=temp;
}
}
}
}
};
```

```
int main()
{
double a[4]={1.1,1.3,1.9,2.2};
doit<double> d(a,4);
for(int i=0;i<4;i++)
{
cout<<a[i]<<endl;
}
return 0;
}
```

C 语言程序：(TC 2.0 通过)

```
void doit(float* in,int count)
{
int x;
int y;
float temp;
for(y=0;y<count-1;y++)
{
for(x=1;x<count-y;x++)
{
if((*(in+x))>(*(in+x-1)))
{
temp=(*(in+x-1));
(*(in+x-1))=(*(in+x));
(*(in+x))=temp;
}
}
}
}
```

5. C++中指针和引用的区别

引用和指针

★ 相同点：

1. 都是地址的概念；

指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名。

★ 区别：

1. 指针是一个实体，而引用仅是个别名；

2. 引用使用时无需解引用(*)，指针需要解引用；

3. 引用只能在定义时被初始化一次，之后不可变；指针可变；

引用“从一而终” ^_^

4. 引用没有 const，指针有 const，const 的指针不可变；

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(所指向的变量或对象的地址)的大小；

typeid(T) == typeid(T&) 恒为真，sizeof(T) == sizeof(T&) 恒为真，

但是当引用作为成员时，其占用空间与指针相同（没找到标准的规定）。

7. 指针和引用的自增(++)运算意义不一样；

★ 联系

1. 引用在语言内部用指针实现（如何实现？）。

2. 对一般应用而言，把引用理解为指针，不会犯严重语义错误。引用是操作受限了的指针（仅容许取内容操作）。

6. const char*, char const*, char*const 的区别

const char*, char const*, char*const 的区别问题几乎是 C++面试中每次都会有的题目。

事实上这个概念谁都有只是三种声明方式非常相似很容易记混。

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。

C++标准规定，const 关键字放在类型或变量名之前等价的。

```
const int n=5;
 //same as belowint const m=10;


const int *p;       //same as below
const (int) * pint const *q;       // (int) const *p


char ** p1;
   //pointer to       pointer to       char
const char **p2;
   //pointer to       pointer to const char
char * const * p3;
   //pointer to const pointer to       char
const char * const * p4;
   //pointer to const pointer to const char
char ** const p5;
   //const pointer to       pointer to       char
const char ** const p6;
```

```
    //const pointer to        pointer to const char
char * const * const p7;
  //const pointer to const pointer to        char
const char * const * const p8;
    //const pointer to const pointer to const char
```

7. C 中可变参数函数实现

发布: 2008-7-27 10:22 | 作者: 剑心通明 | 来源: 互联网 | 查看: 7 次

一、 从 printf()开始

原型：int printf(const char * format, ...);

参数 format 表示如何来格式字符串的指令，…

表示可选参数，调用时传递给"..."的参数可有可无，根据实际情况而定。

系统提供了 vprintf 系列格式化字符串的函数，用于编程人员封装自己的 I/O 函数。

```
int vprintf / vscanf(const char * format, va_list ap); // 从标准输入/输出格式化字符串
int vfprintf / vfsacanf(FILE * stream, const char * format, va_list ap); // 从文件流
int vsprintf / vsscanf(char * s, const char * format, va_list ap); // 从字符串

// 例 1：格式化到一个文件流，可用于日志文件

FILE *logfile;
int WriteLog(const char * format, ...)
{
va_list arg_ptr;

va_start(arg_ptr, format);
int nWrittenBytes = vfprintf(logfile, format, arg_ptr);
va_end(arg_ptr);

return nWrittenBytes;
}
```

二、 va 函数的定义和 va 宏

C 语言支持 va 函数，作为 C 语言的扩展--C++同样支持 va 函数，但在 C++中并不推荐使用，C++引入的多态性同样可以实现参数个数可变的函数。不过，C++的重载功能毕竟只能是有限多个可以预见的参数个数。比较而言，C 中的 va 函数则可以定义无穷多个相当于 C++的重载函数，这方面 C++是无能为力的。va 函数的优势表现在使用的方便性和易用性上，可以使代码更简洁。C 编译器为了统一在不同的硬件架构、硬件平台上的实现，和增加代码的可移植性，提供了一系列宏来屏蔽硬件环境不同带来的差异。

ANSI C 标准下，va 的宏定义在 stdarg.h 中，它们有：va_list，va_start()，va_arg()，va_end()。

三、 编译器如何实现 va

简单地说，va 函数的实现就是对参数指针的使用和控制。

typedef char * va_list; // x86 平台下 va_list 的定义

函数的固定参数部分，可以直接从函数定义时的参数名获得；对于可选参数部分，先将指针指向第一个可选参数，然后依次后移指针，根据与结束标志的比较来判断是否已经获得全部参数。因此，va 函数中结束标志必须事先约定好，否则，指针会指向无效的内存地址，导致出错。

这里，移动指针使其指向下一个参数，那么移动指针时的偏移量是多少呢，没有具体答案，因为这里涉及到内存对齐（alignment）问题，内存对齐跟具体使用的硬件平台有密切关系，比如大家熟知的 32 位 x86 平台规定所有的变量地址必须是 4 的倍数(sizeof(int) = 4)。va 机制中用宏_INTSIZEOF(n)来解决这个问题，没有这些宏，va 的可移植性无从谈起。

首先介绍宏_INTSIZEOF(n)，它求出变量占用内存空间的大小，是 va 的实现的基础。

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

#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )          //第一个可选参数地址

#define va_arg(ap,t) ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) ) //下一个参数地址

#define va_end(ap)     \

( ap=va_list0 )                                  // 将指针置为无效
```

1.va_arg 身兼二职：返回当前参数，并使参数指针指向下一个参数。

初看 va_arg 宏定义很别扭，如果把它拆成两个语句，可以很清楚地看出它完成的两个职责。

```
#define va_arg(ap,t)     ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) ) //下一个参数地址
// 将( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )拆成：
/* 指针 ap 指向下一个参数的地址 */
1）.  ap += _INTSIZEOF(t);              // 当前，ap 已经指向下一个参数了
```

/* ap 减去当前参数的大小得到当前参数的地址，再强制类型转换后返回它的值 */

2）. return *(t *)(ap - _INTSIZEOF(t))

回想到 printf/scanf 系列函数的%d %s 之类的格式化指令，我们不难理解这些它们的用途了- 明示参数强制转换的类型。

(注：printf/scanf 没有使用 va_xxx 来实现，但原理是一致的。)

2．va_end 很简单，仅仅是把指针作废而已。

#define va_end(ap) (ap = (va_list)0) // x86 平台

8. C 程序内存中组成部分

发布: 2008-7-21 16:39 | 作者: 剑心通明 | 来源: 互联网 | 查看: 13 次

由于历史原因，C 程序一直由下列几部分组成：

? 正文段

这是由 CPU 执行的机器指令部分。通常，正文段是可共享的，所以即使是经常执行的程序(如文本编辑程序、C 编译程序、shell 等)在存储器中也只需有一个副本，另外，正文段常常是只读的，以防止程序由于意外事故而修改其自身的指令。

? 初始化数据段

通常将此段称为数据段，它包含了程序中需赋初值的变量。例如， C 程序中任何函数之外的说明：

int maxcount = 99;

使此变量以初值存放在初始化数据段中。

? 非初始化数据段

通常将此段称为 bss 段，这一名称来源于早期汇编程序的一个操作符，意思是“block started by symbol（由符号开始的块）”，在程序开始执行之前，内核将此段初始化为 0。函数外的说明：

long sum[1000] ;

使此变量存放在非初始化数据段中。

? 栈

自动变量以及每次函数调用时所需保存的信息都存放在此段中。每次函数调用时，其返回地址、以及调用者的环境信息（例如某些机器寄存器）都存放在栈中。然后，新被调用的函数在栈上为其自动和临时变量分配存储空间。通过以这种方式使用栈， C 函数可以递归调用。

? 堆

通常在堆中进行动态存储分配。由于历史上形成的惯例，堆位于非初始化数据段顶和栈底之间。

9. C 编程拾粹

发布: 2008-7-27 09:08 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

A：多维数组初始化，如

```
 test[2][3][4]=...{
            ...{
                    ...{000,001,002,003}
                    ...{010,011,012,013}
                    ...{020,021,022,023}
            }
            ...{
                    ...{100,101,102,103}
                    ...{110,111,112,113}
                    ...{120,121,122,123}
            }
}
```

这种初始化风格结构清晰，给人的感觉是数组元素的对应位置准确，如果需要对数组中某几个元素赋值时，这种结构就会显得很合理了。同时最重要的是它的可读性很强，便于维护。其中的花括号起到一种定界的作用。

B：看这两个表达式：

test[i]

*(test+i)

指针比引用数组下标的效率可能要高，但是引用数组下标不可能比指针的效率高。在一般的情况下，我们多采用指针代替数组，但是如果以牺牲程序的可读性和可维护性为代价去换取高效率这不是一个好想法，同时这种做法也违背了软件工程的思想。

C：一点技巧：

CODE:

```
/* find keyword in source file*/
#include<stdio.h>
#include<string.h>
#define null 0
int main()
{    int look_upTarget(char const *t,char const *s[]);
     char const *t="target";
     char const *source[]={
          "hacker",
          "blog",
```

```
            "baby",
            "good",
            "test",
            "target",
            null/*在这里我们用了一个空指针，那么在 look_upTarget 函数中就不需要知道和计算 source 中元素的个数了*/
        };
        printf("%d",look_upTarget(t,source));
        getch();
    }

int look_upTarget(char const *t,char const *s[])
{

    for(;*s!=null;s++)
        if(strcmp(t,*s)==0)
            return 1;
    return -1;
}
```

总结：1,数组名作为一个指针常量，一般来说指向数组的第一个元素的地址，但是也有一些特殊情况，如 sizeof(source)和 sizeof(source[0]),sizeof(source[1])等，

他们分别计算数组 source 所占用的内存空间和每一个数组元素所占用的空间。

2,在一般情况下，我们都将函数形式参数的中的指针类型定义为 const 类型，防止间接寻址对主函数中的值做了不可预料的修改。

10. C 语言中实现数组的动态增长

发布: 2008-7-28 16:40 | 作者: 剑心通明 | 来源: 互联网 | 查看: 17 次

在 c 语言中实现数组的动态增长

原理：在 c 语言中数组下标访问可以看成指针偏移访问

1、对表进行检查，看看它是否真的已满

2、如果表确实已满，使用 realloc()函数扩展表的长度，

并进行检查，确保 realloc()操作成功进行。

3、在表中增加所需要的项目。

代码如下：

```
int current_element=0;
int total_element=128;
char *dynamic=malloc(total_element);
char *ptr;

void add_element(char c)
{
 if(current_element==total_element-1)
 {
   total_element *=2;
   ptr=(char*)realloc(dynamic,total_element);
   if(ptr==NULL)
   {
     printf("can't expand the table!\n");
     return -1;
   }
   else
     dynamic=ptr;
 }
 current_element++;
 dynamic[current_element]=c;
}
```

注：在实践中，不要把 realloc()函数的返回值直接赋给字符指针，

如果 realloc()函数失败，它会使该指针的值变成 NULL，这样就无法

对现有的表进行访问。

11. C 语言中的位运算

在计算机程序中，数据的位是可以操作的最小数据单 位，理论上可以用“位运算”来完成所有的运算和操作。一般的位操作是用来控制硬件的，或者做数据变换使用，但是，灵活的位操作可以有效地提高程序运行的效 率。C 语言提供了位运算的功能， 这使得 C 语言也能像汇编语言一样用来编写系统程序。

位运算符 C 语言提供了六种位运算符：

& 按位与

| 按位或

^ 按位异或

~ 取反

<< 左移

>> 右移

1. 按位与运算 按位与运算符"&"是双目运算符。其功能是参与运算的两数各对应的二进位相与。只有对应的两个二进位均为 1 时，结果位才为 1 ，否则为 0。参与运算的数以补码方式出现。

例如：9&5 可写算式如下：00001001 (9 的二进制补码)&00000101 (5 的二进制补码) 00000001 (1 的二进制补码)可见 9&5=1。

按位与运算通常用来对某些位清 0 或保留某些位。例如把 a 的高八位清 0 ， 保留低八位， 可作 a&255 运算 (255 的二进制数为 0000000011111111)。

```
main(){
int a=9,b=5,c;
c=a&b;
printf("a=%d\nb=%d\nc=%d\n",a,b,c);
}
```

应用：

a. 清零特定位 (mask 中特定位置 0，其它位为 1，s=s&mask)

b. 取某数中指定位 (mask 中特定位置 1，其它位为 0，s=s&mask)

2. 按位或运算 按位或运算符“|”是双目运算符。其功能是参与运算的两数各对应的二进位相或。只要对应的二个二进位有一个为 1 时，结果位就为 1。参与运算的两个数均以补码出现。

例如：9|5 可写算式如下：

00001001|00000101

00001101 (十进制为 13)可见 9|5=13

```
main(){
int a=9,b=5,c;
c=a|b;
printf("a=%d\nb=%d\nc=%d\n",a,b,c);
}
```

应用：

常用来将源操作数某些位置 1，其它位不变。 (mask 中特定位置 1，其它位为 0 s=s|mask)

3. 按位异或运算 按位异或运算符“^”是双目运算符。其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为 1。参与运算数仍以补码出现，例如 9^5 可写成算式如下：

00001001^00000101 00001100 (十进制为 12)

```
main(){
int a=9;
a=a^15;
printf("a=%d\n",a);
}
```

应用：

a. 使特定位的值取反 (mask 中特定位置 1，其它位为 0 s=s^mask)

b. 不引入第三变量，交换两个变量的值 (设 a=a1,b=b1)

目 标 操 作 操作后状态

a=a1^b1 a=a^b a=a1^b1,b=b1

b=a1^b1^b1 b=a^b a=a1^b1,b=a1

a=b1^a1^a1 a=a^b a=b1,b=a1

4. 求反运算 求反运算符～为单目运算符，具有右结合性。 其功能是对参与运算的数的各二进位按位求反。例如～9 的运算为： ~(0000000000001001)结果为：1111111111110110

5. 左移运算 左移运算符“<<”是双目运算符。其功能把“<< ”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数， 高位丢弃，低位补 0。 其值相当于乘 2。例如：a<<4 指把 a 的各二进位向左移动 4 位。如 a=00000011(十进制 3)，左移 4 位后为 00110000(十进制 48)。

```
main(){
unsigned a,b;
printf("input a number: ");
scanf("%d",&a);
b=a>&gt5;
b=b&15;
printf("a=%d\tb=%d\n",a,b);
}

main(){
char a='a',b='b';
int p,c,d;
p=a;
p=(p<&lt8)|b;
d=p&0xff;
c=(p&0xff00)>&gt8;
printf("a=%d\nb=%d\nc=%d\nd=%d\n",a,b,c,d);
}
```

6. 右移运算 右移运算符“>>”是双目运算符。其功能是把“>> ”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数。其值相当于除 2。

例如：设 a=15，a>>2 表示把 000001111 右移为 00000011(十进制 3)。对于左边移出的空位，如果是正数则空位补 0，若为负数， 可能补 0 或补 1，这取决于所用的计算机系统。移入 0 的叫逻辑右移，移入 1 的叫算术右移，Turbo C 采用逻辑右移。

```
main(){
   unsigned a,b;
   printf("input a number: ");
   scanf("%d",&a);
   b=a>>5;
   b=b&15;
   printf("a=%d b=%d ",a,b);
}
```

再看一例:

```
main(){
   char a='a',b='b';
   int p,c,d;
   p=a;
```

```
    p=(p<<8)|b;
    d=p&0xff;
    c=(p&0xff00)>>8;
    printf("a=%d b=%d c=%d d=%d ",a,b,c,d);
}
```

12. 浮点数的存储格式：

浮点数的存储格式是符号+阶码(定点整数)+尾数(定点小数)

SEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMM

即 1 位符号位(0 为正，1 为负)，8 位指数位，23 位尾数位

浮点数存储前先转化成 2 的 k 次方形式，即：

f = A1*2^k + A2*2^(k-1) + ... + Ak +... +An*2^(-m) (Ai = {0, 1}, A1 = 1)

如 5.5=2^2 + 2^0 + 2^(-1)

其中的 k 就是指数，加 127 后组成 8 位指数位

5.5 的指数位就是 2+127 = 129 = 10000001

A2A3.....An 就是尾数位，不足 23 位后补 0

所以 5.5 = 01000000101000000000000000000000 = 40A00000

所以，对浮点数*2、/2 只要对 8 位符号位+、- 即可，但不是左移、右移

关于 unsigned int 和 int 的在位运算上的不同，下面有个 CU 上的例子描述的很清楚：

[问题]：这个函数有什么问题吗？

```
//////////////////////////////////////////////////
/**
*  本函数将两个 16 比特位的值连结成为一个 32 比特位的值。
*  参数：sHighBits  高 16 位
* sLowBits  低 16 位
*  返回：32 位值
**/
long CatenateBits16(short sHighBits, short sLowBits)
{
long lResult = 0; /* 32 位值的临时变量*/

/*  将第一个 16 位值放入 32 位值的高 16 位  */
lResult = sHighBits;
lResult <<= 16;

/*  清除 32 位值的低 16 位  */
lResult &= 0xFFFF0000;
```

```
/* 将第二个 16 位值放入 32 位值的低 16 位 */
lResult |= (long)sLowBits;

return lResult;
}
//////////////////////////////////////////////////
```

[问题的发现]：

我们先看如下测试代码：

```
//////////////////////////////////////////////////
int main()
{
short sHighBits1 = 0x7fff;
short sHighBits2 = 0x8f12;
unsigned short usHighBits3 = 0xff12;
short sLowBits1 = 0x7bcd;
long lResult = 0;

printf("[sHighBits1 + sLowBits1] ";

lResult = CatenateBits16(sHighBits1, sLowBits1);
printf("lResult = %08x ", lResult, lResult);

lResult = CatenateBits16(sHighBits2, sLowBits1);
printf("lResult = %08x ", lResult, lResult);

lResult = CatenateBits16(usHighBits3, sLowBits1);
printf("lResult = %08x ", lResult, lResult);
}
//////////////////////////////////////////////////
```

运行结果为：

```
[sHighBits1 + sLowBits1]
lResult = 7fff7bcd
```

lResult = 8f127bcd

lResult = ff127bcd

嗯，运行很正确嘛……于是我们就放心的在自己的程序中使用起这个函数来了。

可是忽然有一天，我们的一个程序无论如何结果都不对！经过 n 个小时的检查和调试，最后终于追踪到……CatenateBits16() ！？它的返回值居然是错的!!

“郁闷!”你说，“这个函数怎么会有问题呢！？”

可是，更郁闷的还在后头呢，因为你把程序中的输入量作为参数，在一个简单的 main()里面单步调试：

```
//////////////////////////////////////////////////
int main()
{
short sHighBits1 = 0x7FFF;
short sHighBits2 = 0x8F12;
unsigned short usHighBits3 = 0x8F12;

short sLowBits1 = 0x7BCD; //你实际使用的参数
short sLowBits2 = 0x8BCD; //你实际使用的参数

long lResult = 0;

printf("[sHighBits1 + sLowBits1] ";

lResult = CatenateBits16(sHighBits1, sLowBits1);
printf("lResult = %08x ", lResult, lResult);

lResult = CatenateBits16(sHighBits2, sLowBits1);
printf("lResult = %08x ", lResult, lResult);

lResult = CatenateBits16(usHighBits3, sLowBits1);
printf("lResult = %08x ", lResult, lResult);

printf(" [sHighBits1 + sLowBits2] ";

lResult = CatenateBits16(sHighBits1, sLowBits2);
```

```
printf("lResult = %08x ", lResult, lResult);

lResult = CatenateBits16(sHighBits2, sLowBits2);
printf("lResult = %08x ", lResult, lResult);

lResult = CatenateBits16(usHighBits3, sLowBits2);
printf("lResult = %08x ", lResult, lResult);

return 0;
}
////////////////////////////////////////////////
```

发现结果竟然是：

```
[sHighBits1 + sLowBits1]
lResult = 7fff7bcd
lResult = 8f127bcd
lResult = 8f127bcd

[sHighBits1 + sLowBits2]
lResult = ffff8bcd //oops!
lResult = ffff8bcd //oops!
lResult = ffff8bcd //oops!
```

前一次还好好的，后一次就 ffff 了？X 档案？

[X 档案的真相]：

注意那两个我们用来当作低 16 位值的 sLowBits1 和 sLowBits2。

已知：
使用 sLowBits1 = 0x7bcd 时，函数返回正确的值；
使用 sLowBits2 = 0x8bcd 时，函数中发生 X 档案。

那么，sLowBits1 与 sLowBits2 有什么区别？

注意了，sLowBits1 和 sLowBits2 都是 short 型（而不是 unsigned short），所以在这里，sLowBits1

代表一个正数值，而 sLowBits2 却代表了一个负数值（因为 8 即是二进制 1000，sLowBits2 最高位是 1）。

再看 CatenateBits16()函数：

```
//////////////////////////////////////////////////
long CatenateBits16(short sHighBits, short sLowBits)
{
long lResult = 0; /* 32 位值的临时变量*/

/* 将第一个 16 位值放入 32 位值的高 16 位 */
lResult = sHighBits;
lResult <<= 16;

/* 清除 32 位值的低 16 位 */
lResult &= 0xFFFF0000;

/* 将第二个 16 位值放入 32 位值的低 16 位 */
lResult |= (long)sLowBits; //注意这一句!!!!

return lResult;
}
//////////////////////////////////////////////////
```

如果我们在函数中用

```
printf("sLowBits = %04x ", sLowBits);
```

打印传入的 sLowBits 值，会发现

sLowBits = 0x7bcd 时，打印结果为

```
sLowBits = 7bcd
```

而 sLowBits = 0x8bcd 时，打印结果为

```
sLowBits = ffff8bcd
```

是的，即使用%04x 也打印出 8 位十六进制。

因此，我们看出来了：

当 sLowBits = 0x8bcd 时，函数中 "lResult |= (long)sLowBits;" 这一句执行，会先将 sLowBits 转换为

0xffff8bcd

再与 lResult 做或运算。由于现在 lResult 的值为 0xXXXX0000 （其中 XXXX 是任何值），所以显然，无论 sHighBits 是什么值，最后结果都会是

0xffff8bcd

而当 sLowBits = 0x7bcd 时，函数中 "lResult |= (long)sLowBits;" 这一句执行，会先将 sLowBits 转换为

0x00007bcd

再与 lResult 做或运算。这样做或运算出来的结果当然就是对的。

也就是说，CatenateBits16()在 sLowBits 的最高位为 0 的时候表现正常，而在最高位为 1 的时候出现偏差。

[教训：在某些情况下作位运算和位处理的时候，考虑使用无符号数值——因为这个时候往往不需要处理符号。即使你需要的有符号的数值，那么也应该考虑自行在调用 CatenateBits16()前后做转换——毕竟在位处理中，有符号数值相当诡异！]

下面这个 CatenateBits16()版本应该会好一些：

```
//////////////////////////////////////////////////
unsigned long CatenateBits16(unsigned short sHighBits, unsigned short sLowBits)
{
long lResult = 0;

/* 将第一个 16 位值放入 32 位值的高 16 位 */
lResult = sHighBits;
lResult <<= 16;

/* 清除 32 位值的低 16 位 */
```

```
lResult &= 0xFFFF0000;

/* 将第二个 16 位值放入 32 位值的低 16 位 */
lResult |= (long)sLowBits & 0x0000FFFF;

return lResult;
}
//////////////////////////////////////////////////
```

注意其中的 "lResult |= (long)sLowBits & 0x0000FFFF;"。事实上，现在即使我们把 CatenateBits16() 函数的参数（特别是 sLowBits）声明为 short，结果也会是对的。

如果有一天你把一只兔子扔给一只老虎，老虎把兔子吃了，第二天把一只老鼠扔给它，它又吃了，那么说明第一天你看错了：它本来就是一只猫。

13. 位域

有些信息在存储时，并不需要占用一个完整的字节， 而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态， 用一位二进位即可。为了节省存储空间，并使处理简便， C 语言又提供了一种数据结构，称为“位域”或“位段”。所谓“位域”是把一个字节中的二进位划分为几 个不同的区域， 并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。 这样就可以把几个不同的对象用一个字节的二进制位域来表示。 一、位域的定义和位域变量的说明位域定义与结构定义相仿，其形式为：

struct 位域结构名

{ 位域列表 };

其中位域列表的形式为： 类型说明符 位域名：位域长度

例如：

```
struct bs
{
int a:8;
int b:2;
int c:6;
};
```

位域变量的说明与结构变量说明的方式相同。 可采用先定义后说明，同时定义说明或者直接说明这三种方式。例如：

```
struct bs
{
int a:8;
int b:2;
int c:6;
}data;
```

说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。对于位域的定义尚有以下几点说明：

1. 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs
{
unsigned a:4
unsigned :0 /*空域*/
unsigned b:4 /*从下一单元开始存放*/
unsigned c:4
}
```

在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，

c 占用 4 位。

2. 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进位。

3. 位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k
{
int a:1
int :2 /*该 2 位不能使用*/
int b:3
int c:2
};
```

从以上分析可以看出，位域在本质上就是一种结构类型， 不过其成员是按二进位分配的。

二、位域的使用位域的使用和结构成员的使用相同，其一般形式为： 位域变量名•位域名 位域允许用各种格式输出。

```
main(){
struct bs
{
unsigned a:1;
unsigned b:3;
unsigned c:4;
} bit,*pbit;
bit.a=1;
bit.b=7;
bit.c=15;
printf("%d,%d,%d\n",bit.a,bit.b,bit.c);
pbit=&bit;
pbit-&gta=0;
pbit-&gtb&=3;
pbit-&gtc|=1;
printf("%d,%d,%d\n",pbit-&gta,pbit-&gtb,pbit-&gtc);
}
```

上例程序中定义了位域结构 bs，三个位域为 a,b,c。说明了 bs 类型的变量 bit 和指向 bs 类型的指针变量 pbit。这表示位域也是可以使用指针的。

程 序的 9、10、11 三行分别给三个位域赋值。(应注意赋值不能超过该位域的允许范围)程序第 12 行以整型量格式输出三个域的内容。第 13 行把位域变量 bit 的地址送给指针变量 pbit。第 14 行用指针方式给位域 a 重新赋值，赋为 0。第 15 行使用了复合的位运算符"&="， 该行相当于：
pbit-&gtb=pbit-&gtb&3 位域 b 中原有值为 7，与 3 作按位与运算的结果为 3(111&011= 011,十进制值为

3)。同样，程序第 16 行中使用了复合位运算"|="， 相当于： pbit-&gtc=pbit-&gtc|1 其结 果为 15。程序第 17 行用指针方式输出了这三个域的值。

类型定义符 t y p e d e f

C 语言不仅提供了丰富的数据类型，而 且还允许由用户自己定义类型说明符，也就是说允许由用户为数据类型取“别名”。 类型定义符 typedef 即可用来完成此功能。例如，有整型量 a,b,其 说明如下：int aa,b; 其中 int 是整型变量的类型说明符。int 的完整写法为 integer，

为了增加程序的可读性，可把整型说明符 用 typedef 定义为： 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;它等效于： int 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 char NAME[20]; 表示 NAME 是字符数组类型，数组长度为 20。

然后可用 NAME 说明变量，如： NAME a1,a2,s1,s2;完全等效于： char a1[20],a2[20],s1[20],s2[20]

又如：

typedef struct stu{ char name[20];

int age;

char sex;

} STU;

定义 STU 表示 stu 的结构类型，然后可用 STU 来说明结构变量： STU body1,body2;

typedef 定义的一般形式为： typedef 原类型名 新类型名 其中原类型名中含有定义部分，新类型名一般用大写表示， 以

便于区别。在有时也可用宏定义来代替 typedef 的功能，但是宏定义是由预处理完成的，而 typedef 则是在编译时完成的，后者更为灵活方便。

本章小结

1. 枚举是一种基本数据类型。枚举变量的取值是有限的，枚举元素是常量，不是变量。

2. 枚举变量通常由赋值语句赋值，而不由动态输入赋值。枚举元素虽可由系统或用户定义一个顺序值，但枚举元素和整数并不相同，它们属于不同的类型。因此，也不能用 printf 语句来输出元素值(可输出顺序值)。

3. 位运算是C 语言的一种特殊运算功能， 它是以二进制位为单位进行运算的。位运算符只有逻辑运算和移位运算两类。位运算符可以与赋值符一起组成复合赋值符。如&=,|=,^=,>>=,<<=等。

4. 利用位运算可以完成汇编语言的某些功能，如置位，位清零，移位等。还可进行数据的压缩存储和并行运算。

5. 位域在本质上也是结构类型，不过它的成员按二进制位分配内存。其定义、说明及使用的方式都与结构相同。

6. 位域提供了一种手段，使得可在高级语言中实现数据的压缩，节省了存储空间，同时也提高了程序的效率。

7. 类型定义 typedef 向用户提供了一种自定义类型说明符的手段，照顾了用户编程使用词汇的习惯，又增加了程序的可读性。

C 语言之 Main 函数返回值问题

发布: 2008-7-21 15:55 | 作者: 剑心通明 | 来源: 互联网 | 查看: 1 次

很多人甚至市面上的一些书籍，都使用了 void main( ) ，其实这是错误的。C/C++ 中从来没有定义过 void main( ) 。C++ 之父 Bjarne Stroustrup 在他的主页上的 FAQ 中明确地写着 The definition void main( ) { /* ... */ } is not and never has been C++, nor has it even been C.（ void main( ) 从来就不存在于 C++ 或者 C ）。下面我分别说一下 C 和 C++ 标准中对 main 函数的定义。

“The C programming Language（《C 程序设计语言》）用的就是 main( )。”--- 这是因为第一版的 C 语言只有一种类型，那就是 int，没有 char，没有 long，没有 float，……既然只有一种类型，那么就可以不写，后来的改进版为了兼容以前的代码于是规定：不明确标明返回值的，默认返回值为 int，也就是说 main()等同于 int main()，而不是等同于 void main()。在 C99 中，标准要求编译器至少给 main() 这种用法来个警告。

1. C

在 C89 中，main( ) 是可以接受的。Brian W. Kernighan 和 Dennis M. Ritchie 的经典巨著 The C programming Language 2e（《C 程序设计语言第二版》）用的就是 main( )。不过在最新的 C99 标准中，只有以下两种定义方式是正确的：

int main(void)

int main(int argc, char *argv[])

（参考资料：ISO/IEC 9899:1999 (E) Programming languages — C 5.1.2.2.1 Program startup）

当然，我们也可以做一点小小的改动。例如：char *argv[] 可以写成 char **argv；argv 和 argc 可以改成别的变量名（如 intval 和 charval），不过一定要符合变量的命名规则。

如果不需要从命令行中获取参数，请用 int main(void) ；否则请用 int main(int argc, char *argv[]) 。

main 函数的返回值类型必须是 int ，这样返回值才能传递给程序的激活者（如操作系统）。

如果 main 函数的最后没有写 return 语句的话，C99 规定编译器要自动在生成的目标文

件中（如 exe 文件）加入 return 0; ，表示程序正常退出。不过，我还是建议你最好在 main 函数的最后加上 return 语句，虽然没有这个必要，但这是一个好的习惯。注意，vc6 不会在目标文件中加入 return 0; ，大概是因为 vc6 是 98 年的产品，所以才不支持这个特性。现在明白我为什么建议你最好加上 return 语句了吧！不过，gcc3.2（Linux 下的 C 编译器）会在生成的目标文件中加入 return 0; 。

2. C++

C++98 中定义了如下两种 main 函数的定义方式：

int main( )

int main(int argc, char *argv[])

（参考资料：ISO/IEC 14882(1998-9-01)Programming languages — C++ 3.6 Start and termination）

int main( ) 等同于 C99 中的 int main(void) ；int main(int argc, char *argv[]) 的用法也和 C99 中定义的一样。同样，main 函数的返回值类型也必须是 int。如果 main 函数的末尾没写 return 语句，C++98 规定编译器要自动在生成的目标文件中加入 return 0; 。同样，vc6 也不支持这个特性，但是 g++3.2（Linux 下的 C++ 编译器）支持。

3. 关于 void main

在 C 和 C++ 中，不接收任何参数也不返回任何信息的函数原型为“void foo(void);”。可能正是因为这个，所以很多人都误认为如果不需要程序返回值时可以把 main 函数定义成 void main(void) 。然而这是错误的！main 函数的返回值应该定义为 int 类型，C 和 C++ 标准中都是这样规定的。虽然在一些编译器中，void main 可以通过编译（如 vc6），但并非所有编译器都支持 void main ，因为标准中从来没有定义过 void main 。g++3.2 中如果 main 函数的返回值不是 int 类型，就根本通不过编译。而 gcc3.2 则会发出警告。所以，如果你想你的程序拥有很好的可移植性，请一定要用 int main .

总而言之：

void main 主函数没有返回值

main 默认为 int 型，即 int main()， 返回整数。

注意，新标准不允许使用默认返回值，即 int 不能省，而且对应 main 函数不再支持 void 型返回值，因此为了使程序有很好的移植性，强烈建议使用：

```
int main()
{

return 0; /* 新标准主函数的返回值这条语句可以省略 */
}
```

14. C 语言函数二维数组传递方法

发布: 2008-7-22 15:39 | 作者: 剑心通明 | 来源: 互联网 | 查看: 11 次

c 语言中经常需要通过函数传递二维数组，有三种方法可以实现，如下：

方法一， 形参给出第二维的长度。

例如：

```
#include <stdio.h>
void func(int n,char    str[][5])
{
  int i;
  for(i = 0; i < n; i++)
    printf("\nstr[%d] = %s\n", i, str[i]);
}

void main()
{
  char* p[3];
  char str[][5] = {"abc","def","ghi"};
  func(3, str);
}
```

方法二，形参声明为指向数组的指针。

例如：

```
#include <stdio.h>
void func(int n,char （*str）[5])
{
  int i;
  for(i = 0; i < n; i++)
    printf("\nstr[%d] = %s\n", i, str[i]);
}

void main()
```

```
{
  char* p[3];
  char str[][5] = {"abc","def","ghi"};
  func(3, str);
}
```

方法三，形参声明为指针的指针。

例如：

```
#include <stdio.h>
void func(int n,char **str)
{
  int i;
  for(i = 0; i < n; i++)
    printf("\nstr[%d] = %s\n", i, str[i]);
}
void main()
{
  char* p[3];
  char str[][5] = {"abc","def","ghi"};
  p[0] = &str[0][0];
  p[1] = str[1];
  p[2] = str[2];
        func(3, p);
}
```

15. C 语言复杂表达式的执行步骤

发布: 2008-7-28 16:31 | 作者: 剑心通明 | 来源: 互联网 | 查看: 11 次

近日在 CSDN 上闲逛的时候，注意到一个帖子：

((*strDest++=*strSrc++)!='\0'); 哪儿前辈可以解释下里面具体执行的步骤呢？

对于这样的表达式，我们通常会有这样三种看法：

1.这种写法不但没有错误(当然也没有 BUG),而且写法紧凑。

2.这种写法虽然没有错误，但是不够直观，理解起来有点麻烦，可能还会导致理解错误。

3.这种写法中存在未定义的地方，执行结果可能是错误的。

粗略来看，这三种说法都有点道理。我顿时有了刨根问底的兴趣，想对这个问题进行一次深入的分析。对于这种组合表达式，在分析的时候我们应该抓住两个关键的概念：优先级(Precedence)和关联性(Associativity)。

1.优先级(Precedence)。优先级决定了那些表达式的值先被评估，那些表达式的值后被评估。通常情况下，优先级高的表达式的值先被评估出来后，然后用评估的结果再去评估那些优先级低的表达式。所以如果我们将优先级搞反了，评估出来的结果是错误的。

2.(Associativity)。对于二目表达式，关联性决定了左边的表达式还是右边的表达式先被评估，先被评估出来的结果再用来评估另外的表达式。

再抓住这两个关键的同时，我们还应该分清什么是表达式的值，什么是变量的值。我们在评估表达式的时候，我们感兴趣的是表达式的值，而不是构成表达式的某些变量的值。在很多情况下，表达式的值和某些变量的值是一致的，所以我们很容易混淆表达式的值和变量的值。要知道，在有些情况下，表达式的值并不和某些变量的值相同。

有了上面的理论来武装我们，对表达式的分析就显得游刃有余了：

1. 很明显，上面的表达式是一个组合表达式。组合表达式由子表达式组成，子表达式又可能是组合表达式，这样就形成了一个树状的数据结构。对表达式的评估就类似于对树结点的遍历。首先我们应该注意到"()"操作符，它具有最高的优先级，所以从整体来看，整个表达式应该是个"!="操作。"!="左边又是一个组合表达式，而右边是一个常量"\0'，很明显下面的工作就是评估(*strDest++=*strSrc++)。

2.在这一步，我们要对表达式(*strDest++=*strSrc++)进行评估。由于赋值表达式具有较低的优先级，所以表达式　又可以写成：(*strDest++) = (*strSrc++)，所以整个表达式是个"="操作，"="左边又是一个组合表达式，右边也是一个组合表达式，这里就需要从关联性来判断左边还是右边也被评估。由于"="的关联性是从右到左，所以(*strSrc++)先被评估，(*strDest++)后被评估。

2.1 在这一步，我们要对表达式(*strSrc++)进行评估。由于"++"的优先级大于"*"，所以表达式又可以写成：*(strSrc++)。我们要先对表达式 strSrc++进行评估，然后用表达式的值再去评估*(strSrc++)的值。对于表达式 strSrc++,这里要需要注意区分变量的值和表达式的值。对于"后增 1"表达式，表达式的值是变量 strSrc 的值，然后变量 strSrc 的值会"加 1"，也就是说表达式的值是 strSrc 变化前的值，而 strSrc 的值会发生变化。值得注意的是，我们知道 strSrc 的值会发生变化，但是我们却不知道 strSrc 的值发生变化的具体时间，这个变化具体的执行时间由编译器决定了，这就决定了任何依赖 strSrc 的表达式的值是不确定的，具体的值依赖编译器的实现。完成了对 strSrc++的评估后，取值操作符就对表达式的值所对应的内存空间进行取值操作。

2.2 在这一步，我们要对表达式(*strDest++)进行评估。具体的评估的分析完全和 2.1 中的分析一致。

2.3 在这一步，我们要对表达式(*strDest++) = (*strSrc++)进行评估，这是个赋值表达式，将右表达式的值赋给左边表达式的值。值得注意的是，对于赋值表达式，表达式本身的值等于左边子表达式的值。

3.由于"!="表达式左边的子表达式的值已经被评估出来了，下面就执行"!="操作。"!="表达式的是一个布尔值。

通过以上深入的分析，我们知道这个表达式完成了以下多个功能：

1.对于指针 strDest, strSrc，将 strSrc 所指的内存空间的值赋给由 strDest 所指的内存空间。

2.判断赋值后的 strDest 所指的内存空间的指是否等于 0。

3.对于指针 strDest,strSrc,他们的值分别加 1，即指向下一个元素。

我们可以看出，一个表达式完成了三个功能，表达式写的确实"相当紧凑"。而且这个表达式的值是可以确定的，因为所有的分析都是建立在 C 标准的基础上。对于能否在实践的代码中使用这样的代码，这就智者见智了，关键一点就是要遵循项目的代码规范。

16. C 语言字符串函数大全

发布: 2008-7-27 10:29 | 作者: 剑心通明 | 来源: 互联网 | 查看: 209 次

函数名: stpcpy

功　能: 拷贝一个字符串到另一个

用　法: char *stpcpy(char *destin, char *source);

程序例:

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

int main(void)
{
    char string[10];
    char *str1 = "abcdefghi";

    stpcpy(string, str1);
    printf("%s\n", string);
    return 0;
}
```

函数名: strcat

功　能: 字符串拼接函数

用　法: char *strcat(char *destin, char *source);

程序例:

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

int main(void)
{
    char destination[25];
    char *blank = " ", *c = "C++", *Borland = "Borland";
```

```
    strcpy(destination, Borland);
    strcat(destination, blank);
    strcat(destination, c);

    printf("%s\n", destination);
    return 0;
}
```

函数名: strchr
功　能: 在一个串中查找给定字符的第一个匹配之处\
用　法: char *strchr(char *str, char c);
程序例:

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

int main(void)
 {
    char string[15];
    char *ptr, c = 'r';

    strcpy(string, "This is a string");
    ptr = strchr(string, c);
    if (ptr)
        printf("The character %c is at position: %d\n", c, ptr-string);
    else
        printf("The character was not found\n");
    return 0;
 }
```

函数名: strcmp
功　能: 串比较

用　法: int strcmp(char *str1, char *str2);

看 Asic 码，str1>str2，返回值 > 0；两串相等，返回 0

程序例:

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

int main(void)
 {
     char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
     int ptr;

     ptr = strcmp(buf2, buf1);
     if (ptr > 0)
         printf("buffer 2 is greater than buffer 1\n");
     else
         printf("buffer 2 is less than buffer 1\n");

     ptr = strcmp(buf2, buf3);
     if (ptr > 0)
         printf("buffer 2 is greater than buffer 3\n");
     else
         printf("buffer 2 is less than buffer 3\n");

     return 0;
 }
```

函数名: strncmpi

功　能: 将一个串中的一部分与另一个串比较, 不管大小写

用　法: int strncmpi(char *str1, char *str2, unsigned maxlen);

程序例:

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

```
int main(void)
{
    char *buf1 = "BBB", *buf2 = "bbb";
    int ptr;

    ptr = strcmpi(buf2, buf1);

    if (ptr > 0)
        printf("buffer 2 is greater than buffer 1\n");

    if (ptr < 0)
        printf("buffer 2 is less than buffer 1\n");

    if (ptr == 0)
        printf("buffer 2 equals buffer 1\n");

    return 0;
}
```

函数名: strcpy
功　能: 串拷贝
用　法: char *strcpy(char *str1, char *str2);
程序例:

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

int main(void)
 {
    char string[10];
    char *str1 = "abcdefghi";

    strcpy(string, str1);
    printf("%s\n", string);
    return 0;
```

```
}
```

函数名: strcspn

功　能: 在串中查找第一个给定字符集内容的段

用　法: int strcspn(char *str1, char *str2);

程序例:

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

int main(void)
{
    char *string1 = "1234567890";
    char *string2 = "747DC8";
    int length;

    length = strcspn(string1, string2);
    printf("Character where strings intersect is at position %d\n", length);

    return 0;
}
```

函数名: strdup

功　能: 将串拷贝到新建的位置处

用　法: char *strdup(char *str);

程序例:

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

```
int main(void)
 {
     char *dup_str, *string = "abcde";

     dup_str = strdup(string);
     printf("%s\n", dup_str);
     free(dup_str);

     return 0;
 }
```

函数名: stricmp
功　能: 以大小写不敏感方式比较两个串
用　法: int stricmp(char *str1, char *str2);
程序例:

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

int main(void)
{
    char *buf1 = "BBB", *buf2 = "bbb";
    int ptr;

    ptr = stricmp(buf2, buf1);

    if (ptr > 0)
        printf("buffer 2 is greater than buffer 1\n");

    if (ptr < 0)
        printf("buffer 2 is less than buffer 1\n");

    if (ptr == 0)
        printf("buffer 2 equals buffer 1\n");
```

```
    return 0;
}
```

函数名: strerror
功　能: 返回指向错误信息字符串的指针
用　法: char *strerror(int errnum);
程序例:

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

int main(void)
{
    char *buffer;
    buffer = strerror(errno);
    printf("Error: %s\n", buffer);
    return 0;
}
```

函数名: strcmpi
功　能: 将一个串与另一个比较, 不管大小写
用　法: int strcmpi(char *str1, char *str2);
程序例:

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

int main(void)
{
    char *buf1 = "BBB", *buf2 = "bbb";
    int ptr;

    ptr = strcmpi(buf2, buf1);
```

```
    if (ptr > 0)
        printf("buffer 2 is greater than buffer 1\n");

    if (ptr < 0)
        printf("buffer 2 is less than buffer 1\n");

    if (ptr == 0)
        printf("buffer 2 equals buffer 1\n");

    return 0;
}
```

函数名: strncmp
功　能: 串比较
用　法: int strncmp(char *str1, char *str2, int maxlen);
程序例:

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

int    main(void)

{
    char *buf1 = "aaabbb", *buf2 = "bbbccc", *buf3 = "ccc";
    int ptr;

    ptr = strncmp(buf2,buf1,3);
    if (ptr > 0)
        printf("buffer 2 is greater than buffer 1\n");
    else
        printf("buffer 2 is less than buffer 1\n");

    ptr = strncmp(buf2,buf3,3);
    if (ptr > 0)
```

```
        printf("buffer 2 is greater than buffer 3\n");
    else
        printf("buffer 2 is less than buffer 3\n");

    return(0);
}
```

函数名: strncmpi
功　能: 把串中的一部分与另一串中的一部分比较, 不管大小写
用　法: int strncmpi(char *str1, char *str2);
程序例:

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

int main(void)
{
    char *buf1 = "BBBccc", *buf2 = "bbbccc";
    int ptr;

    ptr = strncmpi(buf2,buf1,3);

    if (ptr > 0)
        printf("buffer 2 is greater than buffer 1\n");

    if (ptr < 0)
        printf("buffer 2 is less than buffer 1\n");

    if (ptr == 0)
        printf("buffer 2 equals buffer 1\n");

    return 0;
}
```

函数名: strncpy

功　能: 串拷贝

用　法: char *strncpy(char *destin, char *source, int maxlen);

程序例:

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

int main(void)
{
    char string[10];
    char *str1 = "abcdefghi";

    strncpy(string, str1, 3);
    string[3] = '\0';
    printf("%s\n", string);
    return 0;
}
```

函数名: strnicmp

功　能: 不注重大小写地比较两个串

用　法: int strnicmp(char *str1, char *str2, unsigned maxlen);

程序例:

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

int main(void)
{
    char *buf1 = "BBBccc", *buf2 = "bbbccc";
    int ptr;

    ptr = strnicmp(buf2, buf1, 3);

    if (ptr > 0)
        printf("buffer 2 is greater than buffer 1\n");
```

```
    if (ptr < 0)
        printf("buffer 2 is less than buffer 1\n");

    if (ptr == 0)
        printf("buffer 2 equals buffer 1\n");

    return 0;
}
```

函数名: strnset
功　能: 将一个串中的所有字符都设为指定字符
用　法: char *strnset(char *str, char ch, unsigned n);
程序例:

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

int main(void)
{
    char *string = "abcdefghijklmnopqrstuvwxyz";
    char letter = 'x';

    printf("string before strnset: %s\n", string);
    strnset(string, letter, 13);
    printf("string after    strnset: %s\n", string);

    return 0;
}
```

函数名: strpbrk
功　能: 在串中查找给定字符集中的字符
用　法: char *strpbrk(char *str1, char *str2);

程序例:

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

int main(void)
{
    char *string1 = "abcdefghijklmnopqrstuvwxyz";
    char *string2 = "onm";
    char *ptr;

    ptr = strpbrk(string1, string2);

    if (ptr)
        printf("strpbrk found first character: %c\n", *ptr);
    else
        printf("strpbrk didn't find character in set\n");

    return 0;
}
```

函数名: strrchr
功　能: 在串中查找指定字符的最后一个出现
用　法: char *strrchr(char *str, char c);
程序例:

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

int main(void)
{
    char string[15];
    char *ptr, c = 'r';

    strcpy(string, "This is a string");
```

```
    ptr = strrchr(string, c);
    if (ptr)
        printf("The character %c is at position: %d\n", c, ptr-string);
    else
        printf("The character was not found\n");
    return 0;
}
```

函数名: strrev
功　能: 串倒转
用　法: char *strrev(char *str);
程序例:

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

int main(void)
{
    char *forward = "string";

    printf("Before strrev(): %s\n", forward);
    strrev(forward);
    printf("After strrev():    %s\n", forward);
    return 0;
}
```

函数名: strset
功　能: 将一个串中的所有字符都设为指定字符
用　法: char *strset(char *str, char c);
程序例:

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

```
int main(void)
{
    char string[10] = "123456789";
    char symbol = 'c';

    printf("Before strset(): %s\n", string);
    strset(string, symbol);
    printf("After strset():    %s\n", string);
    return 0;
}
```

函数名: strspn
功　能: 在串中查找指定字符集的子集的第一次出现
用　法: int strspn(char *str1, char *str2);
程序例:

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

int main(void)
{
    char *string1 = "1234567890";
    char *string2 = "123DC8";
    int length;

    length = strspn(string1, string2);
    printf("Character where strings differ is at position %d\n", length);
    return 0;
}
```

函数名: strstr
功　能: 在串中查找指定字符串的第一次出现

用　法: char *strstr(char *str1, char *str2);

程序例:

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

int main(void)
{
    char *str1 = "Borland International", *str2 = "nation", *ptr;

    ptr = strstr(str1, str2);
    printf("The substring is: %s\n", ptr);
    return 0;
}
```

函数名: strtod

功　能: 将字符串转换为 double 型值

用　法: double strtod(char *str, char **endptr);

程序例:

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

int main(void)
{
    char input[80], *endptr;
    double value;

    printf("Enter a floating point number:");
    gets(input);
    value = strtod(input, &endptr);
    printf("The string is %s the number is %lf\n", input, value);
    return 0;
}
```

函数名: strtok

功　能: 查找由在第二个串中指定的分界符分隔开的单词

用　法: char *strtok(char *str1, char *str2);

程序例:

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

int main(void)
{
    char input[16] = "abc,d";
    char *p;

    /* strtok places a NULL terminator
    in front of the token, if found */
    p = strtok(input, ",");
    if (p)      printf("%s\n", p);

    /* A second call to strtok using a NULL
    as the first parameter returns a pointer
    to the character following the token    */
    p = strtok(NULL, ",");
    if (p)      printf("%s\n", p);
    return 0;
}
```

函数名: strtol

功　能: 将串转换为长整数

用　法: long strtol(char *str, char **endptr, int base);

程序例:

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

```
int main(void)
{
    char *string = "87654321", *endptr;
    long lnumber;

    /* strtol converts string to long integer    */
    lnumber = strtol(string, &endptr, 10);
    printf("string = %s    long = %ld\n", string, lnumber);

    return 0;
}
```

函数名: strupr
功　能: 将串中的小写字母转换为大写字母
用　法: char *strupr(char *str);
程序例:

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

int main(void)
{
    char *string = "abcdefghijklmnopqrstuvwxyz", *ptr;

    /* converts string to upper case characters */
    ptr = strupr(string);
    printf("%s\n", ptr);
    return 0;
}
```

函数名: swab
功　能: 交换字节
用　法: void swab (char *from, char *to, int nbytes);

程序例:

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

char source[15] = "rFna koBlrna d";
char target[15];

int main(void)
{
    swab(source, target, strlen(source));
    printf("This is target: %s\n", target);
    return 0;
}
```

17. C 语言宏定义技巧

发布: 2008-7-22 09:16 | 作者: 剑心通明 | 来源: 互联网 | 查看: 7 次

写好 C 语言，漂亮的宏定义很重要，使用宏定义可以防止出错，提高可移植性，可读性，方便性 等等。下面列举一些成熟软件中常用得宏定义。。。。。

1，防止一个头文件被重复包含

```
#ifndef COMDEF_H

#define COMDEF_H

  //头文件内容

#endif
```

2，重新定义一些类型，防止由于各种平台和编译器的不同，而产生的类型字节数差异，方便移植。

```
typedef    unsigned char           boolean;        /* Boolean value type. */


typedef    unsigned long int    uint32;          /* Unsigned 32 bit value */

typedef    unsigned short         uint16;          /* Unsigned 16 bit value */

typedef    unsigned char           uint8;           /* Unsigned 8    bit value */


typedef    signed long int       int32;           /* Signed 32 bit value */

typedef    signed short            int16;           /* Signed 16 bit value */

typedef    signed char             int8;            /* Signed 8    bit value */
```

//下面的不建议使用

```
typedef    unsigned char        byte;          /* Unsigned 8    bit value type. */

typedef    unsigned short       word;          /* Unsinged 16 bit value type. */

typedef    unsigned long        dword;         /* Unsigned 32 bit value type. */


typedef    unsigned char        uint1;         /* Unsigned 8    bit value type. */

typedef    unsigned short       uint2;         /* Unsigned 16 bit value type. */

typedef    unsigned long        uint4;         /* Unsigned 32 bit value type. */


typedef    signed char          int1;          /* Signed 8    bit value type. */

typedef    signed short         int2;          /* Signed 16 bit value type. */

typedef    long int             int4;          /* Signed 32 bit value type. */


typedef    signed long          sint31;        /* Signed 32 bit value */

typedef    signed short         sint15;        /* Signed 16 bit value */

typedef    signed char          sint7;         /* Signed 8    bit value */
```

3，得到指定地址上的一个字节或字

```
#define    MEM_B( x )    ( *( (byte *) (x) ) )

#define    MEM_W( x )    ( *( (word *) (x) ) )
```

4，求最大值和最小值

```
    #define    MAX( x, y ) ( ((x) > (y)) ? (x) : (y) )

    #define    MIN( x, y ) ( ((x) < (y)) ? (x) : (y) )
```

5，得到一个 field 在结构体(struct)中的偏移量

```
#define FPOS( type, field ) \

/*lint -e545 */ ( (dword) &(( type *) 0)-> field ) /*lint +e545 */
```

6,得到一个结构体中 field 所占用的字节数

```
#define FSIZ( type, field ) sizeof( ((type *) 0)->field )
```

7，按照 LSB 格式把两个字节转化为一个 Word

```
#define    FLIPW( ray ) ( (((word) (ray)[0]) * 256) + (ray)[1] )
```

8，按照 LSB 格式把一个 Word 转化为两个字节

```
#define    FLOPW( ray, val ) \

   (ray)[0] = ((val) / 256); \

   (ray)[1] = ((val) & 0xFF)
```

9，得到一个变量的地址（word 宽度）

```
#define    B_PTR( var )    ( (byte *) (void *) &(var) )

#define    W_PTR( var )    ( (word *) (void *) &(var) )
```

10，得到一个字的高位和低位字节

```
#define    WORD_LO(xxx)    ((byte) ((word)(xxx) & 255))

#define    WORD_HI(xxx)    ((byte) ((word)(xxx) >> 8))
```

11，返回一个比 X 大的最接近的 8 的倍数

```
#define RND8( x )            ((((x) + 7) / 8 ) * 8 )
```

12，将一个字母转换为大写

```
#define    UPCASE( c ) ( ((c) >= 'a' && (c) <= 'z') ? ((c) - 0x20) : (c) )
```

13，判断字符是不是 10 进值的数字

```
#define    DECCHK( c ) ((c) >= '0' && (c) <= '9')
```

14，判断字符是不是 16 进值的数字

```
#define    HEXCHK( c ) ( ((c) >= '0' && (c) <= '9') ||\

                        ((c) >= 'A' && (c) <= 'F') ||\

((c) >= 'a' && (c) <= 'f') )
```

15，防止溢出的一个方法

```
#define    INC_SAT( val )    (val = ((val)+1 > (val)) ? (val)+1 : (val))
```

16，返回数组元素的个数

```
#define    ARR_SIZE( a )    ( sizeof( (a) ) / sizeof( (a[0]) ) )
```

17，返回一个无符号数 n 尾的值 MOD_BY_POWER_OF_TWO(X,n)=X%(2^n)

```
#define MOD_BY_POWER_OF_TWO( val, mod_by ) \
```

((dword)(val) & (dword)((mod_by)-1))

18，对于 IO 空间映射在存储空间的结构，输入输出处理

```
#define inp(port)          (*((volatile byte *) (port)))

#define inpw(port)         (*((volatile word *) (port)))

#define inpdw(port)        (*((volatile dword *)(port)))



#define outp(port, val)    (*((volatile byte *) (port)) = ((byte) (val)))

#define outpw(port, val)   (*((volatile word *) (port)) = ((word) (val)))

#define outpdw(port, val) (*((volatile dword *) (port)) = ((dword) (val)))
```

[2005-9-9 添加]

19,使用一些宏跟踪调试

A N S I 标准说明了五个预定义的宏名。它们是：

_ L I N E _

_ F I L E _

_ D A T E _

_ T I M E _

_ S T D C _

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。记住编译程序

也许还提供其它预定义的宏名。

_ L I N E _及_ F I L E _宏指令在有关#l i n e 的部分中已讨论，这里讨论其余的宏名。

_ D AT E _宏指令含有形式为月/日/年的串，表示源文件被翻译到代码时的日期。

源代码翻译到目标代码的时间作为串包含在_ T I M E _中。串形式为时：分：秒。

如果实现是标准的，则宏_ S T D C _含有十进制常量 1。如果它含有任何其它数，则实现是非标准的。

可以定义宏，例如:

当定义了_DEBUG，输出数据信息和所在文件所在行

```
#ifdef _DEBUG
#define DEBUGMSG(msg,date) printf(msg);printf(“%d%d%d”,date,_LINE_,_FILE_)
#else
        #define DEBUGMSG(msg,date)
#endif
```

20，宏定义防止使用是错误

用小括号包含。

例如：#define ADD(a,b) （a+b）

用 do{}while(0)语句包含多语句防止错误

例如：#difne DO(a,b) a+b;\

```
                        a++;
```

应用时：if(….)

DO(a,b); //产生错误

else

解决方法: #difne DO(a,b) do{a+b;\

a++;}while(0)

宏中"#"和"##"的用法

一、一般用法

我们使用#把宏参数变为一个字符串,用##把两个宏参数贴合在一起.

用法:

```
#include<cstdio>
#include<climits>
using namespace std;

#define STR(s)          #s
#define CONS(a,b)    int(a##e##b)

int main()
{
    printf(STR(vck));                  //  输出字符串"vck"
    printf("%d\n", CONS(2,3));    // 2e3  输出:2000
    return 0;
}
```

二、当宏参数是另一个宏的时候

需要注意的是凡宏定义里有用'#'或'##'的地方宏参数是不会再展开.

1, 非'#'和'##'的情况

```
#define TOW            (2)
#define MUL(a,b) (a*b)
```

printf("%d*%d=%d\n", TOW, TOW, MUL(TOW,TOW));

这行的宏会被展开为：

printf("%d*%d=%d\n", (2), (2), ((2)*(2)));

MUL 里的参数 TOW 会被展开为(2).

2, 当有'#'或'##'的时候

```
#define A               (2)
#define STR(s)          #s
#define CONS(a,b)       int(a##e##b)
```

printf("int max: %s\n", STR(INT_MAX)); // INT_MAX #include<climits>

这行会被展开为：

printf("int max: %s\n", "INT_MAX");

printf("%s\n", CONS(A, A)); // compile error

这一行则是：

printf("%s\n", int(AeA));

INT_MAX 和 A 都不会再被展开，然而解决这个问题的方法很简单. 加多一层中间转换宏.

加这层宏的用意是把所有宏的参数在这层里全部展开，那么在转换宏里的那一个宏(_STR)就能得到正确的宏参数.

