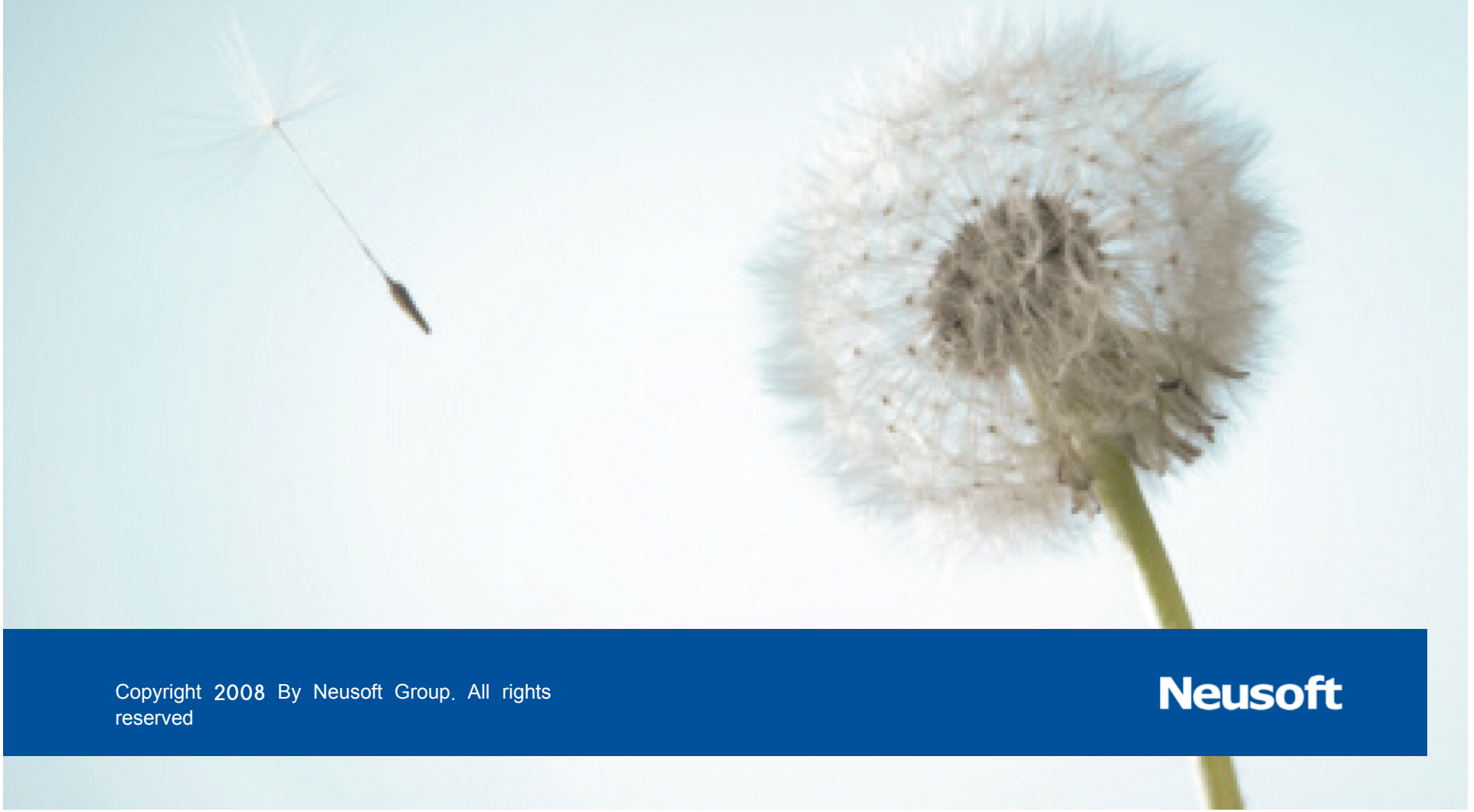
C语言编程效率



Neusoft

目标:

- 认识到编程效率的重要性
- 掌握提高编程方法

学时：2 学时
教学方法：讲授ppt + 上机练习 + 点评

课程概述

- 关于程序效率
- 如何提高程序效率
- 高效率编程的几个技巧
- 实例

1 关于程序效率

※ 实例分析什么是程序效率

没有意识到程序效率的编码者，可能会写如下的代码：

```
for( i=0; i<1000; i++ ){
    GetLocalHostName( hostname );
    ...
 }
```

GetLocalHostName的意思是取得当前计算机名。实际上取得一次机器的名字就可以，而把它放在循环体中，它就被调用了1000次，其中999次是多余的。

如果不意识到无效率行为的危害，程序中多次出现如此没有效率的行为，那么程序运行效率会非常慢，有时会慢得使用用户难以接受。

1 关于程序效率

※什么是程序效率

- 程序效率，是用执行的步骤（step）数——时间复杂度、占内存的多少来衡量的——空间复杂度。
- 完成某项工作，执行的步骤（step）的次数最少、占用内存最小是程序员所追求的。
- 特别是嵌入式系统的开发，内存等资源都是有限的。

