```
  ,
  {
    " menu4", 7, 48, menu4OnOk, menu4OnCancel
  }
  …
};
```

OK 键和 CANCEL 键的处理变成：

```
/* 按下 OK 键 */
void onOkKey()
{
  menu[currentFocusMenu].onOkFun();
}
/* 按下 Cancel 键 */
void onCancelKey()
{
  menu[currentFocusMenu].onCancelFun();
}
```

程序被大大简化了，也开始具有很好的可扩展性！我们仅仅利用了面向对象中的封装思想，就让程序结构清晰，其结果是几乎可以在无需修改程序的情况下在系统中添加更多的菜单，而系统的按键处理函数保持不变。

面向对象，真神了！

模拟 MessageBox 函数

MessageBox 函数，这个 Windows 编程中的超级猛料，不知道是多少入门者第一次用到的函数。还记得我们第一次在 Windows 中利用 MessageBox 输出 "Hello,World!"对话框时新奇的感觉吗？无法统计，这个世界上究竟有多少程序员学习 Windows 编程是从 MessageBox ("Hello,World!",…)开始的。在我本科的学校，广泛流传着一个词汇，叫做"’Hello,World’级程序员"，意指入门级程序员，但似乎"’Hello,World’级"这个说法更搞笑而形象。

图 2 经典的 Hello,World!

图 2 给出了两种永恒经典的 Hello,World 对话框，一种只具有"确定"，一种则包含"确定"、"取消"。是的，MessageBox 的确有，而且也应该有两类！这完全是由特定的应用需求决定的。

嵌入式系统中没有给我们提供 MessageBox，但是鉴于其功能强大，我们需要模拟之，一个模拟的 MessageBox 函数为：

```
/*****************************************
/* 函数名称: MessageBox
/* 功能说明: 弹出式对话框,显示提醒用户的信息
/* 参数说明: lpStr --- 提醒用户的字符串输出信息
/* TYPE --- 输出格式(ID_OK = 0, ID_OKCANCEL = 1)
/* 返回值: 返回对话框接收的键值,只有两种 KEY_OK, KEY_CANCEL
/*****************************************
typedef enum TYPE { ID_OK,ID_OKCANCEL }MSG_TYPE;
extern BYTE MessageBox(LPBYTE lpStr, BYTE TYPE)
{
  BYTE keyValue = -1;

  ClearScreen(); /* 清除屏幕 */
  DisplayString(xPos,yPos,lpStr,TRUE); /* 显示字符串 */
  /* 根据对话框类型决定是否显示确定、取消 */
  switch (TYPE)
  {
    case ID_OK:
      DisplayString(13,yPos+High+1, " 确定 ", 0);
      break;
    case ID_OKCANCEL:
      DisplayString(8, yPos+High+1, " 确定 ", 0);
      DisplayString(17,yPos+High+1, " 取消 ", 0);
      break;
    default:
      break;
  }
  DrawRect(0, 0, 239, yPos+High+16+4); /* 绘制外框 */
  /* MessageBox 是模式对话框，阻塞运行，等待按键 */
  while( (keyValue != KEY_OK) || (keyValue != KEY_CANCEL) )
  {
    keyValue = getSysKey();
  }
  /* 返回按键类型 */
  if(keyValue== KEY_OK)
  {
    return ID_OK;
  }
  else
  {
    return ID_CANCEL;
  }
}
```

上述函数与我们平素在 VC++等中使用的 MessageBox 是何等的神似啊？实现这个函数，你会看到它在嵌入式系统中的妙用是无穷的。

总结

本篇是本系列文章中技巧性最深的一篇，它提供了嵌入式系统屏幕显示方面一些很巧妙的处理方法，灵活使用它们，我们将不再被 LCD 上凌乱不堪的显示内容所困扰。

屏幕乃嵌入式系统生存之重要辅助，面目可憎之显示将另用户逃之夭夭。屏幕编程若处理不好，将是软件中最不系统、最混乱的部分，笔者曾深受其害。.