```
#define A               (2)
#define _STR(s)         #s
#define STR(s)          _STR(s)         //  转换宏
#define _CONS(a,b)      int(a##e##b)
#define CONS(a,b)       _CONS(a,b)      //  转换宏
```

printf("int max: %s\n", STR(INT_MAX)); // INT_MAX,int 型的最大值，为一个变量 #include<climits>

输出为: int max: 0x7fffffff

STR(INT_MAX) --> _STR(0x7fffffff) 然后再转换成字符串；

printf("%d\n", CONS(A, A));

输出为：200

CONS(A, A) --> _CONS((2), (2)) --> int((2)e(2))

三、'#'和'##'的一些应用特例

1、合并匿名变量名

```
#define   ___ANONYMOUS1(type, var, line)   type   var##line
#define   __ANONYMOUS0(type, line)   ___ANONYMOUS1(type, _anonymous, line)
#define   ANONYMOUS(type)   __ANONYMOUS0(type, __LINE__)
```

例：ANONYMOUS(static int); 即: static int _anonymous70; 70 表示该行行号；

```
第一层：ANONYMOUS(static int);   -->   __ANONYMOUS0(static int, __LINE__);
第二层：                                  -->   ___ANONYMOUS1(static int, _anonymous, 70);
第三层：                                  -->   static int   _anonymous70;
```

即每次只能解开当前层的宏，所以__LINE__在第二层才能被解开；

2、填充结构

```
#define   FILL(a)    {a, #a}

enum IDD{OPEN, CLOSE};
typedef struct MSG{
   IDD id;
   const char * msg;
}MSG;

MSG _msg[] = {FILL(OPEN), FILL(CLOSE)};
```

相当于：

```
MSG _msg[] = {{OPEN, "OPEN"},
                    {CLOSE, "CLOSE"}};
```

3、记录文件名

```
#define   _GET_FILE_NAME(f)    #f
#define   GET_FILE_NAME(f)      _GET_FILE_NAME(f)
static char   FILE_NAME[] = GET_FILE_NAME(__FILE__);
```

4、得到一个数值类型所对应的字符串缓冲大小

```
#define   _TYPE_BUF_SIZE(type)   sizeof #type
#define   TYPE_BUF_SIZE(type)    _TYPE_BUF_SIZE(type)
char   buf[TYPE_BUF_SIZE(INT_MAX)];
      -->   char   buf[_TYPE_BUF_SIZE(0x7fffffff)];
      -->   char   buf[sizeof "0x7fffffff"];
```

这里相当于：

```
char   buf[11];
```

18. C 语言实现动态数组

发布: 2008-7-27 10:29 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

test.h

```
#ifndef TEST_H
#define TEST_H
template<typename T>class Array
{
private:
    T* parray;                    //定义动态数组
    int room;                     //存储空间大小的值
public:
    Array()
        :room       (0)
        ,parray     (NULL)
    {
        parray=(int*)malloc(sizeof(int)*room);
    }
    ~Array()
    {
        if(parray!=NULL)
            free(parray);
    }
    int push_back(T m)            //推入一个数组元素
    {
        room++;
        parray=(T*)realloc(parray,sizeof(T)*room);
        *(parray+room-1)=m;
        return *(parray+room-1);
    }
    int sizeof_Array()            //得到数组大小
    {
        return room;
    }
    bool remove(int k)            //删除第 k 个元素
    {
```

```
        printf("删除元素%d 后",k);
        if(k>room-1 || k<0)
        {
            return false;
        }
        else
        {
            memmove(parray+k,parray+k+1,sizeof(T)*(room-1-k));
            room-=1;
            return true;
        }
}
bool insert(int k,T element)            //插入一个元素
{
        printf("插入元素%d 后",k);
        if(k>room || k<0)
        {
            return false;
        }
        else
        {
            room++;
            parray=(T*)realloc(parray,sizeof(T)*room);
            memmove(parray+k+1,parray+k,sizeof(T)*(room-1-k));
            *(parray+k)=element;
        }
}
T query(int k)                          //查询第 k 个元素的值
{
        if(k>room-1 || k<0)
        {
            return -1;
        }
        else
        {
            return *(parray+k);
        }
}
```

```
    void display()                          //显示数组内元素个数
    {
        for(int i=0;i<room;i++)
            cout<<query(i)<<endl;
        cout<<"数组内元素个数"<<sizeof_Array()<<endl;
    }
    T* begin()
    {
        return parray;
    }
    T* end()
    {
        return parray + room;
    }
};
#endif
```

test.cpp

```
#include<iostream>
#include"test.h"

using namespace std;

void main()
{
    Array<int> intArray;
    intArray.push_back(1);
    intArray.push_back(2);
    intArray.push_back(3);
    intArray.push_back(4);
    intArray.push_back(5);
    intArray.push_back(6);
    intArray.push_back(7);
    intArray.push_back(8);
    intArray.push_back(9);
    /*intArray.display();
    intArray.remove(3);
    intArray.display();
    cout<<intArray.begin()<<endl;
```

```
    cout<<intArray.end()<<endl;
    int* ite=NULL;
    for(ite=intArray.begin();ite!=intArray.end();ite++)
    {
        printf("%d ",*ite);
    }
    for(ite=intArray.end();ite!=intArray.begin();ite--)
    {
        printf("%d ",*ite);
    }*/
}
```

19. C 语言笔试-运算符和表达式

发布: 2008-7-27 11:11 | 作者: 剑心通明 | 来源: 互联网 | 查看: 11 次

运算符和表达式

试题 1

设 a = 1,b = 2,c = 3,d = 4,则表达式: a<b?a:c<d?a:d 结果为:

If a = 1,b = 2,c = 3,d = 4,then what is the value of the expersstion a<b?a:c<d?a:d:

A: 4 B: 3 C: 2 D: 1

基本功能题,重在考察程序员对表达式结合性的熟悉程度和对条件运算符的掌握程度.

解题方法:按照表达式自左向右的结合性分布计算表达式的值

a<b?a:c < d?a:d 可写成(a<b?a:c)<d?a:d 先计算 a<b?a:c 结果为 a,在计算 a<d?a:d 结果还是 a

试题 2

设 x 为 int 型变量,则执行以下语句后,x 的值为:

If x is an int value,then what is x value when the following statements are excuted

x = 10; x += x -= x-x;

A: 10 B: 20 C: 30 D: 40

这是一个陷阱题,涉及到的知识点是复合赋值表达式的运算

解题方法:按赋值表达式的右结合性分布计算

x += x -= x-x 可以写成 x += (x -= (x-x))最后,x += 10 就是 20

试题 3

设 x,y,z,t 均为 int 型变量,则执行以下语句后,t 的值为:

If x,y,z,t is int value,then what is t value when the following statements are excuted:

x = y = z = t = 1; t = ++x || ++y && ++z

A: 不定值 B: 2 C: 1 D: 0

此题输入技巧题,考程序员对||运算符的理解

解题方法:在 C 语言中||和&&运算符得到的值为真即为 1,为假即为 0,考虑&&的仅优先级高于||,所以最后计

算||,但对于||来说只要表达式++x 为真则不用再计算后面的,这里++x = 2 显然为真即 1,所以不需计算.

试题 4

设 x,y,z 和 k 都是 int 型变量,初始值都是 0,则执行表达式 x = (y++, z += y, k = z + 2)后,x 的值是:

If x,y,z and k are all int values,the initial values of them are all 0,what is the x value

when the statement "x = (y++, z += y, k = z + 2)" is excuted:

A: 0 B: 1 C: 2 D: 3

此题为基本功能题,考察逗号表达式的求值

解题方法:逗号表达式又叫顺序求值运算符,求解过程是自左至右,依次计算各表达式的值,逗号表达式的值

是最后一个表达式的值,所以如无必要不用计算各个表达式,但这里要依次计算三个表达式才可以,x = 3

试题 5

```
#include <stdio.h>
#define SUB(X, Y)      (X)*Y
void main()
{
  int a = 3, b = 4;
  printf("%d", SUB(a++, b++));
}
```

执行结果是:

what is the output of the program above:

A: 12 B: 13 C: 16 D: 20

陷阱题,考察自增运算和表达式的值

解题方法:把 SUB 的宏展开成(a++)*b++,很显然,(a++)*b = 12(a 先乘以 b,再作自增运算)

试题 6

下面程序的输出是:

What is the output of the following program:

```
main()
{
  enum team(a, b = 4, c, d = c + 10);
  printf("%d,%d,%d,%d", a, b, c, d);
}
```

A: 0 1 2 3 B: 0 4 0 10 C: 0 4 5 15 D: 0 4 4 14

基本题,考察 C 语言的基本数据类型枚举型

枚举型变量非常有用,编程时可一次定义一系列整型常量.一般格式如下:

```
  enum valueName {value0, value1, value2, value3, ..., valuen};
```

value0 的值是 0,以后各常量的值依次递增,如果某个 valueM 用赋值语句被赋值为 M,则 value(M+1)的值就是

M+1(递增一个).此题显然是选 C

```
int main(void)
{
    int a=1, b=2, c=2;
    while (a < b < c) {
        int t;
        t=a; a=b; b=t;
        c--;
    }
    printf("%d\n", c);
    return 0;
}
```

代码很有欺骗性，第一感觉循环不会执行，结果应该是 2，可编译运行后才发现结果是 0。

这是因为 a < b < c 的实际解释为(a < b) < c，如果(a < b)成立，则(a < b)等于 1，否则(a < b)等于 0。

20. C 语言编程准则之稳定篇

发布: 2008-7-22 13:28 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

1. 不要忘记对变量，特别是指针，数组等的初始化过程!!!

2. 不要把 unsigned char(int)的变量或数组当作 char(int)来处理!!!

3. 不要忽略 if,switch 中 else,default 等特殊情况的处理!!!

4. 避免在条件判断 if,while 中出现非 bool 类型结果!!!

5. 尽量避免使用 malloc/free；一旦使用，尽量在同一函数中实现匹配的 malloc 和 free!!!

6. 不要忽略 malloc 失败情况的处理，同样不要忘记 free 后把指针置为 NULL!!!

7. 函数体不宜过大，避免写重复代码，对功能具有独立性的可封装为函数!!!

8. 功能性函数中（即为特定功能而写，可被重复调用的函数），应避免使用全局变量!!!

9. 不要忽略程序编译中出现的任何一个警告!!!

10. 让你的程序代码清楚宜读，变量、函数命名规范!!!

11. 把构架设计放在第一位，做得更细致，给与更多时间，并在需要时继续修改完善!!!

21. C 语言编程常见问题分析

发布: 2008-7-21 19:02 | 作者: 剑心通明 | 来源: 互联网 | 查看: 11 次

C 语言中的一般语法和一些编程技巧在很多书里都讲过了，下面主要讲一些别的书很少讲到、但是又非常重要的问题。这些都是作者在编程过程中总结出的一些经验教训，可以说职业程序员每天编程时都要遇到这些问题。

1.2.1 参数校验问题

在 C 语言的函数中，一般都要对函数的参数进行校验，但是有些情况下不在函数内进行校验，而由调用者在外部校验，到底什么情况下应该在函数内进行校验，什么情况下不需要在函数内进行校验呢？下列原则可供读者参考。

1) 对于需要在大的循环里调用的函数，不需要在函数内对参数进行校验。

例如链表的逐个遍历函数 void *List_EnumNext(LIST *pList)。

在链表的逐个遍历函数里，要不要对 pList 参数的合法性进行校验呢？答案是否定的。为什么呢？因为链表的逐个遍历函数通常是要在一个循环里使用的，比如一个链表有 10 000 个节点，逐个遍历就要遍历 10 000 次。如果上面的函数对参数 pList 进行了校验，那么对整个链表的逐个遍历过程将校验 10 000 次，不如由调用者在调用函数前校验一次就够了。因此，像这种可能频繁地被调用，且在外面校验只要校验一次就够的函数参数是不需要在函数内部进行校验的。

2) 底层的函数调用频度都比较高，一般不校验。

3) 对于调用频度低的函数，参数要校验。

4) 执行时间开销很大的函数，在参数校验相对整个函数来讲占的比例可以忽略不计的情况下，一般最好对函数参数进行校验。

5) 可以大大提升软件的稳定性时，函数参数要进行校验。

1.2.2 return 语句的问题

在函数里，使用 return 语句可能也是一个值得探讨的问题。有些人认为应该让函数的出口尽量的少，因此要减少 return 语句的使用；特别是在函数中有分配资源操作时，在之后的每个 return 语句里都需要进行对应的释放操作，在编程时很容易出现遗漏而出现资源泄漏。但是我们也知道，如果要减少 return 语句，又会带来另外一些问题。首先，很多情况下要减少 return 语句会增加编程的难度；其次，有时候减少了 return 语句又使得大括号嵌套的层数增加不少，使得代码行长度增加不少，很容易因超过 80 字符而使得代码阅读困难。

那么，到底什么情况下可以减少 return 语句，什么情况下又不能减少 return 语句呢？下列原则供读者参考。

1) 对参数进行校验，校验失败，一般要使用 return 语句，这样做可使程序逻辑清晰，阅读也方便，又减少了大括号嵌套的层数。

2) 对于函数内部的不同出错情况，要有不同的 return 语句；否则对外部调用者来说，无法区分出错的真正原因。

3) 对于函数内部的同类出错情况，尽量只使用一个 return 语句，即尽量不要让两个 return 语句返回相同的返回值。

1.2.3 while 循环和 for 循环的问题

1. 两种循环在构造死循环时的区别

用 while 构造死循环时，一般会使用 while(TRUE)来构造死循环；而用 for 来构造死循环时，则使用 for(;;)来构造死循环。这两个死循环的区别是：while 循环里的条件被看成表达式，因此，当用 while

构造死循环时，里面的 TRUE 实际上被看成永远为真的表达式，这种情况容易产生混淆，有些工具软件如 PC-Lint 就会认为出错了，因此构造死循环时，最好使用 for(;;)来进行。

2. 两种循环在普通循环时的区别

对一个数组进行循环时，一般来说，如果每轮循环都是在循环处理完后才讲循环变量增加的话，使用 for 循环比较方便；如果循环处理的过程中就要将循环变量增加时，则使用 while 循环比较方便；还有在使用 for 循环语句时，如果里面的循环条件很长，可以考虑用 while 循环进行替代，使代码的排版格式好看一些。

1.2.4 if 语句的多个判断问题在对参数进行校验时，经常需要校验函数的几个参数，是对每个参数都使用一个单独的 if 语句进行一次校验，还是多个语句都放在一个 if 语句里用逻辑或运算来进行校验呢？还是用实例来说明吧。

例 1-1 参数校验举例。

```
TABLE *CreateTable1( int nRow，int nCol )
{
        if ( nRow > MAX_ROWS )
        {
                return NULL；
        }
        if ( nCol >= MAX_COLS )
        {
                return NULL；
        }
        ......
}
TABLE *CreateTable2( int nRow，int nCol )
{
        if ( nRow > MAX_ROWS || nCol >= MAX_COLS )
        {
                return NULL；
        }
        ......
}
```

在 CreateTable1()和 CreateTable2()两个函数中，到底哪个写法更好呢？答案是第二种写法更优。为什么呢？因为第二种写法中，少了一个 return 语句，编译后，执行代码会比第一种写法小，执行时耗用的内存自然少了，其他方面的性能都是一样的。有兴趣的读者可以将上面两段代码反汇编出来分析一下。

1.2.5 goto 语句问题

goto 语句在结构化编程技术出来后，被当作破坏结构化程序的典型代表，可以说，在结构化程序设计年代，goto 语句就像洪水猛兽一样，程序员都唯恐避之不及；可后来在微软的一些例子程序中经常把 goto 语句用来处理出错，当出错时，goto 到函数要退出的一个 label 那里进行资源释放等操作。那么，

goto 语句是不是只可以用于出错处理，其他地方都不可以用了呢？下列关于使用 goto 语句的原则可以供读者参考。

1) 使用 goto 语句只能 goto 到同一函数内，而不能从一个函数里 goto 到另外一个函数里。

2) 使用 goto 语句在同一函数内进行 goto 时，goto 的起点应是函数内一段小功能的结束处，goto 的目的 label 处应是函数内另外一段小功能的开始处。

3) 不能从一段复杂的执行状态中的位置 goto 到另外一个位置，比如，从多重嵌套的循环判断中跳出去就是不允许的。

1.2.6 switch …case 和 if … else if 的效率区别

许多有关 C 语言的书中都说过，switch … case 的效率比 if … else if 的效率高。其实这个结论并不完全正确，关键看实际代码，少数情况下，if…else if 写法如果得当的话，它的效率是高于 switch…case 的。

例 1-2 if…else if 语句测试举例。

```
void TestIfElse()
{
        unsigned int x ;
        srand( 1 );          /* 初始化随机数发生器 */
        x = rand() % 100;
        if ( x == 0 )
        {
            ......
        }
        else if ( x == 1 )
        {
            ......
        }
        else
        {
            ......
        }
}
```

x 为 0～100 范围内的随机数，当 x 不为 0 和 1 时执行 else 分支，因此上面函数有 98%的概率是执行 else 分支的，而执行 if 分支和 else if 分支的概率各为 1%。在执行 else 分支时，要先执行 if 判断语句，再执行 else if 判断语句后才会执行 else 分支里的内容。因此执行 else 分支需要进行两次判断。如果我们按下列方式将 else 分支的执行改成放到 if 里面去执行，则效率将大大提高。

```
void TestIfElse( )
{
        unsigned int x ;
```

```
        srand( 1 );         /* 初始化随机数发生器 */
        x = rand() % 100;
        if ( x > 1 )
        {
                .....
        }
        else if ( x == 0 )
        {
                .....
        }
        else
        {
                .....
        }
}
```

如果上面的代码用 switch...case 来实现，则效率不如上面的函数，读者可以试验一下。

22. C 语言编程易犯毛病集合

发布: 2008-7-20 23:07 | 作者: 剑心通明 | 来源: 互联网 | 查看: 3 次

C 语言的最大特点是：功能强、使用方便灵活。C 编译的程序对语法检查并不象其它高级语言那么严格，这就给编程人员留下“灵活的余地”，但还是由于这个灵活给程序的调试带来了许多不便，尤其对初学 C 语言的人来说，经常会出一些连自己都不知道错在哪里的错误。看着有错的程序，不知该如何改起，本人通过对 C 的学习，积累了一些 C 编程时常犯的错误，写给各位学员以供参考。

1.书写标识符时，忽略了大小写字母的区别。

```
main()
{
int a=5;
printf("%d",A);
}
```

编译程序把 a 和 A 认为是两个不同的变量名，而显示出错信息。C 认为大写字母和小写字母是两个不同的字符。习惯上，符号常量名用大写，变量名用小写表示，以增加可读性。

2.忽略了变量的类型，进行了不合法的运算。

```
main()
{
float a,b;
printf("%d",a%b);
}
```

%是求余运算，得到 a/b 的整余数。整型变量 a 和 b 可以进行求余运算，而实型变量则不允许进行“求余”运算。

3.将字符常量与字符串常量混淆。

```
char c;
c="a";
```

在这里就混淆了字符常量与字符串常量，字符常量是由一对单引号括起来的单个字符，字符串常量是一对双引号括起来的字符序列。C 规定以“\”作字符串结束标志，它是由系统自动加上的，所以字符串“a”实际上包含两个字符：‘a'和‘\'，而把它赋给一个字符变量是不行的。

4.忽略了“=”与“==”的区别。

在许多高级语言中，用“=”符号作为关系运算符“等于”。如在 BASIC 程序中可以写

```
if (a=3) then …
```

但 C 语言中，“=”是赋值运算符，“==”是关系运算符。如：

if (a==3) a=b;

前者是进行比较，a 是否和 3 相等，后者表示如果 a 和 3 相等，把 b 值赋给 a。由于习惯问题，初学者往往会犯这样的错误。

5.忘记加分号。

分号是 C 语句中不可缺少的一部分，语句末尾必须有分号。

```
a=1
b=2
```

编译时，编译程序在“a=1”后面没发现分号，就把下一行“b=2”也作为上一行语句的一部分，这就会出现语法错误。改错时，有时在被指出有错的一行中未发现错误，就需要看一下上一行是否漏掉了分号。

```
{ z=x+y;
t=z/100;
printf("%f",t);
}
```

对于复合语句来说，最后一个语句中最后的分号不能忽略不写(这是和 PASCAL 不同的)。

6.多加分号。

对于一个复合语句，如：

```
{ z=x+y;
t=z/100;
printf("%f",t);
};
```

复合语句的花括号后不应再加分号，否则将会画蛇添足。

又如：

```
if (a%3==0);
I++;
```

本是如果 3 整除 a，则 I 加 1。但由于 if (a%3==0)后多加了分号，则 if 语句到此结束，程序将执行 I++语句，不论 3 是否整除 a，I 都将自动加 1。

再如：

```
for (I=0;I<5;I++);
{scanf("%d",&x);
printf("%d",x);}
```

本意是先后输入 5 个数，每输入一个数后再将它输出。由于 for()后多加了一个分号，使循环体变为空语句，此时只能输入一个数并输出它。

7.输入变量时忘记加地址运算符“&”。

int a,b;

scanf("%d%d",a,b);

这是不合法的。Scanf 函数的作用是：按照 a、b 在内存的地址将 a、b 的值存进去。“&a”指 a 在内存中的地址。

8.输入数据的方式与要求不符。①scanf("%d%d",&a,&b);

输入时，不能用逗号作两个数据间的分隔符，如下面输入不合法：

3，4

输入数据时，在两个数据之间以一个或多个空格间隔，也可用回车键，跳格键 tab。

②scanf("%d,%d",&a,&b);

C 规定：如果在“格式控制”字符串中除了格式说明以外还有其它字符，则在输入数据时应输入与这些字符相同的字符。下面输入是合法的：

3，4

此时不用逗号而用空格或其它字符是不对的。

3 4 3：4

又如：

scanf("a=%d,b=%d",&a,&b);

输入应如以下形式：

a=3,b=4

9.输入字符的格式与要求不一致。

在用“%c”格式输入字符时，“空格字符”和“转义字符”都作为有效字符输入。

scanf("%c%c%c",&c1,&c2,&c3);

如输入 a b c

字符“a”送给 c1，字符“ ”送给 c2，字符“b”送给 c3，因为%c 只要求读入一个字符，后面不需要用空格作为两个字符的间隔。

10.输入输出的数据类型与所用格式说明符不一致。

例如，a 已定义为整型，b 定义为实型

a=3;b=4.5;

printf("%f%d\n",a,b);

编译时不给出出错信息，但运行结果将与原意不符。这种错误尤其需要注意。

11.输入数据时，企图规定精度。

```
scanf("%7.2f",&a);
```

这样做是不合法的，输入数据时不能规定精度。

12.switch 语句中漏写 break 语句。

例如：根据考试成绩的等级打印出百分制数段。

```
switch(grade)
{ case 'A':printf("85~100\n");
case 'B':printf("70~84\n");
case 'C':printf("60~69\n");
case 'D':printf("<60\n");
default:printf("error\n");
```

由于漏写了 break 语句，case 只起标号的作用，而不起判断作用。因此，当 grade 值为 A 时，printf 函数在执行完第一个语句后接着执行第二、三、四、五个 printf 函数语句。正确写法应在每个分支后再加上“break;”。例如

```
case 'A':printf("85~100\n");break;
```

13.忽视了 while 和 do-while 语句在细节上的区别。

```
(1)main()
{int a=0,I;
scanf("%d",&I);
while(I<=10)
{a=a+I;
I++;
}
printf("%d",a);
}
(2)main()
{int a=0,I;
scanf("%d",&I);
do
{a=a+I;
I++;
}while(I<=10);
printf("%d",a);
}
```

可以看到，当输入 I 的值小于或等于 10 时，二者得到的结果相同。而当 I>10 时，二者结果就不同了。因为 while 循环是先判断后执行，而 do-while 循环是先执行后判断。对于大于 10 的数 while 循环一次也不执行循环体，而 do-while 语句则要执行一次循环体。

14.定义数组时误用变量。

```
int n;
scanf("%d",&n);
int a[n];
```

数组名后用方括号括起来的是常量表达式，可以包括常量和符号常量。即 C 不允许对数组的大小作动态定义。

15.在定义数组时，将定义的“元素个数”误认为是可使的最大下标值。

```
main()
{static int a[10]={1,2,3,4,5,6,7,8,9,10};
printf("%d",a[10]);
}
```

C 语言规定：定义时用 a[10]，表示 a 数组有 10 个元素。其下标值由 0 开始，所以数组元素 a[10]是不存在的。

16.初始化数组时，未使用静态存储。

```
int a[3]={0,1,2};
```

这样初始化数组是不对的。C 语言规定只有静态存储(static)数组和外部存储(exterm)数组才能初始化。应改为：

```
static int a[3]={0,1,2};
```

17.在不应加地址运算符&的位置加了地址运算符。

```
scanf("%s",&str);
```

C 语言编译系统对数组名的处理是：数组名代表该数组的起始地址，且 scanf 函数中的输入项是字符数组名，不必要再加地址符&。应改为：

```
scanf("%s",str);
```

18.同时定义了形参和函数中的局部变量。

```
int max(x,y)
int x,y,z;
{z=x>y?x:y;
return(z);
}
```

形参应该在函数体外定义，而局部变量应该在函数体内定义。应改为：

```
int max(x,y)
int x,y;
{int z;
z=x>y?x:y;
return(z);
}
```

23. C 语言缺陷与陷阱(笔记)

发布: 2008-7-20 22:45 | 作者: 剑心通明 | 来源: 互联网 | 查看: 12 次

C 语言像一把雕刻刀，锋利，并且在技师手中非常有用。和任何锋利的工具一样，C 会伤到那些不能掌握它的人。本文介绍 C 语言伤害粗心的人的方法，以及如何避免伤害。

第一部分研究了当程序被划分为记号时会发生的问题。第二部分继续研究了当程序的记号被编译器组合为声明、表达式和语句时会出现的问题。第三部分研究了由多个部分组成、分别编译并绑定到一起的 C 程序。第四部分处理了概念上的误解：当一个程序具体执行时会发生的事情。第五部分研究了我们的程序和它们所使用的常用库之间的关系。在第六部分中，我们注意到了我们所写的程序也许并不是我们所运行的程序；预处理器将首先运行。最后，第七部分讨论了可移植性问题：一个能在一个实现中运行的程序无法在另一个实现中运行的原因。

词法分析器（lexical analyzer）：检查组成程序的字符序列，并将它们划分为记号（token）一个记号是一个由一个或多个字符构成的序列，它在语言被编译时具有一个（相关地）统一的意义。

C 程序被两次划分为记号，首先是预处理器读取程序，它必须对程序进行记号划分以发现标识宏的标识符。通过对每个宏进行求值来替换宏调用，最后，经过宏替换的程序又被汇集成字符流送给编译器。编译器再第二次将这个流划分为记号。

1.1= 不是 ==：C 语言则是用=表示赋值而用==表示比较。这是因为赋值的频率要高于比较，因此为其分配更短的符号。C 还将赋值视为一个运算符，因此可以很容易地写出多重赋值（如 a = b = c），并且可以将赋值嵌入到一个大的表达式中。

1.2 & 和| 不是&& 和||

1.3 多字符记号

C 语言参考手册说明了如何决定：“如果输入流到一个给定的字符串为止已经被识别为记号，则应该包含下一个字符以组成能够构成记号的最长的字符串” “最长子串原则”

1.4 例外

组合赋值运算符如+=实际上是两个记号。因此，

a + /* strange */ = 1

和

a += 1

是一个意思。看起来像一个单独的记号而实际上是多个记号的只有这一个特例。特别地，

p - > a

是不合法的。它和

p -> a

不是同义词。

另一方面，有些老式编译器还是将=+视为一个单独的记号并且和+=是同义词。

1.5 字符串和字符

包围在单引号中的一个字符只是编写整数的另一种方法。这个整数是给定的字符在实现的对照序列中的一个对应的值。而一个包围在双引号中的字符串，只是编写一个有双引号之间的字符和一个附加的二进制值为零的字符所初始化的一个无名数组的指针的一种简短方法。

使用一个指针来代替一个整数通常会得到一个警告消息（反之亦然），使用双引号来代替单引号也会得到一个警告消息（反之亦然）。但对于不检查参数类型的编译器却除外。

由于一个整数通常足够大，以至于能够放下多个字符，一些 C 编译器允许在一个字符常量中存放多个字符。这意味着用'yes'代替"yes"将不会被发现。后者意味着“分别包含 y、e、s 和一个空字符的四个连续存储器区域中的第一个的地址”，而前者意味着“在一些实现定义的样式中表示由字符 y、e、s 联合构成的一个整数”。这两者之间的任何一致性都纯属巧合。

2 句法缺陷

理解这些记号是如何构成声明、表达式、语句和程序的。

2.1 理解声明

每个 C 变量声明都具有两个部分：一个类型和一组具有特定格式的、期望用来对该类型求值的表达式。

float *g(), (*h)();

表示*g()和(*h)()都是 float 表达式。由于()比*绑定得更紧密，*g()和*(g())表示同样的东西：g 是一个返回指 float 指针的函数，而 h 是一个指向返回 float 的函数的指针。

当我们知道如何声明一个给定类型的变量以后，就能够很容易地写出一个类型的模型（cast）：只要删除变量名和分号并将所有的东西包围在一对圆括号中即可。

float *g();

声明 g 是一个返回 float 指针的函数，所以(float *())就是它的模型。

(*(void(*)())0)();硬件会调用地址为 0 处的子程序

(*0)(); 但这样并不行，因为*运算符要求必须有一个指针作为它的操作数。另外，这个操作数必须是一个指向函数的指针，以保证*的结果可以被调用。需要将 0 转换为一个可以描述“指向一个返回 void 的函数的指针”的类型。(Void(*)())0

在这里，我们解决这个问题时没有使用 typedef 声明。通过使用它，我们可以更清晰地解决这个问题：

typedef void (*funcptr)();// typedef funcptr void (*)();指向返回 void 的函数的指针

(*(funcptr)0)();//调用地址为 0 处的子程序

2.2 运算符并不总是具有你所想象的优先级

绑定得最紧密的运算符并不是真正的运算符：下标、函数调用和结构选择。这些都与左边相关联。

接下来是一元运算符。它们具有真正的运算符中的最高优先级。由于函数调用比一元运算符绑定得更紧密，你必须写(*p)()来调用 p 指向的函数；*p()表示 p 是一个返回一个指针的函数。转换是一元运算符，并且和其他一元运算符具有相同的优先级。一元运算符是右结合的，因此*p++表示*(p++)，而不是(*p)++。

在接下来是真正的二元运算符。其中数学运算符具有最高的优先级，然后是移位运算符、关系运算符、逻辑运算符、赋值运算符，最后是条件运算符。需要记住的两个重要的东西是：

1. 所有的逻辑运算符具有比所有关系运算符都低的优先级。

2. 移位运算符比关系运算符绑定得更紧密，但又不如数学运算符。

乘法、除法和求余具有相同的优先级，加法和减法具有相同的优先级，以及移位运算符具有相同的优先级。

还有就是六个关系运算符并不具有相同的优先级：==和!=的优先级比其他关系运算符要低。

在逻辑运算符中，没有任何两个具有相同的优先级。按位运算符比所有顺序运算符绑定得都紧密，每种与运算符都比相应的或运算符绑定得更紧密，并且按位异或（^）运算符介于按位与和按位或之间。

三元运算符的优先级比我们提到过的所有运算符的优先级都低。

这个例子还说明了赋值运算符具有比条件运算符更低的优先级是有意义的。另外，所有的复合赋值运算符具有相同的优先级并且是自右至左结合的

具有最低优先级的是逗号运算符。赋值是另一种运算符，通常具有混合的优先级。

2.3 看看这些分号！

或者是一个空语句，无任何效果；或者编译器可能提出一个诊断消息，可以方便除去掉它。一个重要的区别是在必须跟有一个语句的 if 和 while 语句中。另一个因分号引起巨大不同的地方是函数定义前面的结构声明的末尾，考虑下面的程序片段：

```
struct foo {
    int x;
}

f() {
    ...
}
```

在紧挨着 f 的第一个}后面丢失了一个分号。它的效果是声明了一个函数 f，返回值类型是 struct foo，这个结构成了函数声明的一部分。如果这里出现了分号，则 f 将被定义为具有默认的整型返回值[5]。

2.4 switch 语句

C 中的 case 标签是真正的标签：控制流程可以无限制地进入到一个 case 标签中。

看看另一种形式，假设 C 程序段看起来更像 Pascal：

```
switch(color) {
case 1: printf ("red");
case 2: printf ("yellow");
case 3: printf ("blue");
}
```

并且假设 color 的值是 2。则该程序将打印 yellowblue，因为控制自然地转入到下一个 printf()的调用。

这既是 C 语言 switch 语句的优点又是它的弱点。说它是弱点，是因为很容易忘记一个 break 语句，从而导致程序出现隐晦的异常行为。说它是优点，是因为通过故意去掉 break 语句，可以很容易实现其他方法难以实现的控制结构。尤其是在一个大型的 switch 语句中，我们经常发现对一个 case 的处理可以简化其他一些特殊的处理。

2.5 函数调用

和其他程序设计语言不同，C 要求一个函数调用必须有一个参数列表，但可以没有参数。因此，如果 f 是一个函数，

```
f();
```

就是对该函数进行调用的语句，而

```
f;
```

什么也不做。它会作为函数地址被求值，但不会调用它[6]。

2.6 悬挂 else 问题

一个 else 总是与其最近的 if 相关联。

3 连接

一个 C 程序可能有很多部分组成，它们被分别编译，并由一个通常称为连接器、连接编辑器或加载器的程序绑定到一起。由于编译器一次通常只能看到一个文件，因此它无法检测到需要程序的多个源文件的内容才能发现的错误。

3.1 你必须自己检查外部类型

假设你有一个 C 程序，被划分为两个文件。其中一个包含如下声明：

int n;

而令一个包含如下声明：

long n;

这不是一个有效的 C 程序，因为一些外部名称在两个文件中被声明为不同的类型。然而，很多实现检测不到这个错误，因为编译器在编译其中一个文件时并不知道另一个文件的内容。因此，检查类型的工作只能由连接器（或一些工具程序如 lint）来完成；如果操作系统的连接器不能识别数据类型，C 编译器也没法过多地强制它。

那么，这个程序运行时实际会发生什么？这有很多可能性：

1. 实现足够聪明，能够检测到类型冲突。则我们会得到一个诊断消息，说明 n 在两个文件中具有不同的类型。

2. 你所使用的实现将 int 和 long 视为相同的类型。典型的情况是机器可以自然地进行 32 位运算。在这种情况下你的程序或许能够工作，好象你两次都将变量声明为 long（或 int）。但这种程序的工作纯属偶然。

3. n 的两个实例需要不同的存储，它们以某种方式共享存储区，即对其中一个的赋值对另一个也有效。这可能发生，例如，编译器可以将 int 安排在 long 的低位。不论这是基于系统的还是基于机器的，这种程序的运行同样是偶然。

4. n 的两个实例以另一种方式共享存储区，即对其中一个赋值的效果是对另一个赋以不同的值。在这种情况下，程序可能失败。

这种情况发生的另一个例子出奇地频繁。程序的某一个文件包含下面的声明：

char filename[] = "etc/passwd";

而另一个文件包含这样的声明：

char *filename;

尽管在某些环境中数组和指针的行为非常相似，但它们是不同的。在第一个声明中，filename 是一个字符数组的名字。尽管使用数组的名字可以产生数组第一个元素的指针，但这个指针只有在需要的时候才产生并且不会持续。在第二个声明中，filename 是一个指针的名字。这个指针可以指向程序员让它指向的任何地方。如果程序员没有给它赋一个值，它将具有一个默认的 0 值（NULL）([译注]实际上，在 C 中一个为初始化的指针通常具有一个随机的值，这是很危险的!)。

这两个声明以不同的方式使用存储区，它们不可能共存。

避免这种类型冲突的一个方法是使用像 lint 这样的工具（如果可以的话）。为了在一个程序的不同编译单元之间检查类型冲突，一些程序需要一次看到其所有部分。典型的编译器无法完成，但 lint 可以。

避免该问题的另一种方法是将外部声明放到包含文件中。这时，一个外部对象的类型仅出现一次[7]。

4 语义缺陷

4.1 表达式求值顺序

一些 C 运算符以一种已知的、特定的顺序对其操作数进行求值。但另一些不能。例如，考虑下面的表达式：

a < b && c < d

C 语言定义规定 a < b 首先被求值。如果 a 确实小于 b，c < d 必须紧接着被求值以计算整个表达式的值。但如果 a 大于或等于 b，则 c < d 根本不会被求值。

要对 a < b 求值，编译器对 a 和 b 的求值就会有一个先后。但在一些机器上，它们也许是并行进行的。

C 中只有四个运算符&&、||、?:和,指定了求值顺序。&&和||最先对左边的操作数进行求值，而右边的操作数只有在需要的时候才进行求值。而?:运算符中的三个操作数：a、b 和 c，最先对 a 进行求值，之后仅对 b 或 c 中的一个进行求值，这取决于 a 的值。,运算符首先对左边的操作数进行求值，然后抛弃它的值，对右边的操作数进行求值[8]。

C 中所有其它的运算符对操作数的求值顺序都是未定义的。事实上，赋值运算符不对求值顺序做出任何保证。

出于这个原因，下面这种将数组 x 中的前 n 个元素复制到数组 y 中的方法是不可行的：

```
i = 0;
while(i < n)
    y[i] = x[i++];
```

其中的问题是 y[i]的地址并不保证在 i 增长之前被求值。在某些实现中，这是可能的；但在另一些实现中却不可能。另一种情况出于同样的原因会失败：

```
i = 0;
while(i < n)
    y[i++] = x[i];
```

而下面的代码是可以工作的：

```
i = 0;
while(i < n) {
    y[i] = x[i];
    i++;
}
```

当然，这可以简写为：

```
for(i = 0; i < n; i++)
    y[i] = x[i];
```

4.2 &&、||和!运算符

4.3 下标从零开始

在很多语言中，具有 n 个元素的数组其元素的号码和它的下标是从 1 到 n 严格对应的。但在 C 中不是这样。

个具有 n 个元素的 C 数组中没有下标为 n 的元素，其中的元素的下标是从 0 到 n - 1。因此从其它语言转到 C 语言的程序员应该特别小心地使用数组：

```
int i, a[10];
for(i = 1; i <= 10; i++)
    a[i] = 0;
```

4.4 C 并不总是转换实参

下面的程序段由于两个原因会失败：

```
double s;
s = sqrt(2);
printf("%g\n", s);
```

第一个原因是 sqrt()需要一个 double 值作为它的参数，但没有得到。第二个原因是它返回一个 double 值但没有这样声名。改正的方法只有一个：

```
double s, sqrt();
s = sqrt(2.0);
printf("%g\n", s);
```

C 中有两个简单的规则控制着函数参数的转换：(1)比 int 短的整型被转换为 int；(2)比 double 短的浮点类型被转换为 double。所有的其它值不被转换。确保函数参数类型的正确性是程序员的责任。

因此，一个程序员如果想使用如 sqrt()这样接受一个 double 类型参数的函数，就必须仅传递给它 float 或 double 类型的参数。常数 2 是一个 int，因此其类型是错误的。

当一个函数的值被用在表达式中时，其值会被自动地转换为适当的类型。然而，为了完成这个自动转换，编译器必须知道该函数实际返回的类型。没有更进一步声名的函数被假设返回 int，因此声名这样的函数并不是必须的。然而，sqrt()返回 double，因此在成功使用它之前必须要声名。

这里有一个更加壮观的例子：

```
main() {
    int i;
    char c;
    for(i = 0; i < 5; i++) {
        scanf("%d", &c);
        printf("%d", i);
    }
    printf("\n");
}
```

表面上看，这个程序从标准输入中读取五个整数并向标准输出写入 0 1 2 3 4。实际上，它并不总是这么做。譬如在一些编译器中，它的输出为 0 0 0 0 0 1 2 3 4。

为什么？因为 c 的声名是 char 而不是 int。当你令 scanf()去读取一个整数时，它需要一个指向一个整数的指针。但这里它得到的是一个字符的指针。但 scanf()并不知道它没有得到它所需要的：它将输入看作是一个指向整数的指针并将一个整数存贮到那里。由于整数占用比字符更多的内存，这样做会影响到 c 附近的内存。

c 附近确切是什么是编译器的事；在这种情况下这有可能是 i 的低位。因此，每当向 c 中读入一个值，i 就被置零。当程序最后到达文件结尾时，scanf()不再尝试向 c 中放入新值，i 才可以正常地增长，直到循环结束。

24. C 语言防止缓冲区溢出方法

发布: 2008-7-22 11:43 | 作者: 剑心通明 | 来源: 互联网 | 查看: 13 次

被过滤广告语言使用直接的内存访问，缓冲区溢出是经常出现的安全问题。
下面将介绍常见的缓冲区溢出，及防止方法。

1、判断边界
例程序:

```
void outstr(int a[10])
{
        for(i=0;a[i]!=0&&i<10;i++)
        {
                printf("%d\n",a[i]);
        }
}
```

当 i=10 时，判断 i<10 同时需要判断 a[i]!=0，
此时 a[10]已经访问到非法区域，可能引起缓冲区溢出问题。

防止方法：将判断条件分成几条语句
例：

```
if(i<10){
        if(a[i]!=0){
                ...
        }
}
```

2、字符串操作
C 语言的字符串库没有安全保护，在使用时要特别小心。
strcpy、strcat 等函数操作时没有检查缓冲区大小，容易引起安全问题。
防止方法：
1)使用 strncpy、strncat，限制拷贝字符串长度。
2)或者是在 strcpy、strcat 之前检查缓冲区大小是否满足要求。

3、free
free 后指针不是 NULL。所以判断指针是否为 NULL 并不能保证指针有效。
防止方法：在 free 后重置指针。

free(p);p=NULL;

4、指针未初始化

指针未初始化便引用。指针初始值未知，无法根据值判定指针是否有效。

指针声明后尽快初始化，如不能初始化为有效值，也要初始化为 NULL。

防止方法：每次使用指针，不能确定指针有效时，先判断指针是否为 NULL。

25. C 语言高效编程秘籍

发布: 2008-7-21 00:13 | 作者: 剑心通明 | 来源: 互联网 | 查看: 6 次

编写高效简洁的 C 语言代码，是许多软件工程师追求的目标。本文就工作中的一些体会和经验做相关的阐述，不对的地方请各位指教。

第 1 招：以空间换时间

计算机程序中最大的矛盾是空间和时间的矛盾，那么，从这个角度出发逆向思维来考虑程序的效率问题，我们就有了解决问题的第 1 招--以空间换时间。

例如：字符串的赋值。

方法 A，通常的办法：

```
#define LEN 32
char string1 [LEN];
memset (string1,0,LEN);
strcpy (string1,"This is an example!!"
```

方法 B:

```
const char string2[LEN]="This is an example!"
char*cp;
cp=string2;
```

(使用的时候可以直接用指针来操作。)

从上面的例子可以看出，A 和 B 的效率是不能比的。在同样的存储空间下，B 直接使用指针就可以操作了，而 A 需要调用两个字符函数才能完成。B 的缺点在于灵活性没有 A 好。在需要频繁更改一个字符串内容的时候，A 具有更好的灵活性；如果采用方法 B，则需要预存许多字符串，虽然占用了 大量的内存，但是获得了程序执行的高效率。

如果系统的实时性要求很高，内存还有一些，那我推荐你使用该招数。

该招数的边招--使用宏函数而不是函数。举例如下：

方法 C:

```
#define bwMCDR2_ADDRESS 4
#define bsMCDR2_ADDRESS 17
int BIT_MASK (int_bf)
{
return ((IU<<(bw##_bf))-1)<<(bs##_bf);
}
void SET_BITS(int_dst,int_bf,int_val)
{
_dst=((_dst) & ~ (BIT_MASK(_bf)))I\ (((_val)<<<(bs##_bf))&(BIT_MASK(_bf)))
```

```
}
SET_BITS(MCDR2,MCDR2_ADDRESS,RegisterNumber);
```

方法 D:

```
#define bwMCDR2_ADDRESS 4
#define bsMCDR2_ADDRESS 17
#define bmMCDR2_ADDRESS BIT_MASK (MCDR2_ADDRESS)
#define BIT_MASK(_bf)(((1U<<(bw##_bf))-1)<< (bs##_bf)
#define SET_BITS(_dst,_bf,_val)\ ((_dst)=((_dst)&~(BIT_MASK(_bf)))I
(((_val)<<(bs##_bf))&(BIT_MASK(_bf))))
SET_BITS(MCDR2,MCDR2_ADDRESS,RegisterNumber);
```

函数和宏函数的区别就在于，宏函数占用了大量的空间，而函数占用了时间。大家要知道的是，函数调用是要使用系统的栈来保存数据的，如果编译器里有栈检查选项，一般在函数的头会嵌入一些汇编语句对当前栈进行检查；同时，CPU 也要在函数调用时保存和恢复当前的现场，进行压栈和弹栈操作，所以，函数调用需要一些 CPU 时间。而宏函数不存在这个问题。宏函数仅仅作为预先写好的代码嵌入到当前程序，不会产生函数调用，所以仅仅是占用了空间，在频繁调用同一个宏函数的时候，该现象尤其突出。

D 方法是我看到的最好的置位操作函数，是 ARM 公司源码的一部分，在短短的三行内实现了很多功能，几乎涵盖了所有的位操作功能。C 方法是其变体，其中滋味还需大家仔细体会。

第 2 招：数学方法解决问题

现在我们演绎高效 C 语言编写的第二招--采用数学方法来解决问题。

数学是计算机之母，没有数学的依据和基础，就没有计算机的发展，所以在编写程序的时候，采用一些数学方法会对程序的执行效率有数量级的提高。

举例如下，求 1~100 的和。

方法 E

```
int I,j;
for (I=1; I<=100; I++){
j+=I;
}
```

方法 F

```
int I;
I=(100*(1+100))/2
```

这个例子是我印象最深的一个数学用例，是我的饿计算机启蒙老师考我的。当时我只有小学三年级，可惜我当时不知道用公式 Nx(N+1)/2 来解决这个问题。方法 E 循环了 100 次才解决问题，也就是说最少用了 100 个赋值、100 个判断、200 个加法(I 和 j)；而方法 F 仅仅用了 1 个加法、1 个乘法、1 次除法。效果自然不言而喻。所以，现在我在编程序的时候，更多的是动脑筋找规律，最大限度地发挥数学的威力来提高程序运行的效率。

第 3 招：使用位操作

实现高效的 C 语言编写的第三招--使用位操作，减少除法和取模的运算。

在计算机程序中，数据的位是可以操作的最小数据单位，理论上可以用“位运算”来完成所有的运算和操作。一般的位操作是用来控制硬件的，或者做数据变换使用，但是，灵活的位操作可以有效地提高程序运行的效率。举例台如下：

方法 G

```
int I,J;
I=257/8;
J=456%32;
```

方法 H

```
int I,J;
I=257>>3;
J=456-(456>>4<<4);
```

在字面上好象 H 比 G 麻烦了好多，但是，仔细查看产生的汇编代码就会明白，方法 G 调用了基本的取模函数和除法函数，既有函数调用，还有很多汇编代码和寄存器参与运算；而方法 H 则仅仅是几句相关的汇编，代码更简洁、效率更高。当然，由于编译器的不同，可能效率的差距不大，但是，以我目前遇到的 MS C,ARM C 来看，效率的差距还是不小。相关汇编代码就不在这里列举了。

运用这招需要注意的是，因为 CPU 的不同而产生的问题。比如说，在 PC 上用这招编写的程序，并在

PC 上调试通过，在移植到一个 16 位机平台上的时候，可能会产生代码隐患。所以只有在一定技术进阶的基础下才可以使用这招。

第 4 招：汇编嵌入

高效 C 语言编程的必杀技，第四招--嵌入汇编。

“在熟悉汇编语言的人眼里，C 语言编写的程序都是垃圾”。这种说法虽然偏激了一些，但是却有它的道理。汇编语言是效率最高的计算机语言，但是，不可能靠着它来写一个操作系统吧？所以，为了获得程序的高效率，我们只好采用变通的方法--嵌入汇编、混合编程。

举例如下，将数组一赋值给数组二，要求每一个字节都相符。char string1[1024], string2[1024];

方法 I

```
int I;
for (I=0; I<1024; I++)
*(string2+I)=*(string1+I)
```

方法 J

```
#int I;
for(I=0; I<1024; I++)
*(string2+I)=*(string1+I);
#else
#ifdef_ARM_
```

```
_asm
{
MOV R0,string1
MOV R1,string2
MOV R2,#0
loop:
LDMIA R0!,[R3-R11]
STMIA R1!,[R3-R11]
ADD R2,R2,#8
CMP R2, #400
BNE loop
}
#endif
```

方法 I 是最常见的方法，使用了 1024 次循环；方法 J 则根据平台不同做了区分，在 ARM 平台下，用嵌入汇编仅用 128 次循环就完成了同样的操作。这里有朋友会说，为什么不用标准的内存拷贝函数呢？这是因为在源数据里可能含有数据为 0 的字节，这样的话，标准库函数会提前结束而不会完成我们要求的操作。这个例程典型应用于 LCD 数据的拷贝过程。根据不同的 CPU，熟练使用相应的嵌入汇编，可以大大提高程序执行的效率。

虽然是必杀技，但是如果轻易使用会付出惨重的代价。这是因为，使用了嵌入汇编，便限制了程序的可移植性，使程序在不同平台移植的过程中，卧虎藏龙、险象环生！同时该招数也与现代软件工程的思想相违背，只有在迫不得已的情况下才可以采用。切记。

使用 C 语言进行高效率编程，我的体会仅此而已。在此已本文抛砖引玉，还请各位高手共同切磋。希望各位能给出更好的方法，大家一起提高我们的编程技巧。

26. C 运算符优先级口诀

发布: 2008-7-21 17:00 | 作者: 剑心通明 | 来源: 互联网 | 查看: 21 次

括号成员第一; //括号运算符[]() 成员运算符. ->

全体单目第二; //所有的单目运算符比如++ -- +(正) -(负) 指针运算*&

乘除余三,加减四; //这个"余"是指取余运算即%

移位五，关系六; //移位运算符：<< >> ，关系：> < >= <= 等

等于(与)不等排第七; //即== !=

位与异或和位或; //这几个都是位运算: 位与(&)异或(^)位或(|)

"三分天下"八九十;

逻辑或跟与; //逻辑运算符:|| 和 &&

十二和十一; //注意顺序:优先级(||) 底于 优先级(&&)

条件高于赋值, //三目运算符优先级排到 13 位只比赋值运算符和","高//需要注意的是赋值运算符很多!

逗号运算级最低! //逗号运算符优先级最低

由于 C 语言的运算符优先级与 C++的不完全一样(主要是增加了几个运算符)，所以这个口诀不能完全实用于 C++.但是应该能够兼容，大家可以比较一下他们的区别应该就能够很快掌握 C++的优先级的!

27. do/while(0)的妙用

发布: 2008-7-22 10:37 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

在 C++中，有三种类型的循环语句：for, while, 和 do...while， 但是在一般应用中作循环时， 我们可能用 for 和 while 要多一些，do...while 相对不受重视。

但是，最近在读我们项目的代码时，却发现了 do...while 的一些十分聪明的用法，不是用来做循环，而是用作其他来提高代码的健壮性。

1. do...while(0)消除 goto 语句。

通常，如果在一个函数中开始要分配一些资源，然后在中途执行过程中如果遇到错误则退出函数，当然，退出前先释放资源，我们的代码可能是这样：

version 1

```
bool Execute()
{
    // 分配资源
    int *p = new int;
    bool bOk(true);

    // 执行并进行错误处理
    bOk = func1();
    if(!bOk)
    {
        delete p;
        p = NULL;
        return false;
    }

    bOk = func2();
    if(!bOk)
    {
        delete p;
        p = NULL;
        return false;
    }

    bOk = func3();
```

```
    if(!bOk)
    {
        delete p;
        p = NULL;
        return false;
    }

    // .....

    // 执行成功，释放资源并返回
     delete p;
     p = NULL;
     return true;

}
```

这里一个最大的问题就是代码的冗余，而且我每增加一个操作，就需要做相应的错误处理，非常不灵活。于是我们想到了 goto:

version 2

```
bool Execute()
{
    // 分配资源
    int *p = new int;
    bool bOk(true);

    // 执行并进行错误处理
    bOk = func1();
    if(!bOk) goto errorhandle;

    bOk = func2();
    if(!bOk) goto errorhandle;

    bOk = func3();
    if(!bOk) goto errorhandle;
```

```
    // .....

    // 执行成功，释放资源并返回
     delete p;
     p = NULL;
     return true;

errorhandle:
     delete p;
     p = NULL;
     return false;

}
```

代码冗余是消除了，但是我们引入了 C++中身份比较微妙的 goto 语句，虽然正确的使用 goto 可以大大提高程序的灵活性与简洁性，但太灵活的东西往往是很危险的，它会让我们的程序捉摸不定，那么怎么才能避免使用 goto 语句，又能消除代码冗余呢，请看 do...while(0)循环：

version3

```
bool Execute()
{
    // 分配资源
    int *p = new int;

    bool bOk(true);
    do
    {
        // 执行并进行错误处理
        bOk = func1();
        if(!bOk) break;

        bOk = func2();
        if(!bOk) break;

        bOk = func3();
        if(!bOk) break;
```

```
        // .....

    }while(0);

    // 释放资源
    delete p;
    p = NULL;
    return bOk;

}
```

“漂亮!”， 看代码就行了，啥都不用说了...

2 宏定义中的 do...while(0)

如果你是 C++程序员，我有理由相信你用过，或者接触过，至少听说过 MFC, 在 MFC 的 afx.h 文件里面， 你会发现很多宏定义都是用了 do...while(0)或 do...while(false)， 比如说：

#define AFXASSUME(cond) do { bool __afx_condVal=!!(cond); ASSERT(__afx_condVal); __analysis_assume(__afx_condVal); } while(0)

粗看我们就会觉得很奇怪，既然循环里面只执行了一次，我要这个看似多余的 do...while(0)有什么意义呢？

当然有!

为了看起来更清晰，这里用一个简单点的宏来演示：

#define SAFE_DELETE(p) do{ delete p; p = NULL} while(0)

假设这里去掉 do...while(0),

#define SAFE_DELETE(p) delete p; p = NULL;

那么以下代码：

if(NULL != p) SAFE_DELETE(p)

else ...do sth...

就有两个问题，

1) 因为 if 分支后有两个语句，else 分支没有对应的 if，编译失败

2) 假设没有 else, SAFE_DELETE 中的第二个语句无论 if 测试是否通过，会永远执行。

你可能发现，为了避免这两个问题，我不一定要用这个令人费解的 do...while, 我直接用{}括起来就可以了

#define SAFE_DELETE(p) { delete p; p = NULL;}

的确，这样的话上面的问题是不存在了，但是我想对于 C++程序员来讲，在每个语句后面加分号是一种约定俗成的习惯，这样的话，以下代码:

if(NULL != p) SAFE_DELETE(p);

else ...do sth...

其 else 分支就无法通过编译了（原因同上），所以采用 do...while(0)是做好的选择了。

也许你会说，我们代码的习惯是在每个判断后面加上{}，就不会有这种问题了，也就不需要 do...while 了，如：

```
if(...)
{
}
else
{
}
```

诚然，这是一个好的，应该提倡的编程习惯，但一般这样的宏都是作为 library 的一部分出现的，而对于一个 library 的作者，他所要做的就是让其库具有通用性，强壮性，因此他不能有任何对库的使用者的假设，如其编码规范，技术水平等。

28. exit()和 return()的区别

发布: 2008-7-22 10:32 | 作者: 剑心通明 | 来源: 互联网 | 查看: 7 次

exit() 结束当前进程/当前程序/，在整个程序中，只要调用 exit ，就结束

return() 是当前函数返回，当然如果是在主函数 main, 自然也就结束当前进程了，如果不是，那就是退回上一层调用

exit(0) 是非正常退出

exit(1) 是正常退出

在多个进程时.如果有时要检测上进程是否正常退出的.就要用到上个进程的返回值.. exit(1)表示进程正常退出. 返回 1; exit(0)表示进程非正常退出. 返回

29. exit 子程序终止函数与 return 的差别

发布: 2008-7-27 10:24 | 作者: 剑心通明 | 来源: 互联网 | 查看: 17 次

exit()子程序终止函数与 return()函数的差别

在 main 函数中我们通常使用 return (0);这样的方式返回一个值。

但这是限定在非 void 情况下的也就是 void main()这样的形式。

exit()通常是用在子程序中用来终结程序用的，使用后程序自动结束跳会操作系统。

但在如果把 exit 用在 main 内的时候无论 main 是否定义成 void 返回的值都是有效的，并且 exit 不需要考虑类型，exit(1)等价于 return (1)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
        exit (1);//等价于 return (1);
}
```

如果以 abort 函数（此函数不带参数，原型为 void abort(void)）终止程序，则会在 debug 模式运行时弹出所示的对话框。

Debug Error!

Program: D:\vcproject\exception\exception\Debug\exception.exe

abnormal program termination

(Press Retry to debug the application)

终止　　重试　　忽略

断言(assert)

assert 宏在 C 语言程序的调试中发挥着重要的作用，它用于检测不会发生的情况，表明一旦发生了这样的情况，程序就实际上执行错误了，例如 strcpy 函数：

```
char *strcpy(char *strDest, const char *strSrc)
{
    char * address = strDest;
    assert((strDest != NULL) && (strSrc != NULL));
    while ((*strDest++ = *strSrc++) != '\0')
       ;
    return address;
}
```

其中包含断言 assert((strDest != NULL) && (strSrc != NULL))，它的意思是源和目的字符串的地址都不能为空，一旦为空，程序实际上就执行错误了，会引发一个 abort。

assert 宏的定义为：

```
#ifdef NDEBUG
#define assert(exp) ((void)0)
#else
#ifdef __cplusplus
extern "C"
{
    #endif

    _CRTIMP void __cdecl _assert(void *, void *, unsigned);
    #ifdef __cplusplus
}
#endif
#define assert(exp) (void)( (exp) || (_assert(#exp, __FILE__, __LINE__), 0) )
#endif /* NDEBUG */
```

如果程序不在 debug 模式下，assert 宏实际上什么都不做；而在 debug 模式下，实际上是对_assert() 函数的调用，此函数将输出发生错误的文件名、代码行、条件表达式。例如下列程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
char * myStrcpy( char *strDest, const char *strSrc )
{
    char *address = strDest;
```

```
    assert( (strDest != NULL) && (strSrc != NULL) );
    while( (*strDest++ = *strSrc++) != '\0' );
      return address;
}
int main(void)
{
    myStrcpy(NULL,NULL);
    return 0;
}
```

在此程序中，为了避免我们的 strcpy 与 C 库中的 strcpy 重名，将其改为 myStrcpy。程序的输出

```
Assertion failed:(strDest != NULL) && (strSrc != NULL). file
D:\vcproject\exception\exception\main.c line?

abnormal program termination
```

失败的断言也会弹出如上所示的对话框，这是因为_assert()函数中也调用了 abort()函数。

一定要记住的是 assert 本质上是一个宏，而不是一个函数，因而不能把带有副作用的表达式放入 assert 的"参数"中。

errno

errno 在 C 程序中是一个全局变量，这个变量由 C 运行时库函数设置，用户程序需要在程序发生异常时检测之。C 运行库中主要在 math.h 和 stdio.h 头文件声明的函数中使用了 errno，前者用于检测数学运算的合法性，后者用于检测 I/O 操作中（主要是文件）的错误，例如：

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
int main(void)
{
    errno = 0;
    if (NULL == fopen("d:\\1.txt", "rb"))
    {
      printf("%d", errno);
    }
```

```
    else
    {
        printf("%d", errno);
    }
    return 0;
}
```

在此程序中，如果文件打开失败（fopen 返回 NULL），证明发生了异常。我们读取 error 可以获知错误的原因，如果 D 盘根目录下不存在"1.txt"文件，将输出 2，表示文件不存在；在文件存在并正确打开的情况下，将执行到 else 语句，输出 0，证明 errno 没有被设置。

30. extern 与 static 存储空间矛盾

发布: 2008-7-27 09:05 | 作者: 剑心通明 | 来源: 互联网 | 查看: 7 次

其实，这两个语句的位置不同，会出现不同的解释。这主要是由于 static 具有的两重意义所导致的：

(1) 如果 static int foo; 这一句位于函数中，则 static 表示的是存储属性，表明 foo 是一个静态变量。

(2) 如果 static int foo; 这一句位于函数外面，则 foo 是一个全局变量，static 不再是表示存储性质，而是作为限制符来使用：用来限制全局变量 foo 的可见范围，将其作用域限制于所在的文件内，在其它文件中是不可见的。

楼主所说的用编译器出错的情况应该是这两句都位于函数外面的情况。在这种情况下

extern int foo;

static int foo;

先声明变量的存在，再定义变量。在 extern int foo; 之前还没有遇到其定义，故 foo 的定义可能出现在这个文件后面或者在其它文件中，因此期望 foo 的作用范围不能仅仅局限于本文件，即不能是 static 全局变量。这就与下面给定的 foo 的定义相矛盾：由于定义语句缩小了被声明的 foo 的作用范围（只局限在本文件中）从而导致声明和定义不一致。

出现的先后顺序不一样，出现的编译信息也不同。例如，如果是下面的情况

static int foo;

extern int foo;

即先定义后声明。此时，声明将遵从定义的规定，继承定义变量的一切属性。

从总体上程序的内存空间可分为代码区和数据区。

从 C++的角度来看数据区又可作如下划分：

1. 自动存储区（栈）：自动（局部）变量、寄存器变量、临时对象以及函数参数。

2. 静态存储区：全局对象、函数中的静态变量、类中的静态数据成员、常量字符串以及 namespace 变量。

3. 自由存储区（堆）：也称为动态内存。

31. PC-Lint 与 C\C++代码质量

发布: 2008-7-27 09:06 | 作者: 剑心通明 | 来源: 互联网 | 查看: 39 次

软件质量对于一个软件设计者来说是相当重要的。

正确性、健壮性、可靠性、效率、易用性、可读性（可理解性）、可扩展性、可复用性、兼容性、可移植性等质量属性就是软件整体质量的体现。
软件质量问题说到底应该是代码质量问题，写出优秀、稳定的代码是一个高水平的程序设计都应该具备的。

在好的数据结构和算法的前提下，如何编写出优秀的代码是我们最应该关心的。

关于如何提高自己编写的 C\C++代码质量，林锐博士的《高质量 C++_C 编程指南》是很好的指导！在那本书的最后提到了 PC-Lint 代码审查工具是很好的助手，这里我简单的介绍一下这个工具。下面是从网络上找来的资料：
/**********
LINT 工具是一种软件质量保证工具，许多国外的大型专业软件公司，如微软公司，都把它作为程序检查工具，在程序合入正试版本或交付测试之前一定要保证通过了 LINT 检查，他们要求软件工程师在使用 LINT 时要打开所有的编译开关，如果一定要关闭某些开关，那么要给出关闭这些开关的正当理由。

可想而知，如果从我们编码后第一次编译程序时就使用 LINT 来检查程序，并且保证消除所有的 LINT 告警，我们就不会遇到象今天这么多的告警信息。即使在今天，我们如果能抽出一定的精力来消除程序中的 LINT 告警，以后再维持这种无告警状态就是很容易的了。我们程序质量的提高也是不言而喻的。

PC-LINT 是 GIMPEL SOFTWARE 公司的产品，其中的内容是非常广泛的，光是选项就有 300 多个，涉及到程序编译及语法使用中的方方面面。本篇培训材料旨在引导读者入门，学会 PC-LINT 的基本使用方法，起抛砖引玉的作用，能让读者从这里起步继续去研究如何娴熟地使用 PC-LINT 的各种选项，能让它充分为我们的开发工作服务。

1.概述

如果要给 LINT 工具下一个形象点的定义，那就是：一种更加严格的编译器。它不仅可以象普通编译器那样检查出一般的语法错误，还可以检查出那些虽然完全合乎语法要求，但很可能是潜在的、不易发现的错误。请看下面的例子：

```
1：
2：char *report( int m, int n, char *p )
3：{
4：  int result;
5：  char *temp;
6：  long nm;
7：  int i, k, kk;
8：  char name[11] = "Joe Jakeson";
9：
10：  nm = n * m;
11：  temp = p == "" ? "null" : p;
12：  for( i = 0; i 13：  {
14：  k++;
15：  kk = i;
16：  }
17：
18：  if( k== 1 ) result = nm;
19：  else if( kk > 0 ) result = 1;
20：  else if( kk < 0 ) result = -1;
21：
22：  if( m == result ) return( temp );
23：  else return( name );
24：}
```

上面的代码用一般的编译器编译是一段有效的代码，但是用 PC-LINT 编译就会有几个告警。首先第 8 行向 name 数组赋值时丢掉了 nul 字符，第 10 行的乘法精度会失准，第 11 行的比较有问题，第 14 行的变量 k 没有初始化，第 15 行的 kk 可能没有被初始化，第 22 行的 result 也有可能没有被初始化，第 23 行返回的是一个局部对象的地址。这段代码在大部分编译器下是可以顺利编译通过的，继续查找其中的错误就要靠人工调试程序，如果程序很大，这将是一项烦琐的工作，没有人可以保证能找出所有的这类问题，但 PC-LINT 只通过一次简单的编译就可做到，显然为我们节省了大量的开发时间。

下面就让我们看看如何安装使用 PC-LINT。

2.如何安装 PC-LINT

PC-LINT 的软件的安装过程比较复杂，选项较多，下面根据安装过程，逐条说明每一步的含义。

0）如果是 zip 文件，将 ZIP 安装文件展开到目录 C:\lint.ins 下，进入 COMMAND PROMPT，先

进行目录映射 subst g: c:\lint.ins，然后转到 G:，执行 install。其他步骤和下面的从软盘安装是一样的。

1）在 A:驱插入 PC-LINT 安装盘，输入 A:\>install 命令，进入开始安装栏，按任意键继续，进入 PC-LINT 介绍栏，再按任意键继续。

2）进入环境选择栏，这一栏中有三个选项：

Windows NT/Windows 95

MS-DOS（DOS extended）

OS/2（32bit）

如果计算机安装了 WIN95、WIN97、WIN98 或 WINNT 要选择 Windows NT/Windows 95，如果只有 DOS 则选择 DOS。

3）进入安装目录选择栏，它推荐的是 C:\>LINT，如不想安装在这个目录下，可输入自己想要安装的目录，然后按回车确认，如果要安装的目录不存在，它会提示为你建立这个目录。我们这里选 C:\>LINT

4）选择安装盘所在的磁盘驱动器，我们这里选 A:

5）判断是否要选择多种编译器或编译库的配置，如果要对不同编译环境下的程序进行 LINT，则选 YES，否则选 NO。然后回车确认。

6）这时看到一个编译器列表，在这个表中选择自己使用的编译器，如果表中没有自己使用的编译器，可选择通用编译器：Generic Compilers。按回车确认。这个选项会体现在 co-xxx.lnt 文件中。

7）接着安装程序会让你选择一个的内存模型，可以根据自己程序区和数据区的实际大小选择一个恰当的内存模型。如果 CPU 为 32 位 68K 系列，则要选择：32-bit Flat Module。内存模型的选项会体现在 STD.LNT 文件中。

8）选完内存模型后，会看到一个库类型的列表，在这里选择一个或多个编译时使用的库。这个选项会体现在 LIB-xxx.LNT 文件中。

9）接着是让你选择为使用 C++编程提出过重要建议的作者，选择的某作者后，他提出的编程建议方面的选项将被打开。与作者选择有关的选项会体现在 AU-xxx.LNT 文件中。

10）下一步是设置包含文件目录。有两种选项，第一种是使用环境变量 INCLUDE，环境变量在批处理文件中设置，环境变量后每个目录用分号隔开，例如可设成“INCLUDE=C:\MRI\MCC68K;D:\LAP\SRC\INC”。第二种选项是使用-i 选项，-i 选项体现在 STD.LNT 文件中，每个目录前以-I 引导，目录间以空格分隔，例如可设成“-IC:\MRI\MCC68K -ID:\LAP\SRC\INC”。如果选择使用-I 选项，安装程序会接着让你输入包含文件所在的目录。

11） 如果前面选择了使用多个编译环境，这里将会问你是否选择更多的编译环境，如果选 YES，将会从第 6 步开使重复。如果选 NO 则会结束编译器选择。

12）接下来将会准备产生一个 反映全局编译信息显示情况的选项文件 OPTIONS.LNT，该文

件的产生方式有两种，一种是安装程序对几个核心选项逐一解释并提问你是否取消该选项，如果你选择取消，则会体现在 OPTIONS.LNT 文件中，具体体现方式是在该类信息编码前加-e，后面第 13~18 步是逐一选择核心选项的过程。如果选择第二种选择方式，安装文件会先生成一个空的 OPTIONS.LNT 文件，等你以后在实际应用时加入必要的选项。

13）是否关闭赋值时的布尔测试告警，如：if(a=f()){...