因此，提高效率的着眼点应该是

- 减少执行次数
- 减少占用空间

2 如何提高程序效率

★效率改善的指导原则

- 满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率；
  - 如果程序的正确性、可靠性得不到保证，提高效率就失去了根本；
  - 如果程序的健壮性得不到保证，提高效率就失去了目标；
  - 如果程序的可读性得不到保证，提高效率就失去了方向；

2 如何提高程序效率

※ 以提高程序的全局效率为主，提高局部效率为辅

- 如果只从局部角度出发，局部效率的改善一般对全局效率改善作用不大，有时甚至影响全局效率的改善；
- 应该从全局角度出发，整体考虑，作出统一改善的调整，有的时候局部利益为配合整体需要应作出牺牲；

2 如何提高程序效率

※在优化程序的效率时，应当先找出限制效率的“瓶颈”

- 非关键点的改善，不会根本改变效率的现状；
- 先进行一些基准数据测量和问题收集，寻找提高效率的关键点；
- 有时一次关键的改动还不能解决问题，还需要进一步的数据测量和持续的改进，直到符合要求；

2 如何提高程序效率

※先优化数据结构和算法，再优化执行代码

- 因为$O(n^2)$的算法写不出$O(n\log_2 n)$的程序，因此写程序前应该考虑数据结构和算法的选择和优化；

- 如果已经选择了正确的算法，那么只有到了写程序的最后，才有可能去关心执行代码的优化问题；

2 如何提高程序效率

※ 时间效率和空间效率可能对立，此时应当分析那个更重要，作出适当的折衷

- 一般来讲，在空间允许的时候，我们会花费空间换取时间效率的大幅提升；
- 当空间受限－－时间效率和空间对立的时候，我们根据需要，在两者之间作出适当折中；

2 如何提高程序效率

※ 时间效率改善的考虑

- 减少内存分配的次数
  - 一次能够申请的内存，就不要多次申请；
  - 被多次访问函数中存储“不变量”的变量请定义成为static。如：

```
GetLocalHostName(char* name)
{
    char funcName[] = “GetLocalHostName”;
    sys_log( “%s begin.....”, funcName );
    ...
    sys_log( “%s end.....”, funcName );
}
```

如果这是一个经常调用的函数，每次调用时都要funcName进行分配内存，这个开销很大。把这个变量声明成static，当函数再次被调用时，就会省去了分配内存的开销，执行效率也很好。

2 如何提高程序效率

※提高循环体效率

- 减少循环次数
- 减少循环体内执行语句的数量
- 在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少CPU 跨越循环层的次数

下面例子a和例子b的功能一样，但效率不同

例a:
```
for (i=0; i<5;i++)
{
    for (j=0; j<100; j++)
    {
    Dothing();
    }
}
```

例b:
```
for (j=0; j<100;j++)
{
    for (i=0; i<5; i++)
    {
Dothing();
    }
}
```

2 如何提高程序效率(续)

※减少指针定位

- 指针(->)使用很方便，但实际运行往往需要进行地址计算；不必要的地址计算会导致性能的下降；
- 尽可能减少->的多级嵌套。

※提高数学表达式的效率

- a*b + a*c = a*(b+c); 减少一次乘法，但不改变表达式的意义。
- b/a + c/a = (1/a)*(b+c); 把两次除法变成一次除法和一次乘法，在几乎所有的平台上，除法都比乘法慢。
- (a || b) && c = c && (a || b); 当c为假时，第一个表达式要计算(a || b)，第二个表达式则不用计算。

2 如何提高程序效率

※空间效率改善的考虑

– 合理结构体成员定义顺序：按照类型从小到大，相同类型连续存放

例a和例b的占用的空间是不同的：

```
例 a
typedef  struct
{
        short   a;
        int       b;
        short   c;
 }AA_t;
```

```
例 b
 typedef struct
{
        short   a;
        short   c;
        int       b;
 } BB_t
```

2 如何提高程序效率

※冗余数据压缩，可以减少对内存的占用

比如矩阵数据的存储。有些矩阵中的非零元素很少，可以考虑通过只存储非零数据来减少对存储空间的占用。

※动态内存使用的方式，可以提高内存的利用率

追求空间的效率和追求时间的效率，往往是矛盾的，需要全面权衡。

2 如何提高程序效率

※减少代码的行数可以减少ROM的占用

对于嵌入系统而言，ROM资源是很宝贵的。减少代码行数是减少ROM占用量的有效途径。

减少代码行的方法:

- 消除冗余代码可以有效减少代码行数
- 通过函数指针和函数表可以减少程序代码的规模

3 高效率编程几个技巧

※时间与空间问题

- 在嵌入式领域,我们经常会遇到时间和空间上的问题, 有的是实时性要求比较高的那么就需要利用空间来换取时间.
  - 方法A:
    #define LEN 32
    char string1 [LEN];
    memset (string1,0,LEN);
    strcpy (string1,“This is a example!!”）;
  - 方法B:
    const char string2[LEN] =“This is a example!”;
    char * cp;
    cp = string2 ;
    虽然这种方法比较笨拙,缺乏灵活性,但在特殊的情况下效率是比较高的.