C 语言嵌入式系统编程修炼之五:键盘操作

作者:宋宝华　　更新日期:2005-07-22

处理功能键

功能键的问题在于，用户界面并非固定的，用户功能键的选择将使屏幕画面处于不同的显示状态下。例如，主画面如图 1：

图 1 主画面

当用户在设置 XX 上按下 Enter 键之后，画面就切换到了设置 XX 的界面，如图 2：

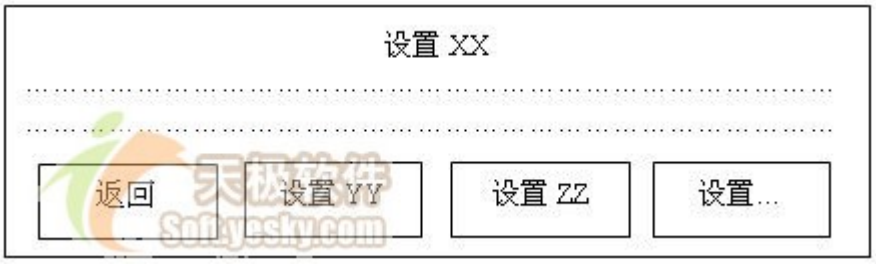

图 2 切换到设置 XX 画面

程序如何判断用户处于哪一画面，并在该画面的程序状态下调用对应的功能键处理函数，而且保证良好的结构，是一个值得思考的问题。

让我们来看看 WIN32 编程中用到的"窗口"概念，当消息（message）被发送给不同窗口的时候，该窗口的消息处理函数

（是一个 callback 函数）最终被调用，而在该窗口的消息处理函数中，又根据消息的类型调用了该窗口中的对应处理函数。通过这种方式，WIN32 有效的组织了不同的窗口，并处理不同窗口情况下的消息。

我们从中学习到的就是：

（1）将不同的画面类比为 WIN32 中不同的窗口，将窗口中的各种元素（菜单、按钮等）包含在窗口之中；

（2）给各个画面提供一个功能键"消息"处理函数，该函数接收按键信息为参数；

（3）在各画面的功能键"消息"处理函数中，判断按键类型和当前焦点元素，并调用对应元素的按键处理函数。

```
/* 将窗口元素、消息处理函数封装在窗口中 */
struct windows
{
  BYTE currentFocus;
  ELEMENT element[ELEMENT_NUM];
  void (*messageFun) (BYTE keyValue);
  …
};
/* 消息处理函数 */
void messageFunction(BYTE keyValue)
{
  BYTE i = 0;
  /* 获得焦点元素 */
  while ( (element [i].ID!= currentFocus)&& (i < ELEMENT_NUM) )
  {
    i++;
  }
  /* "消息映射" */
  if(i < ELEMENT_NUM)
  {
    switch(keyValue)
    {
      case OK:
        element[i].OnOk();
        break;
      …
    }
  }
}
```

在窗口的消息处理函数中调用相应元素按键函数的过程类似于"消息映射"，这是我们从 WIN32 编程中学习到的。编程到了一个境界，很多东西都是相通的了。其它地方的思想可以拿过来为我所用，是为编程中的"拿来主义"。

在这个例子中，如果我们还想玩得更大一点，我们可以借鉴 MFC 中处理 MESSAGE_MAP 的方法，我们也可以学习 MFC 定义几个精妙的宏来实现"消息映射"。

处理数字键

用户输入数字时是一位一位输入的，每一位的输入都对应着屏幕上的一个显示位置（x 坐标，y 坐标）。此外，程序还需要记录该位置输入的值，所以有效组织用户数字输入的最佳方式是定义一个结构体，将坐标和数值捆绑在一起：

```
/* 用户数字输入结构体 */
typedef struct tagInputNum
{
  BYTE byNum; /* 接收用户输入赋值 */
  BYTE xPos; /* 数字输入在屏幕上的显示位置 x 坐标 */
  BYTE yPos; /* 数字输入在屏幕上的显示位置 y 坐标 */
}InputNum, *LPInputNum;
```