14）是否关闭赋值时的有符号量和无符号量间的不匹配告警，通常情况下，这种赋值不会带来问题，选择关闭该告警信息的同时，其他类型的有符号量和无符号量间混合操作的告警仍然是打开的。

15）当把一个整形量赋值给一个比它短的量时，后者会丢失精度，例如把一个 INT 量赋值给给一个 CHAR 量。本步是让你选择是否关闭该类告警。

16）是否关闭左移带符号量的告警。通常 PC-LINT 会对所有带符号量的移动产生告警，但右移一般是由不同的 CPU 来确定是否将符号位移入，左移一般是不会产生什么问题的，所以可以选择关闭该告警。

17）在一个 C 函数被定义或声明前调用它，并不总是会产生错误，在这里可以选择是否关闭该告警选项。该选项对 C++程序不起作用。

18）是否关闭“调用不存在的函数原型”告警。有些程序员不愿遵守严格的函数原形定义约定，但 PC-LINT 会在调用一个没有定义的函数原型时产生一个告警，在这里可以选择关闭该告警。

19）通过上面的步骤确定 OPTIONS.LNT 文件的形式后，接着是选择编译环境。PC-LINT 提供了集成在多种开发环境中工作的功能，例如可集成在 VC、BC、Source Insight 中。假如我们在这里选择 Source Insight。选择后安装程序会继续问你是否还选择其它的环境，可根据自己应用的实际情况选择一种或多种开发环境。开发环境的选择情况记录在 env-xxx.lnt 文件中。

20）安装程序会生成一个 LIN.BAT 文件，该文件是运行 PC-LINT 的批处理文件，为了使该文件能在任何路径下运行，安装程序提供了两种方法供你选择。第一种方法是让你选择把 LIN.BAT 拷贝到任何一个 PATH 目录下，在安装结束运行 LCOPY.BAT 文件时，会把 LIN.BAT 拷贝到

你指定的目录。第二种方法是生成一个 LSET.BAT 文件，在每次使用 PC-LINT 前先运行它来设置路径，或者把 LSET.BAT 文件的内容拷贝到 AUTOEXEC.BAT 文件中。

21）在安装程序执行完后第一件事是在你安装的目录下执行 LCOPY.BAT 文件。它会从安装盘拷贝将一些文件拷贝到安装目录下，并根据你在安装过程中的选择来设置文件中的参数。

3.LINT 一个 C 文件

3.1 用命令行方式进行 LINT

如果使用 LIN.BAT 批处理文件进行 LINT，在 LINT 前要先看一下该批处理文件中的内容，里面包含了 LINT-NT 命令和命令选项，可以根据自己的要求来修改、增减选项。我们看到，在这

个批命令中嵌套了一个 std.lnt 文件，在 std.lnt 文件中还嵌套了 co.lnt、options.lnt 和 l
ib-stl.lnt 文件，原则上*.lnt 文件是可以无限制嵌套，该类文件中一般都是了 LINT 的选项
，可通过修改这些文件来修改 LINT 选项，选项是按照从左到右的顺序执行的。可执行下面
命令行：

C:\abc\src>lin alpha.c beta.c gamma.c

通常对于由多个 C 模块组成的程序，最好先分别对每个 C 模块单元进行 LINT 检查，做单元 LI
NT 时可如下运行：

C:\abs\src>lin -u alpha.c

其中-u 是单元选项，使用-u 后可以关闭一些检查多模块时会产生的告警，例如“函数未被
使用”或“函数没有定义”等。

也可以不使用 LIN.BAT 批处理文件，而直接使用 LINT 命令。在 DOS 环境下 LINT 命令为 LINT.E
XE，在 Windows95/NT 环境下为 LINT-NT.EXE，在 OS2 环境下为 LINT-OS2.EXE。直接使用 LINT
命令要注意的一点是要在使用前预先设置 LINT 目录所在路径，最好的方法是把该路径加在
AUTOEXEC.BAT 文件中。其它的使用方法与使用批处理文件相同。例如：

C:\abs\src>lint-nt -ic:\lint\ std.lnt -os(_lint.tmp) *.c

3.2 用开发环境进行 LINT

也可以使用开发环境来执行 LINT 操作，一般开发环境都支持运行可执行文件，但不一定支
持运行批处理文件，下面用 Source Insight , Ultra Edit, MSVC 6.0 来举例说明如何在
开发环境下进行 LINT。

3.2.1 在 Source Insight 中集成

如果你在安装过程中选定了使用某个开发环境，安装程序会在你安装的目录下生成一个 en
v-xxx.lnt 的文件，例如选择了 Source Insight 就会有一个 env-si.lnt 文件。用编辑器打开
该文件，在该文件开始的注释中说明了如何将 PC-LINT 功能集成在开发环境中，集成在 Sou
rce Insight 中的过程如下：

1）从 Options 菜单中选择“Custom Commands”命令项。

2）在 Name 栏中输入“PC-lint ”，原则上这个名称可以随便起，只要你能搞清楚它的含义
就可以了。

3）在 Run 栏中输入“c:\lint\lint-nt -u -ic:\lint std env-si %f”其中 c:\lint 是你 P
C-LINT 的安装目录。

4）在 Output 栏中选择“Iconic Window”、“Capture Output”。

5）在 Control 栏中选择“Save Files First”。

6）在 Source Links in Output 栏中选择“Parse Links in Output”、“File，then Lin
e”。

7）在 Pattern 栏中输入“^\([^]*\) \([0-9]+\)”。

8）点 Add 键加入该命令。如下图：

9）使用时，在 Source Insight 下打开要 LINT 的文件，打开 Options 菜单中的“Custom Com

mands”命令项，在“Command”栏中选择“PC-lint unit check”命令运行即可。

注意到我的 Run 一栏的参数和上面的提示不一样，其实我的其他古怪参数都放到 c:\lint\std.lnt 中了。请注意，不论你怎样配置参数一定不要忘记了将 si-env.lnt 包含在你的配置文件里，否则就无法进行错误信息和程序的自动对应了。

为了使用方便，你还可以配置一下 Menu 按钮，将它加到系统菜 0 单里，这属于一般性的 Source Insight 应用，笔者就不在此赘述了。

第二笔者在 NT 中使用 Source Insight 时，好象集成不了，原因暂时不明了。上面的例子在 WIN 95 下测试成功。

如果要修改 LINT 选项，可直接在 Run 栏中修改，也可专门编辑一个*.lnt 文件放在 c:\lint 目录下，并将该文件名加入 Run 栏中，和命令行方式是一样的。

3.2.2 在 Ultra Editor 中集成

选取 Menu | Advanced | Tool Configuration ... ， 显示如下图：

1）点按“Insert",

2）在 command line:中填写：c:\lint\lint-nt c:\lint\std.lnt %f

3）在 Menu Item 中填写：PC-LINT

4）在 Command Output 中选择： (x) Output to List Box 和 (x) Capture Output

5）点按"OK"

如图所示的配置笔者在 UE6.0 / NT 4.0 下测试成功。

3.2.3 在 MSVC 6.0 中集成

//这个好使过

基本原理是一样的：

1）选取 menu | tools | customize.....

2）选取 Tools Tab:

3）点按主对话框上方的虚线小方框 New a tool item

4）输入 name: PC-LINT

5）输入 Command: c:\lint\lint-nt.exe

6）输入 Arguments: c:\lint\std.lnt $(FilePath) //注：替 std.lnt 为 lnt\env-vc6.lnt

7) 选择 (x) Use Output Window

8）Close

完成后，在 tools 菜单下就会有一项 PC-LINT 选项。

下面是笔者在 VC6 / Win NT 4.0 的情况下的 TOOL 配置图：

3.3LINT 选项

LINT 选项可以放在注释中，例如：

/*lint option1 option2 ... optional commentary */

//lint option1 option2 ... optional commentary

选项间要以空格分开，lint 命令一定要小写，并且紧跟在/*或//后面，不能有空格。如果选项由类似于操作符和操作数的部分组成，例如-esym(534, printf, scanf, operat or new)，其中最后一个选项是 operator new，那么在 operator 和 new 中间只能有一个空格。

选项还可以放在宏定义中，例如：

#define DIVZERO(x) /*lint -save -e54 */ ((x) /o) /*lint -restore */

LINT 的选项很多共有 300 多种，大体可分为以下几类：

1）错误信息禁止选项

该类选项是用于禁止生成某类错误信息的选项，最常用的是-e 和+e，-e 是禁止生成某类错误信息，+e 是恢复生成某类错误信息。运行 lint 目录下的 msg.exe 可以得到 msg.txt 文件，这个长达 5000 行的文件包含了所有的错误信息号和解释。

-w 对于所有大于级别的告警信息都不显示。

-wlib()对于所有大于级别的关于库函数数的告警信息都不显示。我们可以用-wlib(0)来屏蔽所有的库函数的告警信息，-wlib(1)只显示库函数中的句法错误。

-esym(#,) 可以屏蔽对于特定符号的某告警信息。

2）变量类型大小选项

不同的目标机、编译系统变量类型的的大小（如短整形变量、整形变量等）会有所不同，该类选项用于为目标机设置变量类型的大小。由于默认的设置与大部分的编译器是匹配的，这些专门的设置通常情况下是不需要的，只在特别的目标机结构中才用。例如一个 M68000 目标机，它的 int 类型和指针类型通常是 32bit 的，这时你应该使用选项：-si4 -sp4。这些尺寸参数的当前值可以通过 help 屏来获得，例如可以输入以下命令行：

lin -si4 -sp4 ？

3）冗长信息选项

冗长信息指的是 LINT 过程中产生的一些与编译过程有关的信息，而不是真正的告警信息、错误信息等。是否生成这些信息可以通过-v 和+v 选项来决定。+v 是生成这些信息，-v 是关闭这些信息，这组选项中除+v 外，其它所有选项都可以关闭+v 选项。

4）标记选项

以+f、++f、-f 和--f 开头的选项是标记选项。他们的逻辑含义分别如下：

+f...：通过把标志置为 1 而把它置为 ON

-f...：通过把标志置为 0 而把它置为 OFF

++f...：标志增 1

--f...：标志减 1

后面两个用于你想在局部把一个标志置为 ON 的情况，而不影响全局设置。例如你可以这样使用：

/*lint ++flb */

int printf( );

/*lint --flb */

标记选项的种类很多，基本含义是用于打开或关闭某类语法情况使用，例如允许使用缩写结构体名称，允许使用无名联合体，把所有模块当作 C++编译等。

5）消息显示选项

消息显示选项用于定义消息输出格式。主要有消息高度选项、消息宽度选项、消息格式选项等。

6）其它选项

其它选项中的种类很多，各种类间差异很大，在这里就不一一介绍了，建议大家看一看《PC-LINT》一书，第五章有对每种选项的详细说明。lint 本身也有一些说明信息， lint-nt 2> lint.txt 然后狂按几个回车就可以生成一个 lint 选项的说明文件。

4.LINT 一个工程下的多个 C 文件

4.1 为何要 LINT 多个 C 文件

在程序编码初期，我们关心的可能只是单个 C 模块种中的语法问题，等到编程后期，对于由多个 C 模块组成的程序，我们希望了解当把多个模块连接在一起后是否还有存在于模块间的语法问题。这时编译器虽然能给出一些告警，但 PC-LINT 的连接能给出更多的告警。还有当我们能保证其中的几个模块相对稳定，而另外几个模块仍有问题时可以先将几个稳定的模块编译连接成一个目标文件，文件每次修改完成后先单独编译，然后连接入总的目标文件。

4.2 如何 LINT 一个工程下的多个 C 文件

象我们平时使用的编译工具一样，PC-LINT 在编译连接多个 C 文件时也会先把每个 C 文件编译生成中间的目标文件*.lob，然后再将所有的 LOB 文件连接在一起。LOB 是 Lint Object Module 的缩写。这个文件中包含了一个 C 或 C++模块的所有外部信息。生成 LOB 文件时有三种选项要注意：第一种是-u，如果要 LINT 生成 LOB 文件，就一定要加-u 选项；第二种是-zero 或-zero(500)选项，为了保证 LOB 文件在模块存在错误的情况下也能生成，就一定要加这个选项；第三种是-oo[(filename)]，filename 是生成的 LOB 文件的名称，在-oo 后面，可加，也可不加，如不加，则 LOB 文件名与原 C 模块的名称相同，例如：

lint -u alpha.c -oo(a1)

生成的 LOB 文件名为：a1.lob

lint -u alpha.c -oo

生成的 LOB 文件名为：alpha.lob

LINT 一个工程下的多个 C 模块，在用户的源程序目录下一般需要三个文件： 一个选项文件（*.lnt）、一个批处理文件（*.bat）和一个 MAKEFILE 文件（*.mak）。下面一一讲述如何制作这些文件。

1）选项文件（*.lnt）

选项文件在前面也提到过，你可以把你 LINT 每个 C 文件时时用到的所有公共选项罗列在该文件中，选项生效的顺序按照从左到右，从上到下的原则。该类文件可以层层嵌套，嵌套的

层数没有限制。例如 make.lnt 文件：

-iC:\lint

std.lnt

+os(temp)

-e46

+vm

-zero

2）批处理文件（*.bat）

制作批处理文件时要注意要在该文件中调用 TCMAKE.EXE 文件和 MAKEFILE 文件，例如 lintma
ke.mak 文件：

@echo Lint Making 'makelap':

tcmake -flintmake.mak

@echo End of making

3）MAKEFILE 文件（*.mak）

MAKEFILE 使用的 TCMAKE 的语法，和我们平时开发编译时使用的 MAKEFILE 文件语法格式一
样

，例如下面的 lintmake.mak 文件：

MCCPATH = c:\mcc68k

OPTION = -u make.lnt -oo

GLOBLE = os.h l2lap.h

mail_depend = $(GLOBLE) q931.h mail.h

lapmain_depend = $(GLOBLE) l1pubdef.h q931.h mail.h

lapos_depend = $(GLOBLE)

fhdlc1_depend = $(GLOBLE) cpuhdlc.h bd_prar.h q931.h

OBJ = mail.lob lapmain.lob lapos.lob fhdlc1.lob

project.lob : $(OBJ)

lint-nt make.lnt -e768 -e769 *.lob

mail.lob: mail.c $(mail_depend)

lint-nt $(OPTION) mail.c

lapmain.lob: lapmain.c $(lapmain_depend)

lint-nt $(OPTION) lapmain.c

lapos.lob: lapos.c $(lapos_depend)

lint-nt $(OPTION) lapos.c

fhdlc1.lob: fhdlc1.c $(fhdlc1_depend)

lint-nt $(OPTION) fhdlc1.c

4.3 简单的 LINT 多个文件

假设我们的工程不复杂，我们可以负担起每次都将所有的文件都 lint 一遍的开销，也可以
不使用上面的正规用法。笔者在实践中发现，将所有的*.c 文件放在一个 lint 命令中，同样

能完成 lint 整个工程的目的。

如：

lint-nt c:\lint\std.lnt AllMySource.lnt

在 AllMySource.lnt 中包括你的工程中的所有源文件：

a1.c

a2.c

a3.c

需要注意的是，在 std.lnt 文件中就不需要-u 选项了。因为我们已经提供了所有的信息。

32. spirntf 函数使用大全

发布: 2008-7-28 11:04 | 作者: 剑心通明 | 来源: 互联网 | 查看: 109 次

printf 可能是许多程序员在开始学习 C 语言时接触到的第二个函数（我猜第一个是 main），说起来，自然是老朋友了，可是，你对这个老朋友了解多吗？你对它的那个孪生兄弟 sprintf 了解多吗？在将各种类型的数据构造成字符串时，sprintf 的强大功能很少会让你失望。

由于 sprintf 跟 printf 在用法上几乎一样，只是打印的目的地不同而已，前者打印到字符串中，后者则直接在命令行上输出。这也导致 sprintf 比 printf 有用得多。所以本文着重介绍 sprintf，有时也穿插着用用 pritnf。

sprintf 是个变参函数，定义如下：

int sprintf(char *buffer, const char *format [, argument] ...);

除了前两个参数类型固定外，后面可以接任意多个参数。而它的精华，显然就在第二个参数：格式化字符串上。

printf 和 sprintf 都使用格式化字符串来指定串的格式，在格式串内部使用一些以“%”开头的格式说明符（format specifications）来占据一个位置，在后边的变参列表中提供相应的变量，最终函数就会用相应位置的变量来替代那个说明符，产生一个调用者想要的字符串。

1. 格式化数字字符串

sprintf 最常见的应用之一莫过于把整数打印到字符串中，所以，spritnf 在大多数场合可以替代 itoa。如：

//把整数 123 打印成一个字符串保存在 s 中。

sprintf(s, "%d", 123); //产生"123"

可以指定宽度，不足的左边补空格：

sprintf(s, "%8d%8d", 123, 4567); //产生：" 123 4567"

当然也可以左对齐：

sprintf(s, "%-8d%8d", 123, 4567); //产生："123　　4567"

也可以按照 16 进制打印：

sprintf(s, "%8x", 4567); //小写 16 进制，宽度占 8 个位置，右对齐

sprintf(s, "%-8X", 4568); //大写 16 进制，宽度占 8 个位置，左对齐

这样，一个整数的 16 进制字符串就很容易得到，但我们在打印 16 进制内容时，通常想要一种左边补 0 的等宽格式，那该怎么做呢？很简单，在表示宽度的数字前面加个 0 就可以了。

sprintf(s, "%08X", 4567); //产生："000011D7"

上面以”%d”进行的 10 进制打印同样也可以使用这种左边补 0 的方式。

这里要注意一个符号扩展的问题：比如，假如我们想打印短整数（short）-1 的内存 16 进制表示形式，在 Win32 平台上，一个 short 型占 2 个字节，所以我们自然希望用 4 个 16 进制数字来打印它：

short si = -1;

sprintf(s, "%04X", si);

产生“FFFFFFFF”，怎么回事？因为 spritnf 是个变参函数，除了前面两个参数之外，后面的参数都不是类型安全的，函数更没有办法仅仅通过一个“%X”就能得知当初函数调用前参数压栈时被压进来的到底是个 4 字节的整数还是个 2 字节的短整数，所以采取了统一 4 字节的处理方式，导致参数压栈时做了符号扩展，扩展成了 32 位的整数-1，打印时 4 个位置不够了，就把 32 位整数-1 的 8 位 16 进制都打印出来了。如果你想看 si 的本来面目，那么就应该让编译器做 0 扩展而不是符号扩展（扩展时二进制左边补 0 而不是补符号位）：

sprintf(s, "%04X", (unsigned short)si);

就可以了。或者：

unsigned short si = -1;

sprintf(s, "%04X", si);

sprintf 和 printf 还可以按 8 进制打印整数字符串，使用”%o”。注意 8 进制和 16 进制都不会打印出负数，都是无符号的，实际上也就是变量的内部编码的直接的 16 进制或 8 进制表示。

2. 控制浮点数打印格式

浮点数的打印和格式控制是 sprintf 的又一大常用功能，浮点数使用格式符”%f”控制，默认保留小数点后 6 位数字，比如：

sprintf(s, "%f", 3.1415926); //产生"3.141593"

但有时我们希望自己控制打印的宽度和小数位数，这时就应该使用：”%m.nf”格式，其中 m 表示打印的宽度，n 表示小数点后的位数。比如：

sprintf(s, "%10.3f", 3.1415626); //产生：" 3.142"

sprintf(s, "%-10.3f", 3.1415626); //产生："3.142 "

sprintf(s, "%.3f", 3.1415626); //不指定总宽度，产生："3.142"

注意一个问题，你猜

int i = 100;

sprintf(s, "%.2f", i);

会打出什么东东来？“100.00”？对吗？自己试试就知道了，同时也试试下面这个：

sprintf(s, "%.2f", (double)i);

第一个打出来的肯定不是正确结果，原因跟前面提到的一样，参数压栈时调用者并不知道跟 i 相对应的格式控制符是个”%f”。而函数执行时函数本身则并不知道当年被压入栈里的是个整数，于是可怜的保存整数 i 的那 4 个字节就被不由分说地强行作为浮点数格式来解释了，整个乱套了。

不过，如果有人有兴趣使用手工编码一个浮点数，那么倒可以使用这种方法来检验一下你手工编排的结果是否正确。J

字符/Ascii 码对照

我们知道，在 C/C++语言中，char 也是一种普通的 scalable 类型，除了字长之外，它与 short，int，long 这些类型没有本质区别，只不过被大家习惯用来表示字符和字符串而已。（或许当年该把这个类型叫做“byte”，然后现在就可以根据实际情况，使用 byte 或 short 来把 char 通过 typedef 定义出来，这样更合适些）

于是，使用”%d”或者”%x”打印一个字符，便能得出它的 10 进制或 16 进制的 ASCII 码；反过来，使用”%c”打印一个整数，便可以看到它所对应的 ASCII 字符。以下程序段把所有可见字符的 ASCII 码对照表打印到屏幕上（这里采用 printf，注意”#”与”%X”合用时自动为 16 进制数增加”0X”前缀）：

```
for(int i = 32; i < 127; i++) {

    printf("[ %c ]: %3d 0x%#04X\n", i, i, i);

}
```

3. 连接字符串

sprintf 的格式控制串中既然可以插入各种东西，并最终把它们“连成一串”，自然也就能够连接字符串，从而在许多场合可以替代 strcat，但 sprintf 能够一次连接多个字符串（自然也可以同时在它们中间插入别的内容，总之非常灵活）。比如：

```
char* who = "I";
char* whom = "CSDN";
sprintf(s, "%s love %s.", who, whom); //产生："I love CSDN. "
```

strcat 只能连接字符串（一段以’\0’结尾的字符数组或叫做字符缓冲，null-terminated-string），但有时我们有两段字符缓冲区，他们并不是以’\0’结尾。比如许多从第三方库函数中返回的字符数组，从硬件或者网络传输中读进来的字符流，它们未必每一段字符序列后面都有个相应的’\0’来结尾。如果直接连接，不管是 sprintf 还是 strcat 肯定会导致非法内存操作，而 strncat 也至少要求第一个参数是个 null-terminated-string，那该怎么办呢？我们自然会想起前面介绍打印整数和浮点数时可以指定宽度，字符串也一样的。比如：

```
char a1[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
```

char a2[] = {'H', 'I', 'J', 'K', 'L', 'M', 'N'};

如果：

sprintf(s, "%s%s", a1, a2); //Don't do that!

十有八九要出问题了。是否可以改成：

sprintf(s, "%7s%7s", a1, a2);

也没好到哪儿去，正确的应该是：

sprintf(s, "%.7s%.7s", a1, a2);//产生："ABCDEFGHIJKLMN"

这可以类比打印浮点数的”%m.nf”，在”%m.ns”中，m 表示占用宽度（字符串长度不足时补空格，超出了则按照实际宽度打印），n 才表示从相应的字符串中最多取用的字符数。通常在打印字符串时 m 没什么大用，还是点号后面的 n 用的多。自然，也可以前后都只取部分字符：

sprintf(s, "%.6s%.5s", a1, a2);//产生："ABCDEFHIJKL"

在许多时候，我们或许还希望这些格式控制符中用以指定长度信息的数字是动态的，而不是静态指定的，因为许多时候，程序要到运行时才会清楚到底需要取字符数组中的几个字符，这种动态的宽度/精度设置功能在 sprintf 的实现中也被考虑到了，sprintf 采用”*”来占用一个本来需要一个指定宽度或精度的常数数字的位置，同样，而实际的宽度或精度就可以和其它被打印的变量一样被提供出来，于是，上面的例子可以变成：

sprintf(s, "%.*s%.*s", 7, a1, 7, a2);

或者：

sprintf(s, "%.*s%.*s", sizeof(a1), a1, sizeof(a2), a2);

实际上，前面介绍的打印字符、整数、浮点数等都可以动态指定那些常量值，比如：

sprintf(s, "%-*d", 4, 'A'); //产生"65 "

sprintf(s, "%#0*X", 8, 128); //产生"0X000080"，"#"产生 0X

sprintf(s, "%*.*f", 10, 2, 3.1415926); //产生" 3.14"

4. 打印地址信息

有时调试程序时，我们可能想查看某些变量或者成员的地址，由于地址或者指针也不过是个 32 位的数，你完全可以使用打印无符号整数的”%u”把他们打印出来：

sprintf(s, "%u", &i);

不过通常人们还是喜欢使用 16 进制而不是 10 进制来显示一个地址：

sprintf(s, "%08X", &i);

然而，这些都是间接的方法，对于地址打印，sprintf 提供了专门的”%p”：

sprintf(s, "%p", &i);

我觉得它实际上就相当于：

sprintf(s, "%0*x", 2 * sizeof(void *), &i);

5. 利用 sprintf 的返回值

较少有人注意 printf/sprintf 函数的返回值，但有时它却是有用的，spritnf 返回了本次函数调用最终打印到字符缓冲区中的字符数目。也就是说每当一次 sprinf 调用结束以后，你无须再调用一次 strlen 便已经知道了结果字符串的长度。如：

int len = sprintf(s, "%d", i);

对于正整数来说，len 便等于整数 i 的 10 进制位数。

下面的是个完整的例子，产生 10 个[0, 100)之间的随机数，并将他们打印到一个字符数组 s 中，以逗号分隔开。

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
int main() {
    srand(time(0));
    char s[64];
    int offset = 0;
    for(int i = 0; i < 10; i++) {
        offset += sprintf(s + offset, "%d,", rand() % 100);
    }

    s[offset - 1] = '\n';//将最后一个逗号换成换行符。
    printf(s);
    return 0;
}
```

设想当你从数据库中取出一条记录，然后希望把他们的各个字段按照某种规则连接成一个字符串时，就可以使用这种方法，从理论上讲，他应该比不断的 strcat 效率高，因为 strcat 每次调用都需要先找到最后的那个’\0’的位置，而在上面给出的例子中，我们每次都利用 sprintf 返回值把这个位置直接记下来了。

6. 使用 sprintf 的常见问题

sprintf 是个变参函数，使用时经常出问题，而且只要出问题通常就是能导致程序崩溃的内存访问错误，但好在由 sprintf 误用导致的问题虽然严重，却很容易找出，无非就是那么几种情况，通常用眼睛再把出错的代码多看几眼就看出来了。

? 缓冲区溢出

第一个参数的长度太短了，没的说，给个大点的地方吧。当然也可能是后面的参数的问题，建议变参对应一定要细心，而打印字符串时，尽量使用”%.ns”的形式指定最大字符数。

? 忘记了第一个参数

低级得不能再低级问题，用 printf 用得太惯了。//偶就常犯。:。（

? 变参对应出问题

通常是忘记了提供对应某个格式符的变参，导致以后的参数统统错位，检查检查吧。尤其是对应”*”的那些参数，都提供了吗？不要把一个整数对应一个”%s”，编译器会觉得你欺她太甚了（编译器是 obj 和 exe 的妈妈，应该是个女的，:P）。

7. strftime

sprintf 还有个不错的表妹：strftime，专门用于格式化时间字符串的，用法跟她表哥很像，也是一大堆格式控制符，只是毕竟小姑娘家心细，她还要调用者指定缓冲区的最大长度，可能是为了在出现问题时可以推卸责任吧。这里举个例子：

```
time_t t = time(0);

//产生"YYYY-MM-DD hh:mm:ss"格式的字符串。

char s[32];

strftime(s, sizeof(s), "%Y-%m-%d %H:%M:%S", localtime(&t));
```

sprintf 在 MFC 中也能找到他的知音：CString::Format，strftime 在 MFC 中自然也有她的同道：CTime::Format，这一对由于从面向对象哪里得到了赞助，用以写出的代码更觉优雅。

8. 后记

本文介绍的所有这些功能，在 MSDN 中都可以很容易地查到，笔者只是根据自己的使用经验，结合一些例子，把一些常用的，有用的，而可能为许多初学者所不知的用法介绍了一点，希望大家不要笑话，也希望大家批评指正。

有人认为这种带变参的函数会引起各种问题，因而不提倡使用。但笔者本人每每还是抵挡不了它们强大功能的诱惑，在实际工作中一直在使用。实际上，C#.NET 从开始就支持变参，刚发布不久的 Java5.0 也支持变参了。

33. 二叉树的数据结构

发布: 2008-7-19 21:15 | 作者: 剑心通明 | 来源: 互联网 | 查看: 3 次

```
typedef struct Btree{
        ElemType data;      //先假设为 int
      struct Btree     *lchild, *rchild;
}Btree;
```

recrusive 递归

先序

```
void preorder(Btree *bt){
     printf("%d\t", bt->data);
    preoder(bt->lchild);
    preorder(bt->rchild)
        return;;
}
```

中序

```
void   midorder(Btree *bt){
     midorder(bt->lchild);
    printf("%d\t", bt->data);
  midorder(bt->rchild);
```

```
        return;
}
```

后序

```
void    postorder(Btree *bt){
   postorder(bt->lchild);
   postorder(bt->rchild);
   printf("%d\t", bt->data);
    return;
}
```

将二叉树的左右子树位置调换

```
 void   exchange(Btree *bt){
    Btree *temp;
   if(bt!=NULL){
      temp=bt->lchild;
    bt->lchild=bt->rchild;
   bt->rchild=temp;
  exchange(bt->lchild);
  exchange(bt->rchild);
```

```
    }

}
```

34. 位运算应用口诀和实例

发布: 2008-7-21 14:19 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

位运算应用口诀

清零取反要用与，某位置一可用或

若要取反和交换，轻轻松松用异或

移位运算

要点 1 它们都是双目运算符，两个运算分量都是整形，结果也是整形。

2 "<<" 左移：右边空出的位上补 0，左边的位将从字头挤掉，其值相当于乘 2。

3 ">>"右移：右边的位被挤掉。对于左边移出的空位，如果是正数则空位补 0，若为负数，可能补 0 或补 1，这取决于所用的计算机系统。

4 ">>>"运算符，右边的位被挤掉，对于左边移出的空位一概补上 0。

位运算符的应用 (源操作数 s 掩码 mask)

(1) 按位与-- &

1 清零特定位 (mask 中特定位置 0，其它位为 1，s=s&mask)

2 取某数中指定位 (mask 中特定位置 1，其它位为 0，s=s&mask)

(2) 按位或-- |

常用来将源操作数某些位置 1，其它位不变。 (mask 中特定位置 1，其它位为 0 s=s|mask)

(3) 位异或-- ^

1 使特定位的值取反 (mask 中特定位置 1，其它位为 0 s=s^mask)

2 不引入第三变量，交换两个变量的值 (设 a=a1,b=b1)

目 标	操 作	操作后状态
a=a1^b1	a=a^b	a=a1^b1,b=b1
b=a1^b1^b1	b=a^b	a=a1^b1,b=a1
a=b1^a1^a1	a=a^b	a=b1,b=a1

二进制补码运算公式:

-x = ~x + 1 = ~(x-1)

~x = -x-1

-(~x) = x+1

~(-x) = x-1

x+y = x - ~y - 1 = (x|y)+(x&y)

x-y = x + ~y + 1 = (x|~y)-(~x&y)

x^y = (x|y)-(x&y)

x|y = (x&~y)+y

x&y = (~x|y)-~x

x==y:　　~(x-y|y-x)

x!=y:　　x-y|y-x

x< y:　　(x-y)^((x^y)&((x-y)^x))

x<=y:　　(x|~y)&((x^y)|~(y-x))

x< y:　　(~x&y)|((~x|y)&(x-y))//无符号 x,y 比较

x<=y:　　(~x|y)&((x^y)|~(y-x))//无符号 x,y 比较

应用举例

(1) 判断 int 型变量 a 是奇数还是偶数

a&1　= 0 偶数

　　a&1 =　1 奇数

(2) 取 int 型变量 a 的第 k 位 (k=0,1,2……sizeof(int))，即 a>>k&1

(3) 将 int 型变量 a 的第 k 位清 0，即 a=a&~(1<<k)

(4) 将 int 型变量 a 的第 k 位置 1， 即 a=a|(1<<k)

(5) int 型变量循环左移 k 次，即 a=a<<k|a>>16-k　(设 sizeof(int)=16)

(6) int 型变量 a 循环右移 k 次，即 a=a>>k|a<<16-k　(设 sizeof(int)=16)

(7)整数的平均值

对于两个整数 x,y，如果用 (x+y)/2 求平均值，会产生溢出，因为 x+y 可能会大于 INT_MAX，但是我们知道它们的平均值是肯定不会溢出的，我们用如下算法：

```
int average(int x, int y)　//返回 X,Y 的平均值
{
　　return (x&y)+((x^y)>>1);
}
```

(8)判断一个整数是不是 2 的幂,对于一个数 x >= 0，判断他是不是 2 的幂

```
boolean power2(int x)
{
　　return ((x&(x-1))==0)&&(x!=0);
}
```

(9)不用 temp 交换两个整数

```
void swap(int x , int y)
{
　　x ^= y;
　　y ^= x;
　　x ^= y;
}
```

(10)计算绝对值

```
int abs( int x )
{
int y ;
y = x >> 31 ;
return (x^y)-y ;              //or: (x+y)^y
}
```

(11)取模运算转化成位运算 (在不产生溢出的情况下)

a % (2^n) 等价于 a & (2^n - 1)

(12)乘法运算转化成位运算 (在不产生溢出的情况下)

a * (2^n) 等价于 a<< n

(13)除法运算转化成位运算 (在不产生溢出的情况下)

a / (2^n) 等价于 a>> n

例: 12/8 == 12>>3

(14) a % 2 等价于 a & 1

(15) if (x == a) x= b;

else x= a;

等价于 x= a ^ b ^ x;

(16) x 的 相反数 表示为 (~x+1)

35. 内存对齐与 ANSI C 中 struct 内存布局

发布: 2008-7-27 09:37 | 作者: 剑心通明 | 来源: 互联网 | 查看: 50 次

许多实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数 k(通常它为 4 或 8)的倍数，这就是所谓的内存对齐，而这个 k 则被称为该数据类型的对齐模数(alignment modulus)。

当一种类型 S 的对齐模数与另一种类型 T 的对齐模数的比值是大于 1 的整数，我们就称类型 S 的对齐要求比 T 强(严格)，而称 T 比 S 弱(宽松)。这种强制的要求一来简化了处理器与内存之间传输系统的设计，二来可以提升读取数据的速度。比如这么一种处理器，它每次读写内存的时候都从某个 8 倍数的地址开始，一次读出或写入 8 个字节的数据，假如软件能保证 double 类型的数据都从 8 倍数地址开始，那么读或写一个 double 类型数据就只需要一次内存操作。否则，我们就可能需要两次内存操作才能完成这个动作，因为数据或许恰好横跨在两个符合对齐要求的 8 字节内存块上。某些处理器在数据不满足对齐要求的情况下可能会出错，但是 Intel 的 IA32 架构的处理器则不管数据是否对齐都能正确工作。不过 Intel 奉劝大家，如果想提升性能，那么所有的程序数据都应该尽可能地对齐。

Win32 平台下的微软 C 编译器(cl.exe for 80x86)在默认情况下采用如下的对齐规则: 任何基本数据类型 T 的对齐模数就是 T 的大小，即 sizeof(T)。比如对于 double 类型(8 字节)，就要求该类型数据的地址总是 8 的倍数，而 char 类型数据(1 字节)则可以从任何一个地址开始。Linux 下的 GCC 奉行的是另外一套规则(在资料中查得，并未验证，如错误请指正):任何 2 字节大小(包括单字节吗?)的数据类型(比如 short)的对齐模数是 2，而其它所有超过 2 字节的数据类型(比如 Long 和 double)都以 4 为对齐模数。

现在回到我们关心的 struct 上来。ANSI C 规定一种结构类型的大小是它所有字段的大小以及字段之间或字段尾部的填充区大小之和。填充区就是为了使结构体字段满足内存对齐要求而额外分配给结构体的空间。那么结构体本身也有对齐要求，ANSI C 标准规定结构体类型的对齐要求不能比它所有字段中要求最严格的那个宽松，可以更严格(但此非强制要求，VC7.1 就仅仅是让它们一样严格)。我们来看一个例子(以下所有试验的环境是 Intel Celeron 2.4G + WIN2000 PRO + vc7.1，内存对齐编译选项是"默认"，即不指定/Zp 与/pack 选项):

```
typedef struct ms1    {         char a;         int b;    } MS1;
```

MS1 中有最强对齐要求的是 b 字段(int)，所以根据编译器的对齐规则以及 ANSIC 标准，该结构体的内存布局图如下：

这个方案在 a 与 b 之间多分配了 3 个填充(padding)字节，这样当整个 struct 对象首地址满足 4 字节的对齐要求时，b 字段也一定能满足 int 型的 4 字节对齐规定。那么 sizeof(MS1)显然就应该是 8，而 b 字段相对于结构体首地址的偏移就是 4。非常好理解，对吗？现在我们把 MS1 中的字段交换一下顺序:

```
typedef struct ms2    {         int a;         char b;    } MS2;
```

或许你认为 MS2 比 MS1 的情况要简单，它的布局应该就是

因为 MS2 对象同样要满足 4 字节对齐规定，而此时 a 的地址与结构体的首地址相等，所以它一定也是 4 字节对齐。可是却不全面。让我们来考虑一下定义一个 MS2 类型的数组会出现什么问题。C 标准保证，任何类型(包括自定义结构类型)的数组所占空间的大小一定等于一个单独的该类型数据的大小乘以数组元素的个数。换句话说，数组各元素之间不会有空隙。按照上面的方案，一个 MS2 数组 array 的布局就是:

当数组首地址是 4 字节对齐时，array[1].a 也是 4 字节对齐，可是 array[2].a 呢？array[3].a ....呢？可见这种方案在定义结构体数组时无法让数组中所有元素的字段都满足对齐规定，必须修改成如下形式:

现在无论是定义一个单独的 MS2 变量还是 MS2 数组，均能保证所有元素的所有字段都满足对齐规定。那么 sizeof(MS2)仍然是 8，而 a 的偏移为 0，b 的偏移是 4。尝试分析一个稍微复杂点的类型。typedef struct ms3 { char a; short b; double c; } MS3; 我想你一定能得出如下正确的布局图:

sizeof(short)等于 2，b 字段应从偶数地址开始，所以 a 的后面填充一个字节，而 sizeof(double)等于 8，c 字段要从 8 倍数地址开始，前面的 a、b 字段加上填充字节已经有 4 bytes，所以 b 后面再填充 4 个字节就可以保证 c 字段的对齐要求了。sizeof(MS3)等于 16，b 的偏移是 2，c 的偏移是 8。接着看看结构体中字段还是结构类型的情况:

typedef struct ms4 { char a; MS3 b; } MS4; MS3 中内存要求最严格的字段是 c，那么 MS3 类型数据的对齐模数就与 double 的一致(为 8)，a 字段后面应填充 7 个字节，因此 MS4 的布局应该是:

显然，sizeof(MS4)等于 24，b 的偏移等于 8。

在实际开发中，我们可以通过指定/Zp 编译选项或者在代码中用#pragma pack 指令来更改编译器的对齐规则。比如指定/Zpn(VC7.1 中 n 可以是 1、2、4、8、16)就是告诉编译器最大对齐模数是 n。或者定义结构时

```
#pragma pack(push, n)
typedef struct ms3 { char a; short b; double c; } MS3;
#pragma pack(pop);
```

在这种情况下，所有小于等于 n 字节的基本数据类型的对齐规则与默认的一样，但是大于 n 个字节的数据类型的对齐模数被限制为 n。如果 n = 1,那么结构体的大小就是各个字段的大小之和，在 Moses 中定义结构体的地方随处可见，这样做可以减少结构体所占用的内存空间。

VC7.1 的默认对齐选项就相当于/Zp8。仔细看看 MSDN 对这个选项的描述，会发现它郑重告诫了程序员不要在 MIPS 和 Alpha 平台上用/Zp1 和/Zp2 选项，也不要在 16 位平台上指定/Zp4 和/Zp8(想想为什么？)。 结构体的内存布局依赖于 CPU、操作系统、编译器及编译时的对齐选项，而你的程序可能需要运行在多种平台上，你的源代码可能要被不同的人用不同的编译器编译(试想你为别人提供一个开放源码的库)，那么除非绝对必需，否则你的程序永远也不要依赖这些诡异的内存布局。顺便说一下，如果一个程序中的两个模块是用不同的对齐选项分别编译的，那么它很可能会产生一些非常微妙的错误。如果你的程序确实有很难理解的行为，不防仔细检查一下各个模块的编译选项。

问题：下面的试验，请问如何解释？

```
#pragma pack(push, 2)
struct s
{
```

```
    char a;
};
#pragma pack (pop)

void TestPack()
{
    s c[2];
    assert(sizeof(s)==1);
    assert(sizeof(c)==2);
}

int _tmain(int argc, _TCHAR* argv[])
{
    TestPack();
    return 0;
}
```

解答：

每一种基本的数据类型都有该数据类型的对齐模数(alignment modulus)。Win32 平台下的微软 C 编译器(cl.exe for 80x86)在默认

情况下：任何基本数据类型 T 的对齐模数就是 T 的大小，即 sizeof(T)。

一组可能的对齐模数数据如下：

数据类型	模数
char	1
shor	2
int	4
double	8

ANSI C 规定一种结构类型的大小是它所有字段的大小以及字段之间或字段尾部的填充区大小之和。

注：填充区就是为了使结构体字段满足内存对齐要求而额外分配给结构体的空间。

产生填充区的条件：

当结构体中的成员一种类型 S 的对齐模数与另一种类型 T 的对齐模数不一致的时候，才可能产生填充区。

我们通过编译选项设置/zpn 或#pragma pack(push, n) 来设置内存对齐模数时，当结构体中的某中基本数据类型的对齐模数大于 n 时才会影响填充区的大小，否则将会按照基本数据类型的对齐模数进行对齐。

例子：

当 n = 1 时：

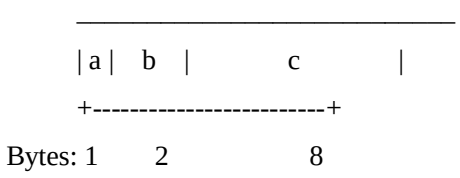
#pragma pack(push,1)

typedef struct ms3{char a; short b; double c; } MS3;

#pragma pack(pop)

这时 n=1,此结构中基本数据类型 short 的对齐模数为 2,double 为 8,大于 n 所以将会影响这两个变量存储时地址的偏移量，必须是 n 的整数倍,而 char 的对齐模数是 1，小于等于 n,将会按照其自身的对齐模数 1 进行对齐.

因为 n=1,所以这三个变量在内存中是连续的而不存在填充区.内存布局如下：



sizeof(MS3) = 11

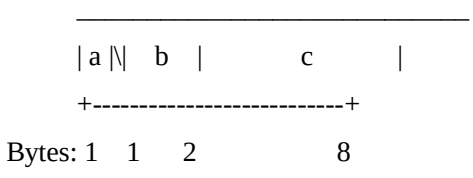
当 n = 2 时：

#pragma pack(push,2)

typedef struct ms3{char a; short b; double c; } MS3;

#pragma pack(pop)

这时 n=2,此结构中基本数据类型 double 的对齐模数为 8,大于 n, 所以将会影响这个变量存储时地址的偏移量，必须是 n 的整数倍,而 char 和 short 的对齐模数小于等于 n, 将会按照其自身的对齐模数分别是 1,2 进行对齐.内存布局如下：



此时变量 c 的存储地址偏移是 4,是 n=2 的整数倍,当然偏移为 6,8 等等时也满足这个条件，但编译器不至于愚蠢到这种地步白白浪费空间，呵。

sizeof(MS3) = 12

当 n = 4 时：与 n=2 时结果是一样的.

当 n = 8 时:

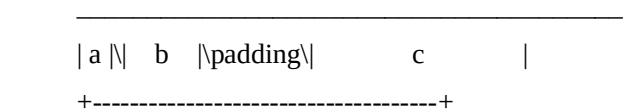
#pragma pack(push,8)

typedef struct ms3{char a; short b; double c; } MS3;

#pragma pack(pop)

这时 n=8,此结构中 char ,short ,double 的对齐模数为都,小于等于 n，将会按照其自身的对齐模数分别是 1,2,8 进行对齐.即：short 变量存储时地址的偏移量是 2 的倍数;double 变量存储时地址的偏移量是 8 的倍数.

内存布局如下：



Bytes: 1 1 2 4 8

此时变量 a 的存储地址偏移是 0,当然也是 char 型对齐模数 1 的整数倍了

变量 b 的存储地址偏移要想是 short 型对齐模数 2 的整数倍，因为前面 a 占了 1 个 byte ，所以至少在 a 与 b 之间再加上 1 个 byte 的 padding.才能满足条件。

变量 c 的存储地址骗移要想是 double 型对齐模数 8 的整数倍，因为前面 a 和 b 加 1 个 byte 的 padding，共 4 bytes 所以最少还需要 4 bytes 的 padding 才能满足条件。

sizeof(MS3) = 16

当 n = 16 时：与 n=8 时结果是一样的.

==========

根据上面的分析，如下定义的结构

```
#pragma pack(push, 2)
struct s
{
    char a;
};
#pragma pack (pop)
```

因为 char 的对齐模数是 1,小于 n=2,所以将按照自身的对齐模数对齐。根本就不会存在填充区,所以 sizeof(s) = 1.对于 s c[2]; sizeof(c)==2 也是必然的。

再看下面的结构:

```
#pragma pack(push, n)//n=(1,2,4,8,16)
struct s
{
    double a;
    double b;
    double c;
};
#pragma pack (pop)
```

对于这样的结构无论 pack 设置的对齐模数为几都不会影响其大小，即无 padding.

double 类型的对齐模数为 8

当 n<8 时，虽然满足前面讲的规则：当结构体中的某中基本数据类型的对齐模数大于 n 时才会影响填充区的大小。但这个时候无论 n 等于几(1,2,4)，double 变量存储时地址的偏移量都是 n 的整数倍，所以根本不需要填充区。当 n>=8 时，自然就按照 double 的对齐模数进行对齐了.因为类型都一样所以变量之间在内存中不会存在填充区.

补充一点：

如果在定义结构的时候，仔细调整顺序，适当明确填充方式，则最终内存结果可以与编译选项/Zpn 或 pack 无关。

举个例子：

typedef struct ms1{ char a; char b; int c; short d; } MS1;

在不同的 /Zpn 下，sizeof(MS1)的长度可能不同，也就是内存布局不同。

如果改成

typedef struct ms2{ char a; char b; short d; int c; } MS2;

即便在不同的/Zpn 或 pack 方式下，编译生成的内存布局总是相同的；

再比如：

typedef struct ms3{ char a; char b; int c; } MS3;

可以改写成：

typedef struct ms4{ char a; char b; short padding; int c; } MS4;　　显式地写上 padding

（通过源代码本身来消除隐患，要比依赖编译选项更加可靠，并易于移植，优质的代码应该做到这一点）

（减少隐含 padding 的另外一个好处是少占内存，当结构的实例数量很大时，内存的节省量是非常可观的）

（以上的变量/结构命名没有遵循命名规范，只为说明用，不可模仿）

36. 冒泡和选择排序实现

发布: 2008-7-20 07:39 | 作者: 剑心通明 | 来源: 互联网 | 查看: 7 次

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//冒泡排序
void bubbleSort(int *a,int len)
{
	int i,j,temp;
	for(i = 0;i<len-1;i++)
	{
		for(j=0;j<len -i-1;j++)
		{
			if(a[j]>a[j+1])
			{
				temp=a[j];
				a[j]=a[j+1];
				a[j+1]=temp;
			}
		}
	}
}
//选择排序
void selectSort(int *a,int len)
{
	int i,j,temp,result;
	for(i=0;i<len-1 ;i++)
	{
		temp=i;
		for(j=i+1;j<len-1;j++)
		{
			if(a[j]<a[temp])
			{
				temp=j;
			}
```

```
            }
            if(temp!=i)
             {
                    result=a[i];
                    a[i]=a[temp];
                    a[i]=result;
             }
       }
}

void print(int *a,int len)
{
      int i=0;
      for(i=0;i<len;i++)
      {
            printf("%d ",a[i]);
      }
      printf("\n");
}


int main()
{
      int value[10]={38,6,14,9,7,33,67,12,34,51};
      printf("bubbleSort result:\n");
      bubbleSort(value,10);
      print(value,10);
      printf("bubbleSort result:\n");
      selectSort(value,10);
      print(value,10);
      return 0;

}
```

写坏 C 程序的几大诀窍

如果你常常按照下面方式写程序，可以保证你的程序经常出错，你也经常头疼。

---

1) 根本不管程序的格式，把程序写得老师也看不懂（自己当然更看不懂）。

用格式迷惑自己也是弄坏程序的绝招。例如：

```
if (a > 0)
    if (x == y) {
        ....
        };
    else {
        ....
    }
```

请自己分析这到底是什么意思。

2）不关心 scanf 或者 printf 中格式串和对应参数类型匹配的问题。例如(假设 x, y 是 double 类型，n 是 int 类型)写：

```
  printf("%d, %f", x, n);
scanf("%d %f", &x, &y, &n);
```

这样可以保证输入输出总出现莫名其妙的问题，而且你和别人都找不到原因。

3）写 scanf 时，在接收输入值的变量名之前不写&符号。

这样做常常还可以顺便摧毁你所使用的计算机系统，给自己再多找点麻烦，有自制病毒之妙。

4）写注释时随便地忘记几个结束符号。例如：

```
x = y + 1; /* ha ha ha
z = x * 2; /* fine fine fine */
```

这样可以保证编译程序“按照你的指示”把你的一段代码吃掉。

5）在比较的时候用 = 代替 ==。例如：

```
if (x = y)
    z = x + 5;
```

这可以保证 x 和 y 一定相同，而是否执行赋值就看你的运气了。

6）定义局部变量后，不初始化就使用。例如：

```
int fun(int n) {
    int m;
    return n + m;
}
```

如果对指针变量这样做，常可以带来隐秘的破坏效果，让最有经验的人也难找到毛病的根源。例如写：

```
int fun(int n) {
    int* p;
    *p = n*n;
    return *p + n;
```

}

这样做有时可以顺便摧毁你用的计算机系统，再给自己多找点麻烦。

7）函数定义为有返回值的，但（有时）却不去写 return（101-102 页）。
这样做可以保证你的程序不时出现古怪行为。

8）不关心变量的范围，例如对循环次数很多的变量用 short 类型做循环等。
这样做可以得到不明不白的结果；有时还可以使程序永不停止，直到你用的计算机累死为止。

9）用 sizeof 运算符去计算函数的数组参数的大小，或者计算字符串的大小。
这样做一般可以保证计算结果错误，出现的问题难以预料。

10）不写函数原型说明，采用过时原型说明形式（103-107 页），或者故意写错误的原型说明。
这样做一般都能骗过编译程序，阻止它帮助你检查程序错误，使你自己麻烦多多。

11）定义带参数宏的时候尽量节约括号，省得写起来麻烦。

这样可以保证在使用宏的地方不时地出现隐含错误，就像埋下的地雷，检查源程序也很难发现。

37. 函数指针数组与返回数组指针的函数

发布: 2008-7-21 00:27 | 作者: 剑心通明 | 来源: 互联网 | 查看: 13 次

int (*a[])(int);

int (*p())[10];

第一种情况为数组里面是函数指针的情况,因为(int (*)(int))是一个强制转换方式, 将里面的 a[]这个数组转换成了一个函数指针的数组, 并且该函数是一个带一个整型变量,并且返回一个整型的函数.

第二种情况为函数返回的为指向一个一维数组的指针的情况. 因为(int (*)[10])将其强制转换成了一个指针,而该指针则是一个指向一维数组的指针.

分别举两个例子进行说明:

对于第一种函数指针数组的情况,如下:

```
#include<stdio.h>
int fun(int a)
{
        return a+1;
}
int main()
{
        int (*p[10])(int);
        int i;
        p[0] = fun;
        i = (p[0])(10);
        printf("i=%d\n", i);
        return 0;
}
```

p[10]为一个指针数组,而该指针数组里面的值为函数指针类型.让 p[0]指向 fun()函数.然后再进行调用,就可以调用到 fun()这个函数了.

对于第二种返回数组指针的函数,如下:

```
#include<stdio.h>
#include<stdlib.h>
int (*p())[10]
{
 int (*m)[10];
 int i;
 m = (int (*)[10])calloc(10, sizeof(int));
 if (m == NULL)
 {
```

```
    printf("calloc error\n");
    exit(1);
  }
  for (i = 0; i < 10; i++)
    *(*m+i) = i+1;
  return m;
}

int main()
{
  int (*a)[10];
  int i;
  a = p();
  for (i = 0; i < 10; i++)
    printf("%d ", *(*a+i));
  printf("\ndone\n");
  return 0;
}
```

其实 int (*m)[10];这种方式一般是用来指向一个二维数组的,例如

int b[4][10];

int (*m)[10] = b;

其指向二维数组中的一维.

使用*(*(m+i)+j);这种方式就可以访问 b[i][j]这个元素.而上面的是使用这种方式来指向一个一维数组,同样也是一样的.只是前面的*(m+i)中的 i 变为 0 了.因为只有一维大小.即

int a[10];

int (*m)[10] = &a;

就使得 m 指向了 a 这个数组了.而平时所用的 int *p = a;只是让 p 指向了 a 的第一个元素.比前面的指向一维数组的指针少了一维.前面的 m+1 跳过的是 10 个整型的长度.而后面的 p+1 则只是跳过了 1 个整型的长度.

38. 右左法则- 复杂指针解析

发布: 2008-7-22 16:12 | 作者: 剑心通明 | 来源: 互联网 | 查看: 66 次

The right-left rule: Start reading the declaration from the innermost parentheses, go right, and then go left. When you encounter parentheses, the direction should be reversed. Once everything in the parentheses has been parsed, jump out of it. Continue till the whole declaration has been parsed.