3 高效率编程几个技巧

※宏的问题

- 一般情况下,不建议使用宏,因为宏破坏了程序的可读性,特别是代码中潜入很多宏的时候排错就很麻烦。

- 但是如果在特殊的情况下,比如追求效率的情况下还是可以使用宏定义的, 一般情况下将这个工作交给编译器来完成, 因为现代的编译器会自动的把小函数或内联函数扩展为宏。(一般只用宏来进行定义常数, 条件编译工作)。

3 高效率编程几个技巧

※采用数学的方法来优化程序

- 有时候采用数学方法来优化程序常常被大家忽略, 对于没有经验的程序员来说更是如此, 然而这对效率的影响是很大的
- 一个经典的例子:
  求 1~100的和。
  - 方法A:

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

  - 方法B:

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

  - 很显然这里的差别也很大, 方法B的代码也更简洁。
  - 呵呵^_^,所以程序员数学好也非常有利于今后的发展。

3 高效率编程几个技巧

※使用位操作来代替乘法和除法

- 大家都知道乘法和除法在cpu中是不能直接进行的，cpu采用加法来现，所以在适当的情况下要使用位移来实现乘法和除法。
- 如
  a=211/4; 也可以写成a=211>>2;(这里的适当时候，当然是运算式不是很复杂的时候，
- 如果需要都用位移来操作,那么调程序的时候会痛苦死的^_^)!

3 高效率编程几个技巧

※汇编潜入

- 大家都知道在计算机里,汇编是跑的最快的程序语言,使用汇编可以时程序的效率提高一定的数量级
- 但是所谓事物的两面性, 使用汇编也减低的程序的可移植性, 而且很可能对程序造成巨大的破坏
- 建议高手使用,而且也仅仅限与对底层处理或性能要求非常苛刻的时候

4 实例

※效率改善的例A

– 请阅读下面的代码

```
for( int i = 0; i < numPixels;i++ )
{
  rendering_context->back_buffer->
            surface->bits[i] = some_value;
}
```

– 做如下修改是否更好

```
unsigned char *back_surface_bits =
            rendering_context->
            back_buffer->surface->bits;
for( int i = 0; i < numPixels; i++ )
{
   back_surface_bits[i] = some_value;
}
```

4 实例

还可以进一步修改

```
unsigned char *back_surface_bits =
                rendering_context->
                back_buffer>surface->bits;

for( int i = 0; i <numPixels;i++,back_surface_bits++ )
{
   *back_surface_bits = some_value;
 }
```

4 实例

※效率改善的例B

问题：有一组处理函数:functionA, functionB,…functionZ,它们的函数形式如下

```
int functionA(int event);
int functionB(int event);
……
int functionZ(int event);
```

他们分别是不同状态(A,B…,Z)的处理。编写这段处理代码，我们该如何做?

4 实例

下面代码可行吗?

```
switch (status) {
        case A:    functionA(event)
                   break;
        case B:    functionB(event)
                   break;
        ……
        case Z:    functionZ(event)
                   break;
}
```

4 实例

可行！但是不好！原因是生成目标代码大，而且可维护性较弱。

这么做可以解决上面提到的缺点

```
typedef  int (*pFunc)(int event);
  typedef enum {   A =0,
                   B,
                   …
                   Z
                } Status_t;
   pFunc  functionlist[Z+1] ={   functionA,functionB,
                                    …
                                 functionZ };
   Status_t   status;
      ……
   status被赋值
         functionlist[status](event);
```

这么做是不是更好?

4 实例

- 请问(i)比 (ii)哪个效率高?

(i)
```
if (condition)
  {   for ( i=0; i< N; i ++)
          DoSomething();   }
 else
  {   for ( i=0; i< N; i ++)
          DoOtherthing();
  }
```

(ii)
```
for ( i=0; i< N; i ++)
  {       if (condition)
            DoSomething();
                else
            DoOtherthing();
  }
```

4 实例

- 改善下面这段程序

```
int f( long n )
{
        int i, s;
        for (s = 0, i = 1;i<= n; i++)
                s += i;
        return( s );
}
```