那么接收用户输入就可以定义一个结构体数组，用数组中的各位组成一个完整的数字：

```
InputNum inputElement[NUM_LENGTH]; /* 接收用户数字输入的数组 */
/* 数字按键处理函数 */
extern void onNumKey(BYTE num)
{
if(num==0|| num==1) /* 只接收二进制输入 */
{
  /* 在屏幕上显示用户输入 */
                                    DrawText(inputElement[currentElementInputPlace].xPos,
inputElement[currentElementInputPlace].yPos, "%1d", num);
  /* 将输入赋值给数组元素 */
  inputElement[currentElementInputPlace].byNum = num;
  /* 焦点及光标右移 */
  moveToRight();
}
}
```

将数字每一位输入的坐标和输入值捆绑后，在数字键处理函数中就可以较有结构的组织程序，使程序显得很紧凑。

整理用户输入

继续第 2 节的例子，在第 2 节的 onNumKey 函数中，只是获取了数字的每一位，因而我们需要将其转化为有效数据，譬如要转化为有效的 XXX 数据，其方法是：

```
/* 从 2 进制数据位转化为有效数据：XXX */
void convertToXXX()
{
```

```
  BYTE i;
  XXX = 0;
  for (i = 0; i < NUM_LENGTH; i++)
  {
    XXX += inputElement[i].byNum*power(2, NUM_LENGTH - i - 1);
  }
}
```

反之，我们也可能需要在屏幕上显示那些有效的数据位，因为我们也需要能够反向转化：

```
/* 从有效数据转化为 2 进制数据位：XXX */
void convertFromXXX()
{
  BYTE i;
  XXX = 0;
  for (i = 0; i < NUM_LENGTH; i++)
  {
    inputElement[i].byNum = XXX / power(2, NUM_LENGTH - i - 1) % 2;
  }
}
```

当然在上面的例子中，因为数据是 2 进制的，用 power 函数不是很好的选择，直接用"<< >>"移位操作效率更高，我们仅是为了说明问题的方便。试想，如果用户输入是十进制的，power 函数或许是唯一的选择了。

总结

本篇给出了键盘操作所涉及的各个方面：功能键处理、数字键处理及用户输入整理，基本上提供了一个全套的按键处理方案。对于功能键处理方法，将 LCD 屏幕与 Windows 窗口进行类比，提出了较新颖地解决屏幕、键盘繁杂交互问题的方案。

计算机学的许多知识都具有相通性，因而，不断追赶时髦技术而忽略基本功的做法是徒劳无意的。我们最多需要"精通"三种语言（精通，一个在如今的求职简历里泛滥成灾的词语），最佳拍档是汇编、C、C++（或 JAVA），很显然，如果你"精通"了这三种语言，其它语言你应该是可以很快"熟悉"的，否则你就没有"精通"它们..

C 语言嵌入式系统编程修炼之六:性能优化

作者:宋宝华　　更新日期:2005-07-22

使用宏定义

在 C 语言中，宏是产生内嵌代码的唯一方法。对于嵌入式系统而言，为了能达到性能要求，宏是一种很好的代替函数的方法。

写一个"标准"宏 MIN ，这个宏输入两个参数并返回较小的一个：

错误做法：

```
#define MIN(A,B)    ( A <= B ? A : B )
```

正确做法：

```
#define MIN(A,B)  ((A) <= (B) ? (A) : (B) )
```

对于宏，我们需要知道三点：

(1)宏定义"像"函数；

(2)宏定义不是函数，因而需要括上所有"参数"；

(3)宏定义可能产生副作用。

下面的代码：

```
least = MIN(*p++, b);
```

将被替换为：

```
( (*p++) <= (b) ?(*p++):(b) )
```

发生的事情无法预料。

因而不要给宏定义传入有副作用的"参数"。

使用寄存器变量

当对一个变量频繁被读写时，需要反复访问内存，从而花费大量的存取时间。为此，C 语言提供了一种变量，即寄存器变量。这种变量存放在 CPU 的寄存器中，使用时，不需要访问内存，而直接从寄存器中读写，从而提高效率。寄存器变量的说明符是 register。对于循环次数较多的循环控制变量及循环体内反复使用的变量均可定义为寄存器变量，而循环计数是应用寄存器变量的最好候选者。