这段英文的翻译如下：

右左法则：首先从最里面的圆括号看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。

笔者要对这个法则进行一个小小的修正，应该是从未定义的标识符开始阅读，而不是从括号读起，之所以是未定义的标识符，是因为一个声明里面可能有多个标识符，但未定义的标识符只会有一个。

现在通过一些例子来讨论右左法则的应用，先从最简单的开始，逐步加深：

```
int (*func)(int *p);
```

首先找到那个未定义的标识符，就是 func，它的外面有一对圆括号，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，也是一个圆括号，这说明(*func)是一个函数，而 func 是一个指向这类函数的指针，就是一个函数指针，这类函数具有 int*类型的形参，返回值类型是 int。

```
int (*func)(int *p, int (*f)(int*));
```

func 被一对括号包含，且左边有一个*号，说明 func 是一个指针，跳出括号，右边也有个括号，那么 func 是一个指向函数的指针，这类函数具有 int *和 int (*)(int*)这样的形参，返回值为 int 类型。再来看一看 func 的形参 int (*f)(int*)，类似前面的解释，f 也是一个函数指针，指向的函数具有 int*类型的形参，返回值为 int。

```
int (*func[5])(int *p);
```

func 右边是一个[]运算符，说明 func 是一个具有 5 个元素的数组，func 的左边有一个*，说明 func 的元素是指针，要注意这里的*不是修饰 func 的，而是修饰 func[5]的，原因是[]运算符优先级比*高，func 先跟[]结合，因此*修饰的是 func[5]。跳出这个括号，看右边，也是一对圆括号，说明 func 数组的元素是函数类型的指针，它所指向的函数具有 int*类型的形参，返回值类型为 int。

int (*(*func)[5])(int *p);

func 被一个圆括号包含，左边又有一个*，那么 func 是一个指针，跳出括号，右边是一个[]运算符号，说明 func 是一个指向数组的指针，现在往左看，左边有一个*号，说明这个数组的元素是指针，再跳出括号，右边又有一个括号，说明这个数组的元素是指向函数的指针。总结一下，就是：func 是一个指向数组的指针，这个数组的元素是函数指针，这些指针指向具有 int*形参，返回值为 int 类型的函数。

int (*(*func)(int *p))[5];

func 是一个函数指针，这类函数具有 int*类型的形参，返回值是指向数组的指针，所指向的数组的元素是具有 5 个 int 元素的数组。

要注意有些复杂指针声明是非法的，例如：

int func(void) [5];

func 是一个返回值为具有 5 个 int 元素的数组的函数。但 C 语言的函数返回值不能为数组，这是因为如果允许函数返回值为数组，那么接收这个数组的内容的东西，也必须是一个数组，但 C 语言的数组名是一个右值，它不能作为左值来接收另一个数组，因此函数返回值不能为数组。

int func[5](void);

func 是一个具有 5 个元素的数组，这个数组的元素都是函数。这也是非法的，因为数组的元素除了类型必须一样外，每个元素所占用的内存空间也必须相同，显然函数是无法达到这个要求的，即使函数的类型一样，但函数所占用的空间通常是不相同的。

作为练习，下面列几个复杂指针声明给读者自己来解析，答案放在第十章里。

int (*(*func)[5][6])[7][8];

int (*(*(*func)(int *))[5])(int *);

int (*(*func[7][8][9])(int*))[5];

实际当中，需要声明一个复杂指针时，如果把整个声明写成上面所示的形式，对程序可读性是一大损害。应该用 typedef 来对声明逐层分解，增强可读性，例如对于声明：

int (*(*func)(int *p))[5];

可以这样分解：

```
typedef    int (*PARA)[5];
typedef PARA (*func)(int *);
```

这样就容易看得多了。

39. 回车和换行的区别

发布: 2008-7-21 18:50 | 作者: 剑心通明 | 来源: 互联网 | 查看: 1 次

在计算机还没有出现之前，有一种叫做电传打字机（Teletype Model 33）的玩意，每秒钟可以打 10 个字符。但是它有一个问题，就是打完一行换行的时候，要用去 0.2 秒，正好可以打两个字符。要是在这 0.2 秒里面，又有新的字符传过来，那么这个字符将丢失。

于是，研制人员想了个办法解决这个问题，就是在每行后面加两个表示结束的字符。一个叫做“回车”，告诉打字机把打印头定位在左边界；另一个叫做“换行”，告诉打字机把纸向下移一行。

这就是“换行”和“回车”的来历，从它们的英语名字上也可以看出一二。

后来，计算机发明了，这两个概念也就被般到了计算机上。那时，存储器很贵，一些科学家认为在每行结尾加两个字符太浪费了，加一个就可以。于是，就出现了分歧。

Unix 系统里，每行结尾只有“<换行>”，即“\n”；Windows 系统里面，每行结尾是“<换行><回车>”，即“\n\r”；Mac 系统里，每行结尾是“<回车>”。一个直接后果是，Unix/Mac 系统下的文件在 Windows 里打开的话，所有文字会变成一行；而 Windows 里的文件在 Unix/Mac 下打开的话，在每行的结尾可能会多出一个^M 符号。

c 语言编程时（windows 系统）

\r 就是 return 回到 本行 行首 这就会把这一行以前的输出 覆盖掉

如：

```
int main() {

cout << "hahaha" << "\r" << "xixi" ;



}
```

最后只显示 xixi 而 hahaha 背覆盖了

\n 是回车＋换行 把光标 先移到 行首 然后换到下一行 也就是 下一行的行首拉

```
int main() {

cout << "hahaha" << "\n" << "xixi" ;
```

}

则 显示

hahaha

xixi

40. 堆和堆栈的区别

发布: 2008-7-21 16:37 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

一、预备知识—程序的内存分配

一个由 c/C++编译的程序占用的内存分为以下几个部分

1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。

3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放

4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放

5、程序代码区—存放函数体的二进制代码。

二、例子程序

这是一个前辈写的，非常详细

//main.cpp

int a = 0; 全局初始化区

char *p1; 全局未初始化区

main()

{

int b; 栈

char s[] = "abc"; 栈

char *p2; 栈

char *p3 = "123456"; 123456\0 在常量区，p3 在栈上。

static int c =0； 全局（静态）初始化区

p1 = (char *)malloc(10);

p2 = (char *)malloc(20);

分配得来得 10 和 20 字节的区域就在堆区。

strcpy(p1, "123456"); 123456\0 放在常量区，编译器可能会将它与 p3 所指向的"123456"优化成一个地方。

}

二、堆和栈的理论知识

2.1 申请方式

stack:

由系统自动分配。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为 b 开辟空间

heap:

需要程序员自己申请，并指明大小，在 c 中 malloc 函数

如 p1 = (char *)malloc(10);

在 C++中用 new 运算符

如 p2 = (char *)malloc(10);

但是注意 p1、p2 本身是在栈中的。

2.2

申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，

会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 delete 语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

2.3 申请大小的限制

栈：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：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。

2.4 申请效率的比较：

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 new 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.

另外，在 WINDOWS 下，最好的方式是用 VirtualAlloc 分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活

2.5 堆和栈中的存储内容

栈： 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

2.6 存取效率的比较

char s1[] = "aaaaaaaaaaaaaa";

char *s2 = "bbbbbbbbbbbbbbbb";

aaaaaaaaaa 是在运行时刻赋值的；

而 bbbbbbbbbb 是在编译时就确定的；

但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。

比如：

```
#include
void main()
{
char a = 1;
char c[] = "1234567890";
char *p ="1234567890";
a = c[1];
a = p[1];
return;
}
```

对应的汇编代码

```
10: a = c[1];
00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]
0040106A 88 4D FC mov byte ptr [ebp-4],cl
11: a = p[1];
0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]
00401070 8A 42 01 mov al,byte ptr [edx+1]
00401073 88 45 FC mov byte ptr [ebp-4],al
```

第一种在读取时直接就把字符串中的元素读到寄存器 cl 中，而第二种则要先把指针值读到 edx 中，在根据 edx 读取字符，显然慢了。

?

2.7 小结：

堆和栈的区别可以用如下的比喻来看出：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

堆和栈的区别主要分：

操作系统方面的堆和栈，如上面说的那些，不多说了。

还有就是数据结构方面的堆和栈，这些都是不同的概念。这里的堆实际上指的就是（满足堆性质的）优先队列的一种数据结构，第 1 个元素有最高的优先权；栈实际上就是满足先进后出的性质的数学或数据结构。

虽然堆栈，堆栈的说法是连起来叫，但是他们还是有很大区别的，连着叫只是由于历史的原因。

41. 堆和堆栈的区别

发布: 2008-7-21 16:37 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

一、预备知识—程序的内存分配

一个由 c/C++编译的程序占用的内存分为以下几个部分

1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。

3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放

4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放

5、程序代码区—存放函数体的二进制代码。

二、例子程序

这是一个前辈写的，非常详细

//main.cpp

int a = 0; 全局初始化区

char *p1; 全局未初始化区

main()

{

int b; 栈

char s[] = "abc"; 栈

char *p2; 栈

char *p3 = "123456"; 123456\0 在常量区，p3 在栈上。

static int c =0； 全局（静态）初始化区

p1 = (char *)malloc(10);

p2 = (char *)malloc(20);

分配得来得 10 和 20 字节的区域就在堆区。

strcpy(p1, "123456"); 123456\0 放在常量区，编译器可能会将它与 p3 所指向的"123456"优化成一个地方。

}

二、堆和栈的理论知识

2.1 申请方式

stack:

由系统自动分配。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为 b 开辟空间

heap:

需要程序员自己申请，并指明大小，在 c 中 malloc 函数

如 p1 = (char *)malloc(10);

在 C++中用 new 运算符

如 p2 = (char *)malloc(10);

但是注意 p1、p2 本身是在栈中的。

2.2

申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，

会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 delete 语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

2.3 申请大小的限制

栈：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：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。

2.4 申请效率的比较：

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 new 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.

另外，在 WINDOWS 下，最好的方式是用 VirtualAlloc 分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活

2.5 堆和栈中的存储内容

栈： 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

2.6 存取效率的比较

char s1[] = "aaaaaaaaaaaaaa";

char *s2 = "bbbbbbbbbbbbbbbb";

aaaaaaaaaa 是在运行时刻赋值的；

而 bbbbbbbbbb 是在编译时就确定的；

但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。

比如：

```
#include
void main()
{
char a = 1;
char c[] = "1234567890";
char *p ="1234567890";
a = c[1];
a = p[1];
return;
}
```

对应的汇编代码

```
10: a = c[1];
00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]
0040106A 88 4D FC mov byte ptr [ebp-4],cl
11: a = p[1];
0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]
00401070 8A 42 01 mov al,byte ptr [edx+1]
00401073 88 45 FC mov byte ptr [ebp-4],al
```

第一种在读取时直接就把字符串中的元素读到寄存器 cl 中，而第二种则要先把指针值读到 edx 中，在根据 edx 读取字符，显然慢了。

?

2.7 小结：

堆和栈的区别可以用如下的比喻来看出：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

堆和栈的区别主要分：

操作系统方面的堆和栈，如上面说的那些，不多说了。

还有就是数据结构方面的堆和栈，这些都是不同的概念。这里的堆实际上指的就是（满足堆性质的）优先队列的一种数据结构，第 1 个元素有最高的优先权；栈实际上就是满足先进后出的性质的数学或数据结构。

虽然堆栈，堆栈的说法是连起来叫，但是他们还是有很大区别的，连着叫只是由于历史的原因。

42. 如何写出专业的 C 头文件

发布: 2008-7-21 16:40 | 作者: 剑心通明 | 来源: 互联网 | 查看: 15 次

做到专业，应该是每个职业程序员应该要求自己做到的。
让我们看看 lua 是怎么写头文件的。

1.License Agreement
License Agreement 应该加在每个头文件的顶部。
Lua Sample:

```
/*
** $Id: lua.h,v 1.175b 2003/03/18 12:31:39 roberto Exp $
** Lua - An Extensible Extension Language
** Tecgraf: Computer Graphics Technology Group, PUC-Rio, Brazil
**http://www.lua.org       mailto:info@lua.org
** See Copyright Notice at the end of this file
*/
```

2.guard define
整个头文件应该在 guard define 之间

```
#ifndef lua_h
#define lua_h

#endif
```

另外，如果这个头文件可能给 c++使用，要加上

```
#ifdef __cplusplus
extern "C" {
#endif
/*The lines within extern "C" */
```

```
#ifdef __cplusplus
}
#endif
```

3.尽量不要在头文件中暴露数据结构

这样可以用户对你的实现的依赖，也减少了用户的编译时间

```
typedef struct lua_State lua_State;
LUA_API lua_State *lua_open (void);
LUA_API void       lua_close (lua_State *L);
```

可以看到虽然用户会一直使用 lua_State,但是并不知道 lua_State 的结构是什么

从一个使用 lua 的例子程序可以看出：

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"


int main(int argc, char *argv[])
{
	lua_State *L = lua_open();
	const char *buf = "var = 100";
	int var ;
	luaopen_base(L);
	luaopen_io(L);
	lua_dostring(L, buf);
	lua_getglobal(L, "var");
	var = lua_tonumber(L, -1);
	lua_close(L);
	return 0;
}
```

4．函数声明前加 XXX_API 已利于拓展

Lua 的例子

```
#ifndef LUA_API

#define LUA_API                extern

#endif
```

```
LUA_API lua_State *lua_open (void);
```

如果定义了 LUA_API 就是给 LUA 内部使用的

如果没定义 LUA_API 就是 for user 的

写 Window dll 程序经常会用到

```
#ifdef DLLTEST_EXPORTS

#define DLLTEST_API __declspec(dllexport)

#else

#define DLLTEST_API __declspec(dllimport)

#endif
```

5.宏的定义

尽量使用括号来包住所定义的对象

```
#define LUA_TNONE		(-1)
```

```
#define lua_register(L,n,f) \
        (lua_pushstring(L, n), \
         lua_pushcfunction(L, f), \
         lua_settable(L, LUA_GLOBALSINDEX))
```

6.目录结构

一般应该使用一个单独的 include 目录来包含要发布的头文件，但不应该把内部使用的头文件包含进去。

Lua 的 include 目录只包含了三个头文件

lauxlib.h , lua.h, lualib.h

非常简洁

43. 打造最快的 Hash 表

发布: 2008-7-27 09:57 | 作者: 剑心通明 | 来源: 互联网 | 查看: 23 次

一个简单的问题：有一个庞大的字符串数组，然后给你一个单独的字符串，让你从这个数组中查找是否有这个字符串并找到它，你会怎么做？有一个方法最简单，老老实实从头查到尾，一个一个比较，直到找到为止，我想只要学过程序设计的人都能把这样一个程序作出来，但要是有程序员把这样的程序交给用户，我只能用无语来评价，或许它真的能工作，但...也只能如此了。

最合适的算法自然是使用 HashTable（哈希表），先介绍介绍其中的基本知识，所谓 Hash，一般是一个整数，通过某种算法，可以把一个字符串"压缩" 成一个整数。当然，无论如何，一个 32 位整数是无法对应回一个字符串的，但在程序中，两个字符串计算出的 Hash 值相等的可能非常小，下面看看在 MPQ 中的 Hash 算法：

以下的函数生成一个长度为 0x500（合 10 进制数：1280）的 cryptTable[0x500]

```
void prepareCryptTable()

{

    unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;



    for( index1 = 0; index1 < 0x100; index1++ )

    {

        for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )

        {

            unsigned long temp1, temp2;



            seed = (seed * 125 + 3) % 0x2AAAAB;

            temp1 = (seed & 0xFFFF) << 0x10;
```

```
            seed = (seed * 125 + 3) % 0x2AAAAB;
            temp2 = (seed & 0xFFFF);

            cryptTable[index2] = ( temp1 | temp2 );
        }
    }
}
```

以下函数计算 lpszFileName 字符串的 hash 值，其中 dwHashType 为 hash 的类型，在下面 GetHashTablePos 函数里面调用本函数，其可以取的值为 0、1、2；该函数返回 lpszFileName 字符串的 hash 值；

```
unsigned long HashString( char *lpszFileName, unsigned long dwHashType )
{
    unsigned char *key  = (unsigned char *)lpszFileName;
unsigned long seed1 = 0x7FED7FED;
unsigned long seed2 = 0xEEEEEEEE;
    int ch;
```

```
    while( *key != 0 )
    {
        ch = toupper(*key++);

        seed1 = cryptTable[(dwHashType << 8) + ch] ^ (seed1 + seed2);
        seed2 = ch + seed1 + seed2 + (seed2 << 5) + 3;
    }
    return seed1;
}
```

Blizzard 的这个算法是非常高效的，被称为"One-Way Hash"(A one-way hash is a an algorithm that is constructed in such a way that deriving the original string (set of strings, actually) is virtually impossible)。举个例子，字符串"unitneutralacritter.grp"通过这个算法得到的结果是 0xA26067F3。

是不是把第一个算法改进一下，改成逐个比较字符串的 Hash 值就可以了呢，答案是，远远不够，要想得到最快的算法，就不能进行逐个的比较，通常是构造一个哈希表(Hash Table)来解决问题，哈希表是一个大数组，这个数组的容量根据程序的要求来定义，例如 1024，每一个 Hash 值通过取模运算 (mod) 对应到数组中的一个位置，这样，只要比较这个字符串的哈希值对应的位置又没有被占用，就可以得到最后的结果了，想想这是什么速度？是的，是最快的 O(1)，现在仔细看看这个算法吧：

```
typedef struct
{
    int nHashA;
    int nHashB;
    char bExists;
   .....
} SOMESTRUCTRUE;
```

一种可能的结构体定义？

lpszString 为要在 hash 表中查找的字符串；lpTable 为存储字符串 hash 值的 hash 表

```
int GetHashTablePos( har *lpszString, SOMESTRUCTURE *lpTable )
{
    int nHash = HashString(lpszString);
    int nHashPos = nHash % nTableSize;

    if ( lpTable[nHashPos].bExists  &&  !strcmp( lpTable[nHashPos].pString, lpszString ) )
    {
        return nHashPos;
```

```
        }

        else

        {

            return -1;

        }

    }
```

看到此，我想大家都在想一个很严重的问题：“如果两个字符串在哈希表中对应的位置相同怎么办？”,毕竟一个数组容量是有限的，这种可能性很大。解决该问题的方法很多，我首先想到的就是用“链表”,感谢大学里学的数据结构教会了这个百试百灵的法宝，我遇到的很多算法都可以转化成链表来解决，只要在哈希表的每个入口挂一个链表，保存所有对应的字符串就 OK 了。事情到此似乎有了完美的结局，如果是把问题独自交给我解决，此时我可能就要开始定义数据结构然后写代码了。然而 Blizzard 的程序员使用的方法则是更精妙的方法。基本原理就是：他们在哈希表中不是用一个哈希值而是用三个哈希值来校验字符串。

MPQ 使用文件名哈希表来跟踪内部的所有文件。但是这个表的格式与正常的哈希表有一些不同。首先，它没有使用哈希作为下标，把实际的文件名存储在表中用于验证，实际上它根本就没有存储文件名。而是使用了 3 种不同的哈希：一个用于哈希表的下标，两个用于验证。这两个验证哈希替代了实际文件名。

当然了，这样仍然会出现 2 个不同的文件名哈希到 3 个同样的哈希。但是这种情况发生的概率平均是 1:18889465931478580854784，这个概率对于任何人来说应该都是足够小的。现在再回到数据结构上，Blizzard 使用的哈希表没有使用链表，而采用"顺延"的方式来解决问题，看看这个算法：

lpszString 为要在 hash 表中查找的字符串；lpTable 为存储字符串 hash 值的 hash 表；nTableSize 为 hash 表的长度；

```
int GetHashTablePos( char *lpszString, MPQHASHTABLE *lpTable, int nTableSize )

{
```

```
    const int    HASH_OFFSET = 0, HASH_A = 1, HASH_B = 2;


    int    nHash = HashString( lpszString, HASH_OFFSET );

    int    nHashA = HashString( lpszString, HASH_A );

    int    nHashB = HashString( lpszString, HASH_B );

    int    nHashStart = nHash % nTableSize;

    int    nHashPos = nHashStart;


    while ( lpTable[nHashPos].bExists )

{

/*如果仅仅是判断在该表中时候存在这个字符串，就比较这两个 hash 值就可以了，不用对

*结构体中的字符串进行比较。这样会加快运行的速度？减少 hash 表占用的空间？这种

*方法一般应用在什么场合？ */

            if (       lpTable[nHashPos].nHashA == nHashA

&&    lpTable[nHashPos].nHashB == nHashB )

{

   return nHashPos;

}

              else
```

```
{
                nHashPos = (nHashPos + 1) % nTableSize;
}


            if (nHashPos == nHashStart)
                break;
        }
        return -1;
}
```

1. 计算出字符串的三个哈希值（一个用来确定位置，另外两个用来校验)
2. 察看哈希表中的这个位置
3. 哈希表中这个位置为空吗？如果为空，则肯定该字符串不存在，返回
4. 如果存在，则检查其他两个哈希值是否也匹配，如果匹配，则表示找到了该字符串，返回
5. 移到下一个位置，如果已经移到了表的末尾，则反绕到表的开始位置起继续查询
6. 看看是不是又回到了原来的位置，如果是，则返回没找到
7. 回到 3

补充 1：其他比较简单一些的 hash 函数：

```
/*key 为一个字符串，nTableLength 为哈希表的长度
*该函数得到的 hash 值分布比较均匀*/
unsigned long getHashIndex( const char *key, int nTableLength )
{
    unsigned long nHash = 0;

    while (*key)
    {
        nHash = (nHash<<5) + nHash + *key++;
    }
```

```
    return ( nHash % nTableLength );

}
```

补充 2：

哈希表的数组是定长的，如果太大，则浪费，如果太小，体现不出效率。合适的数组大小是哈希表的性能的关键。哈希表的尺寸最好是一个质数。当然，根据不同的数据量，会有不同的哈希表的大小。对于数据量时多时少的应用，最好的设计是使用动态可变尺寸的哈希表，那么如果你发现哈希表尺寸太小了，比如其中的元素是哈希表尺寸的 2 倍时，我们就需要扩大哈希表尺寸，一般是扩大一倍。下面是哈希表尺寸大小的可能取值：

17, 37, 79, 163, 331,

673, 1361, 2729, 471, 10949,

21911, 43853, 87719, 175447, 350899,

701819, 1403641, 2807303, 5614657, 11229331,

22458671, 44917381, 89834777, 179669557, 359339171,

718678369, 1437356741, 2147483647

以下为该程序的源代码，在 linux 下测试通过：

```
#include <stdio.h>

/*crytTable[]里面保存的是 HashString 函数里面将会用到的一些数据，在 prepareCryptTable
 *函数里面初始化*/
unsigned long cryptTable[0x500];

 /*************************************************************
  *以下的函数生成一个长度为 0x500（合 10 进制数：1280）的 cryptTable[0x500]
  *
  *
  *************************************************************/
void prepareCryptTable()
{
      unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;
```

```
    for( index1 = 0; index1 < 0x100; index1++ )
    {
        for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )
        {
            unsigned long temp1, temp2;

            seed = (seed * 125 + 3) % 0x2AAAAB;
            temp1 = (seed & 0xFFFF) << 0x10;

            seed = (seed * 125 + 3) % 0x2AAAAB;
            temp2 = (seed & 0xFFFF);

            cryptTable[index2] = ( temp1 | temp2 );
        }
    }
}


/**********************************************************
 *以下函数计算 lpszFileName 字符串的 hash 值，其中 dwHashType 为 hash 的类型，
 *在下面 GetHashTablePos 函数里面调用本函数，其可以取的值为 0、1、2；该函数
 *返回 lpszFileName 字符串的 hash 值；
 **********************************************************/
unsigned long HashString( char *lpszFileName, unsigned long dwHashType )
{
    unsigned char *key  = (unsigned char *)lpszFileName;
unsigned long seed1 = 0x7FED7FED;
unsigned long seed2 = 0xEEEEEEEE;
    int ch;

    while( *key != 0 )
    {
        ch = toupper(*key++);

        seed1 = cryptTable[(dwHashType << 8) + ch] ^ (seed1 + seed2);
        seed2 = ch + seed1 + seed2 + (seed2 << 5) + 3;
    }
    return seed1;
```

```
}


 /************************************************************
  *在 main 中测试 argv[1]的三个 hash 值：
  *   ./hash   "arr\units.dat"
  *   ./hash   "unit\neutral\acritter.grp"
  ************************************************************/
int main( int argc, char **argv )
{
      unsigned long ulHashValue;
      int i = 0;

      if ( argc != 2 )
      {
            printf("please input two arguments\n");
            return -1;
      }

       /*初始化数组：crytTable[0x500]*/
       prepareCryptTable();

       /*打印数组 crytTable[0x500]里面的值*/
       for ( ; i < 0x500; i++ )
       {
            if ( i % 10 == 0 )
            {
                  printf("\n");
            }

            printf("%-12X", cryptTable[i] );
       }

       ulHashValue = HashString( argv[1], 0 );
       printf("\n----%X ----\n", ulHashValue );

       ulHashValue = HashString( argv[1], 1 );
       printf("----%X ----\n", ulHashValue );
```

```
    ulHashValue = HashString( argv[1], 2 );
    printf("----%X ----\n", ulHashValue );

    return 0;
}
```

扫雷程序实现思路讲解

在我大二的时候就编写了一个扫雷程序，现在也有很多源程序下载，我不知道他们的算法是怎么样的，但我想我的算法应是最清晰和简单的。下面就来讲解我的扫雷程序思想。

首先我们在雷区上随机地放上雷，没有雷的地方被点击后就会显示一个数字表示它周围有几个雷，这是怎么实现的呢？我们可以把整个雷区看成一个二维数组 a[i,j]，如雷区：

11 12 13 14 15 16 17 18

21 22 23 24 25 26 27 28

31 32 33 34 35 36 37 38

41 42 43 44 45 46 47 48

51 52 53 54 55 56 57 58

我要知道 a[34]周围有几个雷，就只有去检测

a[23]，a[24]，a[25]

a[33]， a[35]

a[43]，a[44]，a[45]

这 8 个雷区是否放上了雷，仔细观察它们成在数学关系。抽象出来就是：a[i,j]的雷的个数就是由

a[i-1,j-1]，a[i-1,j]，a[i-1,j+1]

a[i ,j-1], a[i ,j+1]

a[i+1,j-1]，a[i+1,j]，a[i+1,j+1]

（如果超出边界再加以判断）这样的 8 个雷区决定的。

扫雷程序还会自动展开已确定没有雷的雷区。如果 a[3,4]周围雷数为 1，a[2,3]已被标示为地雷，那么 a[24]，a[25],a[33]，a[35],a[43]，a[44]，a[45]将被展开，一直波及到不可确定的雷区。这也是实现的关键。我们可以把数组的元素设定为一个类对象，它们所属的类设定这样的一个事件：在被展开时，检查周围的雷数是否与周围标示出来的雷数相等，如果相等则展开周围未标示的雷区。这样新的雷区展开又触发这个事件，就这样递归下去，一直蔓延到不可展开的雷区。

相信在了解以上两个要点后，把雷区这个类编写完全（如添加是否有雷标记，是否展开标记，周围雷数等，双击，左右单击的鼠标事件等），实现扫雷程序应是十分简单的一件事。

44. 指针与数组学习笔记

发布: 2008-7-27 09:17 | 作者: 剑心通明 | 来源: 互联网 | 查看: 13 次

C 语言中指针与数组这两个概念之间的联系是密不可分的，以至于如果不能理解一个概念，就无法彻底理解另一个概念。

C 语言中的数组值得注意的地方有以下两点：

1、C 语言中只有一维数组，而且数组的大小必须在编译期就作为一个常数确定下来。然而，C 语言中数组的元素可以是任何类型的对象，当然也可以是另外一个数组。这样，要“仿真”出一个多维数组就不是一件难事。

2、对于一个数组，我们只能够做两件事：确定该数组的大小，以及获得指向该数组下标为 0 的元素的指针。其他有关数组的操作，哪怕它们看上去是以数组下标进行运算的，实际上都是通过指针进行的。换句话说，任何一个数组下标运算都等同于一个对应的指针运算，因此我们完全可以依据指针行为定义数组下标的行为。

理解 C 语言中数组运作机制的注意点：

一、理解如何声明一个数组

1、例如：int calendar[12][31];

声明了 calendar 是一个数组，该数组拥有 12 个数组类型的元素，其中每个元素都是一个拥有 31 个整型元素的数组。因此，sizeof(calendar)的值是（31×12）与 sizeof(int)的乘积。

如果 calendar 不是用于 sizeof 的操作数，而是用于其他的场合，那么 calendar 总是被转换成一个指向 calendar 数组起始元素的指针。

2、任何指针都是指向某种类型的变量，如：int* ip;表明 ip 是一个指向整型变量的指针。如果一个指针指向的是一个数组中的一个元素，那么我们只要给这个指针加 1，就能够得到指向该数组中下一个元素的指针。同样地，给这个指针减 1，得到的就是指向该数组中前一个元素的指针。

注意，给一个指针加上一个整数，与给该指针的二进制表示加上同样的整数，两者是截然不同。如果 ip 指向一个整数，那么 ip+1 指向的是计算机内存中的下一个整数，在大多数现代计算机中，它都不同于 ip 所指向地址的下一个内存位置。

3、如果两个指针指向的是同一个数组中的元素，把两个指针相减是有意义的。例如：int* q = p + i;那么我们可以通过 q - p 得到 i 的值。需要注意的是如果 q 和 p 指向的不是同一个数组，即使它们所指向的地址在内存中的位置正好间隔一个数组元素的整数被，所得到的结果仍然是无法保证其正确性。

4、如果我们在应该出现指针的地方，却采用了数组名来替换，那么数组名就被当作指向该数组下标为 0 的元素的指针。

例如：int* p;

int a[3];

p = a; 就会把数组 a 中下标为 0 的元素的地址赋值给 p。如果写成 p=&a;在 ANSI C 中是非法的，因为&a 是一个指向数组的指针，而 p 是一个指向整型变量的指针，它们的类型不匹配。

因此，我们可以知道，*a 就是数组 a 中下标为 0 的元素的引用。例如，*a=88;就是将数组 a 中

下标为 0 的元素的值设置为 88。同样的道理，*(a+1)就是数组下标为 1 的元素的引用。以次类推，*（a+i）就是数组 a 中下标为 i 的元素的引用。这种写法是如此常用，因此它被简记为 a[i]。

二、二维数组

二维数组实际上是一数组为元素的一维数组。

例如：int calendar[12][31];

int *p;

int i;

calendar 是一个有 12 个数组类型元素的数组，它的每个数组类型元素是一个有着 31 个整型元素的数组，所以，calendar[4]是 calendar 数组的第 5 个元素，是 calendar 数组中 12 个有着 31 个整型元素的数组之一。因此，calendar[4]的行为也就表现为一个有着 31 个整型元素的数组的行为。

例如：sizeof(calendar[4])的结果是 31 与 sizeof(int)的乘积。

例如：p = calendar[4];使指针 p 指向数组 calendar[4]中下标为 0 的元素。

因此：i=calendar[4][7]=*(calendar[4]+7)=*(*(calendar[4])+7);

显然：用带方括号的下标形式很明显地要比完全用指针来表达要简单得多。

注意：p = calendar;是非法的。

因为 calendar 是一个二维数组，即“数组的数组”，在此处的上下文中使用 calendar 名称会将其转换为一个指向数组的指针；而 p 是一个指向整型变量的指针。

但是：int (*ap)[31];声明了*ap 是一个拥有 31 个整型元素的数组，因此，ap 就是一个指向这样数组的指针。因而，我们可以这样写：

int calendar[12][31];

int (*monthp)[31];

monthp= calendar;

这样，monthp 将指向数组 calendar 的第一个元素，也就是数组 calendar 的 12 个有着 31 个元素的数组类型元素之一。

45. 数组不是指针

发布: 2008-7-21 18:47 | 作者: 剑心通明 | 来源: 互联网 | 查看: 5 次

数组不等于指针。它是内存中一些普通变量的序列。

当我们写下面的代码：

```
int array[3];
array[2]=666;
```

C/C++ 编译器并不把 array[0]看作是一个整型变量的地址，而是直接把他看作一个值，就像下面代码一样：

```
int var;
var=66;
```

显然我们知道 var 不是一个指针，所以 array[2] 也不是。

但是如果我们用指针来代替数组，则代码看起来是一样的，但编译器却编译成不同的汇编代码。例如：

```
int *ptr = new int[3];
ptr[2] = 66;
```

这和第一段代码很像，但对编译器来说却意味着不同的意思。在第一段代码中第二条语句中，编译器生成的代码将做以下事情：

1) 指针下移两个位置并将其所指的值设为 666

但是用指针的那段代码则是：

1、 取 ptr[0]的地址；

2、 将其加 2；

3、 将此地址指向的赋值为 66。

实际上 "array"、"&array" 、"&array[0]"的值相等，但是"&array"的类型却是不同的，它是一个指向数组的地址而不是数组的元素。

为了让大家更好的理解，下面是另外一个例子。我想写一段程序使其从用户那里得到一个整型，将其加 4，然后输出结果。一个用整型指针实现，一个用整型变量实现。

用整型变量的将是：

```
#include<iostream>
main(){
    int int_input;
    cin>>int_input;
```

```
    cout<<(int_input + 4)<<endl;
    return 0;
}
```

用整型指针的是：

```
#include<iostream>
main(){
    int *int_ptr = new int[1];
    cin>>*int_ptr;
    cout<< (*int_ptr + 4)<<endl;
    delete(int_ptr);
    return 0;
}
```

有谁认为这两段程序完全一样？

让我们看一下他们的汇编代码。第一段用整型变量的如下：

```
2212: main(){
00401000    push        ebp
00401001    mov          ebp,esp
00401003    sub          esp,44h
00401006    push        ebx
00401007    push        esi
00401008    push        edi
2213:       int int_input;
2214:       cin>>int_input;
00401009    lea          eax,[ebp-4]
0040100C    push         eax
0040100D    mov          ecx,offset cin (00414c58)
00401012    call         istream::operator>> (0040b7c0)
2215:       cout<<(int_input+4)<<endl;
00401017    push        offset endl (00401070)
0040101C    mov          ecx,dword ptr [ebp-4]
0040101F    add          ecx,4
00401022    push        ecx
00401023    mov          ecx,offset cout (00414c18)
00401028    call         ostream::operator<< (0040b3e0)
0040102D    mov          ecx,eax
```

```
0040102F    call        ostream::operator<< (00401040)
2216:       return 0;
00401034    xor         eax,eax
2217: }
```

用指针的如下：

```
2212: main(){
00401000    push        ebp
00401001    mov         ebp,esp
00401003    sub         esp,4Ch
00401006    push        ebx
00401007    push        esi
00401008    push        edi
2213:       int *int_ptr = new int[1];
00401009    push        4
0040100B    call        operator new (004011b0)
00401010    add         esp,4
00401013    mov         dword ptr [ebp-8],eax
00401016    mov         eax,dword ptr [ebp-8]
00401019    mov         dword ptr [ebp-4],eax
2214:       cin>>*int_ptr;
0040101C    mov         ecx,dword ptr [ebp-4]
0040101F    push        ecx
00401020    mov         ecx,offset cin (00414c38)
00401025    call        istream::operator>> (0040b8a0)
2215:       cout<< (*int_ptr + 4)<<endl;
0040102A    push        offset endl (004010a0)
0040102F    mov         edx,dword ptr [ebp-4]
00401032    mov         eax,dword ptr [edx]
00401034    add         eax,4
00401037    push        eax
00401038    mov         ecx,offset cout (00414bf8)
0040103D    call        ostream::operator<< (0040b4c0)
00401042    mov         ecx,eax
00401044    call        ostream::operator<< (00401070)
2216:       delete(int_ptr);
00401049    mov         ecx,dword ptr [ebp-4]
0040104C    mov         dword ptr [ebp-0Ch],ecx
0040104F    mov         edx,dword ptr [ebp-0Ch]
```

```
00401052      push          edx
00401053      call          operator delete (00401120)
00401058      add           esp,4
2217:         return 0;
0040105B      xor           eax,eax
2218: }
```

分别看第 19 和 32 行你就会明白，一个整型变量和指向整型变量等指针是不同的。整型变量是一个存放整型数的内存位置，但是整型指针存的是整型数的内存地址的地址。编译器知道这个内存地址存放的是整型。因为这篇文章是面向初学者的并且为了简便，所以我不想解释这些汇编代码。

就像我刚才所说的数组是一系列的变量。从上面的例子可以看出，整型指针不是整型变量，所以它更不可能是一系列变量。

46. 标准 C 中字符串分割的方法

发布: 2008-7-21 14:29 | 作者: 剑心通明 | 来源: 互联网 | 查看: 15 次

◆ 使用 strtok 函数分割。

原型：char *strtok(char *s, char *delim);
strtok 在 s 中查找包含在 delim 中的字符并用 NULL('\0')来替换,直到找遍整个字符串。
功能：分解字符串为一组字符串。s 为要分解的字符串，delim 为分隔符字符串。
说明：首次调用时，s 指向要分解的字符串，之后再次调用要把 s 设成 NULL。
strtok 在 s 中查找包含在 delim 中的字符并用 NULL('\0')来替换，直到找遍整个字符串。
返回值：从 s 开头开始的一个个被分割的串。当没有被分割的串时则返回 NULL。
所有 delim 中包含的字符都会被滤掉，并将被滤掉的地方设为一处分割的节点。

使用例：

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


int main(int argc,char **argv)
#include <stdio.h>
#include <string.h>


int main(int argc,char **argv)
{


    char * buf1="aaa, ,a, ,,,bbb-c,,,ee|abc";

    /* Establish string and get the first token: */
    char* token = strtok( buf1, ",-|");
    while( token != NULL )
    {
        /* While there are tokens in "string" */
        printf( "%s ", token );
        /* Get next token: */
```

```
        token = strtok( NULL, ",-|");
    }
    return 0;

}
```

OUT 值：

aaa

a

bbb

c

ee

abc

◆ 使用 strstr 函数分割。

原型：extern char *strstr(char *haystack, char *needle);

用法：#include <string.h>

功能：从字符串 haystack 中寻找 needle 第一次出现的位置（不比较结束符 NULL)。

说明：返回指向第一次出现 needle 位置的指针，如果没找到则返回 NULL。

使用例：

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

int main(int argc,char **argv)
{
char *haystack="aaa||a||bbb||c||ee||";
```

```
char *needle="||";
char* buf = strstr( haystack, needle);
while( buf != NULL )
{
        buf[0]='\0';
        printf( "%s\n ", haystack);
        haystack = buf + strlen(needle);
        /* Get next token: */
        buf = strstr( haystack, needle);
}
     return 0;
}
```

OUT 值：

aaa

a

bbb

c

ee

◆ strtok 比较适合多个字符作分隔符的场合，而 strstr 适合用字符串作分隔符的场合。

47. 汉诺塔源码

发布: 2008-7-12 21:19 | 作者: 剑心通明 | 来源: 互联网 | 查看: 0 次

这个程序的实现可以明确递归方法的使用，理解结构化编程的概念。

原理：

移动 n 个盘子要经历（2 的 n 次方－1）步。具体实现步骤是：1、将 A 上（n-1）个盘借助于 C 先移动到 B 座上；2、将 A 剩下的一个盘移动到 C 上；3、将（n-1）个盘从 B 借助于 A 移动到 C 上。

程序实现：

－－

```
/*hanoi.c*/

#include <stdio.h>

int count=0;   //定义全局变量 count，计算移动的步数

//////////////////////////////////////////////////////////
//函数名：move
//功能：打印出 x-->y，也就是具体的移动方法，并且计算总的移动步数
//入口参数：x-代表第一个座
//          y-代表第二个座
//////////////////////////////////////////////////////////
void move(char x,char y)
{
        printf("\t%c-->%c\n",x,y);
        count++;
}

//////////////////////////////////////////////////////////
//函数名：hanoi
//功能：将 n 个盘从 one 座借助于 two 座，移动到 three 座
//入口参数：n-代表总的盘数
//          one-代表第一个座
//          two-代表第二个座
//          three-代表第三个座
//////////////////////////////////////////////////////////
```

```
voi hanoi(int n,char one,char two,char three)
{
        if(n==1)                //如果只有一个盘，直接从 one 到 three
                move(one,three);
        else {                  //如果有多个 1 个盘
                hanoi(n-1,one,three,two);//第一步：将 n-1 个盘从 one 借助 three 移到 two
                move(one,three);//第二步：将第 n 个盘从 one 移到 three
                hanoi(n-1,two,one,three);//第三步：将 n－1 个盘从 two 借助 one 移到 three
        }
}

////////////////////////////////////////////////////////
//函数名：main
//功能：总的控制，打印出移动方案和移动次数
//入口参数：无
////////////////////////////////////////////////////////
int main()
{
        int m;
        printf("Input the number of disks:");
        scanf("%d",&m);//输入盘的总数
        printf("The step to moving %3d disks:\n\n",m);
        hanoi(m,'A','B','C');//打印出移动方案
        printf("\nThe total times of moving are %d.\n",count);//打印出移动次数
        return(0);
}
```

－－

程序结果：

－－

Makefile

```
CC=gcc
hanoi:hanoi.c
        $(CC) $< -o $@
.PHONY:clean
clean:
        rm -f hanoi
```

```
[armlinux@lqm hanoi]$ ls
hanoi.c    Makefile
[armlinux@lqm hanoi]$ make
gcc hanoi.c -o hanoi
[armlinux@lqm hanoi]$ ls
hanoi    hanoi.c    Makefile
[armlinux@lqm hanoi]$ ./hanoi
Input the number of disks:3
The step to moving      3 disks:
        A-->C
        A-->B
        C-->B
        A-->C
        B-->A
        B-->C
        A-->C
The total times of moving are 7.
```

48. 洗牌算法

发布: 2008-7-21 18:46 | 作者: 剑心通明 | 来源: 互联网 | 查看: 11 次

洗牌即产生指定数据的随机序列。

在网上找了半天大体有两种做法

1、

思路：将 54 个数依次放到随机的位置。关键是每次找一个随机的位置。

下面是找这个随机位置的算法：

1、用一个 Bool 型数组记录各个位置是否已经放置了数，如果放置则置 true，没有则为 false。在算法开始时数组初始化为 false。

2、每次产生一个 0~53 的随机数，看这个位置是否已经放置了数，如果已经放置了，则继续用同样的方法找一个随机位置判断；如果这个位置还未放置，则设置此位置，并标记其已经放置。

3、反复执行（2）直到所有的位置都放置了数为止。（只要设置成功 54 次数就说明所有位置已经设置了数）

例程：

```
void shuffle(int dest[],int n)                    //洗牌算法
{
    int pos,card;
    memset(dest,0,sizeof(int)*n);
    for(card=1;card<=n;card++)
    {
            do
            {
                    pos=rand()%(n+1);

            }while(dest[pos]!=0);
            dest[pos]=card;
    }
}
```

上面方法的问题：随着未设置的数渐渐变少，寻找未设置的位置会越来越难。如果牌数很多则更是不可思议。

2、下面的思路是先对数组进行初始化然后随机交换两个位置，共交换 n 次，其中 n 越大，则随机越接近随机

```
void shuffle ( int a[], int n )          //洗牌算法
{
    int tmp = 0,
```

```
        p1,p2;
    int cnt = rand() % 1023;
    while (cnt--)     //随机交换两个位置的数，共交换 cnt 次
    {
        p1 = rand() % n;
        p2 = rand() % n;

        tmp = a[p1];
        a[p1] = a[p2];
        a[p2] = tmp;
    }
}
```

49. 深入理解 C 语言指针的奥秘

发布: 2008-7-27 09:11 | 作者: 剑心通明 | 来源: 互联网 | 查看: 10 次

指针的概念

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。 要搞清一个指针需要搞清指针的四方面的内容：指针的类型，指针所指向的 类型，指针的值或者叫指针所指向的内存区，还有指针本身所占据的内存区。让我们分别说明。

先声明几个指针放着做例子：

例一：

(1)int*ptr;

(2)char*ptr;

(3)int**ptr;

(4)int(*ptr)[3];

(5)int*(*ptr)[4];

如果看不懂后几个例子的话，请参阅我前段时间贴出的文章<<如何理解 c 和 c ++的复杂类型声明>>。

指针的类型

从语法的角度看，你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。让我们看看例一中各个指针的类型：

(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]

怎么样？找出指针的类型的方法是不是很简单？

指针所指向的类型

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。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]

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对 C 越来越熟悉时，你会发现，把与指针搅和在一起的"类型"这个概念分成"指针的类型"和"指针所指向的类型"两个概念，是精通指针的关键点之一。我看了不少书，发现有些写得差的书中，就把指针的这两个概念搅在一起了，所以看起书来前后矛盾，越看越糊涂。

指针的值，或者叫指针所指向的内存区或地址

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。在 32 位程序里，所有类型的指针的值都是一个 32 位整数，因为 32 位程序里内存地址全都是 32 位长。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 si zeof(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。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

以后，每遇到一个指针，都应该问问：这个指针的类型是什么？指针指的类型是什么？该指针指向了哪里？

指针本身所占据的内存区

指针本身占了多大的内存？你只要用函数 sizeof(指针的类型)测一下就知道了。在 32 位平台里，指针本身占据了 4 个字节的长度。

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。

指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的。例如：

例二：

1、chara[20];

2、int*ptr=a;

...

...

3、ptr++;

在上例中，指针 ptr 的类型是 int*,它指向的类型是 int，它被初始化为指向整形变量 a。接下来的第 3 句中，指针 ptr 被加了 1，编译器是这样处理的：它把指针 ptr 的值加上了 sizeof(int)，在 32 位程序中，是被加上了 4。由于地址是用字节做单位的，故 ptr 所指向的地址由原来的变量 a 的地址向高地址方向增加了 4 个字节。

由于 char 类型的长度是一个字节，所以，原来 ptr 是指向数组 a 的第 0 号单元开始的四个字节，此时指向了数组 a 中从第 4 号单元开始的四个字节。

我们可以用一个指针和一个循环来遍历一个数组，看例子：

例三：

```
intarray[20];
int*ptr=array;
...
//此处略去为整型数组赋值的代码。
...
for(i=0;i<20;i++)
{
   (*ptr)++;
   ptr++;
}
```

这个例子将整型数组中各个单元的值加 1。由于每次循环都将指针 ptr 加 1，所以每次循环都能访问数组的下一个单元。

再看例子：

例四：

1、chara[20];

2、int*ptr=a;

...

...

3、ptr+=5;

在这个例子中，ptr 被加上了 5，编译器是这样处理的：将指针 ptr 的值加上 5 乘 sizeof(int)，在 32 位程序中就是加上了 5 乘 4=20。由于地址的单位是字节，故现在的 ptr 所指向的地址比起加 5 后的 ptr 所指向的地址来说，向高地址方向移动了 20 个字节。在这个例子中，没加 5 前的 ptr 指向数组 a 的第 0 号单元开始的四个字节，加 5 后，ptr 已经指向了数组 a 的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。

如果上例中，ptr 是被减去 5，那么处理过程大同小异，只不过 ptr 的值是被减去 5 乘 sizeof(int)，新的 ptr 指向的地址将比原来的 ptr 所指向的地址向低地址方向移动了 20 个字节。

总结一下，一个指针 ptrold 加上一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值增加了 n 乘 sizeof(ptrold 所指向的类型)个字节。就是说，ptrnew 所指向的内存区将比 ptrold 所指向的内存区向高地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

一个指针 ptrold 减去一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值减少了 n 乘 sizeof(ptrold 所指向的类型)个字节，就是说，ptrnew 所指向的内存区将比 ptrold 所指向的内存区向低地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

运算符&和*

这里&是取地址运算符，*是...书上叫做"间接运算符"。

&a 的运算结果是一个指针，指针的类型是 a 的类型加个*，指针所指向的类型是 a 的类型，指针所指向的地址嘛，那就是 a 的地址。

*p 的运算结果就五花八门了。总之*p 的结果是 p 所指向的东西，这个东西有这些特点：它的类型是 p 指向的类型，它所占用的地址是 p 所指向的地址。

例五：

```
inta=12;
intb;
int*p;
int**ptr;
p=&a;
//&a 的结果是一个指针，类型是 int*，指向的类型是 int，指向的地址是 a 的地址。
*p=24;
//*p 的结果，在这里它的类型是 int，它所占用的地址是 p 所指向的地址，显然，*p 就是变量 a。
ptr=&p;
//&p 的结果是个指针，该指针的类型是 p 的类型加个*，在这里是 int **。该指针所指向的类型是 p 的类型，这里是 int*。该指针所指向的地址就是指针 p 自己的地址。
*ptr=&b;
//*ptr 是个指针，&b 的结果也是个指针，且这两个指针的类型和所指向的类型是一样的，所以用
```

&b 来给*ptr 赋值就是毫无问题的了。

**ptr=34;

//*ptr 的结果是 ptr 所指向的东西，在这里是一个指针，对这个指针再做一次*运算，结果就是一个 int 类型的变量。

指针表达式

一个表达式的最后结果如果是一个指针，那么这个表达式就叫指针表式。

下面是一些指针表达式的例子：

例六：

```
inta,b;
intarray[10];
int*pa;
pa=&a;//&a 是一个指针表达式。
int**ptr=&pa;//&pa 也是一个指针表达式。
*ptr=&b;//*ptr 和&b 都是指针表达式。
pa=array;
pa++;//这也是指针表达式。
```

例七：

```
char*arr[20];
char**parr=arr;//如果把 arr 看作指针的话，arr 也是指针表达式
char*str;
str=*parr;//*parr 是指针表达式
str=*(parr+1);//*(parr+1)是指针表达式
str=*(parr+2);//*(parr+2)是指针表达式
```

由于指针表达式的结果是一个指针，所以指针表达式也具有指针所具有的四个要素：指针的类型，指针所指向的类型，指针指向的内存区，指针自身占据的内存。

好了，当一个指针表达式的结果指针已经明确地具有了指针自身占据的内存的话，这个指针表达式就是一个左值，否则就不是一个左值。

在例七中，&a 不是一个左值，因为它还没有占据明确的内存。*ptr 是一个左值，因为*ptr 这个指针已经占据了内存，其实*ptr 就是指针 pa，既然 pa 已经在内存中有了自己的位置，那么*ptr 当然也有了自己的位置。

数组和指针的关系

如果对声明数组的语句不太明白的话，请参阅我前段时间贴出的文章<<如何理解 c 和 c++的复杂类型声明>>。

数组的数组名其实可以看作一个指针。看下例：

例八：

```
intarray[10]={0,1,2,3,4,5,6,7,8,9},value;
...
...
value=array[0];//也可写成：value=*array;
value=array[3];//也可写成：value=*(array+3);
value=array[4];//也可写成：value=*(array+4);
```

上例中，一般而言数组名 array 代表数组本身，类型是 int[10]，但如果把 array 看做指针的话，它指向数组的第 0 个单元，类型是 int*，所指向的类型是数组单元的类型即 int。因此*array 等于 0 就一点也不奇怪了。同理，array+3 是一个指向数组第 3 个单元的指针，所以*(array+3)等于 3。其它依此类推。

例九：

```
char*str[3]={
   "Hello,thisisasample!",
   "Hi,goodmorning.",
   "Helloworld"
};
```

chars[80]；

strcpy(s,str[0]);//也可写成 strcpy(s,*str);

strcpy(s,str[1]);//也可写成 strcpy(s,*(str+1));

strcpy(s,str[2]);//也可写成 strcpy(s,*(str+2));

上例中，str 是一个三单元的数组，该数组的每个单元都是一个指针，这些指针各指向一个字符串。把指针数组名 str 当作一个指针的话，它指向数组的第 0 号单元，它的类型是 char**，它指向的类型是 char*。

str 也是一个指针，它的类型是 char，它所指向的类型是 char，它指向的地址是字符串 "Hello,thisisasample!"的第一个字符的地址，即'H'的地址。 str+1 也是一个指针，它指向数组的第 1 号单元，它的类型是 char**，它指向的类型是 char*。

(str+1)也是一个指针，它的类型是 char，它所指向的类型是 char，它指向 "Hi,goodmorning." 的第一个字符'H'，等等。

下面总结一下数组的数组名的问题。声明了一个数组 TYPEarray[n]，则数组名称 array 就有了两重含义：第一，它代表整个数组，它的类型是 TYPE[n]；第二 ，它是一个指针，该指针的类型是 TYPE*，该指针指向的类型是 TYPE，也就是数组单元的类型，该指针指向的内存区就是数组第 0 号单元，该指针自己占有单独的内存区，注意它和数组第 0 号单元占据的内存区是不同的。该指针的值是不能修改的，即类似 array++的表达式是错误的。

在不同的表达式中数组名 array 可以扮演不同的角色。

在表达式 sizeof(array)中，数组名 array 代表数组本身，故这时 sizeof 函数测出的是整个数组的大小。

在表达式*array 中，array 扮演的是指针，因此这个表达式的结果就是数组第 0 号单元的值。sizeof(*array)测出的是数组单元的大小。

表达式 array+n（其中 n=0，1，2，....。）中，array 扮演的是指针，故 array+n 的结果是一个指针，它的类型是 TYPE*，它指向的类型是 TYPE，它指向数组第 n 号单元。故 sizeof(array+n)测出的是指针类型的大小。

例十：

intarray[10];

int(*ptr)[10];

ptr=&array;

上例中 ptr 是一个指针，它的类型是 int(*)[10]，他指向的类型是 int[10] ，我们用整个数组的首地址来初始化它。在语句 ptr=&array 中，array 代表数组本身。

本节中提到了函数 sizeof()，那么我来问一问，sizeof(指针名称)测出的究竟是指针自身类型的大小呢还是指针所指向的类型的大小？答案是前者。例如：

int(*ptr)[10];

则在 32 位程序中，有：

sizeof(int(*)[10])==4
sizeof(int[10])==40
sizeof(ptr)==4

实际上，sizeof(对象)测出的都是对象自身的类型的大小，而不是别的什么类型的大小。

指针和结构类型的关系

可以声明一个指向结构类型对象的指针。

例十一：

```
structMyStruct
{
   inta;
   intb;
   intc;
}
MyStructss={20,30,40};
//声明了结构对象 ss，并把 ss 的三个成员初始化为 20，30 和 40。
MyStruct*ptr=&ss;
//声明了一个指向结构对象 ss 的指针。它的类型是 MyStruct*,它指向的类型是 MyStruct。
int*pstr=(int*)&ss;
//声明了一个指向结构对象 ss 的指针。但是它的类型和它指向的类型和 ptr 是不同的。
```

请问怎样通过指针 ptr 来访问 ss 的三个成员变量？

答案：

ptr->a;
ptr->b;
ptr->c;

又请问怎样通过指针 pstr 来访问 ss 的三个成员变量？

答案：

*pstr；//访问了 ss 的成员 a。
*(pstr+1);//访问了 ss 的成员 b。
*(pstr+2)//访问了 ss 的成员 c。

虽然我在我的 MSVC++6.0 上调式过上述代码，但是要知道，这样使用 pstr 来访问结构成员是不正规的，为了说明为什么不正规，让我们看看怎样通过指针来访问数组的各个单元：

例十二：

intarray[3]={35,56,37};
int*pa=array;

通过指针 pa 访问数组 array 的三个单元的方法是：

*pa;//访问了第 0 号单元
*(pa+1);//访问了第 1 号单元
*(pa+2);//访问了第 2 号单元

从格式上看倒是与通过指针访问结构成员的不正规方法的格式一样。

所有的 C/C++编译器在排列数组的单元时，总是把各个数组单元存放在连续的存储区里，单元和单元之间没有空隙。但在存放结构对象的各个成员时，在某种编译环境下，可能会需要字对齐或双字对齐或者是别的什么对齐，需要在相邻两个成员之间加若干个"填充字节"，这就导致各个成员之间可能会有若干个字节的空隙。

所以，在例十二中，即使*pstr 访问到了结构对象 ss 的第一个成员变量 a，也不能保证*(pstr+1)就一定能访问到结构成员 b。因为成员 a 和成员 b 之间可能会有若干填充字节，说不定*(pstr+1)就正好访问到了这些填充字节呢。这也证明了指针的灵活性。要是你的目的就是想看看各个结构成员之间到底有没有填充字节，嘿，这倒是个不错的方法。

通过指针访问结构成员的正确方法应该是象例十二中使用指针 ptr 的方法。

指针和函数的关系

可以把一个指针声明成为一个指向函数的指针。

```
intfun1(char*,int);
int(*pfun1)(char*,int);
pfun1=fun1;
....
....
inta=(*pfun1)("abcdefg",7);//通过函数指针调用函数。
```

可以把指针作为函数的形参。在函数调用语句中，可以用指针表达式来作为实参。

例十三：

```
intfun(char*);
inta;
charstr[]="abcdefghijklmn";
a=fun(str);
...
...
intfun(char*s)
{
intnum=0;
for(inti=0;i{
num+=*s;s++;
```

```
}
returnnum;
}
```

这个例子中的函数 fun 统计一个字符串中各个字符的 ASCII 码值之和。前面说了，数组的名字也是一个指针。在函数调用中，当把 str 作为实参传递给形参 s 后，实际是把 str 的值传递给了 s，s 所指向的地址就和 str 所指向的地址一致，但是 str 和 s 各自占用各自的存储空间。在函数体内对 s 进行自加 1 运算，并不意味着同时对 str 进行了自加 1 运算。

指针类型转换

当我们初始化一个指针或给一个指针赋值时，赋值号的左边是一个指针，赋值号的右边是一个指针表达式。在我们前面所举的例子中，绝大多数情况下，指针的类型和指针表达式的类型是一样的，指针所指向的类型和指针表达式所指向的类型是一样的。

例十四：

1、floatf=12.3;

2、float*fptr=&f;

3、int*p;

在上面的例子中，假如我们想让指针 p 指向实数 f，应该怎么搞？是用下面的语句吗？

p=&f;

不对。因为指针 p 的类型是 int*，它指向的类型是 int。表达式&f 的结果是一个指针，指针的类型是 float*,它指向的类型是 float。两者不一致，直接赋值的方法是不行的。至少在我的 MSVC++6.0 上，对指针的赋值语句要求赋值号两边的类型一致，所指向的类型也一致，其它的编译器上我没试过，大家可以试试。为了实现我们的目的，需要进行"强制类型转换"：

p=(int*)&f;

如果有一个指针 p，我们需要把它的类型和所指向的类型改为 TYEP*TYPE，那么语法格式是：

(TYPE*)p；

这样强制类型转换的结果是一个新指针，该新指针的类型是 TYPE*，它指向的类型是 TYPE，它指向的地址就是原指针指向的地址。而原来的指针 p 的一切属性都没有被修改。

一个函数如果使用了指针作为形参，那么在函数调用语句的实参和形参的结合过程中，也会发生指针类型的转换。

例十五：

```
voidfun(char*);
inta=125,b;
fun((char*)&a);
...
...
voidfun(char*s)
{
charc;
c=*(s+3);*(s+3)=*(s+0);*(s+0)=c;
c=*(s+2);*(s+2)=*(s+1);*(s+1)=c;
}
}
```

注意这是一个 32 位程序，故 int 类型占了四个字节，char 类型占一个字节。函数 fun 的作用是把一个整数的四个字节的顺序来个颠倒。注意到了吗？在函数调用语句中，实参&a 的结果是一个指针，它的类型是 int*，它指向的类型是 int。形参这个指针的类型是 char*，它指向的类型是 char。这样，在实参和形参的结合过程中，我们必须进行一次从 int*类型到 char*类型的转换。结合这个例子，我们可以这样来想象编译器进行转换的过程：编译器先构造一个临时指针 char*temp， 然后执行 temp=(char*)&a，最后再把 temp 的值传递给 s。所以最后的结果是：s 的类型是 char*,它指向的类型是 char，它指向的地址就是 a 的首地址。

我们已经知道，指针的值就是指针指向的地址，在 32 位程序中，指针的值其实是一个 32 位整数。那可不可以把一个整数当作指针的值直接赋给指针呢？就象下面的语句：

```
unsignedinta;
TYPE*ptr;//TYPE 是 int，char 或结构类型等等类型。
...
```

...

a=20345686;

ptr=20345686;//我们的目的是要使指针 ptr 指向地址 20345686（十进制

）

ptr=a;//我们的目的是要使指针 ptr 指向地址 20345686（十进制）

编译一下吧。结果发现后面两条语句全是错的。那么我们的目的就不能达到了吗？不，还有办法：

unsignedinta;

TYPE*ptr;//TYPE 是 int，char 或结构类型等等类型。

...

...

a=某个数，这个数必须代表一个合法的地址；

ptr=(TYPE*)a；//呵呵，这就可以了。

严格说来这里的(TYPE*)和指针类型转换中的(TYPE*)还不一样。这里的(TYPE*)的意思是把无符号整数 a 的值当作一个地址来看待。上面强调了 a 的值必须代表一个合法的地址，否则的话，在你使用 ptr 的时候，就会出现非法操作错误。

想想能不能反过来，把指针指向的地址即指针的值当作一个整数取出来。完 全可以。下面的例子演示了把一个指针的值当作一个整数取出来，然后再把这个整数当作一个地址赋给一个指针：

例十六：

inta=123,b;

int*ptr=&a;

char*str;

b=(int)ptr;//把指针 ptr 的值当作一个整数取出来。

str=(char*)b;//把这个整数的值当作一个地址赋给指针 str。

现在我们已经知道了，可以把指针的值当作一个整数取出来，也可以把一个整数值当作地址赋给一个指针。

指针的安全问题

看下面的例子：

例十七：