(1) 只有局部自动变量和形参才可以定义为寄存器变量。因为寄存器变量属于动态存储方式，凡需要采用静态存储方式的量都不能定义为寄存器变量，包括：模块间全局变量、模块内全局变量、局部 static 变量；

(2) register 是一个"建议"型关键字，意指程序建议该变量放在寄存器中，但最终该变量可能因为条件不满足并未成为寄存器变量，而是被放在了存储器中，但编译器中并不报错（在 C++语言中有另一个"建议"型关键字：inline）。

下面是一个采用寄存器变量的例子：

```
/* 求 1+2+3+….+n 的值 */
WORD Addition(BYTE n)
{
  register i,s=0;
  for(i=1;i<=n;i++)
  {
    s=s+i;
  }
  return s;
}
```

本程序循环 n 次，i 和 s 都被频繁使用，因此可定义为寄存器变量。

内嵌汇编

程序中对时间要求苛刻的部分可以用内嵌汇编来重写，以带来速度上的显著提高。但是，开发和测试汇编代码是一件辛苦的工作，它将花费更长的时间，因而要慎重选择要用汇编的部分。

在程序中，存在一个 80-20 原则，即 20%的程序消耗了 80%的运行时间，因而我们要改进效率，最主要是考虑改进那 20%的代码。

嵌入式 C 程序中主要使用在线汇编，即在 C 程序中直接插入_asm{ }内嵌汇编语句：

```
/* 把两个输入参数的值相加，结果存放到另外一个全局变量中 */
int result;
void Add(long a, long *b)
{
  _asm
  {
    MOV AX, a
    MOV BX, b
    ADD AX, [BX]
    MOV result, AX
  }
}
```

利用硬件特性

首先要明白 CPU 对各种存储器的访问速度，基本上是：

CPU 内部 RAM > 外部同步 RAM > 外部异步 RAM > FLASH/ROM

对于程序代码，已经被烧录在 FLASH 或 ROM 中，我们可以让 CPU 直接从其中读取代码执行，但通常这不是一个好办法，我们最好在系统启动后将 FLASH 或 ROM 中的目标代码拷贝入 RAM 中后再执行以提高取指令速度；

对于 UART 等设备，其内部有一定容量的接收 BUFFER，我们应尽量在 BUFFER 被占满后再向 CPU 提出中断。例如计算机终端在向目标机通过 RS-232 传递数据时，不宜设置 UART 只接收到一个 BYTE 就向 CPU 提中断，从而无谓浪费中断处理时间；

如果对某设备能采取 DMA 方式读取，就采用 DMA 读取，DMA 读取方式在读取目标中包含的存储信息较大时效率较高，其数据传输的基本单位是块，而所传输的数据是从设备直接送入内存的（或者相反）。DMA 方式较之中断驱动方式，减少了 CPU 对外设的干预，进一步提高了 CPU 与外设的并行操作程度。

活用位操作

使用 C 语言的位操作可以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，因而，灵活的位操作可以有效地提高程序运行的效率。举例如下：

```
/* 方法 1 */
int i,j;
i = 879 / 16;
j = 562 % 32;
/* 方法 2 */
int i,j;
i = 879 >> 4;
j = 562 - (562 >> 5 << 5);
```

对于以 2 的指数次方为"*"、"/"或"%"因子的数学运算，转化为移位运算"<< >>"通常可以提高算法效率。因为乘除运算指令周期通常比移位运算大。

C 语言位运算除了可以提高运算效率外，在嵌入式系统的编程中，它的另一个最典型的应用，而且十分广泛地正在被使用着的是位间的与（&）、或（|）、非（~）操作，这跟嵌入式系统的编程特点有很大关系。我们通常要对硬件寄存器进行位设置，譬如，我们通过将 AM186ER 型 80186 处理器的中断屏蔽控制寄存器的第低 6 位设置为 0（开中断 2），最通用的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
outword(INT_MASK, wTemp &~INT_I2_MASK);
```

而将该位设置为 1 的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
outword(INT_MASK, wTemp | INT_I2_MASK);
```