```
chars='a';
int*ptr;
ptr=(int*)&s;
*ptr=1298；
```

指针 ptr 是一个 int*类型的指针，它指向的类型是 int。它指向的地址就是 s 的首地址。在 32 位程序中，s 占一个字节，int 类型占四个字节。最后一条语句不但改变了 s 所占的一个字节，还把和 s 相临的高地址方向的三个字节也改变了。这三个字节是干什么的？只有编译程序知道，而写程序的人是不太可能知道的。也许这三个字节里存储了非常重要的数据，也许这三个字节里正好是程序的一条代码，而由于你对指针的马虎应用，这三个字节的值被改变了！这会造成崩溃性的错误。

让我们再来看一例：

例十八：

1、chara;

2、int*ptr=&a;

...
...

3、ptr++;

4、*ptr=115;

该例子完全可以通过编译，并能执行。但是看到没有？第 3 句对指针 ptr 进行自加 1 运算后，ptr 指向了和整形变量 a 相邻的高地址方向的一块存储区。这块存储区里是什么？我们不知道。有可能它是一个非常重要的数据，甚至可能是一条代码。而第 4 句竟然往这片存储区里写入一个数据！这是严重的错误。所以在使用指针时，程序员心里必须非常清楚：我的指针究竟指向了哪里。在用指针访问数组的时候，也要注意不要超出数组的低端和高端界限，否则也会造成类似的错误。

在指针的强制类型转换：ptr1=(TYPE*)ptr2 中，如果 sizeof(ptr2 的类型)大于 sizeof(ptr1 的类型)，

那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是安全的。如果 sizeof(ptr2 的类型)小于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是不安全的。至于为什么，读者结合例十七来想一想，应该会明白的。

50. 游戏外挂的编写原理

发布: 2008-7-27 10:27 | 作者: 剑心通明 | 来源: 互联网 | 查看: 20 次

一、 前言

所谓游戏外挂，其实是一种游戏外辅程序，它可以协助玩家自动产生游戏动作、修改游戏网络数据包以及修改游戏内存数据等，以实现玩家用最少的时间和金钱去完成功力升级和过关斩将。虽然，现在对游戏外挂程序的“合法”身份众说纷纭，在这里我不想对此发表任何个人意见，让时间去说明一切吧。

不管游戏外挂程序是不是“合法”身份，但是它却是具有一定的技术含量的，在这些小小程序中使用了许多高端技术，如拦截 Sock 技术、拦截 API 技术、模拟键盘与鼠标技术、直接修改程序内存技术等等。本文将对常见的游戏外挂中使用的技术进行全面剖析。

二、认识外挂

游戏外挂的历史可以追溯到单机版游戏时代，只不过当时它使用了另一个更通俗易懂的名字??游戏修改器。它可以在游戏中追踪锁定游戏主人公的各项能力数值。这样玩家在游戏中可以达到主角不掉血、不耗费魔法、不消耗金钱等目的。这样降低了游戏的难度，使得玩家更容易通关。随着网络游戏的时代的来临，游戏外挂在原有的功能之上进行了新的发展，它变得更加多种多样，功能更加强大，操作更加简单，以至有些游戏的外挂已经成为一个体系，比如《石器时代》，外挂品种达到了几十种，自动战斗、自动行走、自动练级、自动补血、加速、不遇敌、原地遇敌、快速增加经验值、按键精灵……几乎无所不包。游戏外挂的设计主要是针对于某个游戏开发的，我们可以根据它针对的游戏的类型可大致可将外挂分为两种大类。

一类是将游戏中大量繁琐和无聊的攻击动作使用外挂自动完成，以帮助玩家轻松搞定攻击对象并可以快速的增加玩家的经验值。比如在《龙族》中有一种工作的设定，玩家的工作等级越高，就可以驾驭越好的装备。但是增加工作等级却不是一件有趣的事情，毋宁说是重复枯燥的机械劳动。如果你想做法师用的杖，首先需要做基本工作--砍树。砍树的方法很简单，在一棵大树前不停的点鼠标就可以了，每 10000 的经验升一级。这就意味着玩家要在大树前不停的点击鼠标，这种无聊的事情通过"按键精灵"就可以解决。外挂的"按键精灵"功能可以让玩家摆脱无趣的点击鼠标的工作。

另一类是由外挂程序产生欺骗性的网络游戏封包，并将这些封包发送到网络游戏服务器，利用这些虚假信息欺骗服务器进行游戏数值的修改，达到修改角色能力数值的目的。这类外挂程序针对性很强，一般在设计时都是针对某个游戏某个版本来做的，因为每个网络游戏服务器与客户端交流的数据包各不相同，外挂程序必须要对欺骗的网络游戏服务器的数据包进行分析，才能产生服务器识别的数据包。这类外挂程序也是当前最流利的一类游戏外挂程序。

另外，现在很多外挂程序功能强大，不仅实现了自动动作代理和封包功能，而且还提供了对网络游戏的客户端程序的数据进行修改，以达到欺骗网络游戏服务器的目的。我相信，随着网络游戏商家的反外挂技术的进展，游戏外挂将会产生更多更优秀的技术，让我们期待着看场技术大战吧.....

三、外挂技术综述

可以将开发游戏外挂程序的过程大体上划分为两个部分：

前期部分工作是对外挂的主体游戏进行分析，不同类型的外挂分析主体游戏的内容也不相同。如外挂为上述谈到的外挂类型中的第一类时，其分析过程常是针对游戏的场景中的攻击对象的位置和分布情况进行分析，以实现外挂自动进行攻击以及位置移动。如外挂为外挂类型中的第二类时，其分析过程常是针对游戏服务器与客户端之间通讯包数据的结构、内容以及加密算法的分析。因网络游戏公司一般都不会公布其游戏产品的通讯包数据的结构、内容和加密算法的信息，所以对于开发第二类外挂成功的关键在于是否能正确分析游戏包数据的结构、内容以及加密算法，虽然可以使用一些工具辅助分析，但是这还是一种坚苦而复杂的工作。

后期部分工作主要是根据前期对游戏的分析结果，使用大量的程序开发技术编写外挂程序以实

现对游戏的控制或修改。如外挂程序为第一类外挂时，通常会使用到鼠标模拟技术来实现游戏角色的自动位置移动，使用键盘模拟技术来实现游戏角色的自动攻击。如外挂程序为第二类外挂时，通常会使用到挡截 Sock 和挡截 API 函数技术，以挡截游戏服务器传来的网络数据包并将数据包修改后封包后传给游戏服务器。另外，还有许多外挂使用对游戏客户端程序内存数据修改技术以及游戏加速技术。

本文主要是针对开发游戏外挂程序后期使用的程序开发技术进行探讨，重点介绍的如下几种在游戏外挂中常使用的程序开发技术：

- 动作模拟技术：主要包括键盘模拟技术和鼠标模拟技术。
- 封包技术：主要包括挡截 Sock 技术和挡截 API 技术。

四、动作模拟技术

我们在前面介绍过，几乎所有的游戏都有大量繁琐和无聊的攻击动作以增加玩家的功力，还有那些数不完的迷宫，这些好像已经成为了角色游戏的代名词。现在，外挂可以帮助玩家从这些繁琐而无聊的工作中摆脱出来，专注于游戏情节的进展。外挂程序为了实现自动角色位置移动和自动攻击等功能，需要使用到键盘模拟技术和鼠标模拟技术。下面我们将重点介绍这些技术并编写一个简单的实例帮助读者理解动作模拟技术的实现过程。

1． 鼠标模拟技术

几乎所有的游戏中都使用了鼠标来改变角色的位置和方向，玩家仅用一个小小的鼠标，就可以使角色畅游天下。那么，我们如何实现在没有玩家的参与下角色也可以自动行走呢。其实实现这个并不难，仅仅几个 Windows API 函数就可以搞定，让我们先来认识认识这些 API 函数。

(1) 模拟鼠标动作 API 函数 mouse_event，它可以实现模拟鼠标按下和放开等动作。

VOID mouse_event(DWORD dwFlags, // 鼠标动作标识。

DWORD dx, // 鼠标水平方向位置。 DWORD dy, // 鼠标垂直方向位置。

DWORD dwData, // 鼠标轮子转动的数量。 DWORD dwExtraInfo // 一个关联鼠标动作辅加信息。);

其中，dwFlags 表示了各种各样的鼠标动作和点击活动，它的常用取值如下：

MOUSEEVENTF_MOVE 表示模拟鼠标移动事件。

MOUSEEVENTF_LEFTDOWN 表示模拟按下鼠标左键。

MOUSEEVENTF_LEFTUP 表示模拟放开鼠标左键。

MOUSEEVENTF_RIGHTDOWN 表示模拟按下鼠标右键。

MOUSEEVENTF_RIGHTUP 表示模拟放开鼠标右键。

MOUSEEVENTF_MIDDLEDOWN 表示模拟按下鼠标中键。

MOUSEEVENTF_MIDDLEUP 表示模拟放开鼠标中键。

(2)、设置和获取当前鼠标位置的 API 函数。获取当前鼠标位置使用 GetCursorPos()函数，设置当前鼠标位置使用 SetCursorPos()函数。

BOOL GetCursorPos(LPPOINT lpPoint // 返回鼠标的当前位置。);

BOOL SetCursorPos(int X, // 鼠标的水平方向位置。 int Y //鼠标的垂直方向位置。);

通常游戏角色的行走都是通过鼠标移动至目的地，然后按一下鼠标的按钮就搞定了。下面我们使用上面介绍的 API 函数来模拟角色行走过程。

```
CPoint oldPoint,newPoint;  GetCursorPos(&oldPoint); //保存当前鼠标位置。
newPoint.x = oldPoint.x+40;  newPoint.y = oldPoint.y+10;
```

SetCursorPos(newPoint.x,newPoint.y); //设置目的地位置。 mouse_event (MOUSEEVENTF_RIGHTDOWN,0,0,0,0);//模拟按下鼠标右键。 mouse_event (MOUSEEVENTF_RIGHTUP,0,0,0,0);//模拟放开鼠标右键。

2. 键盘模拟技术

在很多游戏中，不仅提供了鼠标的操作，而且还提供了键盘的操作，在对攻击对象进行攻击时还可以使用快捷键。为了使这些攻击过程能够自动进行，外挂程序需要使用键盘模拟技术。像鼠标模拟技术一样，Windows API 也提供了一系列 API 函数来完成对键盘动作的模拟。

模拟键盘动作 API 函数 keydb_event，它可以模拟对键盘上的某个或某些键进行按下或放开的动作。 VOID keybd_event(BYTE bVk, // 虚拟键值。 BYTE bScan, // 硬件扫描码。 DWORD dwFlags, // 动作标识。 DWORD dwExtraInfo // 与键盘动作关联的辅加信息。);

其中，bVk 表示虚拟键值，其实它是一个 BYTE 类型值的宏，其取值范围为 1-254。有关虚拟键值表请在 MSDN 上使用关键字“Virtual-Key Codes”查找相关资料。bScan 表示当键盘上某键被按下和放开时，键盘系统硬件产生的扫描码，我们可以 MapVirtualKey()函数在虚拟键值与扫描码之间进行转换。dwFlags 表示各种各样的键盘动作，它有两种取值：KEYEVENTF_EXTENDEDKEY 和 KEYEVENTF_KEYUP。

下面我们使用一段代码实现在游戏中按下 Shift+R 快捷键对攻击对象进行攻击。

keybd_event(VK_CONTROL,MapVirtualKey(VK_CONTROL,0),0,0); //按下 CTRL 键。

keybd_event(0x52,MapVirtualKey(0x52,0),0,0);//键下 R 键。 keybd_event (0x52,MapVirtualKey(0x52,0), KEYEVENTF_KEYUP,0);//放开 R 键。 keybd_event (VK_CONTROL,MapVirtualKey(VK_CONTROL,0), KEYEVENTF_KEYUP,0);//放开 CTRL 键。

3. 激活外挂

上面介绍的鼠标和键盘模拟技术实现了对游戏角色的动作部分的模拟，但要想外挂能工作于游戏之上，还需要将其与游戏的场景窗口联系起来或者使用一个激活键，就象按键精灵的那个激活键一样。我们可以用 GetWindow 函数来枚举窗口，也可以用 Findwindow 函数来查找特定的窗口。另外还有一个 FindWindowEx 函数可以找到窗口的子窗口，当游戏切换场景的时候我们可以用 FindWindowEx 来确定一些当前窗口的特征，从而判断是否还在这个场景，方法很多了，比如可以 GetWindowInfo 来确定一些东西，比如当查找不到某个按钮的时候就说明游戏场景已经切换了等等办法。当使用激活键进行关联，需要使用 Hook 技术开发一个全局键盘钩子，在这里就不具体介绍全局钩子的开发过程了，在后面的实例中我们将会使用到全局钩子，到时将学习到全局钩子的相关知识。

51. 程序实例分析-为什么会陷入死循环

发布: 2008-7-21 15:00 | 作者: 剑心通明 | 来源: 互联网 | 查看: 1 次

看似简单的一段程序如下:

```
 int main()
{
    int i,j[8];
    for(i=0;i<=8;i++)
      j[i]=0;
    return 0;
}
```

gcc 编译运行会陷入死循环.

因为变量 i 和数组 j[8]是保存在栈中，默认是由高地址向低地址方向存储. 输出变量地址可以发现: i 存储位置在 0xbfd90dec, j[0]、j[1]...j[7]在内存的地址分别是 0xbfdab05c、0xbfdab060,...0xbfdab078. 如下所示：

高地址 <----------->低地址

i,j[7],j[6],j[5],j[4],j[3],j[2],j[1],j[0]

如果在 int i,j[8]后面再定义变量 int c,那么 c 就存放在 j[0]的往低方向的下一个地址 0xbfdab058 .

现在不难理解这段程序为什么会出现死循环了。j[8]的位置就是变量 i 所在的位置。这样当 i=8 时的 j[i]=0 语句，实际上就是将 i 的值置为 0，然后 i 又从 0 到 8 循环一直下去. 如果将原句改为 int j[8],i; 就不会出现死循环，而仅仅是一个段越界错误.

另一个程序:

```
#include <stdio.h>
int main()
{
        int i;
        char c;
        for(i=0;i<5;i++)
        {
                scanf("%d",&c);
                printf("i=%d ",i);
        }
        printf("\n");
}
```

编译后运行

[foxman@local~]#./a.out

0　　(输入 0)

i=0　(输出 i 值)

1

i=0

2

i=0

3

i=0

4

i=0

...

这样一直循环下去。

问题在于，c 被声明为 char 类型，而不是 int 类型。当程序要求 scanf 读入一个整数时，应该传递给它一个指向整数的指针。而程序中 scanf 得到的却是一个指向字符的指针，scanf 函数并不能分辨这种情况，只能将这个指向字符的指针作为指向整数的指针而接受，并且在指针指向的位置存储一个整数。因为整数所占的存储空间要大于字符所占的存储空间，所以 c 附近的内存会被覆盖.

由上面分析，i 和 c 是由高地址到低地址存储在栈中，这样在 c 所在位置尝试存储一个 4 字节变量，会占用比 c 高的 3 个字节(覆盖掉 i 字节的低 3 位)，即使 i 总是为零，一直循环下去.

如果每次输入 Ctrl+D 作为字符终止符不存储 int 到 c 处，那么就会输出正常 i=0..4 了.

52. 空指针究竟指向了内存的哪个地方

发布: 2008-7-22 10:52 | 作者: 剑心通明 | 来源: 互联网 | 查看: 15 次

空指针究竟指向了内存的哪个地方？

NULL 在"stdio.h"中被宏定义为 0（或其他什么常数〈视编译器而定〉）

空指针指向你进程私有地址的 0 地址,，它不会被分配出去，主要的不是 null 指在哪，而是如果指向 null,我们的代码就可以用 if(ptr)来判断它是不是有效的指针，不过，如果这个指针不是指向 0，也有可能不是个有效的指针，所以建议程序员在定义指针时把它初始化为 0。

NULL 只是一个概念，叫作空值，其值本身没有任何含义，可以用 0 代替，也可以用 1,...代替，只要这些值不会与系统实际的有效地址冲突即可。

因此，本人在此再次强调，不要自作聪明地认为 NULL 就是 0，要判断的时候还是老老实实地与 NULL 做比较，别想当然地用什么!ptr 之类的写法，因为在某个特定环境下，NULL 可能不是 0，而系统函数返回的是 NULL 不是 0，那时，你的函数就会出现莫名其妙的错误。所以，养成良好的习惯是非常重要的。

NULL 指针的值一定是 0，这点可参照 C 语言标准。以下来自 C99(WG14/N843 199：

> [#3] An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant.46) If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

.....

> 46)The macro NULL is defined in <stddef.h>; as a null pointer constant; see 7.17.

虚拟内存是有权限控制的。0x0 这个位置是规定了不可读写，因此试图读写的时候必然引发 Access Volitation

MEM 中有个地址转换表！空指针指向一个内核保留的只读地址！由 MEM（内存管理）完成映射

如果仅仅声明一个指针，而没有任何赋值，那么这个指针是野指针，它会指到 VM 的任何位置，碰到异常操作，比如对只读区写操作，就会引起硬件中断产生 core，也就是通常的段错误。

良好的编程风格是将指针永远都可控，也就是这个指针的地址，程序可控，通常，对于不使用或初

始的指针都将其地址置为 0，这是约定俗成的，就如同，我们经常使用的进制一样，你非用一个别人都不用的进制表示数，那也随你，只是别人觉得怪而已。再比如，用 free 释放完指针后，相信大家都会将指针置成 NULL 或 0，就是为了再使用这个指针时，便于判断。指针的地址为 0，操作起来就非常方便，比较位操作等，都可对应到机器码，这也就体现了“高级汇编”的美誉。用 NULL 宏，仅仅是为了可读性，编译器会进行优化的。

对于将 NULL 定义成某个地址，然后进行比较，相对 NULL 为 0 地址，然后比较，性质是相同的，在执行过程中，如果重新定义的地址为可操作，可能会对程序的逻辑流程产生影响。

把帖子中出现的几个问题总结了一下，结合 C99 标准试图给出较为明确的答案：

1. 什么是空指针常量（null pointer constant）?

[6.3.2.3-3] An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant.

这里告诉我们：0、0L、'\0'、3 - 3、0 * 17 （它们都是“integer constant expression”）以及 (void*)0 等都是空指针常量（注意 (char*) 0 不叫空指针常量，只是一个空指针值）。至于系统选取哪种形式作为空指针常量使用，则是实现相关的。一般的 C 系统选择 (void*)0 或者 0 的居多（也有个别的选择 0L）；至于 C++ 系统，由于存在严格的类型转化的要求，void* 不能象 C 中那样自由转换为其它指针类型，所以通常选 0 作为空指针常量，而不选择 (void*)0。

2.什么是空指针（null pointer）?

[6.3.2.3-3] If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

因此，如果 p 是一个指针变量，则 p = 0;、p = 0L;、p = '\0';、p = 3 - 3;、p = 0 * 17; 中的任何一种赋值操作之后（对于 C 来说还可以是 p = (void*)0;）， p 都成为一个空指针，由系统保证空指针不指向任何实际的对象或者函数。反过来说，任何对象或者函数的地址都不可能是空指针。

3.什么是 NULL?

[6.3.2.3-Footnote] The macro NULL is defined in <stddef.h> (and other headers) as a null pointer constant

即 NULL 是一个标准规定的宏定义，用来表示空指针常量。因此，除了上面的各种赋值方式之外，还可以用 p = NULL; 来使 p 成为一个空指针。

4.空指针（null pointer）指向了内存的什么地方（空指针的内部实现）?

标准并没有对空指针指向内存中的什么地方这一个问题作出规定，也就是说用哪个具体的地址值（0x0 地址还是某一特定地址）表示空指针取决于系统的实现。我们常见的空指针一般指向 0 地址，即空指针的内部用全 0 来表示（zero null pointer，零空指针）；也有一些系统用一些特殊的地址值或者特殊的方式表示空指针（nonzero null pointer，非零空指针），具体请参见 C FAQ。

幸运的是，在实际编程中不需要了解在我们的系统上空指针到底是一个 zero null pointer 还是 nonzero null pointer，我们只需要了解一个指针是否是空指针就可以了——编译器会自动实现其中的转换，为我们屏蔽其中的实现细节。注意：不要把空指针的内部表示等同于整数 0 的对象表示——如上所述，有时它们是不同的。

5.如何判断一个指针是否是一个空指针？

这可以通过与空指针常量或者其它的空指针的比较来实现（注意与空指针的内部表示无关）。例如，假设 p 是一个指针变量，q 是一个同类型的空指针，要检查 p 是否是一个空指针，可以采用下列任意形式之一——它们在实现的功能上都是等价的，所不同的只是风格的差别。

指针变量 p 是空指针的判断：

```
if ( p == 0 )
if ( p == '\0' )
if ( p == 3 - 3 )
if ( p == NULL ) /* 使用 NULL 必须包含相应的标准库的头文件 */
if ( NULL == p )
if ( !p )
if ( p == q )
...
```

指针变量 p 不是空指针的判断：

```
if ( p != 0 )
if ( p != '\0' )
if ( p != 3 - 3 )
if ( p != NULL ) /* 使用 NULL 必须包含相应的标准库的头文件 */
if ( NULL != p )
if ( p )
if ( p != q )
...
```

6.可以用 memset 函数来得到一个空指针吗？

这个问题等同于：如果 p 是一个指针变量，那么

memset(&p, 0, sizeof(p)); 和 p = 0;

是等价的吗？

答案是否定的，虽然在大多数系统上是等价的，但是因为有的系统存在着“非零空指针”（nonzero null pointer），所以这时两者不等价。由于这个原因，要注意当想将指针设置为空指针的时候不应该使用 memset，而应该用空指针常量或空指针对指针变量赋值或者初始化的方法。

7.可以定义自己的 NULL 的实现吗？兼答"NULL 的值可以是 1、2、3 等值吗？"类似问题

[7.1.3-2] If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 7.1.4), or defines a reserved identifier as a macro name, the behavior is undefined.

NULL 是标准库中的一个符合上述条件的 reserved identifier （保留标识符）。所以，如果包含了相应的标准头文件而引入了 NULL 的话，则再在程序中重新定义 NULL 为不同的内容是非法的，其行为是未定义的。也就是说，如果是符合标准的程序，其 NULL 的值只能是 0，不可能是除 0 之外的其它值，比如 1、2、3 等。

8.malloc 函数在分配内存失败时返回 0 还是 NULL？

malloc 函数是标准 C 规定的库函数。在标准中明确规定了在其内存分配失败时返回的是一个“null pointer”（空指针）：

[7.20.3-1] If the space cannot be allocated, a null pointer is returned.

对于空指针值，一般的文档（比如 man）中倾向于用 NULL 表示，而没有直接说成 0。但是我们应该清楚：对于指针类型来说，返回 NULL 和 返回 0 是完全等价的，因为 NULL 和 0 都表示“null pointer”（空指针）。

53. 算术表达式的计算

发布: 2008-7-12 21:11 | 作者: 剑心通明 | 来源: 互联网 | 查看: 0 次

在计算机中进行算术表达式的计算是通过栈来实现的。这一节首先讨论算术表达式的两种表示方法，即中缀表示法和后缀表示法，接着讨论后缀表达式求值的算法，最后讨论中缀表达式转换为后缀表达式的算法。

1. 算术表达式的两种表示

通常书写的算术表达式是由操作数(又叫运算对象或运算量)和运算符以及改变运算次序的圆括号连接而成的式子。操作数可以是常量、变量和函数，同时还可以是表达式。运算符包括单目运算符和双目运算符两类，单目运算符只要求一个操作数，并被放在该操作数的前面，双目运算符要求有两个操作数，并被放在这两个操作数的中间。单目运算符为取正’+’和取负’-’，双目运算符有加’+’,减’-’，乘’*’和除’/’等。为了简便起见，在我们的讨论中只考虑双目运算符。

如对于一个算术表达式 2+5*6，乘法运算符’*’的两个操作数是它两边的 5 和 6；对于加法运算符’+’的两个操作数，一个是它前面的 2，另一个是它后面的 5*6 的结果即 30。我们把双目运算符出现在两个操作数中间的这种习惯表示叫做算术表达式的中缀表示，这种算术表达式被称为中缀算术表达式或中缀表达式。

中缀表达式的计算比较复杂，它必须遵守以下三条规则：

(1) 先计算括号内，后计算括号外；

(2) 在无括号或同层括号内，先进行乘除运算，后进行加减运算，即乘除运算的优先级高于加减运算的优先级；

(3) 同一优先级运算，从左向右依次进行。

从这三条规则可以看出，在中缀表达式的计算过程中，既要考虑括号的作用，又要考虑运算符的优先级，还要考虑运算符出现的先后次序。因此，各运算符实际的运算次序往往同它们在表达式中出现的先后次序是不一致的，是不可预测的。当然凭直观判别一个中缀表达式中哪个运算符最先算，哪个次之，……，哪个最后算并不困难，但通过计算机处理就比较困难了，因为计算机只能一个字符一个字符地扫描，要想得到哪一个运算符先算，就必须对整个中缀表达式扫描一遍，一个中缀表达式中有多少个运算符，原则上就得扫描多少遍才能计算完毕，这样就太浪费时间了，显然是不可取的。

那么，能否把中缀算术表达式转换成另一种形式的算术表达式，使计算简单化呢? 回答是肯定的。波兰科学家卢卡谢维奇(Lukasiewicz)很早就提出了算术表达式的另一种表示，即后缀表示，又称逆波兰式，其定义是把运算符放在两个运算对象的后面。采用后缀表示的算术表达式被称为后缀算术表达式或后缀表达式。在后缀表达式中，不存在括号，也不存在优先级的差别，计算过程完全按照运算符出现的先后次序进行，整个计算过程仅需一遍扫描便可完成，显然比中缀表达式的计算要简单得多。例如，对于后缀表达式 12!4!-!5!/，其中’!’字符表示成分之间的空格，因减法运算符在前，除法运算符在后，所以应先做减法，后做除法；减法的两个操作数是它前面的 12 和 4，其中第一个数 12 是被减数，第二个数 4 是减数；除法的两个操作数是它前面的 12 减 4 的差(即 8)和 5，其中 8 是被除数，5 是除数。

中缀算术表达式转换成对应的后缀算术表达式的规则是：把每个运算符都移到它的两个运算对象的后面，然后删除掉所有的括号即可。

例如，对于下列各中缀表达式：

(1) 3/5+6

(2) 16-9*(4+3)

(3) 2*(x+y)/(1-x)

(4) (25+x)*(a*(a+b)+b)

对应的后缀表达式分别为：

(1) 3!5!/!6!+

(2) 16!9!4!3!+!*!-

(3) 2!x!y!+!*!1!x!-!/

(4) 25!x!+!a!a!b!+!*!b!+!*

2. 后缀表达式求值的算法

后缀表达式的求值比较简单，扫描一遍即可完成。它需要使用一个栈，假定用 S 表示，其元素类型应为操作数的类型，假定为浮点型 float，用此栈存储后缀表达式中的操作数、计算过程中的中间结果以及最后结果。假定一个后缀算术表达式以字符’@’作为结束符，并且以一个字符串的方式提供。后缀表达式求值算法的基本思路是：把包含后缀算术表达式的字符串定义为一个输入字符串流对象，每次从中读入一个字符（空格作为数据之间的分隔符，不会被作为字符读入）时，若它是运算符，则表明它的两个操作数已经在栈 S 中，其中栈顶元素为运算符的后一个操作数，栈顶元素的下一个元素为运算符的前一个操作数，把它们弹出后进行相应运算即可，然后把运算结果再压入栈 S 中；否则，读入的字符必为操作数的最高位数字，应把它重新送回输入流中，然后把下一个数据作为浮点数输入，并把它压入到栈 S 中。依次扫描每一个字符（对于浮点数只需扫描它的最高位并一次输入整个浮点数）并进行上述处理，直到遇到结束符’@’为止，表明后缀表达式计算完毕，最终结果保存在栈中，并且栈中仅存这一个值，把它弹出返回即可。具体算法描述为：

```
float Compute(char* str)
// 计算由 str 字符串所表示的后缀表达式的值，
// 表达式要以'@'字符结束。
...{
Stack S; // 用 S 栈存储操作数和中间计算结果
InitStack(S); // 初始化栈
istrstream ins(str); // 把 str 定义为输入字符串流对象 ins
char ch; // 用于输入字符
float x; // 用于输入浮点数
ins>>ch; // 从 ins 流对象(即 str 字符串)中顺序读入一个字符
while(ch!='@')
...{ // 扫描每一个字符并进行相应处理
switch(ch)
...{
case '+':
x=Pop(S)+Pop(S);
break;
case '-':
x=Pop(S); // Pop(S)弹出减数
x=Pop(S)-x; // Pop(S)弹出的是被减数
```

```
break;
case '*':
x=Pop(S)*Pop(S);
break;
case '/':
x=Pop(S); // Pop(S)弹出除数
if(x!=0.0)
x=Pop(S)/x; // Pop(S)弹出的是被除数
else ...{ // 除数为 0 时终止运行
cerr<<"Divide by 0!"< exit(1);
}
break;
default: // 读入的必为一个浮点数的最高位数字
ins.putback(ch); // 把它重新回送到输入流中
ins>>x; // 从字符串输入流中读入一个浮点数
}
Push(S,x); // 把读入的一个浮点数或进行相应运算
// 的结果压入到 S 栈中
ins>>ch; // 输入下一个字符，以便进行下一轮循环处理
}
if(!StackEmpty(S))
...{ // 若栈中仅有一个元素，则它是后缀表达式的值，否则为出错
x=Pop(S);
if(StackEmpty(S))
return x;
else ...{
cerr<<"expression error!"< exit(1);
}
}
else ...{ // 若最后栈为空，则终止运行
cerr<<"Stack is empty!"< exit(1);
}
}
```

此算法的运行时间主要花在 while 循环上，它从头到尾扫描后缀表达式中的每一个数据（每个操作数或运算符均为一个数据），若后缀表达式由 n 个数据组成，则此算法的时间复杂度为 O(n)。此算法在运行时所占用的临时空间主要取决于栈 S 的大小，显然，它的最大深度不会超过表达式中操作数的个数，因为操作数的个数与运算符（假定把’@’也看作为一个特殊运算符，即结束运算符）的个数相等，所以此算法的空间复杂度也同样为 O(n)。

假定一个字符串 a 为：

char a[30]="12 3 20 4 / * 8 - 6 * +@";

对应的中缀算术表达式为 12+(3*(20/4)-8)*6@，则使用如下语句调用上述函数得到的输出结果为 54。

cout< 在进行这个后缀算术表达式求值的过程中，从第四个操作数入栈开始，每处理一个操作数或运算符后，栈 S 中保存的操作数和中间结果.

54. 结构体对齐的具体含义

发布: 2008-7-27 10:36 | 作者: 剑心通明 | 来源: 互联网 | 查看: 12 次

最近對結構體對齊比較感興趣,收集了一些資料

结构体对齐的具体含义(#pragma pack)

```
#pragmapack(4)
    class TestB
    {
    public:
        int aa;
        char a;
        short b;
        char c;
    };
    int nSize = sizeof(TestB);
```

这里 nSize 结果为 12，在预料之中。

现在去掉第一个成员变量为如下代码：

```
#pragmapack(4)
class TestC
{
public:
    char a;
    short b;
    char c;
};
int nSize = sizeof(TestC);
```

按照正常的填充方式 nSize 的结果应该是 8，为什么结果显示 nSize 为 6 呢？

事实上，很多人对#pragmapack 的理解是错误的。

#pragmapack 规定的对齐长度，实际使用的规则是：

结构，联合，或者类的数据成员，第一个放在偏移为 0 的地方，以后每个数据成员的对齐，按照 #pragmapack 指定的数值和这个数据成员自身长度中，比较小的那个进行。

也就是说，当#pragmapack 的值等于或超过所有数据成员长度的时候，这个值的大小将不产生任何效果。

而结构整体的对齐，则按照结构体中最大的数据成员 和#pragmapack 指定值 之间，较小的那个进行。

具体解释

#pragmapack(4)

```
class TestB
{
public:
    int aa; //第一个成员，放在[0,3]偏移的位置，
    char a; //第二个成员，自身长为 1，#pragmapack(4),取小值，也就是 1，所以这个成员按一字节对齐，放在偏移[4]的位置。
    short b; //第三个成员，自身长 2，#pragmapack(4)，取 2，按 2 字节对齐，所以放在偏移[6,7]的位置。
    char c; //第四个，自身长为 1，放在[8]的位置。
};
```

这个类实际占据的内存空间是 9 字节

类之间的对齐，是按照类内部最大的成员的长度，和#pragmapack 规定的值之中较小的一个对齐的。

所以这个例子中，类之间对齐的长度是 min(sizeof(int),4)，也就是 4。

9 按照 4 字节圆整的结果是 12，所以 sizeof(TestB)是 12。

如果

#pragmapack(2)

```
class TestB
{
public:
    int aa; //第一个成员，放在[0,3]偏移的位置，
    char a; //第二个成员，自身长为 1，#pragmapack(4),取小值，也就是 1，所以这个成员按一字节对齐，放在偏移[4]的位置。
    short b; //第三个成员，自身长 2，#pragmapack(4)，取 2，按 2 字节对齐，所以放在偏移[6,7]的位置。
    char c; //第四个，自身长为 1，放在[8]的位置。
};
```

//可以看出，上面的位置完全没有变化，只是类之间改为按 2 字节对齐，9 按 2 圆整的结果是 10。

//所以 sizeof(TestB)是 10。

最后看原贴：

现在去掉第一个成员变量为如下代码：

```
#pragmapack(4)
class TestC
{
public:
    char a;//第一个成员，放在[0]偏移的位置，
    short b;//第二个成员，自身长 2，#pragmapack(4)，取 2，按 2 字节对齐，所以放在偏移[2,3]的位置。
    char c;//第三个，自身长为 1，放在[4]的位置。
};
//整个类的大小是 5 字节，按照 min(sizeof(short),4)字节对齐，也就是 2 字节对齐，结果是 6
//所以 sizeof(TestC)是 6。
```

對於位域有如下规定:

C99 规定 int、unsigned int 和 bool 可以作为位域类型，但编译器几乎都对此作了扩展，允许其它类型类型的存在。使用位域的主要目的是压缩存储，其大致规则为：

1) 如果相邻位域字段的类型相同，且其位宽之和小于类型的 sizeof 大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止；

2) 如果相邻位域字段的类型相同，但其位宽之和大于类型的 sizeof 大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍；

3) 如果相邻的位域字段的类型不同，则各编译器的具体实现有差异，VC6 采取不压缩方式，Dev-C++采取压缩方式；

4) 如果位域字段之间穿插着非位域字段，则不进行压缩；

5) 整个结构体的总大小为最宽基本类型成员大小的整数倍。

还是让我们来看看例子。

示例 1：

```
struct BF1
{
    char f1 : 3;
    char f2 : 4;
    char f3 : 5;
};
```

其内存布局为：

```
|_f1__|__f2__|_|____f3___|____|
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
0 3 7 8 1316
```

位域类型为 char，第 1 个字节仅能容纳下 f1 和 f2，所以 f2 被压缩到第 1 个字节中，而 f3 只

能从下一个字节开始。因此 sizeof(BF1)的结果为 2。

示例 2：

```
struct BF2
{
    char f1 : 3;
    short f2 : 4;
    char f3 : 5;
};
```

由于相邻位域类型不同，在 VC6 中其 sizeof 为 6，在 Dev-C++中为 2。

示例 3：

```
struct BF3
{
    char f1 : 3;
    char f2;
    char f3 : 5;
};
```

非位域字段穿插在其中，不会产生压缩，在 VC6 和 Dev-C++中得到的大小均为 3。

55. 连连看 AI 算法

发布: 2008-7-27 09:10 | 作者: 剑心通明 | 来源: 互联网 | 查看: 24 次

大概是两年前无聊的时候写的吧，当时就为了写个外挂，现在看起来代码写的很烂。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_WIDTH 20
#define MAX_HIGH 20
#define MAX_NUMBER 50
#define SAME_NUMBER 4
#define PAUSE
typedef struct tagPOINT {
	int x;
	int y;
} POINT;
int Initmap(int aMap[MAX_WIDTH][MAX_HIGH]);
int PrintMap(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2);
int PrintMap(int aMap[MAX_WIDTH][MAX_HIGH]);
int Find(int aMap[MAX_WIDTH][MAX_HIGH]);
//测试是否结束
bool testEmpty(int aMap[MAX_WIDTH][MAX_HIGH]);
//测试两个点是不是能一条线连接
bool testLine(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2);
//测试一个折线
bool testcorner(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2);
//测试两个折线
bool testtwocorner(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2);
int main(int argc, char* argv[])
{
   int Map[MAX_WIDTH][MAX_HIGH]={0};
  Initmap(Map);
  PrintMap(Map);
  int x=0;
  while (1) {
```

```
    Find(Map);
    if(testEmpty(Map))
      break;
    if(x>5){
      printf("无解，你运气真好！\n");
      break;
    }
  }
  return 0;
}
int Initmap(int aMap[MAX_WIDTH][MAX_HIGH])
{
  srand((unsigned int)time(NULL)); //初始化随机数发生器
  //init map
  for(int map1=0;map1<SAME_NUMBER;map1++){//初始化地图
    for(int i=1;i<=MAX_NUMBER;i++){
      int x=rand()%MAX_WIDTH;
      int y=rand()%MAX_HIGH;
      if(0==aMap[x][y]){
        aMap[x][y]=i;
      }else{
        i--;
      }
    }
  }
  return 1;
}
int PrintMap(int aMap[MAX_WIDTH][MAX_HIGH])
{
  system("cls.exe");
  printf("    ");
  for(int x=0;x<MAX_WIDTH;x++)printf(" %02d",x);
  printf("\n");
  //输出棋盘
  printf("  ┌");
  for(x=0;x<MAX_WIDTH*3/2;x++)printf("─");
  printf("┐\n");
```

```
  for(int a=0;a< MAX_HIGH;a++){
    printf("┃");
    for(int b=0;b<MAX_WIDTH;b++){
      if(0==aMap[b][a]){
        printf("      ");
      }else{
        printf(" %02d",aMap[b][a]);
      }
    }
    printf("┃%d\n",a);
  }
  printf("  ┗");
  for(x=0;x<MAX_WIDTH*3/2;x++)printf("━");
  printf("┛ \n");
  return 1;
}
int PrintMap(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2)
{
// ;
  int temp=aMap[ap1->x][ap1->y];
    system("cls.exe");
  printf("    ");
  for(int x=0;x<MAX_WIDTH;x++)printf(" %02d",x);
  printf("\n");
  printf("  ┏");
  for(x=0;x<MAX_WIDTH*3/2;x++)printf("━");
  printf("┓ \n");
  //输出棋盘
  for(int a=0;a< MAX_HIGH;a++){
    printf("┃");
    for(int b=0;b<MAX_WIDTH;b++){
      if(0==aMap[b][a]){
        printf("      ");
      }else
        if(temp==aMap[b][a]){
          printf("*%02d",aMap[b][a]);
        }else{
          printf(" %02d",aMap[b][a]);
```

```
        }
      }
      printf("丨%d\n",a);
    }
    printf("└");
    for(x=0;x<MAX_WIDTH*3/2;x++)printf("—");
    printf("┘\n");
    aMap[ap1->x][ap1->y]=0;
    aMap[ap2->x][ap2->y]=0;
    return 1;
}
int Find(int aMap[MAX_WIDTH][MAX_HIGH])
{
    POINT p1={0};
    POINT p2={0};
// 列出所有的选择方案
    for(int x1=0;x1<MAX_WIDTH;x1++){
      for(int y1=0;y1<MAX_HIGH;y1++){
        for(int x2=0;x2<MAX_WIDTH;x2++){
          for(int y2=0;y2<MAX_HIGH;y2++){
            p1.x=x1;
            p1.y=y1;
            p2.x=x2;
            p2.y=y2;
            if ((aMap[p1.x][p1.y]==0)||(aMap[p2.x][p2.y]==0) )continue;//不为 0
            if(aMap[p1.x][p1.y]!=aMap[p2.x][p2.y])continue;//相等
            if((x1==x2)&&(y1==y2))continue;//同一个点
            if(testLine(aMap,&p1,&p2)){
              int temp=aMap[p1.x][p1.y];
              PrintMap(aMap,&p1,&p2);
              printf("一直线 数值=%d   p1=%d,%d,p2=%d,%d\n",temp,p1.x,p1.y,p2.x,p2.y);
#ifdef PAUSE
              system("pause");
#endif
              continue;
            }

            if(testcorner(aMap,&p1,&p2)){
```

```
            int temp=aMap[p1.x][p1.y];
            PrintMap(aMap,&p1,&p2);
            printf("二直线 数值=%d   p1=%d,%d,p2=%d,%d\n",temp,p1.x,p1.y,p2.x,p2.y);
#ifdef PAUSE
            system("pause");
#endif
            continue;
          }
          if(testtwocorner(aMap,&p1,&p2)){
            int temp=aMap[p1.x][p1.y];
            PrintMap(aMap,&p1,&p2);
            printf("三直线 数值=%d   p1=%d,%d,p2=%d,%d\n",temp,p1.x,p1.y,p2.x,p2.y);
#ifdef PAUSE
            system("pause");
#endif
            continue;
          }//*/
        }
      }
    }
  }
  return 1;
}

bool testLine(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2)//测试两个点是不是能一条线连接
{
  if((ap1->x==ap2->x)||(ap1->y==ap2->y)){
    if(ap1->x==ap2->x){ //x 轴相等
      int lisum=0;
      int min=0;
      int max=0;
      if(ap1->y > ap2->y){
        min=ap2->y;
        max=ap1->y;
      }else{
        max=ap2->y;
        min=ap1->y;
      }
```

```
      for(int x=min+1;x<max;x++){
        if(aMap[ap1->x][x])return false;
      }
      return true;
    }
    if(ap1->y==ap2->y){ //x 轴相等
      int lisum=0;
      int min=0;
      int max=0;
      if(ap1->x > ap2->x){
        min=ap2->x;
        max=ap1->x;
      }else{
        max=ap2->x;
        min=ap1->x;
      }
      for(int x=min+1;x<max;x++){
        if(aMap[x][ap2->y])return false;
      }
      return true;
    }
  }else{
    return false;
  }
  return false;
}
bool testcorner(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2)
{
  POINT tempPoint={0};
  for(int x=0;x<MAX_WIDTH;x++){
    for(int y=0;y<MAX_HIGH;y++){
      if(0!=aMap[x][y])
        continue;//不是空白
      tempPoint.x=x;
      tempPoint.y=y;
      if(testLine(aMap,ap1,&tempPoint)){
        if(testLine(aMap,&tempPoint,ap2)){
          //printf("tp %d,%d\n",x,y);
          return true;
```

```
      }
     }
    }
   }
  return false;
}
bool testtwocorner(int aMap[MAX_WIDTH][MAX_HIGH],POINT *ap1,POINT* ap2)
{
  POINT tP1={0};
  POINT tP2={0};
  for(int x1=0;x1<MAX_WIDTH;x1++){
    for(int y1=0;y1<MAX_HIGH;y1++){
      for(int x2=0;x2<MAX_WIDTH;x2++){
        for(int y2=0;y2<MAX_HIGH;y2++){
          if(0!=aMap[x1][y1])
            continue;//不是空白
          if(0!=aMap[x2][y2])
            continue;//不是空白
          tP1.x=x1;
          tP1.y=y1;
          tP2.x=x2;
          tP2.y=y2;
          if(testLine(aMap,&tP1,&tP2)){
            if(testLine(aMap,&tP1,ap1)&&testLine(aMap,&tP2,ap2)){
              printf("t1=%d,%d t2=%d,%d\n",x1,y1,x2,y2);
              return true;
            }
          }
        }
      }
    }
  }
  return false;
}
bool testEmpty(int aMap[MAX_WIDTH][MAX_HIGH])
{
  int lisum=0;
  for(int x=0;x<MAX_WIDTH;x++){
    for(int y=0;y<MAX_HIGH;y++){
```

```
      lisum=lisum+aMap[x][y];
    }
   }
   if (0==lisum)
    return true;
   return false;
  }
```

56. 连连看寻路算法的思路

发布: 2008-7-22 10:44 | 作者: 剑心通明 | 来源: 互联网 | 查看: 29 次

图一：

0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 8, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 9, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0

这是一张连连看的地图，假设标 8 和 9 的部分是两张相同的牌。

在数组矩阵中，0 表示没有牌，大于 1 表示有牌。至于是什么牌，那是随机的了。不要告诉我，你说的“布局算法”是指怎么把牌刚刚好放上去，那个无所谓什么算法，你只要首先在地图数组中准备好偶数个 1，在布牌时保证每种牌是偶数个（不同种类的牌用大于 1 的数来表示），相应地放入每个 1 的位置上就可以了。

一、计算地图上这两张牌能不能连通(当然能了，哈哈)。

这是连连看寻路算法的第一步。
先定义一下两张牌能连的充分条件：
1.两张牌是同一种。
2.两张牌之间有一条全是 0 的路可以连通。
3.这一条路不能有两个以上的拐角(corner)
满足这三个条件，就可以认为这两张牌是可以连的。

首先，我们依据前两个条件来完成一个基本的寻路算法。
我们的目的是从 8 到 9 找出一条可以连通的路来。
那么很明显从 8 到 9 的第一步一其有四个方向可以选择，分别是东，南，西，北（e, s, w, n 以中国地图方向为标准）四个方向，在第一步中我们首先假设四个方面没有任何优劣，那么我可以任意选择一个方向移动，那就是东面吧。
图二：

0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 8, －8, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 9, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
我从 8 向东移动了一步，所以到达了-8 的位置，我之所以可以移到-8 位置，很明显，
是因为-8 的位置上原来是一个 0，表示没有牌阻挡。
那么现在寻路的问题就变成了，如何从－8 找连通 9 的路了！
说到这里应该明白了吧，好多费话，有点像娘们在说话。

所以目前的寻路算法归结为一个递归算法的基本问题。
先从 8 到找到下一个结点－8，再用同样的规则，从－8 找到下一个结点，比如－88。。。
图三：
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 8, －8, －88, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0
0, 0, 0, 0, 0, 0, 0, 0 , 9, 0
0, 0, 0, 0, 0, 0, 0, 0 , 0, 0

如果一直都能 OK，没有阻碍的话，最后找到了 9，就算成功以，如要有一步不能走下去了，
就再退回上个结点，向别的方向发展，都不行，就再退回上级结点，再向别的方向发展，
这里的逻辑就是递归的思想了。

用这样的方法写出来的算法已经能在最优的情形下用了，比如从 8，到－88，哈哈。
但在稍微复杂的情况下，会产生奇多的递归结点。P4 机也跑不动啊。我试过，哈哈。

那么第二步就是为（e,s,w,n）四个方向加权，也就是让它们之间有一个优先权，说白了就
是先试哪一条路。决定的法则应该有好几个吧，比如以 9 号的位置来看，它处于 8 号的东南面，
那试路时当然应当优先选择东面和南面，再比如 9 号如果牌 8 号的正东面，那当然是选择正东了。

再比如，当走到－8 的位置时，明显只能再走三个方向，因为它是不能回头的。

经过这样的处理，递归算法生成的结点数会明显变少，会更快的找到成功的路。但性能在最坏情况下没有本质改变。

接下来，第三步，我们把第三个充分条件加进来，来解决根本问题。

3.这一条路不能有两个以上的拐角(corner)

按照面向对象的思想，很自然的，我给每个在递归算法中生成的位置结点加上了个 corner 的属性，来记录这条路到目前为止拐了几个角。

这样一下子就好办了啊。如果发现这个结点已经拐了两个弯时，如果要再拐弯，或者到达 9 之前注定

要再增加 cornor 时，就果断 over，返回上级结点。

好，就说到这儿吧，不能再多说了，否则就是等于把代码公开了，哈哈。

注意，要把二、三两步的条件综合起来详细规划一个个可能性，尽可能提早让不可能的结点 OVER，这就是提高性能的关键吧。你的算法预见性越强，性能就越高吧。

我们的算法在赛扬 500，256M 的机器上，10 万次平均结果是一次运算花时不超过 0.1 毫秒,算得还不

精确，速度确实很快，因为在很坏的情形下，产生的最大结点数是 690 几个，这样必然会很快的，详细的数据已经记不清了。

说了这么多了，应当明白第一步连通算法的思路了吧，我所知道的，都已经尽可能的讲出来了。这个算法完全是自己按照连连看的特点，度身定做的。因为是一步步 test 出来的，所以我个人觉得是很自然的，没有任何高深的地方，完成后，性能也很好，这才觉得是如此的简单。相信大家仔细看看就能写出自己的算法来的吧。

二、整个地图有没有解？？？可以连通的总牌数？

这是一个问题。

解决这个问题之前，我们先来解决提示功能(hint）就是为玩家提供提示，哪一对牌可以连通。

我的做法是，先把整个地图中相同牌归到一起，用数组也好，链表也好。

像这样，

4,4,4,4

5,5,5,5

6,6,6,6

.....

然后计算比如 4 个 4 之间有哪两对可以连，至于如何判断能不能连，第一步算法已经实现了吧，哈哈。

一发现有可以连的牌就退出来，告诉玩家这两张牌可以连啊！

这就 OK 了。

这完全是建立在第一步算法的基础上实现的。

好的，hint 功能可以了，

那么，整个地图没有解的算法是不是出来了？

是的，如果找不到可以 hint 的牌，那当然就是没有解了！

把所有可以 hint 的对数记下来，就是总的可以连通数了。

至此，与连连看所有算法有关的问题解决完毕。

第二步算法的实现，明显开销要大很多，最坏情况应当会是单次连通算法计算量的大约 50 倍以上（与牌的总数量和摆的位置有关）还好在一般的服务器上单次连通计算花的时间实在太少了，实际运行中完全可以很流畅。以上数据都是我估计的理论值，因为实际测试时一直没有问题，我也懒得计算出真正比较可靠的均值了。

这一部分也是我认为可以有所改进的部分，公开出来，也希望大家能提供一些更好，更巧妙的方法，我觉得我计算连通数和有无解的方法是比较笨的方法，几乎是仗着基本的算法快，一个一个算出来的。有没有更好的方法呢？期待中，我也会再想想，太忙，太懒，主要是懒，觉得可以用就行了。

57. 重新认识:指向函数的指针

发布: 2008-7-27 09:34 | 作者: 剑心通明 | 来源: 互联网 | 查看: 15 次

摘要：

本文主要介绍了指向函数的指针的详细用法。

又读了两天的 DirectFB 的源码，头都大了！不过不的不承认，她的结构很有个性，我也比较欣赏。今天在阅读其中的/src/gfx/generic/Generic.c 里面的函数 bool gAcquire(CardState *state, DFBAccelerationMask accel){...}时，发现有一个定义：

GenefxFunc *funcs;

funcs 指向一个 GenefxFunc funcs[32];

以后对 funcs 的操作多为：

...

*funcs++ = Dacc_premultiply;

*funcs++ = Dacc_xor;

...

而 GenefxFunc 的定义格式为：

typedef void (*GenefxFunc)(GenefxState *gfxs);

以前没有见国，呵呵。后来在网上发现了一篇文章，贴出来，如下：

作者:csumck 来源:CSDN

函数指针与 typedef

关于 C++中函数指针的使用(包含对 typedef 用法的讨论)

（一）简单的函数指针的应用。

```
//形式 1：返回类型(*函数名)(参数表)
char (*pFun)(int);
char glFun(int a){ return;}
void main()
{
    pFun = glFun;
```

```
    (*pFun)(2);
}
```

第一行定义了一个指针变量 pFun。首先我们根据前面提到的“形式 1”认识到它是一个指向某种函数的指针，这种函数参数是一个 int 型，返回值是 char 类型。只有第一句我们还无法使用这个指针，因为我们还未对它进行赋值。

第二行定义了一个函数 glFun()。该函数正好是一个以 int 为参数返回 char 的函数。我们要从指针的层次上理解函数——函数的函数名实际上就是一个指针，函数名指向该函数的代码在内存中的首地址。

然后就是可爱的 main()函数了，它的第一句您应该看得懂了——它将函数 glFun 的地址赋值给变量 pFun。main()函数的第二句中“*pFun”显然是取 pFun 所指向地址的内容，当然也就是取出了函数 glFun()的内容，然后给定参数为 2。

（二）使用 typedef 更直观更方便。

```
//形式 2：typedef 返回类型(*新类型)(参数表)
typedef char (*PTRFUN)(int);
PTRFUN pFun;
char glFun(int a){ return;}
void main()
{
    pFun = glFun;
    (*pFun)(2);
}
```

typedef 的功能是定义新的类型。第一句就是定义了一种 PTRFUN 的类型，并定义这种类型为指向某种函数的指针，这种函数以一个 int 为参数并返回 char 类型。后面就可以像使用 int,char 一样使用 PTRFUN 了。

第二行的代码便使用这个新类型定义了变量 pFun，此时就可以像使用形式 1 一样使用这个变量了。

（三）在 C++类中使用函数指针。

```
//形式 3：typedef 返回类型(类名::*新类型)(参数表)
class CA
{
 public:
    char lcFun(int a){ return; }
};
CA ca;
typedef char (CA::*PTRFUN)(int);
PTRFUN pFun;
void main()
{
    pFun = CA::lcFun;
    ca.(*pFun)(2);
```

}

在这里，指针的定义与使用都加上了“类限制”或“对象”，用来指明指针指向的函数是那个类的这里的类对象也可以是使用 new 得到的。比如：

```
CA *pca = new CA;
pca->(*pFun)(2);
delete pca;
```

而且这个类对象指针可以是类内部成员变量，你甚至可以使用 this 指针。比如：

类 CA 有成员变量 PTRFUN m_pfun;

```
void CA::lcFun2()
{
    (this->*m_pFun)(2);
}
```

一句话，使用类成员函数指针必须有“->*”或“.*”的调用。

58. 链表的源码

发布: 2008-7-20 08:32 | 作者: 剑心通明 | 来源: 互联网 | 查看: 7 次

```
/* ========== Program Description ============== */
/*  程序名称：createList.c                                                    */
/*  程序目的：设计一个将输入的数据建立成链表的程序      */
/* Written By Kuo-Yu Huang. (WANT Studio.)                          */
/* ============ Program Description ============ */

#include <stdio.h>
#include <malloc.h>
#define Max 10

struct List       /*节点结构声明*/
{
  int Number;
  char Name[Max];
  struct List *Next;
};
typedef struct List Node;
typedef Node *Link;

/*释放链表*/
void Free_List(Link Head)
{
  Link Pointer;     /*节点声明*/
  while(Head!=NULL)   /*当节点为 NULL，结束循环*/
  {
    Pointer=Head;
    Head=Head->Next; /*指向下一个节点*/
    free(Pointer);
  }
}

/*输出链表*/
void Print_List(Link Head)
```

```
{
  Link Pointer;      /*节点声明*/
  Pointer = Head;      /*Pointer 指针设为首节点*/
  while(Pointer != NULL) /*当节点为 NULL 结束循环*/
  {
    printf("##Input Data##\n");
    printf("Data Number: %d\n", Pointer->Number);
    printf("Data Name: %s\n", Pointer->Name);
    Pointer = Pointer->Next;    /*指向下一个节点*/
  }
}

/*建立链表*/
Link Create_List(Link Head)
{
  int DataNum;              /*数据编号*/
  char DataName[Max];      /*数据名称*/
  Link New;                 /*节点声明*/
  Link Pointer;        /*节点声明*/
  int i;
  Head = (Link)malloc(sizeof(Node));      /*分配内存*/
  if(Head == NULL)
    printf("Memory allocate Failure!\n");      /*内存分配失败*/
  else
  {
    DataNum = 1; /*初始数据编号*/
    printf("Please input the data name:");
    scanf("%s",DataName);
    Head->Number = DataNum; /*定义首节点数据编号*/
    for(i=0; i <= Max; i++ )
      Head->Name[i] = DataName[i]; /*  将 DataName 的值赋给 Name */
    Head->Next = NULL;
    Pointer=Head; /*Pointer 指针设为首节点*/

    while(1)
    {
      DataNum++ ; /*数据编号递增*/
```

```
    New = (Link)malloc(sizeof(Node));   /*分配内存*/
    printf("Please input the data Name:");
    scanf("%s",DataName);
    if(DataName[0] == '0') /*输入 0 则结束*/
      break;
    New->Number = DataNum;
    for(i=0; i < Max; i++ )
    {
      New->Name[i] = DataName[i];
    }
    New->Next = NULL;
    Pointer->Next = New;   /*将新节点串连在原列表尾端*/
    Pointer=New;           /*列表尾端节点为新节点*/
   }
  }
  return Head;
}

/*主程序*/
void main()
{
  Link Head = NULL;          /*节点声明*/
  Head = Create_List(Head);   /*调用建立链表函数*/
  if(Head != NULL)
  {
   Print_List(Head);   /*调用输出链表数据函数*/
   Free_List(Head);   /*调用释放链表函数*/
  }
}
```

59. 高质量的子程序

发布: 2008-7-21 16:15 | 作者: 剑心通明 | 来源: 互联网 | 查看: 3 次

总体问题

- 创建子程序的理由充分吗？
- 如果把一个子程序中的某些部分独立成另一个子程序会更好的话，你这样做了吗？
- 是否用了明显而清楚的动宾词组对过程进行命名？是否是用返回值的描述来命名函数？
- 子程序的名称是否描述了它做的所有工作？
- 子程序的内聚性是不是很强的功能内聚性？它只做一件工作并做得很好吗？
- 子程序的耦合是不是松散的？两个子程序之间的联系是不是小规模、密切、可见和灵活的？
- 子程序的长度是不是它的功能和逻辑自然地决定的：而不是由人为标准决定的？

防错性编程

- 断言是否用于验证假设？
- 子程序对于非法输入数据进行防护了吗？
- 子程序是否能很好地进行程序终止？
- 子程序是否能很好地处理修改情况？
- 是否不用很麻烦地启用或去掉调试帮助？
- 是否信息隐蔽、松散耦合，以及使用“防火墙”数据检查，以使得它不影响子程序之外的代码？
- 子程序是否检查返回值？
- 产品代码中的防错性代码是否帮助用户，而不是程序员？

参数传递问题

- 形式参数与实际参数匹配吗？
- 子程序中参数的排列合理吗？与相似子程序中的参数排列顺序匹配吗？
- 接口假设说明了吗？
- 子程序中参数个数是不是 7 个或者更少，
- 是否只传递了结构化变量中另一个子程序用得到的部分？
- 是否用到了每一个输入参数？
- 是否用到了每一个输出参数?
- 如果子程序是一函数，是否在所有情况下它都会返回一个值？

总结：

- 建立子程序的最重要原因是加强可管理性（即降低复杂性），其它原因还有节省空间、改进正确性、可靠性、可修改性等等。

- 强调强内聚性和松散耦合的首要原因是它们提供了较高层次的抽象性，你可以认为一个具备这种特性的子程序运行是独立的，这可以使你集中精力完成其它任务。

- 有些情况下，放入子程序而带来巨大收益的操作可能是非常简单的。

- 子程序的名称表明了它的质量，如果名称不好但却是精确的，那么说明它的设计也是非常令人遗憾的。如果一个子程序的名称既不好又不精确，那它根本就无法告诉你程序作了些什么。无论哪种情况，都说明程序需要改进。

- 防错性编程可以使错误更容易被发现和修复，对最终软件的危害性显著减小

60. 高级C语言程序员测试必过的十六道最佳题目+答案详解

发布: 2008-2-18 20:49 | 作者: 剑心通明 | 来源: 互联网 | 查看: 80 次

◆假定在所有的程序中必须的头文件都已经被正确包含。

考虑如下的数据类型：

◆char 为 1 个字节

◆int 为 4 个字节

◆long int 为 4 个字节

◆float 为 4 个字节

◆double 为个 8 字节

◆long double 为 8 个字节

◆指针为 4 个字节

1、Consider the following program：

```
#include<setjmp.h>

static jmp_buf   buf;

main()

{

volatile   int b;

b =3;

if(setjmp(buf)!=0)

{
```

```
printf("%d ", b);

exit(0);

}

b=5;

longjmp(buf , 1);

}
```

The output for this program is：

(a) 3

(b) 5

(c) 0

(d) None of the above

2、Consider the following program：

```
main()

{

struct node

{

int a;

int b;

int c;

};
```

```
struct node   s= { 3, 5,6 };

struct node *pt = &s;

printf("%d" ,   *(int*)pt);

}
```

The output for this program is：

(a) 3

(b) 5

(c) 6

(d) 7

3、Consider the following code segment：

```
int   foo ( int x , int   n)

{

int val;

val =1;

if (n>0)

{

if (n%2 == 1)   val = val *x;

val = val * foo(x*x , n/2);

}

return val;
```

```
}
```

What function of x and n is compute by this code segment?

(a) x^n

(b) x*n

(c) n^x

(d) None of the above

4、Consider the following program：

```
main()

{

int   a[5] = {1,2,3,4,5};

int *ptr =   (int*)(&a+1);

printf("%d %d" , *(a+1), *(ptr-1) );

}
```

The output for this program is：

(a) 2 2

(b) 2 1

(c) 2 5

(d) None of the above

5、Consider the following program：

```
void foo(int [][3] );

main()
```

```
{

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

foo(a);

printf("%d" , a[2][1]);

}

void foo( int b[][3])

{

++ b;

b[1][1] =9;

}
```

The output for this program is：

(a) 8

(b) 9

(c) 7

(d) None of the above

6、Consider the following program：

```
main()

{

int a, b,c, d;

a=3;
```

```
b=5;

c=a,b;

d=(a,b);

printf("c=%d" ,c);

printf("d=%d" ,d);

}
```

The output for this program is：

(a) c=3 d=3

(b) c=5 d=3

(c) c=3 d=5

(d) c=5 d=5

7、Consider the following program：

```
main()

{

int a[][3] = { 1,2,3 ,4,5,6};

int (*ptr)[3] =a;

printf("%d %d "   ,(*ptr)[1], (*ptr)[2] );

++ptr;

printf("%d %d"   ,(*ptr)[1], (*ptr)[2] );

}
```

The output for this program is：

(a) 2 3 5 6

(b) 2 3 4 5

(c) 4 5 0 0

(d) None of the above

8、Consider following function：

```
int *f1(void)

{

int x =10;

return(&x);

}

int *f2(void)

{

int*ptr;

*ptr =10;

return ptr;

}

int *f3(void)

{

int *ptr;

ptr=(int*) malloc(sizeof(int));
```

```
return ptr;

}
```

Which of the above three functions are likely to cause problem with pointers

(a) Only f3

(b) Only f1 and f3

(c) Only f1 and f2

(d) f1 , f2 ,f3

9、Consider the following program：

```
main()

{

int i=3;

int j;

j = sizeof(++i+ ++i);

printf("i=%d j=%d", i ,j);

}
```

The output for this program is：

(a) i=4 j=2

(b) i=3 j=2

(c) i=3 j=4

(d) i=3 j=6

10、Consider the following program：

```
void f1(int *, int);

void f2(int *, int);

void(*p[2]) ( int *, int);

main()

{

int a;

int b;

p[0] = f1;

p[1] = f2;

a=3;

b=5;

p[0](&a , b);

printf("%d\t %d\t" , a ,b);

p[1](&a , b);

printf("%d\t %d\t" , a ,b);

}

void f1( int* p , int q)

{

int tmp;
```

```
tmp =*p;

*p = q;

q= tmp;

}

void f2( int* p , int q)

{

int tmp;

tmp =*p;

*p = q;

q= tmp;

}
```

The output for this program is：

(a) 5 5 5 5

(b) 3 5 3 5

(c) 5 3 5 3

(d) 3 3 3 3

11、Consider the following program：

```
void e(int );

main()

{
```

```
int a;

a=3;

e(a);

}

void e(int n)

{

if(n>0)

{

e(--n);

printf("%d" , n);

e(--n);

}

}
```

The output for this program is：

(a) 0 1 2 0

(b) 0 1 2 1

(c) 1 2 0 1

(d) 0 2 1 1

12、Consider following declaration

```
typedef int (*test) ( float * , float*)
```

```
test tmp;
```

type of tmp is

(a) Pointer to function of having two arguments that is pointer to float

(b) int

(c) Pointer to function having two argument that is pointer to float and return int

(d) None of the above

13、Consider the following program：

```
main()
{
char *p;
char buf[10] ={ 1,2,3,4,5,6,9,8};
p = (buf+1)[5];
printf("%d" , p);
}
```

The output for this program is：

(a) 5

(b) 6

(c) 9

(d) None of the above

14、Consider the following program：

```
Void f(char**);
```

```
main()

{

char * argv[] = { "ab" ,"cd" , "ef" ,"gh", "ij" ,"kl" };

f( argv );

}

void f( char **p )

{

char* t;

t= (p+= sizeof(int))[-1];

printf( "%s" , t);

}
```

The output for this program is：

(a) ab

(b) cd

(c) ef

(d) gh

15、Consider the following program：

```
#include<stdarg.h>

int ripple ( int , ...);

main()
```

```
{

int num;

num = ripple ( 3, 5,7);

printf( " %d" , num);

}

int ripple (int n, ...)

{

int i , j;

int k;

va_list p;

k= 0;

j = 1;

va_start( p , n);

for (; j<n;   ++j)

{

i =   va_arg( p , int);

for (; i;     i &=i-1   )

++k;

}
```

```
return k;

}
```

The output for this program is：

(a) 7

(b) 6

(c) 5

(d) 3

16、Consider the following program：

```
int counter (int i)

{

static int count =0;

count = count +i;

return (count );

}

main()

{

int i , j;

for (i=0; i <=5; i++)

j = counter(i);

}
```

The value of j at the end of the execution of the this program is:

(a) 10

(b) 15

(c) 6

(d) 7

Answer With Detailed Explanation

Answer 1.

The answer is (b)

volatile variable isn't affected by the optimization. Its value after the longjump is the last value variable assumed.

b last value is 5 hence 5 is printed.

setjmp : Sets up for nonlocal goto /* setjmp.h*/

Stores context information such as register values so that the lomgjmp function can return control to the statement following the one calling setjmp.Returns 0 when it is initially called.

Lonjjmp： longjmp Performs nonlocal goto /* setjmp.h*/

Transfers control to the statement where the call to setjmp (which initialized buf) was made. Execution continues at this point as if longjmp cannot return the value 0.A

nonvolatile automatic variable might be changed by a call to longjmp.When you use setjmp and longjmp, the only automatic variables guaranteed to remain valid are those declared volatile.

Note: Test program without volatile qualifier (result may very)

Answer 2.

The answer is (a)

The members of structures have address in increasing order of their declaration. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member.

Answer 3.

The answer is (a)

Non recursive version of the program

```
int   what ( int x , int   n)

{

int val;
```

```
int product;

product =1;

val =x;

while(n>0)

{

if (n%2 == 1)

product = product*val;

n = n/2;

val = val* val;

}

}
```

/* Code raise a number (x) to a large power (n) using binary doubling strategy */

Algorithm description

```
(while n>0)

{

if   next most significant binary digit of   n( power)   is one

then multiply accumulated product by current val,

reduce n(power)   sequence by a factor of two using integer division.

get next val by multiply current value of itself
```

```
}
```

Answer 4.

The answer is (c)

type of a is array of int

type of &a is pointer to array of int

Taking a pointer to the element one beyond the end of an array is sure to work.

Answer 5.

The answer is (b)

Answer 6.

The answer is (c)

The comma separates the elements of a function argument list. The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them. the left operand E1 is evaluated as a void expression, then E2 is evaluated to give the result and type of the comma expression. By recursion, the expression E1, E2, ..., En results in the left-to-right evaluation of each Ei, with the value and type of En giving the result of the whole expression.

```
c=a,b;   / *yields c=a* /


d=(a,b); /* d =b   */
```

Answer 7.

The answer is (a)

/* ptr is pointer to array of 3 int */

Answer 8.

The answer is (c)

f1 and f2 return address of local variable ,when function exit local variable disappeared

Answer 9.

The answer is (c)

sizeof operator gives the number of bytes required to store an object of the type of its operand . The operands is either an expression, which is not evaluated ((++i + ++ i) is not evaluated so i remain 3 and j is sizeof int that is 2) or a parenthesized type name.

Answer 10.

The answer is (a)

void(*p[2]) (int *, int);

define array of pointer to function accept two argument that is pointer to int and return int. p[0] = f1; p[1] = f2 contain address of function .function name without parenthesis represent address of function Value and address of variable is passed to function only argument that is effected is a (address is passed). Because of call by value f1, f2 can not effect b

Answer 11.

The answer is (a)

Answer 12.

The answer is (c)

C provide a facility called typedef for creating new data type names, for example declaration

```
typedef char string
```

Makes the name string a synonym for int .The type string can be used in declaration, cast, etc, exactly the same way that the type int can be. Notice that the type being declared in a typedef appears in the position of a variable name not after the word typedef.

Answer 13.

The answer is (c)

If the type of an expression is "array of T" for some type T, then the value of the expression is a pointer to the first object in the array, and the type of the expression is altered to "pointer to T"

So (buf+1)[5] is equvalent to *(buf +6) or buf[6]

Answer 14.

The answer is (d)

p+=sizeof(int) point to argv[2]

(p+=sizeof(int))[-1] points to argv[1]

Answer 15.

The answer is (c)

When we call ripple value of the first argument passed to ripple is collected in the n that is 3. va_start initialize p to point to first unnamed argument that is 5 (first argument).Each call of va_arg return an argument and step p to the next argument. va_arg uses a type name to determine what type to return and how big a step to take Consider inner loop

```
(; i; i&=i-1) k++ /* count number of   1 bit in i *
```

in five number of 1 bits is (101) 2

in seven number of 1 bits is (111) 3

hence k return 5

example

```
let   i= 9 = 1001

i-1   = 1000

(i-1) +1 = i

1000

+1

1 001
```

The right most 1 bit of i has corresponding 0 bit in i-1 this way i & i-1, in a two complement number system will delete the right most 1 bit I(repeat until I become 0 gives number of 1 bits)

Answer 16.

The answer is (b)

Static variable count remain in existence rather than coming and going each time function is called

so first call counter(0) count =0

second call counter(1) count = 0+1;

third call counter(2) count = 1+2; /* count = count +i */

fourth call counter(3) count = 3+3;

fifth call counter(4) count = 6+4;

sixth call counter(5) count = 10+5;

61. C 语言常见错误

发布: 2008-2-18 20:53 | 作者: 剑心通明 | 来源: 互联网 | 查看: 37 次

1.用 malloc 或 farmalloc 动态分配内存时，如

char *buffer;

buffer=(char *)malloc(300);

因为并不是在所有的情况下，都会分配成功

所以应加 if(buffer==NULL) {.....}

2.char far *buffer;

buffer=(char far *)farmalloc(size);

当 size 的值大于 64K 时，很可能不会分配成功，

但是不会返回 NULL，很容易出错。

因此最好 size 的值最好不要大于 64K

3.char 的取值范围为 0-127

unsigned char 的取值范围为 0-255

新手常常在使用 char 时，出错，比如说，

在 256 色的屏幕模式下，颜色索引值为 0-255，

而很多新手用 char color,这样会出错。

4.编程时常会对键盘进行处理

key=bioskey(1);

用 bioskey(1)时，程序从缓冲区取一个键，但是不

会把这个键从缓冲区移走，因此 key 的值将永远保持不变

5.用 if(bioskey(1)!=0) key=bioskey(0);

这样按键时，会从缓冲区移走一个键。

但是当不按键时，key 的值将保持不变，直到按下一键为止

因此应加 else key=0;

6.使用 textcolor(int newcolor);

和 textbackground(int newcolor);

这两个函数来改变字符颜色和背景色时，

用 stdio.h 中的 printf,puts 等向屏幕输出的函数是，并不能

按 textcolor(int),textbackground(int)这两个函数所要求的颜色来显示。

用该用 conio.h 中的 cprintf,cputs 等函数

7.printf("\n");光标会在屏幕上换行并移动到第一列。

但 cprintf("\n");不会，printf 是标准输出函数，与 DOS 相关

DOS 会自动把'\n'处理成'\n'和'\r',而 cprintf 是控制台级函数，不会这样

因此应该用 cprintf("\n\r");

8.float a=1.9;

if(a==1.9) printf("OK");

运行时却不会显示 OK;

不要用 float 型数字进行关系运算中的等于运算

改为 double 双精度型即可解决问题.

9.在 TC2.0 中

```
main()
{
 ......
 .....
}
```

这样编译时不会出错

但是在有些 C 语言版本中,系统会自动当作

```
int main()
{
 }
```

而显示"..should return a value "

10.float a;

```
a=1/2;
 printf("%f",a);
```

运行的结果是 0.000000,而不是 0.500000.

因为/号两边的数字是 1 和 2,系统会当作整型处理,因此出

现这种情况,解决办法是改为 a=1.0/2 或 a=1/2.0 或 1.0/2.0

11.一个函数

```
char *buf()
{
  char *data;
  char str[]="HELLO";
  data=str;
 return data;
}
```

因为系统在调用完函数时,会自动释放为 str 开辟的空间,

因此 return data 无意义

12.对于 findfirst(char *name,struct ffblk,int attrib);

findnext(struct ffblk);

当寻找目录时，用 findfirst("*.*",&ffblk,FA_DIREC);

虽然所找文件的属性为目录，但是会将当前目录下的所有文件找出来

用 findfirst("*.",&ffbli,FA_DIREC)时，找出来的是目录以及没有扩

展名的文件。

13.看下面一个函数

```
/*640x480x256*/
void plot(int x,int y,unsigned char color)
{
  char far *video=(char far *)(0xa0000000L);
 long int offest;

  offest=(y-1)*640+x-1;
  *(video+offest)=color;
}
```

看似没有错误，但是对于 offest=(y-1)*640+x-1 这句，

虽然 offest 是长整型，但是 y 是短整型，在计算(y-1)*640 时，系统

会按短整型处理，当 y 的值取 110 以上时，（y-1)*640 的值显然会超出

短整型数字的取值范围，从而产生错误，因此应改为

offest=(long int)(y-1)*640+x-1;

14.对于一个数组 int data[5][4];

data 表示 data[0][0]的地址

data[2]表示 data[2][0]的地址

因此有人认为 data[4][2]的地址可表示为

(1).data+4*4+2;

(2).data[4]+2;

其中(2)是正确的，而(1)却是错误的.

因为 data 的基类型是 8 个字节的，

data+4 表示的就是 data[4][0]的地址

data+1 表示的就是 data[1][0]的地址

15.许多人认为转义字符'\64'中的 64 是十进制数，

但实际上这个 64 是八进制数,这个八进制数前面可以加 0 也可不

加 0，当然转义字符\后面也可以用十六进制数，但要

用'\x..'或'\X..'的形式，如'\x14'。

16.看下面这个程序

```
#include <stdio.h>
void display(int data[]，int num)
{
```

```
 int i;
 for(i=0;i<num;i++)
  printf("%d\n",data[i]);
}
void main()
{
  int da[10]={0,1,2,3,4,5,6,7,8,9};
  display(da,10);
}
```

一般认为当程序里调用这个函数时，系统会为形参 data[]开辟一个 10 个整型数字的存储空间，实际上，这个函数只为 data[]开辟一个整型指针类型的空间，使用指针指向 data[]的首地址，因此在 display(int data[],int num)这个函数中，若使用赋值语句对数组 data[]的某项赋值，如 data[0]=2,会使主程序中的数组 da[]的值改变.

17.

```
void main()
{
   unsigned char index;
   for(index=0;index<256;index++)
  {
    printf("%d\n",index);
  }
}
```

在编译时，会出现警告。

在 TC2.0 中，若改为 index<=255，编译会通过，但是运行程序时却会陷入死循环，大家不禁会产生疑问，unsigned char 类型数据的取值范围为 0-255，为什么上述程序会出现错误呢。

让我们先看下面一个程序：

```
void main()
{
  int i;
  for(i=0;i<10;i++)
  {;}
 printf("%d",i);
}
```

运行结果为 10，当 i=9 的循环完成后，i 自加 1,使得 i=10，这时不符合 i<10，才跳出循环，因此运行结果为 10;

同理，若改为 i<=10;则运行结果为 11。

通过这个程序，应该能明白为什么前一个程序会出错了。

62. 超强的指针学习笔记

发布: 2008-2-18 20:55 | 作者: 剑心通明 | 来源: 互联网 | 查看: 57 次

C 语言所有复杂的指针声明，都是由各种声明嵌套构成的。如何解读复杂指针声明呢？右左法则是一个既著名又常用的方法。不过，右左法则其实并不是 C 标准里面的内容，它是从 C 标准的声明规定中归纳出来的方法。C 标准的声明规则，是用来解决如何创建声明的，而右左法则是用来解决如何辩识一个声明的，两者可以说是相反的。右左法则的英文原文是这样说的：

The right-left rule: Start reading the **declaration** from the innermost parentheses, go right, and then go left. When you encounter parentheses, the direction should be reversed. Once everything in the parentheses has been parsed, jump out of it. Continue till the whole declaration has been parsed.

这段英文的翻译如下：

右左法则：首先从最里面的圆括号看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。

笔者要对这个法则进行一个小小的修正，应该是从未定义的标识符开始阅读，而不是从括号读起，之所以是未定义的标识符，是因为一个声明里面可能有多个标识符，但未定义的标识符只会有一个。

现在通过一些例子来讨论右左法则的应用，先从最简单的开始，逐步加深：

int (*func)(int *p);

首先找到那个未定义的标识符，就是 func，它的外面有一对圆括号，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，也是一个圆括号，这说明(*func)是一个函数，而 func 是一个指向这类函数的指针，就是一个函数指针，这类函数具有 int*类型的形参，返回值类型是 int。

int (*func)(int *p, int (*f)(int*));

func 被一对括号包含，且左边有一个 *号，说明 func 是一个指针，跳出括号，右边也有个括号，那么 func 是一个指向函数的指针，这类函数具有 int *和 int (*)(int*)这样的形参，返回值为 int 类型。再来看一看 func 的形参 int (*f)(int*)，类似前面的解释，f 也是一个函数指针，指向的函数具有 int*类型的形参，返回值为 int。

int (*func[5])(int *p);

func 右边是一个[]运算符，说明 func 是一个具有 5 个元素的数组，func 的左边有一个*，说明 func 的元素是指针，要注意这里的*不是修饰 func 的，而是修饰 func[5]的，原因是[]运算符优先级比*高，func 先跟[]结合，因此*修饰的是 func[5]。跳出这个括号，看右边，也是一对圆括号，说明 func 数组的元素是函数类型的指针，它所指向的函数具有 int*类型的形参，返回值类型为 int。

int (*(*func)[5])(int *p);

func 被一个圆括号包含，左边又有一个*，那么 func 是一个指针，跳出括号，右边是一个[]运算符号，说明 func 是一个指向数组的指针，现在往左看，左边有一个*号，说明这个数组的元素是指针，再跳出括号，右边又有一个括号，说明这个数组的元素是指向函数的指针。总结一下，就是：func 是一个指向数组的指针，这个数组的元素是函数指针，这些指针指向具有 int*形参，返回值为 int 类型的函数。

int (*(*func)(int *p))[5];

func 是一个函数指针，这类函数具有 int*类型的形参，返回值是指向数组的指针，所指向的数组的元素是具有 5 个 int 元素的数组。

要注意有些复杂指针声明是非法的，例如：

int func(void) [5];

func 是一个返回值为具有 5 个 int 元素的数组的函数。但 C 语言的函数返回值不能为数组，这是因为如果允许函数返回值为数组，那么接收这个数组的内容的东西，也必须是一个数组，但 C 语言的数组名是一个右值，它不能作为左值来接收另一个数组，因此函数返回值不能为数组。

int func[5](void);

func 是一个具有 5 个元素的数组，这个数组的元素都是函数。这也是非法的，因为数组的元素除了类型必须一样外，每个元素所占用的内存空间也必须相同，显然函数是无法达到这个要求的，即使函数的类型一样，但函数所占用的空间通常是不相同的。

作为练习，下面列几个复杂指针声明给读者自己来解析，答案放在第十章里。

int (*(*func)[5][6])[7][8];

int (*(*(*func)(int *))[5])(int *);

int (*(*func[7][8][9])(int*))[5];

实际当中，需要声明一个复杂指针时，如果把整个声明写成上面所示的形式，对程序可读性是一大损害。应该用 typedef 来对声明逐层分解，增强可读性，例如对于声明：

int (*(*func)(int *p))[5];

可以这样分解：

typedef int (*PARA)[5];

typedef PARA (*func)(int *);

这样就容易看得多了。

==========

转载述： 这是一篇比较老的关于指针的文章，作者站在初学者的角度对指针作了深入的剖析。如果你在学习指针的时候有什么问题，看一看这篇文章定有收获。

一。指针的概念

1。指针的类型

2。指针所指向的类型

3。指针的值

二。指针的算术运算

三。运算符&和*

四。指针表达式

五。数组和指针的关系

一。指针的概念

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。

要搞清一个指针需要搞清指针的四方面的内容：指针的类型，指针所指向的类型，指针的值或者叫指针所指向的内存区，还有指针本身所占据的内存区。让我们分别说明。

先声明几个指针放着做例子：

例一：

(1)int *ptr;

(2)char *ptr;

(3)int **ptr;

(4)int (*ptr)[3];

(5)int *(*ptr)[4];

如果看不懂后几个例子的话，请参阅我前段时间贴出的文章<<如何理解 c 和 c++的复杂类型声明>>。

1。 指针的类型

从语法的角度看，你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。让我们看看例一中各个指针的类型：

(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]

怎么样？找出指针的类型的方法是不是很简单？

2。指针所指向的类型

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。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]

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对 C 越来越熟悉时，你会发现，把与指针搅和在一起的“类型”这个概念分成“指针的类型”和“指针所指向的类型”两个概念，是精通指针的关键点之一。我看了不少书，发现有些写得差的书中，就把指针的这两个概念搅在一起了，所以看起书来前后矛盾，越看越糊涂。

3。 指针的值

指针的值，或者叫指针所指向的内存区或地址。 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。在 32 位程序里，所有类型的指针的值都是一个 32 位整数，因为 32 位程序里内存地址全都是 32 位长。

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。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

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？

4。 指针本身所占据的内存区。

指针本身占了多大的内存？你只要用函数 sizeof(指针的类型)测一下就知道了。在 32 位平台里，指针本身占据了 4 个字节的长度。

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。

二。指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的。例如：

例二：

```
1。 char a[20];
2。 int *ptr=a;
...
...
3。 ptr++;
```

在上例中，指针 ptr 的类型是 int*,它指向的类型是 int，它被初始化为指向整形变量 a。接下来的第 3 句中，指针 ptr 被加了 1，编译器是这样处理的：它把指针 ptr 的值加上了 sizeof(int)，在 32 位程序中，是被加上了 4。由于地址是用字节做单位的，故 ptr 所指向的地址由原来的变量 a 的地址向高地址方向增加了 4 个字节。由于 char 类型的长度是一个字节，所以，原来 ptr 是指向数组 a 的第 0 号单元开始的四个字节，此时指向了数组 a 中从第 4 号单元开始的四个字节。

我们可以用一个指针和一个循环来遍历一个数组，看例子：

例三：

```
int array[20];
int *ptr=array;
...
//此处略去为整型数组赋值的代码。
...
for(i=0;i<20;i++)
{
(*ptr)++;
ptr++;
}
```

这个例子将整型数组中各个单元的值加 1。由于每次循环都将指针 ptr 加 1，所以每次循环都能访问数组的下一个单元。

再看例子：

例四：

```
1。 char a[20];
2。 int *ptr=a;
...
...
```

3。 ptr+=5;

在这个例子中，ptr 被加上了 5，编译器是这样处理的：将指针 ptr 的值加上 5 乘 sizeof(int)，在 32 位程序中就是加上了 5 乘 4=20。由于地址的单位是字节，故现在的 ptr 所指向的地址比起加 5 后的 ptr 所指向的地址来说，向高地址方向移动了 20 个字节。在这个例子中，没加 5 前的 ptr 指向数组 a 的第 0 号单元开始的四个字节，加 5 后，ptr 已经指向了数组 a 的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。

如果上例中，ptr 是被减去 5，那么处理过程大同小异，只不过 ptr 的值是被减去 5 乘 sizeof(int)，新的 ptr 指向的地址将比原来的 ptr 所指向的地址向低地址方向移动了 20 个字节。

总结一下，一个指针 ptrold 加上一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值增加了 n 乘 sizeof(ptrold 所指向的类型)个字节。就是说， ptrnew 所指向的内存区将比 ptrold 所指向的内存区向高地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

一个指针 ptrold 减去一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值减少了 n 乘 sizeof(ptrold 所指向的类型)个字节，就是说， ptrnew 所指向的内存区将比 ptrold 所指向的内存区向低地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

三。运算符&和*

这里&是取地址运算符，*是...书上叫做“间接运算符”。&a 的运算结果是一个指针，指针的类型是 a 的类型加个*，指针所指向的类型是 a 的类型，指针所指向的地址嘛，那就是 a 的地址。*p 的运算结果就五花八门了。总之*p 的结果是 p 所指向的东西，这个东西有这些特点：它的类型是 p 指向的类型，它所占用的地址是 p 所指向的地址。

例五：

```
int a=12;
int b;
int *p;
int **ptr;
```

p=&a;//&a 的结果是一个指针，类型是 int*，指向的类型是 int，指向的地址是 a 的地址。

*p=24;//*p 的结果，在这里它的类型是 int，它所占用的地址是 p 所指向的地址，显然，*p 就是变量 a。

ptr=&p;//&p 的结果是个指针，该指针的类型是 p 的类型加个*，在这里是 int **。该指针所指向的类型是 p 的类型，这里是 int*。该指针所指向的地址就是指针 p 自己的地址。

*ptr=&b;//*ptr 是个指针，&b 的结果也是个指针，且这两个指针的类型和所指向的类型是一样的，所以用&b 来给*ptr 赋值就是毫无问题的了。

**ptr=34;//*ptr 的结果是 ptr 所指向的东西，在这里是一个指针，对这个指针再做一次*运算，结果就是一个 int 类型的变量。

四。指针表达式

一个表达式的最后结果如果是一个指针，那么这个表达式就叫指针表达式。

下面是一些指针表达式的例子：

例六：

```
int a,b;
int array[10];
int *pa;
```

pa=&a;//&a 是一个指针表达式。

int **ptr=&pa;//&pa 也是一个指针表达式。

*ptr=&b;//*ptr 和&b 都是指针表达式。

pa=array;

pa++;//这也是指针表达式。

例七：

char *arr[20];

char **parr=arr;//如果把 arr 看作指针的话，arr 也是指针表达式

char *str;

str=*parr;//*parr 是指针表达式

str=*(parr+1);//*(parr+1)是指针表达式

str=*(parr+2);//*(parr+2)是指针表达式

由于指针表达式的结果是一个指针，所以指针表达式也具有指针所具有的四个要素：指针的类型，指针所指向的类型，指针指向的内存区，指针自身占据的内存。

好了，当一个指针表达式的结果指针已经明确地具有了指针自身占据的内存的话，这个指针表达式就是一个左值，否则就不是一个左值。在例七中，&a 不是一个左值，因为它还没有占据明确的内存。*ptr 是一个左值，因为*ptr 这个指针已经占据了内存，其实*ptr 就是指针 pa，既然 pa 已经在内存中有了自己的位置，那么*ptr 当然也有了自己的位置。

五。数组和指针的关系

如果对声明数组的语句不太明白的话，请参阅我前段时间贴出的文章<<如何理解 c 和 c++的复杂类型声明>>。

数组的数组名其实可以看作一个指针。看下例：

例八：

int array[10]={0,1,2,3,4,5,6,7,8,9},value;

...

...

value=array[0];//也可写成：value=*array;

value=array[3];//也可写成：value=*(array+3);

value=array[4];//也可写成：value=*(array+4);

上例中，一般而言数组名 array 代表数组本身，类型是 int [10]，但如果把 array 看做指针的话，它指向数组的第 0 个单元，类型是 int *，所指向的类型是数组单元的类型即 int。因此*array 等于 0 就一点也不奇怪了。同理，array+3 是一个指向数组第 3 个单元的指针，所以* (array+3)等于 3。其它依此类推。

例九：

char *str[3]={

"Hello,this is a sample!",

"Hi,good morning.",

"Hello world"

};

char s[80];

strcpy(s,str[0]);//也可写成 strcpy(s,*str);

strcpy(s,str[1]);//也可写成 strcpy(s,*(str+1));

strcpy(s,str[2]);//也可写成 strcpy(s,*(str+2));

上例中，str 是一个三单元的数组，该数组的每个单元都是一个指针，这些指针各指向一个字符串。把指针数组名 str 当作一个指针的话，它指向数组的第 0 号单元，它的类型是 char**，它指向的类型是 char *。

str 也是一个指针，它的类型是 char，它所指向的类型是 char，它指向的地址是字符串"Hello,this is a sample!"的第一个字符的地址，即'H'的地址。

str+1 也是一个指针，它指向数组的第 1 号单元，它的类型是 char**，它指向的类型是 char *。

(str+1)也是一个指针，它的类型是 char，它所指向的类型是 char，它指向"Hi,good morning."的第一个字符'H'，等等。

下面总结一下数组的数组名的问题。声明了一个数组 TYPE array[n]，则数组名称 array 就有了两重含义：第一，它代表整个数组，它的类型是 TYPE [n]；第二，它是一个指针，该指针的类型是 TYPE*，该指针指向的类型是 TYPE，也就是数组单元的类型，该指针指向的内存区就是数组第 0 号单元，该指针自己占有单独的内存区，注意它和数组第 0 号单元占据的内存区是不同的。该指针的值是不能修改的，即类似 array++的表达式是错误的。

在不同的表达式中数组名 array 可以扮演不同的角色。

在表达式 sizeof(array)中，数组名 array 代表数组本身，故这时 sizeof 函数测出的是整个数组的大小。

在表达式*array 中，array 扮演的是指针，因此这个表达式的结果就是数组第 0 号单元的值。sizeof(*array)测出的是数组单元的大小。

表达式 array+n（其中 n=0，1，2，....。）中，array 扮演的是指针，故 array+n 的结果是一个指针，它的类型是 TYPE*，它指向的类型是 TYPE，它指向数组第 n 号单元。故 sizeof(array+n)测出的是指针类型的大小。

例十：

int array[10];

int (*ptr)[10];

ptr=&array;

上例中 ptr 是一个指针，它的类型是 int (*)[10]，他指向的类型是 int [10]，我们用整个数组的首地址来初始化它。在语句 ptr=&array 中，array 代表数组本身。

本节中提到了函数 sizeof()，那么我来问一问，sizeof(指针名称)测出的究竟是指针自身类型的大小呢还是指针所指向的类型的大小？答案是前者。例如：

int (*ptr)[10];

则在 32 位程序中，有：

sizeof(int(*)[10])==4

sizeof(int [10])==40

sizeof(ptr)==4

实际上，sizeof(对象)测出的都是对象自身的类型的大小，而不是别的什么类型的大小。

六。指针和结构类型的关系

七。指针和函数的关系

八。指针类型转换

九。指针的安全问题

十、指针与链表问题

六。指针和结构类型的关系

可以声明一个指向结构类型对象的指针。

例十一：

```
struct MyStruct
{
int a;
int b;
int c;
}
```

MyStruct ss={20,30,40};//声明了结构对象 ss，并把 ss 的三个成员初始化为 20，30 和 40。

MyStruct *ptr=&ss;//声明了一个指向结构对象 ss 的指针。它的类型是 MyStruct*,它指向的类型是 MyStruct。

int *pstr=(int*)&ss;//声明了一个指向结构对象 ss 的指针。但是它的类型和它指向的类型和 ptr 是不同的。

请问怎样通过指针 ptr 来访问 ss 的三个成员变量？

答案：

```
ptr->a;
ptr->b;
ptr->c;
```

又请问怎样通过指针 pstr 来访问 ss 的三个成员变量？

答案：

```
*pstr；//访问了 ss 的成员 a。
*(pstr+1);//访问了 ss 的成员 b。
*(pstr+2)//访问了 ss 的成员 c。
```

呵呵，虽然我在我的 MSVC++6.0 上调式过上述代码，但是要知道，这样使用 pstr 来访问结构成员是不正规的，为了说明为什么不正规，让我们看看怎样通过指针来访问数组的各个单元：

例十二：

```
int array[3]={35,56,37};
int *pa=array;
```

通过指针 pa 访问数组 array 的三个单元的方法是：

```
*pa;//访问了第 0 号单元
*(pa+1);//访问了第 1 号单元
*(pa+2);//访问了第 2 号单元
```

从格式上看倒是与通过指针访问结构成员的不正规方法的格式一样。所有的 C/C++编译器在排列数组的单元时，总是把各个数组单元存放在连续的存储区里，单元和单元之间没有空隙。但在存放结构对象的各个成员时，在某种编译环境下，可能会需要字对齐或双字对齐或者是别的什么对齐，需要在相邻两个成员之间加若干个“填充字节”，这就导致各个成员之间可能会有若干个字节的空隙。

所以，在例十二中，即使*pstr 访问到了结构对象 ss 的第一个成员变量 a，也不能保证*(pstr+1)就一定能访问到结构成员 b。因为成员 a 和成员 b 之间可能会有若干填充字节，说不定*(pstr+1)就正好访问到了这些填充字节呢。这也证明了指针的灵活性。要是你的目的就是想看看各个结构成员之间到底有没有填充字节，

嘿，这倒是个不错的方法。

通过指针访问结构成员的正确方法应该是象例十二中使用指针 ptr 的方法。

七。指针和函数的关系

可以把一个指针声明成为一个指向函数的指针。

```
int fun1(char*,int);
int (*pfun1)(char*,int);
pfun1=fun1;
....
....
int a=(*pfun1)("abcdefg",7);//通过函数指针调用函数。
```

可以把指针作为函数的形参。在函数调用语句中，可以用指针表达式来作为实参。

例十三：

```
int fun(char*);
int a;
char str[]="abcdefghijklmn";
a=fun(str);
...
...
int fun(char*s)
{
int num=0;
for(int i=0;i {
num+=*s;s++;
}
return num;
}
```

这个例子中的函数 fun 统计一个字符串中各个字符的 ASCII 码值之和。前面说了，数组的名字也是一个指针。在函数调用中，当把 str 作为实参传递给形参 s 后，实际是把 str 的值传递给了 s，s 所指向的地址就和 str 所指向的地址一致，但是 str 和 s 各自占用各自的存储空间。在函数体内对 s 进行自加 1 运算，并不意味着同时对 str 进行了自加 1 运算。

八。指针类型转换

当我们初始化一个指针或给一个指针赋值时，赋值号的左边是一个指针，赋值号的右边是一个

指针表达式。在我们前面所举的例子中，绝大多数情况下，指针的类型和指针表达式的类型是一样的，指针所指向的类型和指针表达式所指向的类型是一样的。

例十四：

```
1。 float f=12.3;
2。 float *fptr=&f;
3。 int *p;
```

在上面的例子中，假如我们想让指针 p 指向实数 f，应该怎么搞？是用下面的语句吗？

```
p=&f;
```

不对。因为指针 p 的类型是 int*，它指向的类型是 int。表达式&f 的结果是一个指针，指针的类型是 float*,它指向的类型是 float。两者不一致，直接赋值的方法是不行的。至少在我的 MSVC++6.0 上，对指针的赋值语句要求赋值号两边的类型一致，所指向的类型也一致，其它的编译器上我没试过，大家可以试试。为了实现我们的目的，需要进行“强制类型转换”：

```
p=(int*)&f;
```

如果有一个指针 p，我们需要把它的类型和所指向的类型改为 TYEP*和 TYPE，那么语法格式是：

```
(TYPE*)p;
```

这样强制类型转换的结果是一个新指针，该新指针的类型是 TYPE*，它指向的类型是 TYPE，它指向的地址就是原指针指向的地址。而原来的指针 p 的一切属性都没有被修改。

一个函数如果使用了指针作为形参，那么在函数调用语句的实参和形参的结合过程中，也会发生指针类型的转换。 例十五：

```
void fun(char*);
int a=125,b;
fun((char*)&a);
...
...
void fun(char*s)
{
char c;
c=*(s+3);*(s+3)=*(s+0);*(s+0)=c;
c=*(s+2);*(s+2)=*(s+1);*(s+1)=c;
}
}
```

注意这是一个 32 位程序，故 int 类型占了四个字节，char 类型占一个字节。函数 fun 的作用是把一个整数的四个字节的顺序来个颠倒。注意到了吗？在函数调用语句中，实参&a 的结果是一个指针，它的类型是 int *，它指向的类型是 int。形参这个指针的类型是 char*，它指向的类型是 char。这样，在实参和形参的结合过程中，我们必须进行一次从 int*类型到 char*类型的转换。结合这个例子，我们可以这样来想象编译器进行转换的过程：编译器先构造一个临时指针 char*temp，然后执行 temp=(char*)&a，最后再把 temp 的值传递给 s。所以最后的结果是：s 的类型是 char*,它指向的类型是 char，它指向的地址就是 a 的首地址。

我们已经知道，指针的值就是指针指向的地址，在 32 位程序中，指针的值其实是一个 32 位整数。那可不可以把一个整数当作指针的值直接赋给指针呢？就象下面的语句：

```
unsigned int a;
```

TYPE *ptr;//TYPE 是 int，char 或结构类型等等类型。

...

...

a=20345686;

ptr=20345686;//我们的目的是要使指针 ptr 指向地址 20345686（十进制）

ptr=a;//我们的目的是要使指针 ptr 指向地址 20345686（十进制）

编译一下吧。结果发现后面两条语句全是错的。那么我们的目的就不能达到了吗？不，还有办法：

unsigned int a;

TYPE *ptr;//TYPE 是 int，char 或结构类型等等类型。

...

...

a=某个数，这个数必须代表一个合法的地址；

ptr=(TYPE*)a；//呵呵，这就可以了。

严格说来这里的(TYPE*)和指针类型转换中的(TYPE*)还不一样。这里的(TYPE*)的意思是把无符号整数 a 的值当作一个地址来看待。上面强调了 a 的值必须代表一个合法的地址，否则的话，在你使用 ptr 的时候，就会出现非法操作错误。

想想能不能反过来，把指针指向的地址即指针的值当作一个整数取出来。完全可以。下面的例子演示了把一个指针的值当作一个整数取出来，然后再把这个整数当作一个地址赋给一个指针：

例十六：

int a=123,b;

int *ptr=&a;

char *str;

b=(int)ptr;//把指针 ptr 的值当作一个整数取出来。

str=(char*)b;//把这个整数的值当作一个地址赋给指针 str。

好了，现在我们已经知道了，可以把指针的值当作一个整数取出来，也可以把一个整数值当作地址赋给一个指针。

九。指针的安全问题

看下面的例子：

例十七：

char s='a';

int *ptr;

ptr=(int*)&s;

*ptr=1298;

指针 ptr 是一个 int*类型的指针，它指向的类型是 int。它指向的地址就是 s 的首地址。在 32 位程序中，s 占一个字节，int 类型占四个字节。最后一条语句不但改变了 s 所占的一个字节，还把和 s 相临的高地址方向的三个字节也改变了。这三个字节是干什么的？只有编译程序知道，而写程序的人是不太可能知道的。也许这三个字节里存储了非常重要的数据，也许这三个字节里正好是程序的一条代码，而由于你对指针的马虎应用，这三个字节的值被改变了！这会造成崩溃性的错误。

让我们来看一例：

例十八：

```
1。 char a;
2。 int *ptr=&a;
...
...
3。 ptr++;
4。 *ptr=115;
```

该例子完全可以通过编译，并能执行。但是看到没有？第 3 句对指针 ptr 进行自加 1 运算后，ptr 指向了和整形变量 a 相邻的高地址方向的一块存储区。这块存储区里是什么？我们不知道。有可能它是一个非常重要的数据，甚至可能是一条代码。而第 4 句竟然往这片存储区里写入一个数据！这是严重的错误。所以在使用指针时，程序员心里必须非常清楚：我的指针究竟指向了哪里。在用指针访问数组的时候，也要注意不要超出数组的低端和高端界限，否则也会造成类似的错误。

在指针的强制类型转换：ptr1=(TYPE*)ptr2 中，如果 sizeof(ptr2 的类型)大于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是安全的。如果 sizeof(ptr2 的类型)小于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是不安全的。至于为什么，读者结合例十七来想一想，应该会明白的。

十、指针与链表问题

红色部分所示的程序语句有问题，改正后的程序在下面。

list1.c

```
#include
#include
struct listNode{
    int data;
     struct listNode *nextPtr;
};
typedef struct listNode LISTNODE;
typedef LISTNODE * LISTNODEPTR;
void list(LISTNODEPTR *, int);
void printlist(LISTNODEPTR);
main()
{
    LISTNODEPTR newPtr=NULL;
    int i,a;
    for(i=0;i<3;i++){
       printf("please enter a number\n");
       scanf("%d,",&a);
       list(&newPtr,a);
       // 此处给的是 newPtr 的地址， 注意！
```

```
        }
        printlist(newPtr);
    free(newPtr);
    // 链表的释放不能这样写，这样，只释放了 newPtr 指向的一个节点。
    // 可以先找到链表的尾，然后反向释放；或者，利用 printlist 的顺序释放，
    // 改函数 printlist，或在此函数里释放。
    return 0;
}
void list(LISTNODEPTR *sPtr, int a)
{
    LISTNODEPTR newPtr,currentPtr;
    newPtr=malloc(sizeof(LISTNODEPTR));
    // 此处错， LISTNODEPTR 是指针类型，不是结构类型，
    // malloc 返回 void 指针，应该强制转换类型，此处会告警不报错，但应有良好的编程风格与习惯。
    if(newPtr!=NULL){
      newPtr->data=a;
      newPtr->nextPtr=NULL;
       currentPtr=*sPtr;
    }
    if(currentPtr==NULL){
    // 此处条件不确切，因为 currentPtr 没有初始化，
    // 如 newPtr 一旦为 NULL，此句及以下就有问题。
    newPtr->nextPtr=*sPtr;
    *sPtr=newPtr;}
    // 在第一个数来的时候，main 里的 newPtr 通过 sPtr 被修改指向此节点。
    // 在第二个数来的时候，main 里的 newPtr 通过 sPtr 被修改指向此节点。
    // 在第三个数来的时候，main 里的 newPtr 通过 sPtr 被修改指向此节点。
    // 最后，main 里的 newPtr 指向第三个数。
}
void printlist(LISTNODEPTR currentPtr)
{
    if(currentPtr==NULL)
      printf("The list is empty\n");
    else{
      printf("This list is :\n");
     while(currentPtr!=NULL){
```

```
            printf("%d-->",currentPtr->data);
            // main 里的 newPtr 指向第三个数。你先打印了最后一个数。
            // currentPtr=currentPtr->nextPtr->data;
            // 此句非法， 类型不同， 有可能让你只循环一次，如 data 为 0。
        }
        printf("NULL\n\n");
    }
}
    // 对类似程序能运行，但结果似是而非的情况，应该多利用跟踪调试，看变量的变化。

改正后的正确程序
#include
#include
struct listNode{
    int data;
    struct listNode *nextPtr;
};
typedef struct listNode LISTNODE;
typedef LISTNODE * LISTNODEPTR;
LISTNODEPTR list(LISTNODEPTR , int); // 此处不同
void printlist(LISTNODEPTR);
void freelist(LISTNODEPTR); // 增加
main()
{
    LISTNODEPTR newPtr=NULL;
    int i,a;
    for(i=0;i<3;i++){
        printf("please enter a number\n");
        scanf("%d,",&a);
        newPtr = list(newPtr,a); // 此处注意
    }
    printlist(newPtr);
    freelist(newPtr); // 此处
    return 0;
}
LISTNODEPTR list(LISTNODEPTR sPtr, int a)
{
```

```
    if ( sPtr != NULL )
       sPtr->nextPtr = list( sPtr->nextPtr, a ); // 递归，向后面的节点上加数据。
    else
    {
       sPtr =(LISTNODEPTR) malloc(sizeof(LISTNODE)); // 注意，是节点的尺寸,类型转换
       sPtr->nextPtr = NULL;
       sPtr->data = a;
    }
    return sPtr;
}
void freelist(LISTNODEPTR sPtr )
{
    if ( sPtr != NULL )
    {
       freelist( sPtr->nextPtr ); // 递归， 先释放后面的节点
       free( sPtr ); // 再释放本节点
    }
    else //
    return ; // 此两行可不要
}
void printlist(LISTNODEPTR currentPtr)
{
    if(currentPtr==NULL)
       printf("The list is empty\n");
    else
    {
       printf("This list is :\n");
       while(currentPtr!=NULL)
       {
          printf("%d-->",currentPtr->data);
          currentPtr=currentPtr->nextPtr; // 这里不一样
       }
       printf("NULL\n\n");
    }
}
```

63. 程序员之路——关于代码风格

发布: 2008-2-18 20:56 | 作者: 剑心通明 | 来源: 互联网 | 查看: 44 次

优秀的代码风格如同一身得体的打扮，能够给人以良好的印象。初学程序设计，首先必须建立良好的编程习惯，这其中就包括代码风格。本文就代码风格中的几个重点问题进行了讨论，并在文后给出了一份优秀的代码作为风格模板。代码风格不必花费太多专门的时间研究，在使用中不断模仿模板代码，轻轻松松就能写出“专业的代码”。

一、80 字符，代码行极限

无论时空怎么转变，世界怎样改变，一行 80 字符应始终铭记心间。古老的 Unix 终端以 80 列的格式显示文本，为了让源代码与手册具有最佳的可读性， Unix 系统始终坚持着 80 列的传统。80 列不多不少，足够写出一行有意义的代码，同时也足够显示在终端屏幕，足够打印在 A4 纸上。虽然时至今日，我们的屏幕分辨率早已足够高，一行能够显示的内容远超超过 80 字符，但我们的优秀传统已经形成——几乎所有的 Unix/Linux 内核源代码以及联机用户手册都严格地遵守着 80 列极限。如果你正好在使用 Windows 平台下的 Dev C++，你是否有注意到代码编辑框里那条细细的灰色竖线？不错，那正是代码行极限。除了 HTML、XML 等冗长繁复的标记式语言，几乎所有的语言都需要严格遵守代码行极限，这包括 C、C++、Java、C#、Python、PHP 等等。不过有时，比如当 PHP 跟 HTML 打交道的时候，这个限制是可以暂时放松的。过长的代码行总是不好的，好的代码要始终保持苗条的身材。

二、Tab 还是 Space，众说纷纭的缩进方式

代码离不开缩进，关于缩进主要有两个争论，一个是该用空格（Space）还是用制表符（Tab），另外一个是该用 4 格缩进还是 8 格缩进甚至都不是。

先来谈谈 Space 与 Tab 的问题。坚持用 Space 的程序员会告诉你，如果你从来都不用 Tab，那么你的代码放到所有的地方看都是一样的。没错，这是用 Space 缩进的优点，可惜的是，这是它唯一的优点。代码层次越多，内层代码最前面的缩进便越多，这意味着你需要敲很多很多次空格。即使你能忍受不厌其烦地按空格键直到它坏掉，你也一定会被 IDE 总是自作聪明地插入一些 Tab 字符的行为烦恼不已。建议总是使用 Tab 缩进，因为几乎所有的代码（不仅仅是 C 代码）都在使用 Tab 缩进。

Tab 到底是 4 格还是 8 格？这是 Tab 缩进会被某些人诟病的根源。当你写程序时使用的 Tab 大小与别人读程序时使用的 Tab 大小不同时，再漂亮的排版也会变得杂乱无章。标准的 Tab 是 8 格的，而不幸的是，几乎所有的 Windows 平台下的 IDE，包括 Visual Studio、Dev C++，甚至跨平台的 Eclipse 等，都默认使用 4 格 Tab。我使用的 **FreeBSD** 系统的所有的内核源代码都采用 8 格缩进，所以我一直坚持使用 8 格缩进。也许你不习惯太大的间距，如果不是在 Unix 平台下，或者不是 C 语言，那就采用 4 格 Tab 吧。如果你在 Unix 下编写 C 代码，使用 8 格的标准 Tab 是更好的习惯。

三、折行原则，容易被忽略的角落

既然有代码行极限，很多情况下我们不得不断开一个完整的代码行，这就带来了一个问题：折行后应该如何缩进？好的做法是，第一次折行后，在原来缩进的基础上增加 1/2 的 Tab 大小的空格，之后的折行全部对齐第二行。可能这样的文字描述过于晦涩了，还是举个例子罢（以 8 格缩进为例）：

```
if (value > a && value > b && value > c && value < d && value < e && value < f
    value < h && value < h) { /* 注意折行后的缩进 */
value = value + 1;
            value = value * value * value * value * value * value * value * value
              * value * value + value * value * value * value * value * value
              * value * value; /* 注意再次折行后的缩进 */
```

}

显然这个段代码没有任何实际用处，只是为了说明折行缩进而编造的。

四、无处不在的空格，无处不在的空行

需要空格的位置有：

1）if、while、switch 等关键字与之后的左括号(之间。

2）左花括号{之前。

3）双目运算符两侧，例如 p == NULL。

4）逗号,与分号;之后，例如 for (i = 0; i < 10; i++)。

不要空格的位置有：

1）函数名与之后的左括号(，包括带参数的宏与之后的左括号(，例如 max(a, b)。

2）分号;与冒号:之前。

3）左括号(右边，右括号)左边，例如 if (p == NULL)。

需要空行的位置有：

1）函数的定义之前、函数的定义之后

2）一组联系紧密的代码段之前和之后

这些规则并不完全，当你碰到上面没有列举出来的情况时，请参考本文提供的模板代码。

五、左花括号的争议——换行乎？不换乎？

这又是一个仁者见仁智者见智的问题了。从使代码更清晰的角度看，作为代码段开头标识的左花括号{应该另起一行：

```
if (p == NULL)
{
printf("error!\n");
exit(0);
}
```

可是，这看起来实在不够紧凑，所以大部分的 C 代码（至少 Unix 上如此）都采用了这样的方式：

```
if (p == NULL) {
printf("error!\n");
exit(0);
}
```

我的建议是采用后者，这会使你的代码显得更加紧凑，也更加专业。需要说明一个特例，在定义函数时，我们总是要给左花括号{换行：

```
static int
maxint(int a, int b)
{
return a > b ? a : b;
```

}

六、坚持美观、灵活对待

代码风格远远不止上面提到的五点，事实上，很多公司都有规定的代码风格，包括命名规则、缩进规则等。如果你在一个开发团队中，应该始终以团队的标准为主，而如果你只是在学习 C 语言并试着形成自己的代码风格，我将在这里给一份最专业的源代码供你参考。你的任何疑问，在这份代码里一定可以找到答案：

文件:	Code Style.zip
大小:	10KB
下载:	下载

（注意查看时将 Tab 设置为 8 格）。这份代码来自优秀的 FreeBSD 操作系统内核源代码（版本 6.2），你一定不会置疑它的权威性。更多内容请参考 FreeBSD 内核代码编写规范，你也许需要一些耐心才能读完这篇英文文档。

64. 指针、结构体、联合体的安全规范

指针赋予了 C 编程最大的灵活性；结构体使得 C 程序整齐而紧凑；联合体在某些要求注重效率的场合有精彩的表现，这三个要素是 C 语言的精华。

然而，精华并不意味着完美，C 语言在赋予程序员足够灵活性的同时，也给了程序员很多犯错误的机会。所以有必要关注指针、结构体和联合体的实现细节，从而保障程序的安全性。

在此．第一部分介绍《MISRA—C：2004》中与指针相关的部分规则，第二部分讲解结构体和联合体的操作规范。下文中凡是未加特殊说明的都是强制(required)规则，个别推荐(advisory)规则加了“推荐”标示。

1 指针的安全规范

《MISRA—C：2004》关于指针的规范主要分为三个部分：指针的类型转换规则、指针运算的规则和指针的有效性规则。

1.1 指针的类型转换

指针类型转换是个高风险的操作，所以应该尽量避免进行这个操作。MISRA—C 对其中可能造成严重错误的情况作了严格的限定，选择其中两条作简要分析。

规则 11.4(推荐)：指向不同数据类型的指针之间不能相互转换。

思考如下程序：

```
uint8_t*pl;
uint32)_t*p2;
p2=(uint32_t*)pl;
/ *注：uint8_t 表示 8 位无符号整型，uint3_t 表示 32 位无符号整型。* /
```

程序员希望将从 p1 单元开始的 4 个字节组成一个 32 付的整型来参与运算。

如果 CPU 允许各种数据对象存放在任意的存储单元，则以上转换没有问题。但某些 CPU 对某种(些)数据类型加强了对齐限制，要求这些数据对象占用一定的地址空间，比如某些字节寻址的 CPU 会要求 32 位(4 字节)整型存放在 4 的整倍数地址上。在这个前提下．思考程序中的指针转换：假设 pl 一开始指向的是 0x00O3 单元(对 uint8_t 型的整型没有对齐要求)，则执行最后一行强制转换后，p2 到底指向哪个单元就无法预料了。

规则 1 1.5：指针转换过程中不允许丢失指针的 const、volatile 属性。按如下定义指针：

```
uIntl6 一 t     x;
uint16_t*const cpi=&x;    / *const 指针* /
uintl6_t*const *pcpi;     / *指向 const 指针的指针* /
const uintl6_t* *ppci;     / *指向 const 整型指针的指针* /
```

uIntl6_t*　　*ppi ;

const uint16_t　*pci;　　/ *指向 const 整型的指针* /

volatik uint16_t　　*pvi;　　/ *指向 volatile 整型的指针* /

uintl6_t　　*pi;

则以下指针转换是允许的：

pl=cpi;

以下指针转换是不允许的：

pi=(umtl6_t*)pci;

pi=(uintl6_t*)pvil

ppi=(uintl6_t* *)pcpi;

ppi=(uintl6_I**)ppci+

以上非法指针类型转换将会丢失 const 或者 volatile 类型。丢失 const 属性，将有可能导致在对只读内容进行写操作时，编译器不会发出警告，编译器将不对具有 volatile 属性的变量作优化；丢失 volatile 属性，编译器的优化可能导致程序员预先设计的硬件时序操作失效，这样的错误很难发现。关于 const 和 volatile 关键字的详细作用，读者可参考 ISOC 获取更多信息。

1．2 指针的运算

ISOC 标准中，对指向数组成员的指针运算(包括算术运算、比较等)做了规范定义，除此以外的指针运算属于未定义(undefined)范围，具体实现有赖于具体编译器，其安全性无法得到保障，MISRA—C 中对指针运算的合法范围做了如下限定。

规则 17.1：只有指向数组的指针才允许进行算术运算①。

规则 17 2：只有指向同一个数组的两个指针才允许相减②。

规则 17 3：只有指向同一个数组的两个指针才允许用>，>=，<，<=等关系运算符进行比较。

为了尽最大可能减少直接进行指针运算带来的隐患，尤其是程序动态运行时可能发生的数组越界等问题，MISRA—C 对指针运算作了更为严格的规定。规则 17 4：只允许用数组索引做指针运算。按如下方式定义数组和指针：

uint8_t　　a[10];

uint8_t　　*p;

则*(p+5)=O 是不允许的．而 p[5]=O 则是允许的，尽管就这段程序而言，二者等价。

以下给出一段程序，读者可参照相应程序行的注释，细细品味上述规则的含义。

void my_fn(uInt*_t*p1．uint8_t p2[]){

①其实此处的算术运算仅限定于指针加减某个整数．比如 ppoint=point 一 5．ppoint++等。0 两

个指针可指向不同的散组成员。

```
uint8_t index=0；
 uint8_t     *p3
uint8_t     *p4；
 *pl=O；
 p1++；       / *不允许，pl 不是指向数组的指针* /
 p1=p1+5; / *不允许，pl 不是指向数组的指针* /
 pl[5]=O；    / *不允许，p1 不是指向数组的指针* /
 p3=&p1[5]; / *不允许，pl 不是指向数组的指针* /
 p2[0]=O；
 index++;
index=index+5：
 p2[index]=0；       / *允许* /
 *(p2+index)=O；    / *不允许* /
 p4=&p2[5]；        / *允许* /
 }
```

1．3 指针的有效性

下面介绍《MISRA—C：2004》中关于指针有效性的规则。

规则 17 6：动态分配对象的地址不允许在本对象消亡后传给另外一个对象。

这条规则的实际意义是不允许将栈对象的地址传给外部作用域的对象。

请看以下这段程序：

```
#include″stdi0．h″
char*getm(void){
char p[]=″hello world″；
return p；
intmain(){
    char* str=NULL;
    str=getm();
    printf(str)；
```

程序员希望最后的输出结果是″hello world″这个字符串，然而实际运行时，却出现乱码(具体内容依赖于编译环境)。

简单分析一下，由于 chat p[]=″hell0 world″这条语句是在栈中分配空间存储″hell0 world″这个字

符串，当函数 getm()返回的时候，已分配的空间将会被释放(但内容并不会被销毁)，而 priM(str)涉及系统调用，有数据压栈，会修改从前分配给数组 p[]存储空间的内容，导致程序无法得到预期的效果。

倘若将 getm()函数体中的 char p[]=″hell0 world″程序行改成 char*q=″hello world″，则执行 main( )的时候可以正确输出″hello world″，这是由于 q 指向的是静态数据区，而非栈中的某个单元。

所以，数组名是指针不假，但在实现细节上还是有很大的差异，程序员在使用指针的时候必须慎之又慎。

2 结构体、联合体的安全规范

规则 18 4：不允许使用联合体。这是一个不太近情理的规定，在具体阐述为何《MIS—RA—C：2004》如此“痛恨”联合体之前，首先需要明确与联合体相关的细节：

①联合体的末尾有多少个填充单元?

②联合体中的各个成员如何对齐?

③多字节的数据类型高低字节如何排放顺序?

④如果包含位字段(bit—field)，各位如何排放顺序?

针对细节 3 举个例子。

程序段 2．1

```
typedef union{
    uilat32_t word;
    uint8_t bytes[4];
}word_msg_t;
unit32_t read_nasg(void){
 word_rnsg_t tmp;
/ *注：tmn bvte[O]对府干 tmp．word 的高 8 位，tmp byter[l]对应于
tmp.WOfO 的次高 8 位，依次类推。* /
tmp．bytes[O]=read_byte();
tmp．bytes[1]=read_byte();
tmp．bytes[2]=read_byte();
tmp．bytes[3]=read_byte();
retlarn(trap．word);
}
```

以上代码格式在各种通信协议中使用的频率很高，接收端接收到的数据一般都以字节为单位存放，主控程序需要根据相应的协议将接收到的多个字节进行组合。为了实现相同的功能，《MISRA-C：2004》推荐了 read_msg()函数的另外一种写法。

程序段 2．2

```
uint32_trcad_msg(void){
```

```
uint32_t word;
Word=((unit32_t)read_byte())<<24;
word=word|(((unit32_t)read_byte())<<16);
word=word|(((unit32_t)read_byted_byte<<8);
word=word|(((unit32_t)read_byte());
return(word);
}
```

无论从程序的清晰程度还是执行效率来讲，程序段 2．1 都优于程序段 2．2。然而，程序段 2．1 在 Intel 80x86／Pentlurn 体系(1ittle—endian，存储多字节整数的时候低字节存放在低地址，高字节存放在高地址)CPU 中和在 Mo—torola 68K 体系(big—endian，存储多字节整数的时候高字节存放在低地址，低字节存放在高地址)cPu 中的执行结果完全不一样。假设 read_byte()函数返回的数据依次是 0x01、0x02、0x03 和 0x04，则在 Intel 体系中，程序段 2．1

中 read_msg()函数的返回值是 0x432l；在 Motorola 体系中，read_msg()的返回值是 0x1234。

无论在 Intel 体系还是 Motorola 体系中，程序段 2．2 中 read_msg()的返回值都是 0x1 234。

以上是联合体中多字节整型字节排放顺序不定导致漏洞的一个例子。倘若不明确联合体末尾填充的细节，或者不清楚联合体成员的对齐方式，或者不注意联合体中位字段成员的位排列次序，都有可能导致错误。作为将安全性放在第一位的 C 标准，MlSRA—C 禁止使用联合体并非不可理喻。

然而，联合体毕竟是 C 语言的一个重要元素，所以 MISRA—C 主张禁止使用联合体的同时，也为效率和资源要求比较苛刻的情况开了一扇门，程序员在明确联合体各个实现细节的前提下，在万不得已的时候，仍可谨慎使用联合体，在不同体系的 CPU 间移植程序的时候要注意做相应的修改。

65. C 指针讲解

发布: 2008-2-18 21:07 | 作者: 剑心通明 | 来源: 互联网 | 查看: 49 次

第一章。指针的概念

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。要搞清一个指针需要搞清指针的四方面的内容：指针的类型，指针所指向的类型，指针的值或者叫指针所指向的内存区，还有指针本身所占据的内存区。让我们分别说明。

先声明几个指针放着做例子：

例一：

(1)int *ptr;

(2)char *ptr;

(3)int **ptr;

(4)int (*ptr)[3];

(5)int *(*ptr)[4];

如果看不懂后几个例子的话，请参阅我前段时间贴出的文?lt;<如何理解 c 和 c++的复杂类型声明>>。

1。 指针的类型。

从语法的角度看，你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。让我们看看例一中各个指针的类型：

(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]

怎么样？找出指针的类型的方法是不是很简单？

2。指针所指向的类型。

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。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]

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对 C 越来越熟悉时，你会发现，把与指针搅和在一起的"类型"这个概念分成"指针的类型"和"指针所指向的类型"两个概念，是精通指针的关键点之一。我看了不少书，发现有些写得差的书中，就把指针的这两个概念搅在一起了，所以看起书来前后矛盾，越看越糊涂。

3。 指针的值，或者叫指针所指向的内存区或地址。

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。在 32 位程序里，所有类型的指针的值都是一个 32 位整数，因为 32 位程序里内存地址全都是 32 位长。

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。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

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？

4。 指针本身所占据的内存区。

指针本身占了多大的内存？你只要用函数 sizeof(指针的类型)测一下就知道了。在 32 位平台里，指针本身占据了 4 个字节的长度。

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。

第二章。指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的。例如：

例二：

```
1。 char a[20];
2。 int *ptr=a;
...
...
3。 ptr++;
```

在上例中，指针 ptr 的类型是 int*,它指向的类型是 int，它被初始化为指向整形变量 a。接下来的第 3 句中，指针 ptr 被加了 1，编译器是这样处理的：它把指针 ptr 的值加上了 sizeof(int)，在 32 位程序中，是被加上了 4。由于地址是用字节做单位的，故 ptr 所指向的地址由原来的变量 a 的地址向高地址方向增加了 4 个字节。

由于 char 类型的长度是一个字节，所以，原来 ptr 是指向数组 a 的第 0 号单元开始的四个字节，此时指向了数组 a 中从第 4 号单元开始的四个字节。

我们可以用一个指针和一个循环来遍历一个数组，看例子：

例三：

```
int array[20];
int *ptr=array;
...
//此处略去为整型数组赋值的代码。
...
for(i=0;i<20;i++)
{
(*ptr)++;
ptr++;
}
```

这个例子将整型数组中各个单元的值加 1。由于每次循环都将指针 ptr 加 1，所以每次循环都能访问数组的下一个单元。再看例子：

例四：

```
1。 char a[20];
2。 int *ptr=a;
...
...
3。 ptr+=5;
```

在这个例子中，ptr 被加上了 5，编译器是这样处理的：将指针 ptr 的值加上 5 乘 sizeof(int)，在 32 位程序中就是加上了 5 乘 4=20。由于地址的单位是字节，故现在的 ptr 所指向的地址比起加 5 后的 ptr 所指向的地址来说，向高地址方向移动了 20 个字节。在这个例子中，没加 5 前的 ptr 指向数组 a 的第 0 号单元开始的四个字节，加 5 后，ptr 已经指向了数组 a 的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。

如果上例中，ptr 是被减去 5，那么处理过程大同小异，只不过 ptr 的值是被减去 5 乘 sizeof(int)，新的 ptr 指向的地址将比原来的 ptr 所指向的地址向低地址方向移动了 20 个字节。

总结一下，一个指针 ptrold 加上一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值增加了 n 乘 sizeof(ptrold 所指向的类型)个字节。就是说，ptrnew 所指向的内存区将比 ptrold 所指向的内存区向高地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

a 一个指针 ptrold 减去一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值减少了 n 乘 sizeof(ptrold 所指向的类型)个字节，就是说，ptrnew 所指向的内存区将比 ptrold 所指向的内存区向低地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。

第三章。运算?amp;和*

这里&是取地址运算符，*是...书上叫做"间接运算符"。

&a 的运算结果是一个指针，指针的类型是 a 的类型加个*，指针所指向的类型是 a 的类型，指针所指向的地址嘛，那就是 a 的地址。

*p 的运算结果就五花八门了。总之*p 的结果是 p 所指向的东西，这个东西有这些特点：它的类型是 p 指向的类型，它所占用的地址是 p 所指向的地址。

例五：

int a=12;

int b;

int *p;

int **ptr;

p=&a;//&a 的结果是一个指针，类型是 int*，指向的类型是 int，指向的地址是 a 的地址。

*p=24;//*p 的结果，在这里它的类型是 int，它所占用的地址是 p 所指向的地址，显然，*p 就是变量 a。

ptr=&p;//&p 的结果是个指针，该指针的类型是 p 的类型加个*，在这里是 int**。该指针所指向的类型是 p 的类型，这里是 int*。该指针所指向的地址就是指针 p 自己的地址。

*ptr=&b;//*ptr 是个指针，&b 的结果也是个指针，且这两个指针的类型和所指向的类型是一样的，所以用&b 来给*ptr 赋值就是毫无问题的了。

**ptr=34;//*ptr 的结果是 ptr 所指向的东西，在这里是一个指针，对这个指针再做一次*运算，结果就是一个 int 类型的变量。

第四章。指针表达式。

一个表达式的最后结果如果是一个指针，那么这个表达式就叫指针表达式。

下面是一些指针表达式的例子：

例六：

int a,b;

int array[10];

int *pa;

pa=&a;//&a 是一个指针表达式。

int **ptr=&pa;//&pa 也是一个指针表达式。

*ptr=&b;//*ptr 和&b 都是指针表达式。

pa=array;

pa++;//这也是指针表达式。

例七：

char *arr[20];

char **parr=arr;//如果把 arr 看作指针的话，arr 也是指针表达式

char *str;

str=*parr;//*parr 是指针表达式

str=*(parr+1);//*(parr+1)是指针表达式

str=*(parr+2);//*(parr+2)是指针表达式

由于指针表达式的结果是一个指针，所以指针表达式也具有指针所具有的四个要素：指针的类型，指针所指向的类型，指针指向的内存区，指针自身占据的内存。

好了，当一个指针表达式的结果指针已经明确地具有了指针自身占据的内存的话，这个指针表达式就是一个左值，否则就不是一个左值。

在例七中，&a 不是一个左值，因为它还没有占据明确的内存。*ptr 是一个左值，因为*ptr 这个指针已经占据了内存，其实*ptr 就是指针 pa，既然 pa 已经在内存中有了自己的位置，那么*ptr 当然也有了自己的位置。

第五章。数组和指针的关系

如果对声明数组的语句不太明白的话，请参阅我前段时间贴出的文?lt;<如何理解 c 和 c++的复杂类型声明>>。

数组的数组名其实可以看作一个指针。看下例：

例八：

int array[10]={0,1,2,3,4,5,6,7,8,9},value;

...