判断该位是否为 1 的做法是：

```
#define INT_I2_MASK 0x0040
wTemp = inword(INT_MASK);
if(wTemp & INT_I2_MASK)
{
… /* 该位为 1 */
}
```

上述方法在嵌入式系统的编程中是非常常见的，我们需要牢固掌握。

总结

在性能优化方面永远注意 80-20 准备，不要优化程序中开销不大的那 80%，这是劳而无功的。

宏定义是 C 语言中实现类似函数功能而又不具函数调用和返回开销的较好方法，但宏在本质上不是函数，因而要防止宏展开后出现不可预料的结果，对宏的定义和使用要慎而处之。很遗憾，标准 C 至今没有包括 C++中 inline 函数的功能，inline 函数兼具无调用开销和安全的优点。

使用寄存器变量、内嵌汇编和活用位操作也是提高程序效率的有效方法。

除了编程上的技巧外，为提高系统的运行效率，我们通常也需要最大可能地利用各种硬件设备自身的特点来减小其运转开销，例如减小中断次数、利用 DMA 传输方式等。.

C/C++语言 void 及 void 指针深层探索

1.概述

许多初学者对 C/C++语言中的 void 及 void 指针类型不甚理解，因此在使用上出现了一些错误。本文将对 void 关键字的深刻含义进行解说，并详述 void 及 void 指针类型的使用方法与技巧。

2.void 的含义

void 的字面意思是“无类型”，void *则为“无类型指针”，void *可以指向任何类型的数据。

void 几乎只有“注释”和限制程序的作用，因为从来没有人会定义一个 void 变量，让我们试着来定义：

```
void a;
```

这行语句编译时会出错，提示“illegal use of type ‘void‘”。不过，即使 void a 的编译不会出错，它也没有任何实际意义。

void 真正发挥的作用在于：

（1） 对函数返回的限定；

（2） 对函数参数的限定。

我们将在第三节对以上二点进行具体说明。

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

例如：

```
float *p1;
int *p2;
p1 = p2;
```

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

p1 = (float *)p2;

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

```
void *p1;
int *p2;
p1 = p2;
```

但这并不意味着，void *也可以无需强制类型转换地赋给其它类型的指针。因为“无类型”可以包容“有类型”，而“有类型”则不能包容“无类型”。道理很简单，我们可以说“男人和女人都是人”，但不能说“人是男人”或者“人是女人”。下面的语句编译出错：

```
void *p1;
int *p2;
p2 = p1;
```

提示“ ‘= ‘ : cannot convert from ‘void * ‘ to ‘int * ‘”。

3. void 的使用

下面给出 void 关键字的使用规则：

规则一 如果函数没有返回值，那么应声明为 void 类型

在 C 语言中，凡不加返回值类型限定的函数，就会被编译器作为返回整型值处理。但是许多程序员却误以为其为 void 类型。例如：

```
add ( int a, int b )
{
return a + b;
}
int main(int argc, char* argv[])
{
printf ( "2 + 3 = %d", add ( 2, 3) );
}
```

程序运行的结果为输出：

2 + 3 = 5

这说明不加返回值说明的函数的确为 int 函数。

林锐博士《高质量 C/C++编程》中提到：“C++语言有很严格的类型安全检查，不允许上述情况（指函数不加类型声明）发生”。可是编译器并不一定这么认定，譬如在 Visual C++6.0 中上述 add 函数的编译无错也无警告且运行正确，所以不能寄希望于编译器会做严格的类型检查。

因此，为了避免混乱，我们在编写 C/C++程序时，对于任何函数都必须一个不漏地指定其类型。如果函数没有返回值，一定要声明为 void 类型。这既是程序良好可读性的需要，也是编程规范性的要求。另外，加上 void 类型声明后，也可以发挥代码的“自注释”作用。代码的“自注释”即代码能自己注释自己。