...

value=array[0];//也可写成：value=*array;

value=array[3];//也可写成：value=*(array+3);

value=array[4];//也可写成：value=*(array+4);

上例中，一般而言数组名 array 代表数组本身，类型是 int [10]，但如果把 array 看做指针的话，它指向数组的第 0 个单元，类型是 int *，所指向的类型是数组单元的类型即 int。因此*array 等于 0 就一点也不奇怪了。同理，array+3 是一个指向数组第 3 个单元的指针，所以*(array+3)等于 3。其它依此类推。

例九：

```
char *str[3]={
"Hello,this is a sample!","Hi,good morning.","Hello world"

};
char s[80];
strcpy(s,str[0]);//也可写成 strcpy(s,*str);
strcpy(s,str[1]);//也可写成 strcpy(s,*(str+1));

strcpy(s,str[2]);//也可写成 strcpy(s,*(str+2));
```

上例中，str 是一个三单元的数组，该数组的每个单元都是一个指针，这些指针各指向一个字符串。把指针数组名 str 当作一个指针的话，它指向数组的第 0 号单元，它的类型是 char**，它指向的类型是 char *。

str 也是一个指针，它的类型是 char，它所指向的类型是 char，它指向的地址是字符串"Hello,this is a sample!"的第一个字符的地址，即'H'的地址。

str+1 也是一个指针，它指向数组的第 1 号单元，它的类型是

char**，它指向的类型是 char *。

(str+1)也是一个指针，它的类型是 char，它所指向的类型是

char，它指向"Hi,good morning."的第一个字符'H'，等等。

下面总结一下数组的数组名的问题。声明了一个数组 TYPE array[n]，则数组名称 array 就有了两重含义：第一，它代表整个数组，它的类型是 TYPE [n]；第二，它是一个指针，该指针的类型是 TYPE*，该指针指向的类型是 TYPE，也就是数组单元的类型，该指针指向的内存区就是数组第 0 号单元，该指针自己占有单独的内存区，注意它和数组第 0 号单元占据的内存区是不同的。该指针的值是不能修改的，即类似 array++的表达式是错误的。

在不同的表达式中数组名 array 可以扮演不同的角色。

在表达式 sizeof(array)中，数组名 array 代表数组本身，故这时 sizeof 函数测出的是整个数组的大小。

在表达式*array 中，array 扮演的是指针，因此这个表达式的结果就是数组第 0 号单元的值。sizeof(*array)测出的是数组单元的大小。

表达式 array+n（其中 n=0，1，2，....。）中，array 扮演的是指针，故 array+n 的结果是一个指针，它的类型是 TYPE*，它指向的类型是 TYPE，它指向数组第 n 号单元。故 sizeof(array+n)测出的是指针类型的大小。

例十：

int array[10];

int (*ptr)[10];

ptr=&array;

上例中 ptr 是一个指针，它的类型是 int (*)[10]，他指向的类型是 int [10]，我们用整个数组的首地址来初始化它。在语句 ptr=&array 中，array 代表数组本身。

本节中提到了函数 sizeof()，那么我来问一问，sizeof(指针名称)测出的究竟是指针自身类型的大小呢还是指针所指向的类型的大小？答案是前者。例如：

int (*ptr)[10];

则在 32 位程序中，有：

sizeof(int(*)[10])==4

sizeof(int [10])==40

sizeof(ptr)==4

实际上，sizeof(对象)测出的都是对象自身的类型的大小，而不是别的什么类型的大小。

第六章。指针和结构类型的关系

可以声明一个指向结构类型对象的指针。

例十一：

```
struct MyStruct
{
int a;
```

int b;

int c;

}

MyStruct ss={20,30,40};//声明了结构对象 ss，并把 ss 的三个成员初始化为 20，30 和 40。

MyStruct *ptr=&ss;//声明了一个指向结构对象 ss 的指针。它的类型是 MyStruct*,它指向的类型是 MyStruct。

int *pstr=(int*)&ss;//声明了一个指向结构对象 ss 的指针。但是它的类型和它指向的类型和 ptr 是不同的。

请问怎样通过指针 ptr 来访问 ss 的三个成员变量？

答案：

ptr->a;

ptr->b;

ptr->c;

又请问怎样通过指针 pstr 来访问 ss 的三个成员变量？

答案：

*pstr；//访问了 ss 的成员 a。

*(pstr+1);//访问了 ss 的成员 b。

*(pstr+2)//访问了 ss 的成员 c。

呵呵，虽然我在我的 MSVC++6.0 上调式过上述代码，但是要知道，这样使用 pstr 来访问结构成员是不正规的，为了说明为什么不正规，让我们看看怎样通过指针来访问数组的各个单元：

例十二：

int array[3]={35,56,37};

int *pa=array;

通过指针 pa 访问数组 array 的三个单元的方法是：

*pa;//访问了第 0 号单元

*(pa+1);//访问了第 1 号单元

*(pa+2);//访问了第 2 号单元

从格式上看倒是与通过指针访问结构成员的不正规方法的格式一样。

所有的 C/C++编译器在排列数组的单元时，总是把各个数组单元存放在连续的存储区里，单元和单元之间没有空隙。但在存放结构对象的各个成员时，在某种编译环境下，可能会需要字对齐或双字对齐或者是别的什么对齐，需要在相邻两个成员之间加若干"填充字节"，这就导致各个成员之间可能会有若干个字节的空隙。

所以，在例十二中，即使*pstr 访问到了结构对象 ss 的第一个成员变量 a，也不能保证*(pstr+1)就一定能访问到结构成员 b。因为成员 a 和成员 b 之间可能会有若干填充字节，说不定*(pstr+1)就正好访问

到了这些填充字节呢。这也证明了指针的灵活性。要是你的目的就是想看看各个结构成员之间到底有没有填充字节，嘿，这倒是个不错的方法。

通过指针访问结构成员的正确方法应该是象例十二中使用指针 ptr 的方法。

第七章。指针和函数的关系

可以把一个指针声明成为一个指向函数的指针。

```
int fun1(char*,int);
int (*pfun1)(char*,int);
pfun1=fun1;
....
....
int a=(*pfun1)("abcdefg",7);//通过函数指针饗煤       ?
```

可以把指针作为函数的形参。在函数调用语句中，可以用指针表达式来作为 实参。

例十三：

```
int fun(char*);
int a;
char str[]="abcdefghijklmn";
a=fun(str);
...
...
int fun(char*s)
{
int num=0;
for(int i=0;i
{
num+=*s;s++;
}
return num;
}
```

这个例子中的函数 fun 统计一个字符串中各个字符的 ASCII 码值之和。前面说了，数组的名字也是一个指针。在函数调用中，当把 str 作为实参传递给形参 s 后，实际是把 str 的值传递给了 s，s 所指向的地址就和 str 所指向的地址一致，但是 str 和 s 各自占用各自的存储空间。在函数体内对 s 进行自加 1 运算，并不意味着同时对 str 进行了自加 1 运算。

第八章。指针类型转换

当我们初始化一个指针或给一个指针赋值时，赋值号的左边是一个指针，赋值号的右边是一个指针表达式。在我们前面所举的例子中，绝大多数情况下，指针的类型和指针表达式的类型是一样的，指针所指向的类型和指针表达式所指向的类型是一样的。

例十四：

1。 float f=12.3;

2。 float *fptr=&f;

3。 int *p;

在上面的例子中，假如我们想让指针 p 指向实数 f，应该怎么搞？是用下面的 语句吗？

p=&f;

不对。因为指针 p 的类型是 int*，它指向的类型是 int。表达式&f 的结果是一个指针，指针的类型是 float*,它指向的类型是 float。两者不一致，直接赋值的方法是不行的。至少在我的 MSVC++6.0 上，对指针的赋值语句要求赋值号两边的类型一致，所指向的类型也一致，其它的编译器上我没试过，大家可以试试。为了实现我们的目的，需要进行"强制类型转换"：

p=(int*)&f;

如果有一个指针 p，我们需要把它的类型和所指向的类型改为 TYEP*和 TYPE，那么语法格式是：

(TYPE*)p;

这样强制类型转换的结果是一个新指针，该新指针的类型是 TYPE*，它指向的类型是 TYPE，它指向的地址就是原指针指向的地址。而原来的指针 p 的一切属性都没有被修改。

一个函数如果使用了指针作为形参，那么在函数调用语句的实参和形参的结合过程中，也会发生指针类型的转换。

例十五：

```
void fun(char*);
int a=125,b;
fun((char*)&a);
...
...
void fun(char*s)
{
char c;
c=*(s+3);*(s+3)=*(s+0);*(s+0)=c;
c=*(s+2);*(s+2)=*(s+1);*(s+1)=c;
}
}
```

注意这是一个 32 位程序，故 int 类型占了四个字节，char 类型占一个字节。函数 fun 的作用是把一个整数的四个字节的顺序来个颠倒。注意到了吗？在函数调用语句中，实参&a 的结果是一个指针，它的类型是 int *，它指向的类型是 int。形参这个指针的类型是 char*，它指向的类型是 char。这样，在实参和形参的结合过程中，我们必须进行一次从 int*类型到 char*类型的转换。结合这个例子，我们可以这样来想象编译器进行转换的过程：编译器先构造一个临时指针 char*temp，然后执行 temp=(char*)&a，最后再把 temp 的值传递给 s。所以最后的结果是：s 的类型是 char*,它指向的类型是 char，它指向的地

址就是 a 的首地址。

我们已经知道，指针的值就是指针指向的地址，在 32 位程序中，指针的值其实是一个 32 位整数。那可不可以把一个整数当作指针的值直接赋给指针呢？就象下面的语句：

unsigned int a;

TYPE *ptr;//TYPE 是 int，char 或结构类型等等类型。

...

...

a=20345686;

ptr=20345686;//我们的目的是要使指针 ptr 指向地址 20345686（十进制）

ptr=a;//我们的目的是要使指针 ptr 指向地址 20345686（十进制）

编译一下吧。结果发现后面两条语句全是错的。那么我们的目的就不能达到了吗？不，还有办法：

unsigned int a;

TYPE *ptr;//TYPE 是 int，char 或结构类型等等类型。

...

...

a=某个数，这个数必须代表一个合法的地址；

ptr=(TYPE*)a；//呵呵，这就可以了。

严格说来这里的(TYPE*)和指针类型转换中的(TYPE*)还不一样。这里的(TYPE*)的意思是把无符号整数 a 的值当作一个地址来看待。

上面强调了 a 的值必须代表一个合法的地址，否则的话，在你使用 ptr 的时候，就会出现非法操作错误。

想想能不能反过来，把指针指向的地址即指针的值当作一个整数取出来。完全可以。下面的例子演示了把一个指针的值当作一个整数取出来，然后再把这个整数当作一个地址赋给一个指针：

例十六：

int a=123,b;

int *ptr=&a;

char *str;

b=(int)ptr;//把指针 ptr 的值当作一个整数取出来。

str=(char*)b;//把这个整数的值当作一个地址赋给指针 str。

好了，现在我们已经知道了，可以把指针的值当作一个整数取出来，也可以把一个整数值当作地址赋给一个指针。

第九章。指针的安全问题

看下面的例子：

例十七：

```
char s='a';
int *ptr;
ptr=(int*)&s;
*ptr=1298;
```

指针 ptr 是一个 int*类型的指针，它指向的类型是 int。它指向的地址就是 s 的首地址。在 32 位程序中，s 占一个字节，int 类型占四个字节。最后一条语句不但改变了 s 所占的一个字节，还把和 s 相临的高地址方向的三个字节也改变了。这三个字节是干什么的？只有编译程序知道，而写程序的人是不太可能知道的。也许这三个字节里存储了非常重要的数据，也许这三个字节里正好是程序的一条代码，而由于你对指针的马虎应用，这三个字节的值被改变了！这会造成崩溃性的错误。

让我们再来看一例：

例十八：

```
1。 char a;
2。 int *ptr=&a;
...
...
3。 ptr++;
4。 *ptr=115;
```

该例子完全可以通过编译，并能执行。但是看到没有？第 3 句对指针 ptr 进行自加 1 运算后，ptr 指向了和整形变量 a 相邻的高地址方向的一块存储区。这块存储区里是什么？我们不知道。有可能它是一个非常重要的数据，甚至可能是一条代码。而第 4 句竟然往这片存储区里写入一个数据！这是严重的错误。所以在使用指针时，程序员心里必须非常清楚：我的指针究竟指向了哪里。

在用指针访问数组的时候，也要注意不要超出数组的低端和高端界限，否则也会造成类似的错误。

在指针的强制类型转换：ptr1=(TYPE*)ptr2 中，如果 sizeof(ptr2 的类型)大于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是安全的。如果 sizeof(ptr2 的类型)小于 sizeof(ptr1 的类型)，那么在使用指针 ptr1 来访问 ptr2 所指向的存储区时是不安全的。至于为什么，读者结合例十七来想一想，应该会明白的。

66. 关于指向指针的指针

发布: 2008-2-18 21:16 | 作者: 剑心通明 | 来源: 互联网 | 查看: 50 次

就指向指针的指针,很早以前在说指针的时候说过,但后来发现很多人还是比较难以理解,这一次我们再次仔细说一说指向指针的指针!

先看下面的代码,注意看代码中的注解!

########################

剑心通明注：原文的程序因为都比较早，如果你原样照抄，可能会通不过，感谢下面评论的网友，我把两段程序代码全部整理了一下，发在本文的下面。

########################

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <string>
usingnamespacestd;

voidprint_char(char* array[],intlen);//函数原形声明

voidmain(void)
{
//----------------------------段 1----------------------------------------
char*a[]={"abc","cde","fgh"};//字符指针数组
char* *b=a;//定义一个指向指针的指针,并赋予指针数组首地址所指向的第一个字符串的地址也就是 abc\0 字符串的首地址
cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<
//------------------------------------------------------------------------

//----------------------------段 2----------------------------------------
char* test[]={"abc","cde","fgh"};//注意这里是引号,表示是字符串,以后的地址每加 1 就是加 4 位(在 32 位系统上)
intnum=sizeof(test)/sizeof(char*);//计算字符串个数
print_char(test,num);
cin.get();
//------------------------------------------------------------------------
}
```

voidprint_char(char* array[],intlen)//当调用的时候传递进来的不是数组,而是字符指针他每加 1 也就是加上 sizeof(char*)的长度

{

for(inti=0;i

{

cout<<*array++<

}

}

下面我们来仔细说明一下字符指针数组和指向指针的指针,段 1 中的程序是下面的样子:

char*a[]={"abc","cde","fgh"};

char* *b=a;

cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<

char *a[]定义了一个指针数组,注意不是 char[], char[]是不能同时初始化为三个字符的,定义以后的 a[]其实内部有三个内存位置,分别存储了 abc\0,cde\0,fgh\0,三个字符串的起始地址,而这三个位置的内存地址却不是这三个字符串的起始地址,在这个例子中 a[]是存储在栈空间内的,而三个字符串却是存储在静态内存空间内的 const 区域中的,接下去我们看到了 char* *b=a;这里是定义了一个指向指针的指针,如果你写成 char *b=a;那么是错误的,因为编译器会返回一个无法将 char* *[3]转换给 char *的错误,b=a 的赋值,实际上是把 a 的首地址赋给了 b,由于 b 是一个指向指针的指针,程序的输出 cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<

结果是

abc

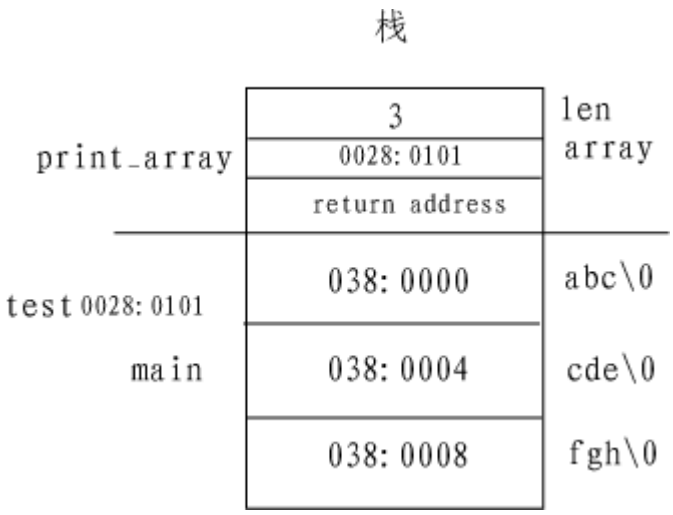
cde

fgh

可以看出每一次内存地址的+1 操作事实上是一次加 sizeof(char*)的操作,我们在 32 位的系统中 sizeof(char*)的长度是 4,所以每加 1 也就是+4,实际上是*a[]内部三个位置的+1,所以*(b+1)的结果自然就是 cde 了,我们这时候可能会问,为什么输出是 cde 而不是 c 一个呢?答案是这样的,在 c++中,输出字符指针就是输出字符串,程序会自动在遇到\0 后停止.

我们最后分析一下段 2 中的代码,段 2 中我们调用了 print_array()这个函数,这个函数中形式参数是 char *array[]和代码中的 char *test[]一样,同为字符指针,当你把参数传递过来的时候,事实上不是把数组内容传递过来,test 的首地址传递了进来,由于 array 是指针,所以在内存中它在栈区,具有变量一样的性质,可以为左值,所以我们输出写成了,cout<<*array++<<ENDL;当然我们也可以改写为 COUT<<ARRAY[I]<

到这里这两个非常重要的知识点我们都说完了,说归说,要想透彻理解希望读者多动手,多观察,熟能生巧!



内存结构示意图!

程序代码 1

```
#include <iostream>
#include <string>
using namespace std;
void print_char(char* array[],int len);
int main(void)
{
char*a[]={"abc","cde","fgh"};
char* *b=a;
cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<<endl;
char* test[]={"abc","cde","fgh"};
int num=sizeof(test)/sizeof(char*);
print_char(test,num);
cin.get();
return 0;
}
void print_char(char* array[],int len)
{
for(int i=0;i<len;i++){
cout<<*array++<<endl;}
}
```

程序代码 2

```
#include <iostream>
#include <string>
using namespace std;
void print_char(char* array[],int len);
```

```
int main(void)
{
char*a[]={"abc","cde","fgh"};
char* *b=a;
cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<<endl;

char* test[]={"abc","cde","fgh"};
int num=sizeof(test)/sizeof(char*);
print_char(test,num);
cin.get();

return 0;
}
void print_char(char* array[],int len)
{
for(int i=0;i<len;i++)
{
cout<<*array++<<endl;
}
}
```

67. C/C++ 误区一：void main()

发布: 2008-2-18 21:17 | 作者: 剑心通明 | 来源: 互联网 | 查看: 104 次

很多人甚至市面上的一些书籍，都使用了 **void main( )**，其实这是错误的。C/C++中从来没有定义过 **void main( )**。**C++**之父 **Bjarne Stroustrup** 在他的主页上的 **FAQ** 中明确地写着 The definition void main( ) { /* ... */ } is not and never has been C++, nor has it even been C.（void main( )从来就不存在于 C++或者 C）。下面我分别说一下 C 和 C++标准中对 main 函数的定义。

1. C

在 **C89** 中，**main( )**是可以接受的。**Brian W. Kernighan** 和 **Dennis M. Ritchie** 的经典巨著 The C programming Language 2e（《C 程序设计语言第二版》）用的就是 main( )。不过在最新的 **C99** 标准中，只有以下两种定义方式是正确的：

```
int main( void )
int main( int argc, char *argv[] )
```

（**参考资料**：ISO/IEC 9899:1999 (E) Programming languages—C 5.1.2.2.1 Program startup）

当然，我们也可以做一点小小的改动。例如：char *argv[]可以写成 char **argv；argv 和 argc 可以改成别的变量名（如 intval 和 charval），不过一定要符合变量的命名规则。

如果不需要从命令行中获取参数，请用 int main(void)；否则请用 int main(int argc, char *argv[])。

main 函数的返回值类型必须是 int，这样返回值才能传递给程序的调用者（如操作系统）。

如果 main 函数的最后没有写 return 语句的话，C99 规定编译器要自动在生成的目标文件中（如 exe 文件）加入 return 0;，表示程序正常退出。不过，我还是建议你最好在 main 函数的最后加上 return 语句，虽然没有这个必要，但这是一个好的习惯。注意，**vc6** 不会在目标文件中加入 return 0;，大概是因为 vc6 是 98 年的产品，所以才不支持这个特性。现在明白我为什么建议你最好加上 return 语句了吧！不过，**gcc3.2**（Linux 下的 C 编译器）会在生成的目标文件中加入 return 0;。

2. C++

C++98 中定义了如下两种 main 函数的定义方式：

```
int main( )
int main( int argc, char *argv[] )
```

（**参考资料**：ISO/IEC 14882(1998-9-01)Programming languages—C++3.6Start and termination）

int main( )等同于 C99 中的 int main(void)；int main(int argc, char *argv[])的用法也和 C99 中定义的一样。同样，main 函数的返回值类型也必须是 int。如果 main 函数的末尾没写 return 语句，C++98 规定编译器要自动在生成的目标文件中加入 return 0;。同样，vc6 也不支持这个特性，但是 g++3.2（Linux 下的 C++编译器）支持。

3.关于 void main

在 C 和 C++中，不接收任何参数也不返回任何信息的函数原型为“void foo(void);”。可能正是因为这个，所以很多人都误认为如果不需要程序返回值时可以把 main 函数定义成 void main(void)。然而这是错误的！main 函数的返回值应该定义为 int 类型，C 和 C++标准中都是这样规定的。虽然在一些编译器中，void main 可以通过编译（如 vc6），但并非所有编译器都支持 void main，因为**标准中从来没有定义过 void main**。**g++3.2 中如果 main 函数的返回值不是 int 类型，就根本通不过编译**。而 gcc3.2 则会发出警告。所以，如果你想你的程序拥有很好的**可移植性**，请一定要用 int main。

4.返回值的作用

main 函数的返回值用于说明程序的退出状态。如果返回 0，则代表程序正常退出；返回其它数字的含义则由系统决定。通常，返回非零代表程序异常退出。下面我们在 **winxp** 环境下做一个小实验。首先编译下面的程序：

```
int main( void )
{
    return 0;
}
```

然后打开附件里的“命令提示符”，在命令行里运行刚才编译好的可执行文件，然后输入“echo**%ERRORLEVEL%**”，回车，就可以看到程序的返回值为 0。假设刚才编译好的文件是 a.exe，如果输入“**a && dir**”，则会列出当前目录下的文件夹和文件。但是如果改成“return -1”，或者别的非 0 值，重新编译后输入“a && dir”，则 dir 不会执行。因为&&的含义是：如果&&前面的程序正常退出，则继续执行&&后面的程序，否则不执行。也就是说，利用程序的返回值，我们可以控制要不要执行下一个程序。这就是 int main 的好处。如果你有兴趣，也可以把 main 函数的返回值类型改成非 int 类型（如 float），重新编译后执行“a && dir”，看看会出现什么情况，想想为什么会出现那样的情况。顺便提一下，如果输入 **a || dir** 的话，则表示如果 a 异常退出，则执行 dir。

5. 那么 int main(int argc, char *argv[], char *envp[]) 呢？

这当然也不是标准 C/C++ 里面定义的东西！char *envp[] 是某些编译器提供的扩展功能，用于

获取系统的环境变量。因为不是标准，所以并非所有编译器都支持，故而移植性差，不推荐使用。

==========

如果您觉得我不够权威，那么就让 C++ 之父 **Bjarne Stroustrup** 来说服您吧！

请点击进入**我可以写"void main()"吗？**

68. C/C++ 误区二：fflush(stdin)

发布: 2008-2-18 21:18 | 作者: 剑心通明 | 来源: 互联网 | 查看: 119 次

为什么 fflush(stdin)是错的

首先请看以下程序：

```
#include<stdio.h>

int main( void )
{
  int i;
  for (;;) {
     fputs("Please input an integer: ", stdout);
     scanf("%d", &i);
     printf("%d\n", i);
  }
  return 0;
}
```

这个程序首先会提示用户输入一个整数，然后等待用户输入，如果用户输入的是整数，程序会输出刚才输入的整数，并且再次提示用户输入一个整数，然后等待用户输入。但是一旦用户输入的不是整数（如小数或者字母），假设 **scanf** 函数最后一次得到的整数是 2，那么程序会不停地输出“Please input an integer: 2”。这是因为 **scanf("%d", &i);**只能接受整数，如果用户输入了字母，则这个字母会遗留在“**输入缓冲区**”中。因为**缓冲**中有数据，故而 **scanf** 函数不会等待用户输入，直接就去缓冲中读取，可是缓冲中的却是字母，这个字母再次被遗留在缓冲中，如此反复，从而导致不停地输出“Please input an integer: 2”。

也许有人会说：“居然这样，那么在 scanf 函数后面加上‘fflush(**stdin**);’，把**输入缓冲**清空掉不就行了？”然而这是错的！**C** 和 **C++**的**标准**里从来没有定义过 fflush(stdin)。也许有人会说：“可是我用 fflush(stdin)解决了这个问题，你怎么能说是错的呢？”的确，**某些**编译器（如 **VC6**）支持用 fflush(stdin)来清空输入缓冲，但是并非所有编译器都要支持这个功能（linux 下的 gcc 就不支持），因为标准中根本没有定义 fflush(stdin)。**MSDN** 文档里也清楚地写着 fflush on input stream is an**extension**to the C standard（fflush 操作输入流是对 C 标准的**扩充**）。当然，如果你毫不在乎程序的**移植性**，用 fflush(stdin)也没什么大问题。以下是 **C99** 对 **fflush** 函数的定义：

int fflush(**FILE*stream**);

如果 **stream** 指向**输出流**或者**更新流**（update stream），并且这个更新流

最近执行的操作不是输入，那么 fflush 函数将把这个流中任何待写数据传送至

宿主环境（**host** environment）写入文件。否则，它的行为是**未定义**的。

原文如下：

int fflush(**FILE*stream**);
If stream points to an output stream or an update stream in which
the most recentoperation was not input, the fflush function causes
any unwritten data for thatstream to be delivered to the host environment
to be written to the file;otherwise, the behavior is undefined.

其中，宿主环境可以理解为操作系统或内核等。

由此可知，如果 stream 指向输入流（如 stdin），那么 fflush 函数的行为是不确定的。故而使用 **fflush(stdin)** 是不正确的，至少是**移植性不好**的。

2. 清空输入缓冲区的方法

虽然不可以用 fflush(stdin)，但是我们可以自己写代码来清空**输入缓冲区**。只需要在 scanf 函数后面加上几句简单的代码就可以了。

```
/* C 版本 */
#include<stdio.h>

int main( void )
{
  int i, c;
  for ( ; ; )
  {
     fputs("Please input an integer: ", stdout);
    scanf("%d", &i);
  if ( feof(stdin) || ferror(stdin) )
    {/* 如果用户输入文件结束标志（或文件已被读完）， */
      /* 或者发生读写错误，则退出循环          */

      /* do something */
       break;
    }
```

```
        /* 没有发生错误，清空输入流。                */
        /* 通过 while 循环把输入流中的余留数据“吃”掉 */
          while ( (c = getchar()) != '\n' && c != EOF ) ;
        /* 使用 scanf("%*[^\n]"); 也可以清空输入流，  */
        /* 不过会残留\n 字符。                       */
                                        printf("%d\n", i);
      }

                    return 0;
  }


/* C++ 版本 */
  #include<iostream>
  #include<limits>// 为了使用 numeric_limits

using std::cout;
  using std::endl;
 using std::cin;
  using std::numeric_limits;
  using std::streamsize;

int main()
  {
     int value;
      for ( ; ; )
      {
        cout << "Enter an integer: ";
         cin >> value;
        if ( cin.eof() || cin.bad() )
         {// 如果用户输入文件结束标志（或文件已被读完），
          // 或者发生读写错误，则退出循环
         // do something
           break;
        }
       // 读到非法字符后，输入流将处于出错状态，
       // 为了继续获取输入，首先要调用 clear 函数
```

```
        // 来清除输入流的错误标记，然后才能调用
       // ignore 函数来清除输入流中的数据。
        cin.clear();
      // numeric_limits<streamsize>::max() 返回输入缓冲的大小。
        // ignore 函数在此将把输入流中的数据清空。
       // 这两个函数的具体用法请读者自行查询。
        cin.ignore( numeric_limits<streamsize>::max(), '\n' );

       cout << value << '\n';
     }
   return 0;
  }
```

参考资料：

ISO/IEC 9899:1999 (E) Programming languages—**C 7.19.5.2 The fflush function**

The C Programming Language 2nd Edition By Kernighan & Ritchie

ISO/IEC 14882(1998-9-01)Programming languages—**C++**

69. C/C++ 误区三：强制转换 malloc() 的返回值

发布: 2008-2-18 21:18 | 作者: 剑心通明 | 来源: 互联网 | 查看: 56 次

首先要说的是，使用 **malloc** 函数，请包含 stdlib.h（C++ 中是 cstdlib），而不是 malloc.h 。因为 malloc.h 从来没有在 C 或者 C++ 标准中出现过！因此并非所有编译器都有 malloc.h 这个头文件。但是所有的 C 编译器都应该有 stdlib.h 这个头文件。

在 C++ 中，强制转换 malloc() 的返回值是必须的，否则不能通过编译。但是在 C 中，这种强制转换却是多余的，并且不利于代码维护。

起初，C 没有 void 指针，那时 char* 被用来作为泛型指针（generic pointer），所以那时 malloc 的返回值是 char* 。因此，那时必须强制转换 malloc 的返回值。后来，ANSI C（即 C89） 标准定义了 void 指针作为新的泛型指针。void 指针可以不经转换，直接赋值给任何类型的指针（函数指针除外）。从此，malloc 的返回值变成了 void* ，再也不需要强制转换 malloc 的返回值了。以下程序在 VC6 编译无误通过。

```
#include<stdlib.h>
int main( void )
{
   double *p = malloc( sizeof *p );/* 不推荐用 sizeof( double ) */
   free(p);
  return 0;
}
```

当然，强制转换 malloc 的返回值并没有错，但画蛇添足！例如，日后你有可能把 double *p 改成 int *p。这时，你就要把所有相关的 (double *) malloc (sizeof(double))改成(int *)malloc(sizeof(int))。如果改漏了，那么你的程序就存在 bug 。就算你有把握把所有相关的语句都改掉，但这种无聊乏味的工作你不会喜欢吧！不使用强制转换可以避免这样的问题，而且书写简便，何乐而不为呢？使用以下代码，无论以后指针改成什么类型，都不用作任何修改。

```
double *p = malloc( sizeof *p );
```

类似地，使用 calloc ，realloc 等返回值为 void* 的函数时，也不需要强制转换返回值。

参考资料：

ISO/IEC 9899:1999 (E) Programming languages — C 7.20.3.3 The malloc function

ISO/IEC 9899:1999 (E) Programming languages — C P104 (6.7.2.2)

70. C/C++ 误区四：char c = getchar();

发布: 2008-2-18 21:20 | 作者: 剑心通明 | 来源: 互联网 | 查看: 105 次

许多初学者都习惯用 char 型变量接收 getchar、getc，fgetc 等函数的返回值，其实这么做是不对的，并且隐含着足以**致命的错误**。getchar 等函数的返回值类型都是 **int** 型，当这些函数**读取出错**或者**读完文件后**，会返回 **EOF**。EOF 是一个宏，标准规定它的值必须是一个 **int 型的负数常量**。通常编译器都会把 EOF 定义为 **-1**。问题就出在这里，使用 char 型变量接收 getchar 等函数的返回值会导致对 EOF 的辨认出错，或者错把好的数据误认为是 EOF，或者把 EOF 误认为是好的数据。例如：

```
int c;  /* 正确。应该使用 int 型变量接收 fgetc 的返回值 */
while ( (c = fgetc(fp)) != EOF )
{
   putchar(c);
}
```

如上例所示，我们很多时候都需要先用一个变量接收 fgetc 等函数的返回值，然后再用这个变量和 EOF 比较，判断是否已经读完文件。上面这个例子是正确的，把 c 定义为 int 型保证了它能正确接收 fgetc 返回的 EOF，从而保证了这个比较的正确性。但是，**如果把 c 定义为 char 型，则会导致意想不到的后果**。

首先，因为 fgetc 等函数的返回值是 int 型的，当赋值给 char 型变量时，会发生降级，从而导致数据截断。例如：

十进制	int	char
10	00 00 00 0A	0A
-1	FF FF FF FF	FF
-2	FF FF FF FE	FE

在此，我们假设 int 和 char 分别是 32 位和 8 位的。由上表可得，从 int 型到 char 型，损失了 3 个字节的数据。而当我们要拿 char 型和 int 型比较的时候，char 型会自动升级为 int 型。char 型升级为 int 型后的值会因为它到底是 **signed char** 还是 **unsigned char** 而有所不同。不幸的是，如果我们没有使用 signed 或者 unsigned 来修饰 char，那么我们无从知晓 char 到底是指 unsigned char 还是指 signed char，因为这是**由编译器决定**的。不过，无论 char 是 signed 的也好，unsigned 的也罢，都不能改变使用 char 型变量接收 fgetc 等函数的返回值是**错误**的这个事实。唯一能改变的是该错误导致的后果。前面我们说了，char 型和 int 型比较时，char 会自动升级为 int，下面我们来看看 signed char 和 unsigned char 在转换成 int 后，它们的值有什么不同：

char	unsigned	signed
10	00 00 00 0A	00 00 00 0A
FF	00 00 00 FF	FF FF FF FF
FE	00 00 00 FE	FF FF FF FE

由上表可知，当 char 是 unsigned 的时候，其转换为 int 后的值是**正数**。也就是说，假如我们把 c 定义为 char 型变量，而编译器默认 char 为 unsigned char，那么以下表达式将**永远成立**。

```
(c = fgetc(fp)) != EOF  /* c 的值永远为正数，而标准规定 EOF 为负数 */
```

也就是说以下循环是一个**死循环**。

```
while ( (c = fgetc(fp)) != EOF )
{
  putchar(c);
}
```

读到这里，可能有些读者朋友会说：“那么我明确把 c 定义为 signed char 型的就没问题了吧！”很遗憾，就算把 c 定义为 signed char，仍然是错误的。假设 fgetc 等函数读到一个字节的值为 **FF**，那么返回值就是 **00 00 00 FF**。把这个值赋值给 c 后， c 的值变成 FF。然后 c 的值为了和 EOF 比较，会自动升级为 int 型的值，也就是 **FF FF FF FF**。从而导致以下**表达式不成立**。

```
(c = fgetc(fp)) != EOF  /* 读到值为 FF 的字符，误认为 EOF */
```

也就是说以下循环在**没有读完文件**的情况下**提前退出**。

```
while ( (c = fgetc(fp)) != EOF )
{
  putchar(c);
}
```

综上所述，使用 char 型变量接收 fgetc 等函数的返回值是错误的，我们**必须使用 int 型变量接收这些函数的返回值**，然后判断接收到的值是否 EOF。只有判断发现该返回值并非 EOF，我们才可以把该值赋值给 char 型变量。

同理，C++ 中，用 char 型变量接收 **cin.get()** 的返回值也是错误的。不过，把 char 型变量当作**参数**传递给 **cin.get** 则是正确的。例如：

```
char c = cin.get();  // 错误，理由同上
char c;
cin.get(c);          // 正确
```

71. C/C++ 误区五：检查 new 的返回值

发布: 2008-2-18 21:20 | 作者: 剑心通明 | 来源: 互联网 | 查看: 97 次

首先澄清一下，这个误区仅对 C++ 成立，这里不过是沿用“C/C++ 误区”这个衙头罢了。

我们都知道，使用 malloc/calloc 等分配内存的函数时，一定要检查其返回值是否为“空指针”（亦即检查分配内存的操作是否成功），这是良好的编程习惯，也是编写可靠程序所必需的。但是，如果你简单地把这一招应用到 new 上，那可就不一定正确了。我经常看到类似这样的代码：

```
int* p = new int[SIZE];
if ( p == 0 ) // 检查 p 是否空指针
  return -1;
// 其它代码
```

其实，这里的 if (p == 0) 完全是没啥意义的。C++ 里，如果 new 分配内存失败，默认是**抛出异常**的。所以，如果分配成功，p == 0 就绝对不会成立；而如果分配失败了，也不会执行 if (p == 0)，因为分配失败时，new 就会**抛出异常跳过后面的代码**。如果你想检查 new 是否成功，应该**捕捉异常**：

```
try {
  int* p = new int[SIZE];
  // 其它代码
} catch ( const bad_alloc& e ) {
  return -1;
}
```

据说一些老的编译器里，new 如果分配内存失败，是不抛出异常的（大概是因为那时 C++ 还没加入异常机制），而是和 malloc 一样，返回空指针。不过我从来都没遇到过 new 返回空指针的情况。

当然，标准 C++ 亦提供了一个方法来**抑制 new 抛出异常**，而返回空指针：

```
int* p = new (std::nothrow) int; // 这样如果 new 失败了，就不会抛出异常，而是返回空指针
if ( p == 0 ) // 如此这般，这个判断就有意义了
  return -1;
// 其它代码
```

72. C 是 C++ 的子集吗？

发布: 2008-5-12 08:20 | 作者: 剑心通明 | 来源: 互联网 | 查看: 15 次

许多初学者都以为 C 是 C++ 的子集，其实这种观点是不正确的。C++ 之父 Bjarne Stroustrup（简称 B.S）在其 FAQ 上解答了这个问题。

C 是 C++ 的子集吗？

严格按照数学上的定义来说，C 不是 C++ 的子集。有些程序在 C 里面是合法的，但在 C++ 里却是不合法的；甚至有些编写代码的方式在 C 和 C++ 中有不同的含义。然而，C++ 支持 C 所支持的全部编程技巧。任何 C 程序都能被 C++ 用基本相同的方法写出来，并且运行效率和空间效率都一样。把数万行 ANSI C 代码转换成 C 风格的 C++ 代码，通常只需要几个小时。因此，C++ 是 ANSI C 的超集程度和 ANSI C 是 K&R C 的超集程度以及 ISO C++ 是 1985 年的 C++ 的超集程度差不多。

编写风格好的 C 程序通常会是合法的 C++ 程序。例如，Kernighan 和 Ritchie 合著的《C 程序设计语言（第二版）》中的所有例子都是 C++ 程序。

C/C++ 兼容性问题的一些例子：

```
int main()
{
  double sq2 = sqrt(2);  /* 不是 C++：调用了未经声明的函数 */
  int s = sizeof('a');   /* 隐蔽的区别：C++ 中是 1，而 C 中却是 sizeof(int) */
}
```

调用未经声明的函数在 C 里是不良风格，而在 C++ 里是非法的。同样的情况还有，传递参数给一个没有在其声明中列出参数类型的函数：

```
void f();    /* 没有注明参数类型 */
void g()
{
  f(2);    /* C 中是不良风格。C++ 中不合法 */
}
```

C 里面，void* 可以隐式转换为任何指针类型，并且堆空间分配通常是通过 malloc() 进行的，无法检查是否已经分配了足够的内存：

```
void* malloc(size_t);
void f(int n)
{
  int* p = malloc(n*sizeof(char));  /* C++ 中非法。C++ 使用 new 分配堆空间 */
  char c;
  void* pv = &c;
  int* pi = pv;   /* 隐式转换 void* 为 int*。C++ 中非法 */
}
```

请注意，隐式转换 void* 为 int* 引起了潜在的数据对齐错误。请参阅 **C++ 中 void* 和 malloc() 的代替品**。

把 C 代码转换成 C++ 时，请紧记 C++ 的关键字比 C 多：

```
int class = 2;    /* C 中合法。C++ 中是语法错误 */
int virtual = 3;  /* C 中合法。C++ 中是语法错误 */
```

除了少数例外，如上面所述的例子（以及 C++ 标准和 **C++ 程序设计语言第三版**附录 B 里详细列举的例子），C++ 是 C 的超集。（附录 B 可以通过下载获取）

请注意，“C”在以上段落中指的是经典 C 和 C89。C++ 不是 C99 的后裔；C++ 和 C99 是兄弟。C99 引入了一些新特性，造成 **C/C++ 的不兼容性**进一步增大。这里有一篇 **C++98 和 C99 的不同点**的描述。

原文地址：http://www.research.att.com/~bs/bs_faq.html#C-is-subset

==========

antigloss 注：原文说“Kernighan 和 Ritchie 合著的《C 程序设计语言（第二版）》（简称 K&R）中的所有例子都是 C++ 程序”，我对此有不同看法。C++ 要求函数必须显式注明其返回值类型，而 K&R 中所有的 main 函数都没有注明返回值类型，这显然不是合法的 C++ 代码。或许 B.S 指的是标准前的 C++ 吧。标准前的 C++ 是允许不显式注明函数返回值类型的，如果省略返回值类型，则默认为 int。而且，C++ 中的标准头文件已经不用 .h 了，但是 K&R 中用的都是 .h。当然，C 中也只能用 .h。同样，标准前的 C++ 用的也是 .h。就此，我已发了一封 e-mail 给他老人家，得到他的回复如下：

> I wrote this mail to ask you a question on an article at your FAQ: Is C a
>subset of C++? In this article, you wrote "every example in Kernighan &
>Ritchie: 'The C Programming Language (2nd Edition)' is also a C++ program",
>which is what I doubt about. In C++, every function must explicitly write
>down its return type. But in K&R, none of the main function had its return
>type written. So those programs shouldn't be C++ progroms. I know in K&R
>C, if return type is omitted, then `int' is implied, so does pre-standard
>C++.

and that was allowed in C++ at the time. You could then say that K&R2 is neither
C nor C++, but I find that a bit pedantic. Especially as C and C++ changed in
exactly the same way.

> Also, in K&R, each header uses .h postfix which is not used in C++ but
>used in pre-standard C++.

but the .h form is allowed for the C headers in C++, even today.

大意为：

>C++ 要求函数必须显式注明其返回值类型，而 K&R 中所有的 main 函数都没有注明返回值类型，

>这显然不是合法的 C++ 代码。标准前的 C++ 允许不显式注明函数返回值类型，如果省略返回值类型，

>则默认为 int。

那时的 C++ 也允许这种做法。你可以说 K&R2 既不是 C 也不是 C++，但我觉得这样有点咬文嚼字了。尤其是，C 和 C++ 都朝同样的方向作了改变。

>C++ 中的标准头文件已经不用 .h 了，但是 K&R 中用的都是 .h。同样，标准前的 C++ 用的也是 .h。

但 C++ 仍然允许 C 头文件使用 .h 形式，甚至今天。

73. C 和 C++的区别是什么？

发布: 2008-5-12 08:28 | 作者: 剑心通明 | 来源: 互联网 | 查看: 20 次

Q: 对于小项目来说 C 比 C++好，是吗？

A: 我不这么认为。我从未见过任何一个项目是优于 C++的，除非是因为缺乏一个好的 C++编译器。

Q: C 和 C++的区别是什么？

A: C++是 C 的一个直接后代，它几乎包含整个 C 作为一个子集。C++提供了比 C 更强的类型检查，并支持更多的编程风格。因为 C++支持 C 语言的编程风格，同时提供更好的类型检查和更多概念上的支持，又不损失效率，所以从这个意义上来说，C++是“一个更好的 C”。同样，ANSI C 也是一个比 K&R C 更好的 C。另外，C++还支持数据抽象、面向对象编程和泛型编程（见 The C++ Programming Language (3rd Edition)；附录 B 讨论了 C 兼容性的问题，该章可以下载。）

我从没有见过一个问题用 C 来表达会比 C++更好（我也不相信会有这样的问题，因为每个 C 的构造显然都有 C++的等价物。）然而，还是有一些环境对 C++的支持不是很好，而用 C 会有好处。

关于 C++的设计以及它与 C 的关系的讨论，可以看 The Design and Evolution of C++。

请注意上面所说的 C 指的是经典 C 和 C89。C++不是 C99 的后代，C++和 C99 是兄弟。C99 引入的一些特性有可能会造成 C 和 C++的不兼容。这里有一个关于 C++98 和 C99 的不同点的描述。

74. 无条件循环

发布: 2008-5-12 08:30 | 作者: 剑心通明 | 来源: 互联网 | 查看: 6 次

通常，如果要写一个**无限**循环，我们可以这么写：

```
while ( 1 ) {
   /* ... ... */
}
```

这么写完全正确。不过，while (1) 是一个**有条件循环**，每次循环都会判断 **1** 的真假性，因为 1 永远为真，所以这个循环会无限执行。正因为 while (1) 是**有条件循环**，所以在效率上比**无条件循环**要略孙一筹。哪怎么写**无条件循环**呢？很简单。请看：

```
for (/* 可以有代码 */; /* 必须为空 */; /* 可以有代码 */) {
  /* ... ... */
}
```

上面的代码就是一个**无条件循环**，每次循环不用进行任何判断，所以效率比 while (1) 高。

不过，有些**编译器**非常聪明，可以在**编译时**分析出 while (1) 条件永远为真，是一个**无限循环**，从而对生成的**二进制代码**进行优化，使得每次循环不用对 1 的真假性进行判断。

75. 产生随机数的方法

发布: 2008-5-12 08:52 | 作者: 剑心通明 | 来源: 互联网 | 查看: 17 次

1. 如何产生一定范围内的随机数？

直接的方法是：

```
rand() % N;
```

返回从 0 到 N - 1 的数字。但这个方法不好，因为许多随机数发生器的低位比特并不随机。一个较好的方法是：

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N);
```

如果你不希望使用 double，另一个方法是：

```
rand() / (RAND_MAX / N + 1);
```

两种方法都需要知道 RAND_MAX，而且假设 N 要远远小于 RAND_MAX。ANSI 规定标准头文件 stdlib.h 中包含 RAND_MAX 的 #define。顺便提一下，RAND_MAX 是个常数，它告诉你 C 库函数 rand() 的固定范围。你不可以设 RAND_MAX 为其它的值，也没有办法要求 rand() 返回其它范围的值。如果你用的随机数发生器返回的是 0 到 1 的浮点值，要取得范围在 0 到 N - 1 内的整数，只要将随机数乘以 N 就可以了。

2. 为什么每次执行程序，rand() 都返回相同顺序的数字？

你可以调用 srand() 来初始化伪随机数发生器的种子，传递给 srand() 的值应该是真正的随机数，例如当前时间：

```
#include<stdlib.h>
#include<time.h>
srand((unsigned int)time((time_t *)NULL));
```

请注意，在一个程序执行中多次调用 srand() 并不见得有帮助！不要为了取得“真随机数”而在每次调用 rand() 前都调用 srand()！

3. 我需要随机的真/假值，所以我用直接用 rand() % 2，可是我得到交替的 0, 1, 0, 1, 0 。

这是个低劣的伪随机数生成器，在低位比特中不随机！很不幸，某些系统就提供这样的伪随机数生成器。请试着使用高位比特，具体请参考本文第 1 点。

76. 顺序表及其操作

发布: 2008-5-12 08:59 | 作者: 剑心通明 | 来源: 互联网 | 查看: 21 次

这个程序足足花了我一整天才写出来！希望对各位有帮助。如果发现我的代码存在问题，请不吝指出。谢谢！请使用 VC++6 打开 main.dsw 文件，然后点击运行按钮（就是那个红色的叹号）编译执行。

顺序表是线性表的一种最简单的存储结构。顺序表存储方式为：在内存中开辟一片连续空间，每个元素相互毗邻地存放。

顺序表的优点：可以随机访问表中任一位置的元素。

缺点：插入和删除效率低；

我这个程序支持的操作如下：

1. 创建顺序表
2. 输入数据
3. 插入数据
4. 删除数据
5. 求顺序表并集
6. 删除重复元素
7. 冒泡排序
8. 比较顺序表大小
9. 前 N 个元素和后 M 个元素互换
10. 删除重复元素(2)

0. 退出

77. 单链表的实现及其操作

发布: 2008-5-12 09:00 | 作者: 剑心通明 | 来源: 互联网 | 查看: 59 次

请使用 VC++6 打开 main.dsw 文件，然后点击运行按钮（就是那个红色的叹号）编译执行。

==========

单链表

1、链接存储方法

链接方式存储的线性表简称为链表（Linked List）。

链表的具体存储表示为：

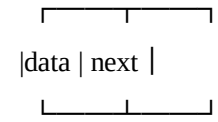
① 用一组任意的存储单元来存放线性表的结点（这组存储单元既可以是连续的，也可以是不连续的）

② 链表中结点的逻辑次序和物理次序不一定相同。为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其后继结点的地址（或位置）信息（称为指针（pointer）或链(link)）

注意：

链式存储是最常用的存储方式之一，它不仅可用来表示线性表，而且可用来表示各种非线性的数据结构。

2、链表的结点结构



data 域--存放结点值的数据域

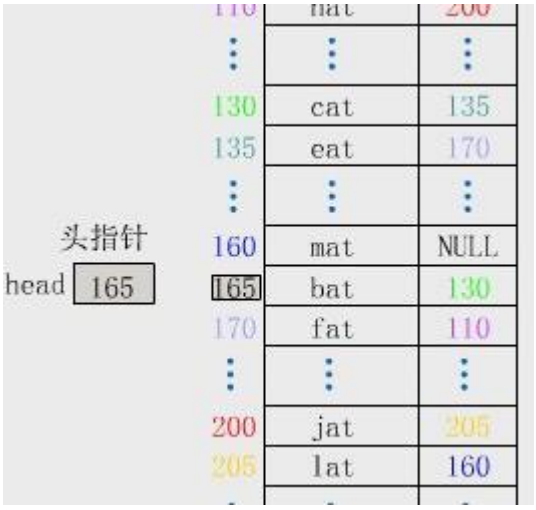
next 域--存放结点的直接后继的地址（位置）的指针域（链域）

注意：

①链表通过每个结点的链域将线性表的 n 个结点按其逻辑顺序链接在一起的。

②每个结点只有一个链域的链表称为单链表（Single Linked List）。

【例】线性表（bat，cat，eat，fat，hat，jat，lat，mat）的单链表示如示意图



3、头指针 head 和终端结点指针域的表示

单链表中每个结点的存储地址是存放在其前趋结点 next 域中，而开始结点无前趋，故应设头指针 head 指向开始结点。

注意：

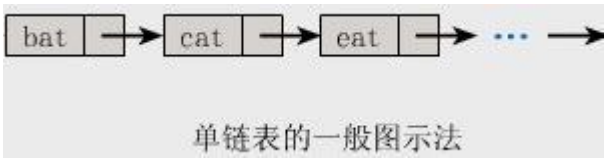
链表由头指针唯一确定，单链表可以用头指针的名字来命名。

【例】头指针名是 head 的链表可称为表 head。

终端结点无后继，故终端结点的指针域为空，即 NULL。

4、单链表的一般图示法

由于我们常常只注重结点间的逻辑顺序，不关心每个结点的实际位置，可以用箭头来表示链域中的指针，线性表（bat，cat，fat，hat，jat，lat，mat）的单链表就可以表示为下图形式。



5、单链表类型描述

```
typedef char DataType; /*  假设结点的数据域类型为字符  */
typedef struct node {     /*  结点类型定义  */
    DataType data;        /*  结点的数据域  */
    struct node *next;    /*  结点的指针域  */
} ListNode;
typedef ListNode *LinkList;
ListNode *p;
LinkList head;
```

注意：

①LinkList 和 ListNode *是不同名字的同一个指针类型（命名的不同是为了概念上更明确）

②LinkList 类型的指针变量 head 表示它是单链表的头指针

③ListNode *类型的指针变量 p 表示它是指向某一结点的指针

6、指针变量和结点变量

	指针变量	结点变量
定义	在变量说明部分显式定义 函数 malloc 生成	在程序执行时，通过标准
取值	非空时，存放某类型结点的地址	实际存放结点各域内容
操作方式	通过指针变量名访问	通过指针生成、访问和释放

①生成结点变量的标准函数

p = malloc(sizeof(ListNode));

/* 函数 malloc 分配一个类型为 ListNode 的结点变量的空间,并将其首地址放入指针变量 p 中 */

②释放结点变量空间的标准函数

free(p); /* 释放 p 所指的结点变量空间 */

③结点分量的访问

利用结点变量的名字*p 访问结点分量

方法一：(*p).data 和(*p).next

方法二：p- > data 和 p- > next

④指针变量 p 和结点变量*p 的关系

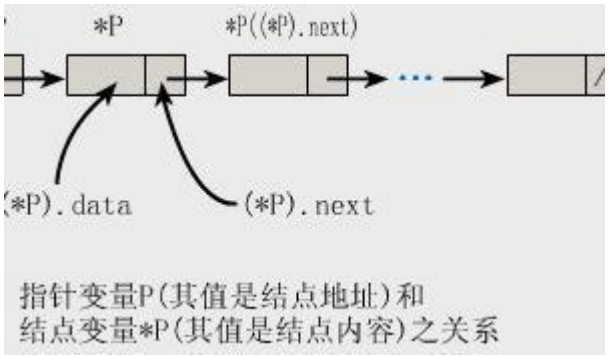
指针变量 p 的值——结点地址

结点变量*p 的值——结点内容

(*p).data 的值——p 指针所指结点的 data 域的值

(*p).next 的值——*p 后继结点的地址

*((*p).next)——*p 后继结点



注意：

① 若指针变量 p 的值为空（NULL），则它不指向任何结点。此时，若通过*p 来访问结点就意味着访问一个不存在的变量，从而引起程序的错误。

② 有关指针类型的意义和说明方式的详细解释，【参考 C 语言的有关资料】。

78. 双向链表

发布: 2008-5-12 09:04 | 作者: 剑心通明 | 来源: 互联网 | 查看: 44 次

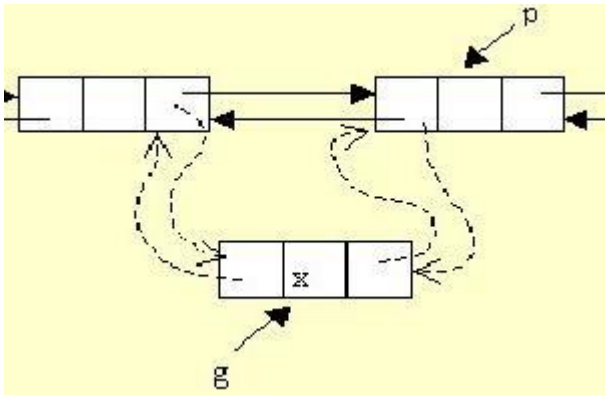
请使用 VC++6 打开 main.dsw 文件，然后点击运行按钮（就是那个红色的叹号）编译执行。

==========

双链表

线性表的插入运算（双向链表存储结构）

在双向循环链表 L 的位置 p 处插入一个新元素 x 的过程 Insert 可实现如下。



算法：

```
typedef struct dnode {
      elemtype data;
      struct dnode *prior, *next;
} DNODE;
int link_ins(DNODE **head, int i, elemtype x)
{
    int j = 1; /*  双向循环链表  */
    DNODE *p, *q;
    q = malloc( sizeof *p );
    q->data = x;
    if ( i == 0 ) {
       q->prior = *head;
       q->next = *head;
       *head = q;
       return 0;
    }
    p = *head;
    j = 0;
    while ( ++j < i && p != NULL )
        p = p->next;
```

```
    if ( i < 0 || j < i )
       return 1;
    else {
        q->next = p->next;
        p->next = q;
        p->next->prior = q;
        q->prior = p;
        return 0;
    }
}
```

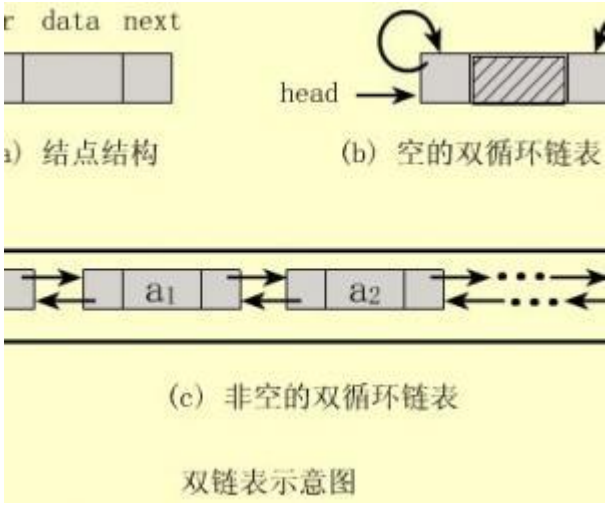
分析：在上面的插入算法中，不需要移动别的元素，但必须

从头开始查找第 i 结点的地址，一旦找到插入位置，则插入结点

只需两条语句就可完成。该算法的时间复杂度为 O(n)

1、双向链表（Doubly Linked List）

双（向）链表中有两条方向不同的链，即每个结点中除 next 域存放后继结点地址外，还增加一个指向其直接前趋的指针域 prior。



双链表示意图

注意：

①双链表由头指针 head 惟一确定的。

②带头结点的双链表的某些运算变得方便。

③将头结点和尾结点链接起来，为双（向）循环链表。

2、双向链表的结点结构和形式描述

①结点结构(见上图 a)

②形式描述

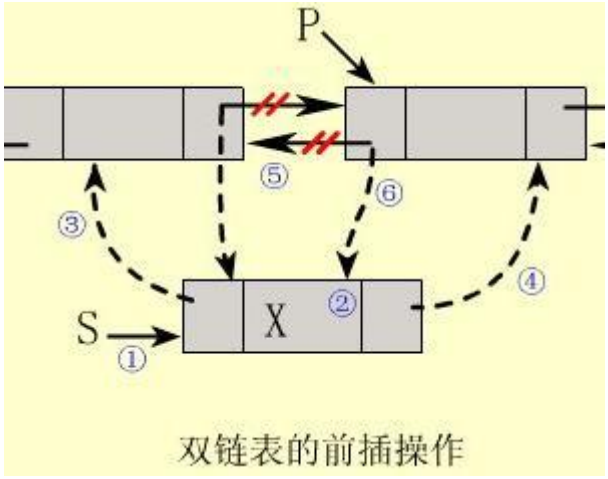
```
typedef struct dlistnode {
```

```
    DataType data;
    struct dlistnode *prior,*next;
} DListNode;
typedef DListNode *DLinkList;
DLinkList head;
```

3、双向链表的前插和删除本结点操作

由于双链表的对称性，在双链表能能方便地完成各种插入、删除操作。

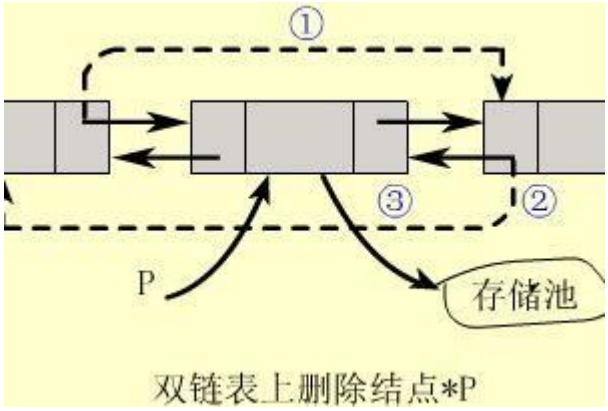
①双链表的前插操作



双链表的前插操作

```
void DInsertBefore(DListNode *p, DataType x)
{ /* 在带头结点的双链表中，将值为 x 的新结点插入*p 之前，设 p≠NULL */
    DListNode *s = malloc( sizeof(DListNode) ); /* ① */
    s->data = x; /* ② */
    s->prior = p->prior; /* ③ */
    s->next = p; /* ④ */
    p->prior->next = s; /* ⑤ */
    p->prior = s; /* ⑥ */
}
```

②双链表上删除结点*p 自身的操作



双链表上删除结点*P

```
void DDeleteNode(DListNode *p)
{ /* 在带头结点的双链表中，删除结点*p，设*p 为非终端结点 */
    p->prior->next = p->next; /* ① */
    p->next->prior = p->prior; /* ② */
    free(p); /* ③ */
}
```

注意：

与单链表上的插入和删除操作不同的是，在双链表中插入和删除必须同时修改两个方向上的指针。

上述两个算法的时间复杂度均为 O(1)。

79. 程序员数据结构笔记

发布: 2008-6-20 08:55 | 作者: 剑心通明 | 来源: 互联网 | 查看: 18 次

第一天

真想不到,第一次上课竟然会是"9.11"事件纪念日.美国竟然还是不改老毛病,伊拉克战争死了多少平民百姓啊?!!!在此请先为死难者默哀 3 分钟,老美如果再这样多管闲事下去,上帝会二度惩罚美国人的啊!