规则二 如果函数无参数，那么应声明其参数为 void

在 C++语言中声明一个这样的函数：

```
int function(void)
{
return 1;
}
```

则进行下面的调用是不合法的：

```
function(2);
```

因为在 C++中，函数参数为 void 的意思是这个函数不接受任何参数。

我们在 Turbo C 2.0 中编译：

```
#include "stdio.h"
fun()
{
return 1;
}
main()
{
printf("%d",fun(2));
getchar();
}
```

编译正确且输出 1，这说明，在 C 语言中，可以给无参数的函数传送任意类型的参数，但是在 C++编译器中编译同样的代码则会出错。在 C++中，不能向无参数的函数传送任何参数，出错提示“‘fun‘ : function does not take 1 parameters”。

所以，无论在 C 还是 C++中，若函数不接受任何参数，一定要指明参数为 void。

规则三 小心使用 void 指针类型

按照 ANSI(American National Standards Institute)标准，不能对 void 指针进行算法操作，即下列操作都是不合法的：

```
void * pvoid;
pvoid++; //ANSI：错误
pvoid += 1; //ANSI：错误
//ANSI 标准之所以这样认定，是因为它坚持：进行算法操作的指针必须是确定知道其指向数据类型大小的。
//例如：
int *pint;
pint++; //ANSI：正确
```

pint++的结果是使其增大 sizeof(int)。

但是大名鼎鼎的 GNU(GNU‘s Not Unix 的缩写)则不这么认定，它指定 void *的算法操作与 char *一致。
因此下列语句在 GNU 编译器中皆正确：

```
pvoid++; //GNU：正确
pvoid += 1; //GNU：正确
```

pvoid++的执行结果是其增大了 1。

在实际的程序设计中，为迎合 ANSI 标准，并提高程序的可移植性，我们可以这样编写实现同样功能的代码：

```
void * pvoid;
(char *)pvoid++; //ANSI：正确；GNU：正确
(char *)pvoid += 1; //ANSI：错误；GNU：正确
```

GNU 和 ANSI 还有一些区别，总体而言，GNU 较 ANSI 更“开放”，提供了对更多语法的支持。但是我们在真实设计时，还是应该尽可能地迎合 ANSI 标准。

规则四 如果函数的参数可以是任意类型指针，那么应声明其参数为 void *

典型的如内存操作函数 memcpy 和 memset 的函数原型分别为：

```
void * memcpy(void *dest, const void *src, size_t len);
void * memset ( void * buffer, int c, size_t num );
```

这样，任何类型的指针都可以传入 memcpy 和 memset 中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么类型。如果 memcpy 和 memset 的参数类型不是 void *，而是 char *，那才叫真的奇怪了！这样的 memcpy 和 memset 明显不是一个“纯粹的，脱离低级趣味的”函数！

下面的代码执行正确：

```
//示例：memset 接受任意类型指针
int intarray[100];
memset ( intarray, 0, 100*sizeof(int) ); //将 intarray 清 0

//示例：memcpy 接受任意类型指针
int intarray1[100], intarray2[100];
memcpy ( intarray1, intarray2, 100*sizeof(int) ); //将 intarray2 拷贝给 intarray1
```

有趣的是，memcpy 和 memset 函数返回的也是 void *类型，标准库函数的编写者是多么地富有学问啊！

规则五 void 不能代表一个真实的变量

下面代码都企图让 void 代表一个真实的变量，因此都是错误的代码：

```
void a; //错误
function(void a); //错误
```

void 体现了一种抽象，这个世界上的变量都是“有类型”的，譬如一个人不是男人就是女人（还有人妖？）。

void 的出现只是为了一种抽象的需要，如果你正确地理解了面向对象中“抽象基类”的概念，也很容易理解 void 数据类型。正如不能给抽象基类定义一个实例，我们也不能定义一个 void（让我们类比的称 void 为“抽象数据类型”）变量。