能听到周 SIR 讲课真的挺开心的,我觉得他讲课比某些高程(高级工程师)还深动[转载时注：此处应该是 生动](当然,他的数据结构水平应该不亚于高程),为了考中程,上学期的<算法分析>选修课我也去揍了揍热闹.可惜由于SARS的关系,课只上了将近一半就停了.可以说我报程序员的原因就是因为有周 SIR 开导我们,听他的课真的是一种享受,不像大学里的某些人,水平不怎么高还在这里作威作福.

好了,发了这么多劳骚,开始转入正题了.

读这文章的人想必都是想报考或者将考程序员的同志们吧!首先一点必须问问自己,我考程序员的目的到底是为了什么?如果你的答案是:"只是为了拿一本证书".那我劝你早点 GIVE UP 吧!那样学起来会很累,因为没有兴趣.如果你想加入程序员的行列,为将来开发打下坚实的基础,那就试着看完这一小篇读书笔记吧!或许会对你有所帮助.

有句话必须记住:程序员考试仅仅是为了检验自己学到的而已,仅此而已!我想这便是程序员考试的最终意义所在吧!有些事情更注重过程!

数据结构

知识:

1.数据结构中对象的定义,存储的表示及操作的实现.

2.线性:线性表、栈、队列、数组、字符串（广义表不考）

树：二叉树

集合：查找，排序

图(不考)

能力：

分析，解决问题的能力

过程：

- 确定问题的数据。
- 确定数据间的关系。
- 确定存储结构（顺序－数组、链表－指针）
- 确定算法
- 编程
- 算法评价（时间和空间复杂度，主要考时间复杂度）

一、数组

1、存放于一个连续的空间

2、一维～多维数组的地址计算方式

已知 data[0][0]的内存地址，且已知一个元素所占内存空间 S 求 data[i][j]在内存中的地址。

公式：（add+(i*12+j)*S)(假设此数组为 data[10][12])

注意：起始地址不是 data[0][0]时候的情况。起始地址为 data[-3][8]和情况；

3、顺序表的定义

存储表示及相关操作

4、顺序表操作中时间复杂度估计

5、字符串的定义（字符串就是线性表），存储表示

模式匹配算法（简单和 KMP（不考））

6、特殊矩阵：存储方法（压缩存储（按行，按列））

三对角：存储于一维数组

三对角问题：已知 A_{ij} 能求出在一维数组中的下标 k;已知下标 k 求 A_{ij}。

7、稀疏矩阵：定义，存储方式：三元组表、十字链表（属于图部分，不考）

算法

- 数组中元素的原地逆置； 对换
- 在顺序表中搜索值为 X 的元素；
- 在有序表中搜索值为 X 的元素；（折半查找）
- 在顺序表中的第 i 个位置插入元素 X；
- 在顺序表中的第 i 个位置删除元素 X；
- 两个有序表的合并；算法？

线性表数据结构定义：

```
Typedef struct {
   int data[max_size];
   int len;
}linear_list;
```

- 模式匹配
- 字符串相加
- 求子串
- （i,j）<=>K 注意：不同矩阵所用的公式不同；
- 稀疏矩阵的转置（两种方式，后种为妙）
- 和数组有关的算法

例程：求两个长整数之和。

a=13056952168

b=87081299

数组:

a[]:1 3 0 5 6 9 5 2 1 6 8

b[]:8 7 0 8 1 2 9 9

由于以上的结构不够直观(一般越是直观越容易解决) 将其改为:

a[]:11　8 6 1 2 5 9 6 5 0 3 1 a[0]=11(位数)

b[]: 8　9 9 2 1 8 0 7 8 0 0 0 b[0]=8

c 进位　0 1 1 0 0 1 1 1 1 0 0

c[]:11　7 6 4 3 3 0 4 4 2 3 1 c[0]的值(C 位数)由 c[max_s+1]决定!

注意:在求 C 前应该将 C(max_s+1)位赋 0.否则为随机数; 较小的整数高位赋 0.

算法:已知 a,b 两个长整数,结果:c=a+b;

总共相加次数: max_s=max(a[],b[])

程序:

```
for(i=1;i<=max_s;i++) {
   k=a[i]+b[i]+c[i];
   c[i]=k%10;
   c[i+1]=k/10;
}
```

求 c 位数:

```
if(c[max_s+1]==0)
   c[0]=max_s;
else
   c[0]=max_s+1;
```

以下代码是我编的(毕竟是初学者.不太简洁大家不要见怪!):

```
/*两长整数相加*/
 #include<stdio.h>
 #include<string.h>
#define PRIN printf("\n");

int flag=0; /*a[0]>b[0]?1:0*/

/* max(a[],b[]) {}*/

change(char da[],char db[],int a[],int b[],int c[]) {
   int i;
   if(a[0]>b[0]) {
      for(i=1;i<=a[0];a[i]=da[a[0]-i]-'0',i++); /*a[0]-'0' so good!*/
      for(i=1;i<=b[0];b[i]=db[b[0]-i]-'0',i++);
      for(i=b[0]+1;i<=a[0];b[i]=0,i++);
      for(i=1;i<=a[0]+1;c[i]=0,i++);
      flag=1;
   }
```

```
    else {
        for(i=1;i<=b[0];b[i]=db[b[0]-i]-'0',i++);
        for(i=1;i<=a[0];a[i]=da[a[0]-i]-'0',i++);
        for(i=a[0]+1;i<=b[0];a[i]=0,i++);
        for(i=1;i<=b[0]+1;c[i]=0,i++);
    }
}

add(int a[],int b[],int c[]) {
    int i,sum;
    if(flag==1) {
        for(i=1;i<=a[0];i++) {
            sum=a[i]+b[i]+c[i];
            c[i+1]=sum/10;
            c[i]=sum%10;
        }
        if(c[a[0]+1]==0)
            c[0]=a[0];
        else
            c[0]=a[0]+1;
    }
    else {
        for(i=1;i<=b[0];i++) {
            sum=a[i]+b[i]+c[i];
            c[i+1]=sum/10;
            c[i]=sum%10;
        }
        if(c[b[0]+1]==0)
            c[0]=b[0];
        else
            c[0]=b[0]+1;
    }
}

void print(int m[]) {
    int i;
    for(i=m[0];i>=1;i--)
        printf("%d,",m[i]); PRIN
}
```

```
main(){
    int s;
    int a[20],b[20],c[20];
    char da[]={"123456789"};
    char db[]={"12344443"};
    a[0]=strlen(da);
    b[0]=strlen(db);
    printf("a[0]=%d\t",a[0]);
    printf("b[0]=%d",b[0]); PRIN
    change(da,db,a,b,c);
    printf("flag=%d\n",flag); PRIN
    printf("-----------------\n");
    if(flag==1) {
        print(a); PRIN
        s=abs(a[0]-b[0]);
        printf("+");
            for(s=s*2-1;s>0;s--)
                printf(" ");
                print(b); PRIN
    }
    else {
        s=abs(a[0]-b[0]);
        printf("+");
        for(s=s*2-1;s>0;s--)
            printf(" ");
            print(a); PRIN
            print(b); PRIN
    }
    add(a,b,c);
    printf("-----------------\n");
    print(c);
}
```

时间复杂度计算:

- 确定基本操作
- 计算基本操作次数
- 选择 T(n)

● lim(F(n)/T(n))=c

● 0(T(n))为时间复杂度

上例子的时间复杂度为 O(max_s);

二:链表

1、知识点

●逻辑次序与物理次序不一致存储方法；

●单链表的定义：术语（头结点、头指针等）

●注意带头结点的单链表与不带头结点的单链表区别。（程序员考试一般不考带头结点，因为稍难理解）

●插入、删除、遍历（p==NULL 表明操作完成）等操作

● 循环链表：定义，存储表示，操作；

● 双向链表：定义，存储方法，操作；

单链表和循环链表区别在最后一个指针域值不同。

2、操作

●单链表：插入 X，删除 X，查找 X，计算结点个数

●单链表的逆置（中程曾考）

head->NULL/p->a1/p->a2/p->a3/p.....an/NULL 注：p 代表指针;NULL/p 代表头结点

＝》 head->NULL/p->an/p->an-1/p->an-2/p.....a1/NULL

●循环链表的操作：插入 X，删除 X，查找 X，计算结点个数；

用 p=head->next 来判断一次计算结点个数完成；

程序段如下：

```
k=0;
do{
   k++;
   p=p->next;
}while(p!=head->next);
```

● 双向链表

●多项式相加

● 有序链表合并

例程：已知两个字符串 S，T，求 S 和 T 的最长公子串；

1、逻辑结构：字符串

2、存储结构：数组

3、算法： 精化（精细化工）****老顽童注：此处“精细化工”说明好像不对！**

s="abaabcacb"

t="abdcabcaabcda"

当循环到 s.len-1 时，有两种情况：s="abaabcacb"、s="abaabcacb"

s.len-2 时，有三种情况：s="abaabcacb"、s="abaabcacb"、s="abaabcacb"

.

.

.

1 s.len 种情况

程序思路：

tag=0 //没有找到

for(l=s.len;l>0&&!tag;l--) {

判断长度为 l 的 s 中的子串是否为 t 的子串;

若是：tag=1;

}

长度为 l 的 s 的子串在 s 中有（s.len-l+1）个。

子串 0: 0～l-1

1: 1～l

2: 2～l+1

3: 3～l+2

.....

.....

s.len-l: s.len-l～s.len-1

由上面可得：第 j 个子串为 j～l+j-1。

判断长度为 l 的 s 中的子串是否为 t 的子串:

for(j=0;j<s.len-l+1&&!tag;j++){

判断 s 中长度为 l 的第 j 个子串是否为 t 的子串;

如果是：tag=1;

}

模式结构：

tag=0;

for(l=s.len;l>0&&tag==0;l--) {

for(j=0;j<s.len-l+1&&!tag;j++) {

?? 用模式匹配方法确定 s[j]～s[l+j-1]这个字符串是否为 t 的子串； //好好想想

若是，tag=1;

}

}

80. Hashtable 和 HashMap 的区别

发布: 2008-6-20 09:13 | 作者: 剑心通明 | 来源: 互联网 | 查看: 11 次

HashTable 的应用非常广泛，**HashMap** 是新框架中用来代替 **HashTable** 的类，也就是说建议使用 HashMap，不要使用 HashTable。可能你觉得 HashTable 很好用，为什么不用呢？这里简单分析他们的区别。

1.HashTable 的方法是同步的，HashMap 未经同步，所以在多线程场合要手动同步 HashMap 这个区别就像 Vector 和 ArrayList 一样。

2.HashTable 不允许 null 值(key 和 value 都不可以),HashMap 允许 null 值(key 和 value 都可以)。

3.HashTable 有一个 contains(Object value)，功能和 containsValue(Object value)功能一样。

4.HashTable 使用 Enumeration，HashMap 使用 Iterator。 以上只是表面的不同，它们的实现也有很大的不同。

5.HashTable 中 hash 数组默认大小是 11，增加的方式是 old*2+1。HashMap 中 hash 数组的默认大小是 16，而且一定是 2 的指数。

6.哈希值的使用不同，HashTable 直接使用对象的 hashCode，代码是这样的：

```
int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;
```

而 HashMap 重新计算 hash 值，而且用与代替求模：

```
int hash = hash(k);
int i = indexFor(hash, table.length);

static int hash(Object x) {
        int h = x.hashCode();
        h += ~(h << 9);
        h ^= (h >>> 14);
        h += (h << 4);
        h ^= (h >>> 10);
        return h;
}
```

```
static int indexFor(int h, int length) {
        return h & (length-1);
}
```

以上只是一些比较突出的区别，当然他们的实现上还是有很多不同的，比如 HashMap 对 null 的操作。

Hashtable 和 HashMap 的区别：

1.Hashtable 是 Dictionary 的子类，HashMap 是 Map 接口的一个实现类；

2.Hashtable 中的方法是同步的，而 HashMap 中的方法在缺省情况下是非同步的。即是说，在多线程应用程序中，不用专门的操作就安全地可以使用 Hashtable 了；而对于 HashMap，则需要额外的同步机制。但 HashMap 的同步问题可通过 Collections 的一个静态方法得到解决：

Map Collections.synchronizedMap(Map m)

这个方法返回一个同步的 Map，这个 Map 封装了底层的 HashMap 的所有方法，使得底层的 HashMap 即使是在多线程的环境中也是安全的。

3.在 HashMap 中，null 可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为 null。当 get()方法返回 null 值时，即可以表示 HashMap 中没有该键，也可以表示该键所对应的值为 null。因此，在 HashMap 中不能由 get()方法来判断 HashMap 中是否存在某个键，而应该用 containsKey()方法来判断。

Hashtable 继承自 Dictionary 类，而 HashMap 是 Java1.2 引进的 Map interface 的一个实现

HashMap 允许将 null 作为一个 entry 的 key 或者 value，而 Hashtable 不允许

还有就是，HashMap 把 Hashtable 的 contains 方法去掉了，改成 containsvalue 和 containsKey。因为 contains 方法容易让人引起误解。

最大的不同是，Hashtable 的方法是 Synchronize 的，而 HashMap 不是，在

多个线程访问 Hashtable 时，不需要自己为它的方法实现同步，而 HashMap

就必须为之提供外同步。

Hashtable 和 HashMap 采用的 hash/rehash 算法都大概一样，所以性能不会有很大的差异。

81. hash 表学习笔记

发布: 2008-6-20 09:15 | 作者: 剑心通明 | 来源: 互联网 | 查看: 9 次

1. hash 表的概念

hash 表技术是直接查找技术的推广， 其主要目标是提高查找效率。

2. 几种 hash 表

<1>. 链式 hash 表。 这种 hash 表是在实际应用中最常用的 hash 表。如图所示

```
|------|
|      |--> 1 --> 11
|------|
|      |--> 2 --> 12
|------|
|      |--> 3 --> 13
  ...
  ...
```

链式 hash 表的填入：

· 计算关键字 K 的 hash 码 i = i(k).

· 获得新的结点 P, 并将关键字 K 及有关信息填入结点 P 中。

· 将结点 P 链入以 H(i) 为头指针的链表的链头。

链式 hash 表的取出

· 计算关键字 K 的 hash 码 i = i(k).

· 在以 H(i) 为头指针的链表中查找关键字为 K 的结点。

<2>. 溢出 hash 表。

溢出 hash 表包括 hash 表与溢出表两部分。

在 hash 表填入过程中，将冲突的元素顺序填入到溢出表中。

而当查找过程中发现冲突时，就在溢出表中进行顺序查找。

<3>. 顺序 hash 表是一种简单的 hash 表，其中存在很多问题，在实际开发中基本上不使用。

<4>. 随机 hash 表。它与顺序 hash 表基本相同，只不过当发生冲突时它使用随即数来解决冲突。

82. C 程序设计常用算法源代码

发布: 2008-6-20 09:46 | 作者: 剑心通明 | 来源: 互联网 | 查看: 18 次

算法（Algorithm）：计算机解题的基本思想方法和步骤。算法的描述：是对要解决一个问题或要完成一项任务所采取的方法和步骤的描述，包括需要什么数据（输入什么数据、输出什么结果）、采用什么结构、使用什么语句以及如何安排这些语句等。通常使用自然语言、结构化流程图、伪代码等来描述算法。

一、计数、求和、求阶乘等简单算法

此类问题都要使用循环，要注意根据问题确定循环变量的初值、终值或结束条件，更要注意用来表示计数、和、阶乘的变量的初值。

例：用随机函数产生 100 个[0，99]范围内的随机整数，统计个位上的数字分别为 1，2，3，4，5，6，7，8，9，0 的数的个数并打印出来。

本题使用数组来处理，用数组 a[100]存放产生的确 100 个随机整数，数组 x[10]来存放个位上的数字分别为 1，2，3，4，5，6，7，8，9，0 的数的个数。即个位是 1 的个数存放在 x[1]中，个位是 2 的个数存放在 x[2]中，.....个位是 0 的个数存放在 x[10]。

```
void main()
{ int a[101],x[11],i,p;
for(i=0;i<=11;i++)
x[i]=0;
for(i=1;i<=100;i++)
{ a[i]=rand() % 100;
printf("%4d",a[i]);
if(i%10==0)printf("\n");
}
for(i=1;i<=100;i++)
{ p=a[i]%10;
if(p==0) p=10;
x[p]=x[p]+1;
}
for(i=1;i<=10;i++)
{ p=i;
if(i==10) p=0;
printf("%d,%d\n",p,x[i]);
}
printf("\n");
}
```

二、求两个整数的最大公约数、最小公倍数

分析：求最大公约数的算法思想：(最小公倍数=两个整数之积/最大公约数)

(1) 对于已知两数 m，n，使得 m>n；

(2) m 除以 n 得余数 r；

(3) 若 r=0，则 n 为求得的最大公约数，算法结束；否则执行(4)；

(4) m←n，n←r，再重复执行(2)。

例如: 求 m=14 ,n=6 的最大公约数. m n r

14 6 2

6 2 0

```
void main()
{ int nm,r,n,m,t;
printf("please input two numbers:\n");
scanf("%d,%d",&m,&n);
nm=n*m;
if (m<n)
{ t=n; n=m; m=t; }
r=m%n;
while (r!=0)
{ m=n; n=r; r=m%n; }
printf("最大公约数:%d\n",n);
printf("最小公倍数:%d\n",nm/n);
}
```

三、判断素数

只能被 1 或本身整除的数称为素数 基本思想：把 m 作为被除数，将 2—INT（ ）作为除数，如果都除不尽，m 就是素数，否则就不是。（可用以下程序段实现）

```
void main()
{ int m,i,k;
printf("please input a number:\n");
scanf("%d",&m);
k=sqrt(m);
for(i=2;i<k;i++)
if(m%i==0) break;
if(i>=k)
printf("该数是素数");
else
printf("该数不是素数");
}
```

将其写成一函数,若为素数返回 1，不是则返回 0

```
int prime( m%)
{int i,k;
k=sqrt(m);
for(i=2;i<k;i++)
```

```
if(m%i==0) return 0;
return 1;
}
```

四、验证哥德巴赫猜想

（任意一个大于等于 6 的偶数都可以分解为两个素数之和）

基本思想：n 为大于等于 6 的任一偶数，可分解为 n1 和 n2 两个数，分别检查 n1 和 n2 是否为素数，如都是，则为一组解。如 n1 不是素数，就不必再检查 n2 是否素数。先从 n1=3 开始，检验 n1 和 n2（n2=N-n1）是否素数。然后使 n1+2 再检验 n1、n2 是否素数，... 直到 n1=n/2 为止。

利用上面的 prime 函数，验证哥德巴赫猜想的程序代码如下：

```
#include "math.h"
int prime(int m)
{ int i,k;
k=sqrt(m);
for(i=2;i<k;i++)
if(m%i==0) break;
if(i>=k)
return 1;
else
return 0;
}
main()
{ int x,i;
printf("please input a even number(>=6):\n");
scanf("%d",&x);
if (x<6||x%2!=0)
printf("data error!\n");
else
for(i=2;i<=x/2;i++)
if (prime(i)&&prime(x-i))
{
printf("%d+%d\n",i,x-i);
printf("验证成功!");
break;
}
}
```

五、排序问题

1．选择法排序（升序）

基本思想：

1）对有 n 个数的序列（存放在数组 a(n)中），从中选出最小的数，与第 1 个数交换位置；

2）除第 1 个数外，其余 n-1 个数中选最小的数，与第 2 个数交换位置；

3）依次类推，选择了 n-1 次后，这个数列已按升序排列。

程序代码如下：

```
void main()
{ int i,j,imin,s,a[10];
printf("\n input 10 numbers:\n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
for(i=0;i<9;i++)
{ imin=i;
for(j=i+1;j<10;j++)
if(a[imin]>a[j]) imin=j;
if(i!=imin)
{s=a[i]; a[i]=a[imin]; a[imin]=s; }
printf("%d\n",a[i]);
}
}
```

2．冒泡法排序（升序）

基本思想：(将相邻两个数比较，小的调到前头)

1）有 n 个数（存放在数组 a(n)中），第一趟将每相邻两个数比较，小的调到前头，经 n-1 次两两相邻比较后，最大的数已“沉底”，放在最后一个位置，小数上升“浮起”；

2）第二趟对余下的 n-1 个数（最大的数已“沉底”）按上法比较，经 n-2 次两两相邻比较后得次大的数；

3）依次类推，n 个数共进行 n-1 趟比较，在第 j 趟中要进行 n-j 次两两比较。

程序段如下

```
void main()
{ int a[10];
int i,j,t;
printf("input 10 numbers\n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
printf("\n");
for(j=0;j<=8;j++)
for(i=0;i<9-j;i++)
if(a[i]>a[i+1])
{t=a[i];a[i]=a[i+1];a[i+1]=t;}
printf("the sorted numbers:\n");
```

```
for(i=0;i<10;i++)
printf("%d\n",a[i]);
}
```

3．合并法排序（将两个有序数组 A、B 合并成另一个有序的数组 C，升序）

基本思想：

1）先在 A、B 数组中各取第一个元素进行比较，将小的元素放入 C 数组；

2）取小的元素所在数组的下一个元素与另一数组中上次比较后较大的元素比较，重复上述比较过程，直到某个数组被先排完；

3）将另一个数组剩余元素抄入 C 数组，合并排序完成。

程序段如下：

```
void main()
{ int a[10],b[10],c[20],i,ia,ib,ic;
printf("please input the first array:\n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
for(i=0;i<10;i++)
scanf("%d",&b[i]);
printf("\n");
ia=0;ib=0;ic=0;
while(ia<10&&ib<10)
{ if(a[ia]<b[ib])
{ c[ic]=a[ia];ia++;}
else
{ c[ic]=b[ib];ib++;}
ic++;
}
while(ia<=9)
{ c[ic]=a[ia];
ia++;ic++;
}
while(ib<=9)
{ c[ic]=b[ib];
b++;ic++;
}
for(i=0;i<20;i++)
printf("%d\n",c[i]);
}
```

六、查找问题

1．①顺序查找法（在一列数中查找某数 x）

基本思想：一列数放在数组 a[1]---a[n]中，待查找的数放在 x 中，把 x 与 a 数组中的元素从头到尾一一进行比较查找。用变量 p 表示 a 数组元素下标，p 初值为 1，使 x 与 a[p]比较，如果 x 不等于 a[p]，则使 p=p+1，不断重复这个过程；一旦 x 等于 a[p]则退出循环；另外，如果 p 大于数组长度，循环也应该停止。（这个过程可由下语句实现）

```
void main()
{ int a[10],p,x,i;
printf("please input the array:\n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
printf("please input the number you want find:\n");
scanf("%d",&x);
printf("\n");
p=0;
while(x!=a[p]&&p<10)
p++;
if(p>=10)
printf("the number is not found!\n");
else
printf("the number is found the no%d!\n",p);
}
```

思考：将上面程序改写一查找函数 Find，若找到则返回下标值，找不到返回-1

②基本思想：一列数放在数组 a[1]---a[n]中，待查找的关键值为 key，把 key 与 a 数组中的元素从头到尾一一进行比较查找，若相同，查找成功，若找不到，则查找失败。(查找子过程如下。index：存放找到元素的下标。)

```
void main()
{ int a[10],index,x,i;
printf("please input the array:\n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
printf("please input the number you want find:\n");
scanf("%d",&x);
printf("\n");
index=-1;
for(i=0;i<10;i++)
if(x==a[i])
{ index=i; break;
}
if(index==-1)
printf("the number is not found!\n");
```

```
else
printf("the number is found the no%d!\n",index);
}
```

83. C 语言有头结点链表的经典实现

发布: 2008-6-29 21:09 | 作者: 剑心通明 | 来源: 互联网 | 查看: 20 次

经常用到链表，但每次写都不那么顺利，终于有点时间整理一下，把感觉写的不错的代码拿出来分享，希望大家能指出问题，那我算没白写。 该链表以存放整型数据为例。

头文件：

```
#ifndef __LINK_H__
#define __LINK_H__
#define ERROR ( -1 )
#define OK      ( 0   )
#define TRUE   ( 1 == 1 )
#define FALSE ( !TRUE   )

typedef int BOOL;
typedef int elem_t;          //定义元素的数据类型
typedef struct node
...{
    elem_t data;
    struct node * next;
}tagNode_t,tagList_t;

/**//* 把已经存在的头节点进行初始化为一个空链表 */
void initList ( tagList_t * list );
/**//* 销毁链表 */
void destroyList( tagList_t * list );
/**//* 将链表重置为空 */
void clearList( tagList_t * list );
BOOL listEmpty( tagList_t * list );
int listLength( tagList_t * list );
/**//* 得到指定位置 pos 的元素，如果顺利得到则返回 TRUE，否则返回 FALSE*/
BOOL getElem( tagList_t * list, int iPos, elem_t * e);
/**//* 返回第一个满足 compare 关系的元素的位置，如果不存在则返回 -1 */
int locateElem( tagList_t * list, elem_t e,
    BOOL (*compare)(elem_t e1,elem_t e2));
/**//* 将元素 e 插到链表的指定位置 ,若 iPos > listLength() 则插到最后*/
BOOL listInsert( tagList_t * list, int iPos, elem_t e );
BOOL listInsertFront( tagList_t * list, elem_t e );
```

```
/**//* 按照费递减序插入元素 e */
BOOL listSortInsert( tagList_t * list, elem_t e );
BOOL listDeleteFront( tagList_t * list, elem_t * e );
/**//* 用 visit 函数遍历链表 */
void listTraverse( tagList_t * list, void (*visit)(elem_t e));

#endif
```

源文件：

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "link.h"

/**//* 把已经存在的头节点进行初始化为一个空链表 */
void initList ( tagList_t * list )
...{
    list->next = NULL;
}
/**//* 销毁链表,没有释放头结点 */
void destroyList( tagList_t * list )
...{
    tagNode_t * pn = NULL;
    assert( list != NULL );

    pn = list->next;
    while( pn != NULL)
    ...{
        list->next = pn->next;
        free( pn );
        pn = list->next;
    }
}
/**//* 将链表重置为空 */
void clearList( tagList_t * list )
...{
```

```
    destroyList( list );   //同 销毁
}
BOOL listEmpty( tagList_t * list )
...{
    return ( NULL == list->next );
}
int listLength( tagList_t * list )
...{
    int iLen = 0;
    tagNode_t * ptn = NULL;
    assert( list != NULL );

    ptn = list->next;
    while( ptn != NULL )
    ...{
       iLen += 1;
       ptn = ptn->next;
    }
    return iLen;
}
/**//* 得到指定位置 pos 的元素，如果顺利得到则返回 TRUE，否则返回 FALSE*/
BOOL getElem( tagList_t * list, int iPos, elem_t * e)
...{
    BOOL bRet = FALSE;
    tagNode_t * ptn = NULL;
    assert( list );        //list != NULL

    if( list || ( iPos >= 0 ) )
    ...{
       ptn = list->next;
       while( iPos-- && ptn )
       ...{
          ptn = ptn->next;
       }
       if( ptn != NULL )
       ...{
          *e = ptn->data;
```

```
            bRet = TRUE;
        }
    }
    return bRet;
}
/**//* 返回第一个满足 compare 关系的元素的位置，如果不存在则返回 0 */
int locateElem( tagList_t * list, elem_t e,
    BOOL (*compare)(elem_t e1,elem_t e2))
...{
    int iPos = 0;
    tagNode_t * ptn = NULL;
    assert( list );        //list != NULL

    ptn = list->next;
    while( ptn )
    ...{
        iPos += 1;
        if( compare( e, ptn->data ) )
        ...{
            break;
        }
        ptn = ptn->next;
    }
    if( NULL == ptn )
    ...{
        iPos = 0;
    }
    return iPos;

}
/**//* 将元素 e 插到链表的指定位置 ,若 iPos > listLength() 则插到最后*/
BOOL listInsert( tagList_t * list, int iPos, elem_t e )
...{
    BOOL bRet = FALSE;
    tagNode_t * ptn = NULL, * pstNew = NULL;
    assert( list );        //list != NULL
```

```
    if( list || (iPos >= 0) )
    ...{
        ptn = list->next;
        while( iPos-- && ptn->next )
        ...{
            ptn = ptn->next;
        }
        pstNew = ( tagNode_t * )malloc( sizeof( tagNode_t ) );
        if( pstNew != NULL )
        ...{
            pstNew->data = e;
            pstNew->next = ptn->next;
            ptn->next = pstNew;

            bRet = TRUE;
        }
    }
    return bRet;
}
BOOL listInsertFront( tagList_t * list, elem_t e )
...{
    BOOL bRet = FALSE;
    tagNode_t * pstNew =
        ( tagNode_t * )malloc( sizeof( tagNode_t ) );
    assert( list );

    if( pstNew != NULL )
    ...{
        pstNew->data = e;
        pstNew->next = list->next;
        list->next = pstNew;

        bRet = TRUE;
    }
    return bRet;
}
/**//*  按照非递减序插入元素  e */
```

```
BOOL listSortInsert( tagList_t * list, elem_t e )
...{
    BOOL bRet = FALSE;
    tagNode_t * ptn = NULL, * pstNew = NULL;
    assert( list );

    ptn = list;
    while( ptn->next )
    ...{
       if( ptn->next->data > e )
       ...{
          bRet = TRUE;
          break;
       }
       ptn = ptn->next;
    }

    pstNew = ( tagNode_t * )malloc( sizeof( tagNode_t ) );
    if( pstNew != NULL)
    ...{
       pstNew->data = e;
       pstNew->next = ptn->next;
       ptn->next = pstNew;

       bRet = TRUE;
    }
    return bRet;
}
BOOL listDeleteFront( tagList_t * list, elem_t * e )
...{
    BOOL bRet = FALSE;
    tagNode_t * ptn = NULL;

    if( list != NULL )
    ...{
       ptn = list->next;
       *e = ptn->data;
```

```
        list->next = ptn->next;
        free( ptn );
        bRet = TRUE;
    }
    return bRet;
}
/**//* 用 visit 函数遍历链表 */
void listTraverse( tagList_t * list, void (*visit)(elem_t e))
...{
    tagNode_t * ptn = NULL, * pstNew = NULL;
    assert( list );

    ptn = list->next;
    while( ptn )
    ...{
        visit( ptn->data );
        ptn = ptn->next;
    }
}
```

测试文件：

```
include <stdio.h>
#include "link.h"
void tst_link()
...{
    elem_t e;
    tagList_t list;

    printf("********Test Stack******** ");

    initList( &list );

    printf("IsEmpty: %s ",(listEmpty( &list ) ? "true":"false"));

    listSortInsert( &list, 4 );
```

```
    listSortInsert( &list, 3 );

    printf("5 in pos: %d ",locateElem( &list,5,elemEqual ));

    listSortInsert( &list, 5 );

    printf("5 in pos: %d ",locateElem( &list,5,elemEqual ));

    listSortInsert( &list, 7 );
    listSortInsert( &list, 2 );

    printf("getElem (3) %s : %d ",(getElem( &list, 3, &e )?"right":"error"),e);
    printf("getElem (6) %s : %d ",(getElem( &list, 6, &e )?"right":"error"),e);

    listPrint( &list );
    printf(" ");

    printf("len: %d ",listLength( &list ));
    listDeleteFront( &list, &e );
    printf("After delete front len: %d ",listLength( &list ));
    printf("IsEmpty: %s ",(listEmpty( &list ) ? "true":"false"));
    printf("5 in pos: %d ",locateElem( &list,5,elemEqual ));

    destroyList( &list );

    printf("IsEmpty: %s ",(listEmpty( &list ) ? "true":"false"));
}
```

84. C 语言惠通面试题

发布: 2008-7-03 07:56 | 作者: 剑心通明 | 来源: 互联网 | 查看: 15 次

什么是预编译，何时需要预编译：

总是使用不经常改动的大型代码体。

程序由多个模块组成，所有模块都使用一组标准的包含文件和相同的编译选项。在这种情况下，可以将所有包含文件预编译为一个预编译头。

```
char * const p;
char const * p
const char *p
```

上述三个有什么区别？

```
char * const p; //常量指针，p 的值不可以修改
char const * p；//指向常量的指针，指向的常量值不可以改
const char *p；  //和 char const *p

char str1[] = "abc";
char str2[] = "abc";

const char str3[] = "abc";
const char str4[] = "abc";

const char *str5 = "abc";
const char *str6 = "abc";

char *str7 = "abc";
char *str8 = "abc";

cout << ( str1 == str2 ) << endl;
cout << ( str3 == str4 ) << endl;
cout << ( str5 == str6 ) << endl;

cout << ( str7 == str8 ) << endl;
```

结果是：0 0 1 1

解答：str1,str2,str3,str4 是数组变量，它们有各自的内存空间；而 str5,str6,str7,str8 是指针，它们指向相同的常量区域。

以下代码中的两个 sizeof 用法有问题吗？

```
void UpperCase( char str[] ) // 将 str 中的小写字母转换成大写字母
{
    for( size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i )
   if( 'a'<=str[i] && str[i]<='z' )
  str[i] -= ('a'-'A' );
}
char str[] = "aBcDe";
cout << "str 字符长度为: " << sizeof(str)/sizeof(str[0]) << endl;
UpperCase( str );
cout << str << endl;
```

答：函数内的 sizeof 有问题。根据语法，sizeof 如用于数组，只能测出静态数组的大小，无法检测动态分配的或外部数组大小。函数外的 str 是一个静态定义的数组，因此其大小为 6，函数内的 str 实际只是一个指向字符串的指针，没有任何额外的与数组相关的信息，因此 sizeof 作用于上只将其当指针看，一个指针为 4 个字节，因此返回 4。

一个 32 位的机器,该机器的指针是多少位？

指针是多少位只要看地址总线的位数就行了。80386 以后的机子都是 32 的数据总线。所以指针的位数就是 4 个字节了。

```
main()
{
   int a[5]={1,2,3,4,5};
   int *ptr=(int *)(&a+1);
   printf("%d,%d",*(a+1),*(ptr-1));
}
```

输出：2,5

(a+1）就是 a[1]，(ptr-1)就是 a[4],执行结果是 2，5

&a+1 不是首地址+1，系统会认为加一个 a 数组的偏移，是偏移了一个数组的大小（本例是 5

个 int）

int *ptr=(int *)(&a+1);

则 ptr 实际是&(a[5]),也就是 a+5

原因如下：

&a 是数组指针，其类型为 int (*)[5];

而指针加 1 要根据指针类型加上一定的值，不同类型的指针+1 之后增加的大小不同。

a 是长度为 5 的 int 数组指针，所以要加 5*sizeof(int)

所以 ptr 实际是 a[5]

但是 prt 与(&a+1)类型是不一样的(这点很重要)

所以 prt-1 只会减去 sizeof(int*)

a,&a 的地址是一样的，但意思不一样，a 是数组首地址，也就是 a[0]的地址，&a 是对象（数组）首地址，a+1 是数组下一元素的地址，即 a[1],&a+1 是下一个对象的地址，即 a[5].

请问以下代码有什么问题：

```
int   main()
{
   char a;
   char *str=&a;
   strcpy(str,"hello");
   printf(str);
   return 0;
}
```

没有为 str 分配内存空间，将会发生异常。问题出在将一个字符串复制进一个字符变量指针所指地址。虽然可以正确输出结果，但因为越界进行内在读写而导致程序崩溃。

```
char* s="AAA";
printf("%s",s);
s[0]='B';
printf("%s",s);
```

有什么错？

"AAA"是字符串常量。s 是指针，指向这个字符串常量，所以声明 s 的时候就有问题。

cosnt char* s="AAA";

然后又因为是常量，所以对是 s[0]的赋值操作是不合法的。

写一个“标准”宏，这个宏输入两个参数并返回较小的一个。

.#define Min(X, Y) ((X)>(Y)?(Y):(X))//结尾没有;

嵌入式系统中经常要用到无限循环，你怎么用 C 编写死循环。

while(1){}或者 for(;;)

关键字 static 的作用是什么？

定义静态变量

关键字 const 有什么含意？

表示常量不可以修改的变量。

关键字 volatile 有什么含意？并举出三个不同的例子？

提示编译器对象的值可能在编译器未监测到的情况下改变。

int (*s[10])(int) 表示的是什么？

int (*s[10])(int) 函数指针数组，每个指针指向一个 int func(int param)的函数。

有以下表达式：

```
int a=248; b=4;
int const c=21;
const int *d=&a;
int *const e=&b;
```

int const *f const =&a;

请问下列表达式哪些会被编译器禁止？为什么？

*c=32;d=&b;*d=43;e=34;e=&a;f=0x321f;

*c 这是个什么东东，禁止

*d 说了是 const， 禁止

e = &a 说了是 const 禁止

const *f const =&a; 禁止

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;

or

a = a^b;// 只能对 int,char..

b = a^b;

a = a^b;

or

a ^= b ^= a;

c 和 c++中的 struct 有什么不同？

c 和 c++中 struct 的主要区别是 c 中的 struct 不可以含有成员函数，而 c++中的 struct 可以。c++中 struct 和 class 的主要区别在于默认的存取权限不同，struct 默认为 public，而 class 默认为 private。

```
#include <stdio.h>
#include <stdlib.h>
void getmemory(char *p)
{
   p=(char *) malloc(100);
   strcpy(p,"hello world");
}
int main( )
{
   char *str=NULL;
```

```
    getmemory(str);
    printf("%s/n",str);
    free(str);
    return 0;
}
```

程序崩溃，getmemory 中的 malloc 不能返回动态内存， free（）对 str 操作很危险

```
char szstr[10];
  strcpy(szstr,"0123456789");
```

产生什么结果？为什么？

长度不一样，会造成非法的 OS

列举几种进程的同步机制，并比较其优缺点。

原子操作
信号量机制
自旋锁
管程，会合，分布式系统

进程之间通信的途径

共享存储系统
消息传递系统
管道：以文件系统为基础

进程死锁的原因

资源竞争及进程推进顺序非法

死锁的 4 个必要条件

互斥、请求保持、不可剥夺、环路

死锁的处理

鸵鸟策略、预防策略、避免策略、检测与解除死锁

操作系统中进程调度策略有哪几种？

FCFS(先来先服务)，优先级，时间片轮转，多级反馈

类的静态成员和非静态成员有何区别？

类的静态成员每个类只有一个，非静态成员每个对象一个

纯虚函数如何定义？使用时应注意什么？

```
virtual void f()=0;
```

是接口，子类必须要实现

数组和链表的区别

数组：数据顺序存储，固定大小

连表：数据可以随机存储，大小可动态改变

ISO 的七层模型是什么？tcp/udp 是属于哪一层？tcp/udp 有何优缺点？

应用层

表示层

会话层

运输层

网络层

物理链路层

物理层

tcp /udp 属于运输层

TCP 服务提供了数据流传输、可靠性、有效流控制、全双工操作和多路复用技术等。

与 TCP 不同， UDP 并不提供对 IP 协议的可靠机制、流控制以及错误恢复功能等。由于 UDP 比较简单， UDP 头包含很少的字节，比 TCP 负载消耗少。

tcp: 提供稳定的传输服务，有流量控制，缺点是包头大，冗余性不好

udp: 不提供稳定的服务，包头小，开销小

1：(void *)ptr 和 (*(void**))ptr 的结果是否相同？其中 ptr 为同一个指针(void *)ptr 和 (*(void**))ptr 值是相同的

2：

```
int main()
{
   int x=3;
   printf("%d",x);
   return 1;
}
```

问函数既然不会被其它函数调用，为什么要返回 1？

mian 中，c 标准认为 0 表示成功，非 0 表示错误。具体的值是某中具体出错信息

要对绝对地址 0x100000 赋值，我们可以用(unsigned int*)0x100000 = 1234;那么要是想让程序跳转到绝对地址是 0x100000 去执行，应该怎么做？

*((void (*)( ))0x100000) ( );

首先要将 0x100000 强制转换成函数指针,即:

(void (*)())0x100000

然后再调用它:

*((void (*)())0x100000)();

用 typedef 可以看得更直观些:

typedef void(*)() voidFuncPtr;

*((voidFuncPtr)0x100000)();

已知一个数组 table，用一个宏定义，求出数据的元素个数

#define NTBL

#define NTBL (sizeof(table)/sizeof(table[0]))

面试题: 线程与进程的区别和联系？线程是否具有相同的堆栈? dll 是否有独立的堆栈?

进程是死的，只是一些资源的集合，真正的程序执行都是线程来完成的，程序启动的时候操作系统就帮你创建了一个主线程。

每个线程有自己的堆栈。DLL 中有没有独立的堆栈？

这个问题不好回答，或者说这个问题本身是否有问题。因为 DLL 中的代码是被某些线程所执行，只有线程拥有堆栈，如果 DLL 中的代码是 EXE 中的线程所调用，那么这个时候是不是说这个 DLL 没有自己独立的堆栈？如果 DLL 中的代码是由 DLL 自己创建的线程所执行，那么是不是说 DLL 有独立的堆栈？

以上讲的是堆栈，如果对于堆来说，每个 DLL 有自己的堆，所以如果是从 DLL 中动态分配的内存，最好是从 DLL 中删除，如果你从 DLL 中分配内存，然后在 EXE 中，或者另外一个 DLL 中删除，很有可能导致程序崩溃。

```
unsigned short A = 10;
printf("~A = %u\n", ~A);

char c=128;
printf("c=%d\n",c);
```

输出多少？并分析过程

第一题，～A ＝0xfffffff5,int 值 为－11，但输出的是 uint。所以输出 4294967285

第二题，c＝0x10,输出的是 int，最高位为 1，是负数，所以它的值就是 0x00 的补码就是 128，所以输出－128。

这两道题都是在考察二进制向 int 或 uint 转换时的最高位处理。

分析下面的程序：

```
void GetMemory(char **p,int num)
{
   *p=(char *)malloc(num);
}
int main()
{
    char *str=NULL;
   GetMemory(&str,100);
```

```
    strcpy(str,"hello");
    free(str);
    if(str!=NULL)
    {
       strcpy(str,"world");
    }
    printf("\n str is %s",str);软件开发网  www.mscto.com
    getchar();
}
```

问输出结果是什么？

输出 str is world。

free 只是释放的 str 指向的内存空间,它本身的值还是存在的.所以 free 之后，有一个好的习惯就是将 str=NULL.

此时 str 指向空间的内存已被回收,如果输出语句之前还存在分配空间的操作的话,这段存储空间是可能被重新分配给其他变量的,

尽管这段程序确实是存在大大的问题（上面各位已经说得很清楚了），但是通常会打印出 world 来。

这是因为，进程中的内存管理一般不是由操作系统完成的，而是由库函数自己完成的。

当你 malloc 一块内存的时候，管理库向操作系统申请一块空间（可能会比你申请的大一些），然后在这块空间中记录一些管理信息（一般是在你申请的内存前面一点），并将可用内存的地址返回。但是释放内存的时候，管理库通常都不会将内存还给操作系统，因此你是可以继续访问这块地址的。

char a[10],strlen(a)为什么等于 15？运行的结果

```
#include "stdio.h"
#include "string.h"

void main()
{
   char aa[10];
   printf("%d",strlen(aa));
软件开发网  www.mscto.com

}
```

sizeof()和初不初始化，没有关系；

strlen()和初始化有关。

char (*str)[20];/*str 是一个数组指针，即指向数组的指针．*/

char *str[20];/*str 是一个指针数组，其元素为指针型数据．*/

long a=0x801010;

a+5=?

0x801010 用二进制表示为：“1000 0000 0001 0000 0001 0000”，十进制的值为 8392720，再加上 5 就是 8392725 罗

1)给定结构

```
struct A
{
  char t:4;
  char k:4;
  unsigned short i:8;
  unsigned long m;
};
```

问 sizeof(A) = ?

给定结构

```
struct A
{
  char t:4; 4 位
  char k:4; 4 位
  unsigned short i:8; 8 位
   unsigned long m; //  偏移 2 字节保证 4 字节对齐
}; //  共 8 字节
```

2)下面的函数实现在一个数上加一个数，有什么错误？请改正。

```
int add_n ( int n )
{
   static int i = 100;
   i += n;
   return i;
}
```

当你第二次调用时得不到正确的结果，难道你写个函数就是为了调用一次？问题就出在 static 上？

```
// 帮忙分析一下
#include<iostream.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
typedef struct   AA
{
    int b1:5;
     int b2:2;
}AA;
void main()
{
   AA aa;
   char cc[100];
   strcpy(cc,"0123456789abcdefghijklmnopqrstuvwxyz");
   memcpy(&aa,cc,sizeof(AA));
   cout << aa.b1 <<endl;
```



```
   cout << aa.b2 <<endl;
}
```

答案是 -16 和 1

首先 sizeof(AA)的大小为 4,b1 和 b2 分别占 5bit 和 2bit.经过 strcpy 和 memcpy 后,aa 的 4 个字节所存放的值是: 0,1,2,3 的 ASC 码，即 00110000,00110001,00110010,00110011 所以，最后一步：显示的是这 4 个字节的前 5 位，和之后的 2 位分别为：10000,和 01，因为 int 是有正负之分

所以：答案是-16 和 1

求函数返回值，输入 x=9999;

```
int func （ x ）
```

```
{
    int countx = 0;
     while ( x )
     {
        countx ++;
       x = x&(x-1);
     }
     return countx;
}
```

结果呢？

知道了这是统计 9999 的二进制数值中有多少个 1 的函数，且有 9999＝9×1024＋512＋256＋15

9×1024 中含有 1 的个数为 2；
512 中含有 1 的个数为 1；
256 中含有 1 的个数为 1；
15 中含有 1 的个数为 4；软件开发网 www.mscto.com
故共有 1 的个数为 8，结果为 8。
1000 - 1 = 0111，正好是原数取反。这就是原理。
用这种方法来求 1 的个数是很效率很高的。
不必去一个一个地移位。循环次数最少。

int a,b,c 请写函数实现 C=a+b ,不可以改变数据类型,如将 c 改为 long int,关键是如何处理溢出问题

```
bool add (int a, int b,int *c)
{
   *c=a+b;
   return (a>0 && b>0 &&(*c<a || *c<b) || (a<0 && b<0 &&(*c>a || *c>b)));
}
```

分析：

```
struct bit
{
   int a:3;
   int   b:2;
    int c:3;
};
```

```
int main()
{
  bit s;
  char *c=(char*)&s;
   cout<<sizeof(bit)<<endl;
  *c=0x99;
   cout << s.a <<endl <<s.b<<endl<<s.c<<endl;
      int a=-1;
    printf("%x",a);
  return 0;
}
```

输出为什么是

4

1

-1

-4

ffffffff

因为 0x99 在内存中表示为 100 11 001 , a = 001, b = 11, c = 100。当 c 为有符合数时, c = 100, 最高 1 为表示 c 为负数，负数在计算机用补码表示，所以 c = -4;同理 b = -1;当 c 为有符合数时, c = 100,即 c = 4, 同理 b = 3。

位域 ：

有些信息在存储时，并不需要占用一个完整的字节， 而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态， 用一位二进位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为“位域”或“位段”。所谓“位域”是把一个字节中的二进位划分为几个不同的区域， 并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。 这样就可以把几个不同的对象用一个字节的二进制位域来表示。一、位域的定义和位域变量的说明位域定义与结构定义相仿，其形式为：

struct 位域结构名 { 位域列表 }; 其中位域列表的形式为： 类型说明符 位域名：位域长度

例如：

```
struct bs
{
```

```
    int a:8;
    int b:2;
    int c:6;
};
```

位域变量的说明与结构变量说明的方式相同。 可采用先定义后说明，同时定义说明或者直接说明这三种方式。例如：

```
struct bs
{
    int a:8;
    int b:2;
    int c:6;
}data;
```

说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。对于位域的定义尚有以下几点说明：

一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs
{
    unsigned a:4
    unsigned :0 /*空域*/
    unsigned b:4 /*从下一单元开始存放*/
```



```
    unsigned c:4
}
```

在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进位。

位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k
```

```
{
   int a:1
   int :2 /*该 2 位不能使用*/
   int b:3
   int c:2
};
```

从以上分析可以看出，位域在本质上就是一种结构类型， 不过其成员是按二进位分配的。

位域的使用位域的使用和结构成员的使用相同，其一般形式为： 位域变量名•位域名
位域允许用各种格式输出。

```
main()
{
   struct bs
   {
      unsigned a:1;
```



```
      unsigned b:3;
      unsigned c:4;
   }
   bit,*pbit;
   bit.a=1;
   bit.b=7;
   bit.c=15;
   pri
```

改错：

```
#include <stdio.h>
int main(void)
{
   int **p;
   int arr[100];
   p = &arr;
   return 0;
}
```

解答：搞错了,是指针类型不同,int **p; //二级指针&arr; //得到的是指向第一维为 100 的数组的指针

```
#include <stdio.h>
int main(void)
{
   int **p, *q;
   int arr[100];
   q = arr;
   p = &q;
   return 0;
}
```

下面这个程序执行后会有什么错误或者效果:

```
#define MAX 255
int main()
{
   unsigned char A[MAX],i;//i 被定义为 unsigned char
   for (i=0;i<=MAX;i++)
   A[i]=i;



}
```

解答：死循环加数组越界访问（C/C++不进行数组越界检查）MAX=255 数组 A 的下标范围为:0..MAX-1,这是其一..

其二.当 i 循环到 255 时,循环内执行:A[255]=255;这句本身没有问题..但是返回 for (i=0;i<=MAX;i++)语句时,由于 unsigned char 的取值范围在(0..255),i++以后 i 又为 0 了..无限循环下去。

```
struct name1
{
   char   str;
   short x;
   int    num;
}
struct name2
{
   char str;
```

```
    int num;
    short x;
}
```

sizeof(struct name1)=8,sizeof(struct name2)=12

在第二个结构中，为保证 num 按四个字节对齐，char 后必须留出 3 字节的空间；同时为保证整个结构的自然对齐（这里是 4 字节对齐），在 x 后还要补齐 2 个字节，这样就是 12 字节。

intel：

A.c 和 B.c 两个 c 文件中使用了两个相同名字的 static 变量,编译的时候会不会有问题?这两个 static 变量会保存到哪里（栈还是堆或者其他的）？

static 的全局变量，表明这个变量仅在本模块中有意义，不会影响其他模块。他们都放在数据区，但是编译器对他们的命名是不同的。如果要使变量在其他模块也有意义的话，需要使用 extern 关键字。

```
struct s1
{
    int i: 8;
    int j: 4;
    int a: 3;
    double b;
};

struct s2
{
    int i: 8;
    int j: 4;
    double b;
    int a:3;
};

printf("sizeof(s1)= %d\n", sizeof(s1));
printf("sizeof(s2)= %d\n", sizeof(s2));
result: 16, 24
第一个 struct s1
{
    int i: 8;
```

```
    int j: 4;
    int a: 3;
    double b;
};
```

理论上是这样的，首先是 i 在相对 0 的位置，占 8 位一个字节，然后，j 就在相对一个字节的位置，由于一个位置的字节数是 4 位的倍数，因此不用对齐，就放在那里了，然后是 a，要在 3 位的倍数关系的位置上，因此要移一位，在 15 位的位置上放下，目前总共是 18 位，折算过来是 2 字节 2 位的样子，由于 double 是 8 字节的，因此要在相对 0 要是 8 个字节的位置上放下，因此从 18 位开始到 8 个字节之间的位置被忽略，直接放在 8 字节的位置了，因此，总共是 16 字节。

第二个最后会对照是不是结构体内最大数据的倍数，不是的话，会补成是最大数据的倍数。

85. C 语言常用宏定义

发布: 2008-6-29 22:01 | 作者: 剑心通明 | 来源: 互联网 | 查看: 11 次

01:防止一个头文件被重复包含

```
#ifndef COMDEF_H
#define COMDEF_H
//头文件内容
#endif
```

02:重新定义一些类型,防止由于各种平台和编译器的不同,而产生的类型字节数差异,方便移植。

```
typedef  unsigned char       boolean;     /* Boolean value type. */
typedef  unsigned long int  uint32;       /* Unsigned 32 bit value */
typedef  unsigned short     uint16;       /* Unsigned 16 bit value */
typedef  unsigned char       uint8;        /* Unsigned 8  bit value */
typedef  signed long int     int32;        /* Signed 32 bit value */
typedef  signed short        int16;        /* Signed 16 bit value */
typedef  signed char         int8;        /* Signed 8  bit value */

//下面的不建议使用
typedef  unsigned char     byte;          /* Unsigned 8  bit value type. */
typedef  unsigned short     word;          /* Unsinged 16 bit value type. */
typedef  unsigned long     dword;         /* Unsigned 32 bit value type. */
typedef  unsigned char     uint1;        /* Unsigned 8  bit value type. */
typedef  unsigned short     uint2;        /* Unsigned 16 bit value type. */
typedef  unsigned long     uint4;        /* Unsigned 32 bit value type. */
typedef  signed char         int1;         /* Signed 8  bit value type. */
typedef  signed short       int2;         /* Signed 16 bit value type. */
typedef  long int           int4;         /* Signed 32 bit value type. */
typedef  signed long         sint31;       /* Signed 32 bit value */
typedef  signed short       sint15;       /* Signed 16 bit value */
typedef  signed char         sint7;        /* Signed 8  bit value */
```

03:得到指定地址上的一个字节或字

```
#define  MEM_B(x) (*((byte *)(x)))
#define  MEM_W(x) (*((word *)(x)))
```

04:求最大值和最小值

```
#define  MAX(x,y) (((x)>(y)) ? (x) : (y))
```

```
#define  MIN(x,y) (((x) < (y)) ? (x) : (y))
```

05:得到一个 field 在结构体(struct)中的偏移量

```
#define FPOS(type,field) ((dword)&((type *)0)->field)
```

06:得到一个结构体中 field 所占用的字节数

```
#define FSIZ(type,field) sizeof(((type *)0)->field)
```

07:按照 LSB 格式把两个字节转化为一个 Word

```
#define FLIPW(ray) ((((word)(ray)[0]) * 256) + (ray)[1])
```

08:按照 LSB 格式把一个 Word 转化为两个字节

```
#define FLOPW(ray,val) (ray)[0] = ((val)/256); (ray)[1] = ((val) & 0xFF)
```

09:得到一个变量的地址（word 宽度）

```
#define B_PTR(var)  ((byte *) (void *) &(var))
#define W_PTR(var)  ((word *) (void *) &(var))
```

10:得到一个字的高位和低位字节

```
#define WORD_LO(xxx)  ((byte) ((word)(xxx) & 255))
#define WORD_HI(xxx)  ((byte) ((word)(xxx) >> 8))
```

11:返回一个比 X 大的最接近的 8 的倍数

```
#define RND8(x) ((((x) + 7)/8) * 8)
```

12:将一个字母转换为大写

```
#define UPCASE(c) (((c)>='a' && (c) <= 'z') ? ((c) - 0x20) : (c))
```

13:判断字符是不是 10 进值的数字

```
#define  DECCHK(c) ((c)>='0' && (c)<='9')
```

14:判断字符是不是 16 进值的数字

```
#define HEXCHK(c) (((c) >= '0' && (c)<='9') ((c)>='A' && (c)<= 'F') \
((c)>='a' && (c)<='f'))
```

15:防止溢出的一个方法

```
#define INC_SAT(val) (val=((val)+1>(val)) ? (val)+1 : (val))
```

16:返回数组元素的个数

#define ARR_SIZE(a) (sizeof((a))/sizeof((a[0])))

17:返回一个无符号数 n 尾的值 MOD_BY_POWER_OF_TWO(X,n)=X%(2^n)

#define MOD_BY_POWER_OF_TWO(val, mod_by) ((dword)(val) & (dword)((mod_by)-1))

18:对于 IO 空间映射在存储空间的结构,输入输出处理

#define inp(port) (*((volatile byte *)(port)))

#define inpw(port) (*((volatile word *)(port)))

#define inpdw(port) (*((volatile dword *)(port)))

#define outp(port,val) (*((volatile byte *)(port))=((byte)(val)))

#define outpw(port, val) (*((volatile word *)(port))=((word)(val)))

#define outpdw(port, val) (*((volatile dword *)(port))=((dword)(val)))

19:使用一些宏跟踪调试

ANSI 标准说明了五个预定义的宏名。它们是：

__LINE__

__FILE__

__DATE__

__TIME__

__STDC__

C++中还定义了 __cplusplus

如果编译器不是标准的,则可能仅支持以上宏名中的几个,或根本不支持。记住编译程序也许还提供其它预定义的宏名。

__LINE__ 及 __FILE__ 宏指示，#line 指令可以改变它的值，简单的讲，编译时，它们包含程序的当前行数和文件名。

__DATE__ 宏指令含有形式为月/日/年的串,表示源文件被翻译到代码时的日期。

__TIME__ 宏指令包含程序编译的时间。时间用字符串表示，其形式为： 分：秒

__STDC__ 宏指令的意义是编译时定义的。一般来讲，如果__STDC__已经定义，编译器将仅接受不包含任何非标准扩展的标准 C/C++代码。如果实现是标准的,则宏__STDC__含有十进制常量 1。如果它含有任何其它数,则实现是非标准的。

__cplusplus 与标准 c++一致的编译器把它定义为一个包含至少 6 为的数值。与标准 c++不一致的编译器将使用具有 5 位或更少的数值。

可以定义宏,例如:

当定义了_DEBUG,输出数据信息和所在文件所在行

```
#ifdef _DEBUG
#define DEBUGMSG(msg,date) printf(msg);printf("%d%d%d",date,_LINE_,_FILE_)
#else
#define DEBUGMSG(msg,date)
#endif
```

20：宏定义防止错误使用小括号包含。

例如：

有问题的定义：#define DUMP_WRITE(addr,nr) {memcpy(bufp,addr,nr); bufp += nr;}

应该使用的定义： #difne DO(a,b) do{a+b;a++;}while(0)

例如：

```
if(addr)
    DUMP_WRITE(addr,nr);
else
    do_somethong_else();
```

宏展开以后变成这样:

```
if(addr)
    {memcpy(bufp,addr,nr); bufp += nr;};
else
    do_something_else();
```

gcc 在碰到 else 前面的"；"时就认为 if 语句已经结束，因而后面的 else 不在 if 语句中。而采用 do{} while(0)的定义，在任何情况下都没有问题。而改为 #difne DO(a,b) do{a+b;a++;}while(0) 的定义则在任何情况下都不会出错。

来自：http://blog.chinaunix.net/u/12467/showart.php?id=1009960