4.总结

小小的 void 蕴藏着很丰富的设计哲学，作为一名程序设计人员，对问题进行深一个层次的思考必然使我们受益匪浅。

C/C++语言可变参数表深层探索

作者: 宋宝华 **e-mail:**21cnbao@21cn.com

1.引言

C/C++语言有一个不同于其它语言的特性，即其支持可变参数，典型的函数如 printf、scanf 等可以接受数量不定的参数。如：

```
printf ( "I love you" );
printf ( "%d", a );
printf ( "%d,%d", a, b );
```

第一、二、三个 printf 分别接受 1、2、3 个参数，让我们看看 printf 函数的原型：

```
int printf ( const char *format, ... );
```

从函数原型可以看出，其除了接收一个固定的参数 format 以外，后面的参数用“…”表示。在 C/C++语言中，“…”表示可以接受不定数量的参数，理论上来讲，可以是 0 或 0 以上的 n 个参数。

本文将对 C/C++可变参数表的使用方法及 C/C++支持可变参数表的深层机理进行探索。

2.可变参数表的用法

2.1 相关宏

标准 C/C++包含头文件 stdarg.h，该头文件中定义了如下三个宏：

```
void va_start ( va_list arg_ptr, prev_param ); /* ANSI version */
type va_arg ( va_list arg_ptr, type );
void va_end ( va_list arg_ptr );
```

在这些宏中，va 就是 variable argument(可变参数)的意思；arg_ptr 是指向可变参数表的指针；prev_param 则指可变参数表的前一个固定参数；type 为可变参数的类型。va_list 也是一个宏，其定义为 typedef char * va_list，实质上是一 char 型指针。char 型指针的特点是++、--操作对其作用的结果是增 1 和减 1（因为 sizeof(char)为 1），与之不同的是 int 等其它类型指针的++、--操作对其作用的结果是增 sizeof(type)或减 sizeof(type)，而且 sizeof(type)大于 1。

通过 va_start 宏我们可以取得可变参数表的首指针，这个宏的定义为：

```
#define va_start ( ap, v ) ( ap = (va_list)&v + _INTSIZEOF(v) )
```

显而易见，其含义为将最后那个固定参数的地址加上可变参数对其的偏移后赋值给 ap，这样 ap 就是可变参数表的首地址。其中的_INTSIZEOF 宏定义为：

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

va_arg 宏的意思则指取出当前 arg_ptr 所指的可变参数并将 ap 指针指向下一可变参数,其原型为：

```
#define va_arg(list, mode) ((mode *)(list =\
 (char *) ((((int)list + (__builtin_alignof(mode)<=4?3:7)) &\
 (__builtin_alignof(mode)<=4?-4:-8))+sizeof(mode))))[-1]
```

对这个宏的具体含义我们将在第 3 节深入讨论。

而 va_end 宏被用来结束可变参数的获取，其定义为：

```
#define va_end ( list )
```

可以看出，va_end (list)实际上被定义为空，没有任何真实对应的代码，用于代码对称，与 va_start 对应；另外，它还可能发挥代码的“自注释”作用。所谓代码的“自注释”，指的是代码能自己注释自己。

下面我们以具体的例子来说明以上三个宏的使用方法。

2.2 一个简单的例子

```
#include <stdarg.h>
/*  函数名：max
 *  功能：返回 n 个整数中的最大值
 *  参数：num：整数的个数 ...：num 个输入的整数
 *  返回值：求得的最大整数
 */
int max ( int num, ... )
{
    int m = -0x7FFFFFFF;  /* 32 系统中最小的整数 */
    va_list ap;
    va_start ( ap, num );
    for ( int i= 0; i< num; i++ )
    {
        int t = va_arg (ap, int);
        if ( t > m )
        {
            m = t;
    }
    }
    va_end (ap);
    return m;
}

/* 主函数调用 max */
int main ( int argc, char* argv[] )
{
    int n = max ( 5, 5, 6 ,3 ,8 ,5);  /* 求 5 个整数中的最大值 */
    cout << n;
    return 0;
}
```

函数 max 中首先定义了可变参数表指针 ap，而后通过 va_start (ap, num)取得了参数表首地址（赋给了 ap），其后的 for 循环则用来遍历可变参数表。这种遍历方式与我们在数据结构教材中经常看到的遍历方式是类似的。

函数 max 看起来简洁明了，但是实际上 printf 的实现却远比这复杂。max 函数之所以看起来简单，是因为：

(1) max 函数可变参数表的长度是已知的，通过 num 参数传入；

(2) max 函数可变参数表中参数的类型是已知的，都为 int 型。

而 printf 函数则没有这么幸运。首先，printf 函数可变参数的个数不能轻易的得到，而可变参数的类型也不是固定的，需由格式字符串进行识别（由%f、%d、%s 等确定），因此则涉及到可变参数表的更复杂应用。

下面我们以实例来分析可变参数表的高级应用。

2.3 高级应用

下面这个程序是我们为某嵌入式系统（该系统中 CPU 的字长为 16 位）编写的在屏幕上显示格式字符串

的函数 DrawText，它的用法类似于 int printf (const char *format, ...)函数，但其输出的目标为嵌入式系统的液晶显示屏幕（LED）。

```
//////////////////////////////////////////////////////////////////////////////
//  函数名称: DrawText
//  功能说明: 在显示屏上绘制文字
//  参数说明: xPos ---横坐标的位置  [0 .. 30]
//            yPos ---纵坐标的位置  [0 .. 64]
//            ...  可以同数字一起显示，需设置标志(%d、%l、%x、%s)
//////////////////////////////////////////////////////////////////////////////
extern void DrawText ( BYTE xPos, BYTE yPos, LPBYTE lpStr, ... )
{
    BYTE   lpData[100];  //缓冲区
    BYTE   byIndex;
    BYTE   byLen;
    DWORD  dwTemp;
    WORD   wTemp;
    int    i;
    va_list lpParam;

    memset( lpData, 0, 100);
    byLen  = strlen( lpStr );
    byIndex = 0;
    va_start ( lpParam, lpStr );

    for ( i = 0; i < byLen; i++ )
    {
        if( lpStr[i] != '%' )    //不是格式符开始
        {
            lpData[byIndex++] = lpStr[i];
        }
        else
        {
            switch (lpStr[i+1])
            {
            //整型
            case   'd':
            case   'D':
                wTemp  = va_arg ( lpParam, int );
                byIndex += IntToStr( lpData+byIndex, (DWORD)wTemp );
                i++;
                break;
            //长整型
            case   'l':
            case   'L':
```

```
                dwTemp = va_arg ( lpParam, long );
                byIndex += IntToStr ( lpData+byIndex, (DWORD)dwTemp );
                i++;
                break;
            //16 进制（长整型）
            case   'x':
            case   'X':
                dwTemp = va_arg ( lpParam, long );
                byIndex += HexToStr ( lpData+byIndex, (DWORD)dwTemp );
                i++;
                break;
            default:
               lpData[byIndex++] = lpStr[i];
                break;
            }
        }
    }
    va_end ( lpParam );
    lpData[byIndex]   = '\0';
    DisplayString ( xPos, yPos, lpData, TRUE);  //在屏幕上显示字符串 lpData
}
```

在这个函数中，需通过对传入的格式字符串（首地址为 lpStr）进行识别来获知可变参数个数及各个可变参数的类型，具体实现体现在 for 循环中。譬如，在识别为%d 后，做的是 va_arg (lpParam, int)，而获知为%l 和%x 后则进行的是 va_arg (lpParam, long)。格式字符串识别完成后，可变参数也就处理完了。

在项目的最初，我们一直苦于不能找到一个好的办法来混合输出字符串和数字，我们采用了分别显示数字和字符串的方法，并分别指定坐标，程序条理被破坏。而且，在混合显示的时候，要给各类数据分别人工计算坐标，我们感觉头疼不已。以前的函数为：

```
//显示字符串
showString ( BYTE xPos, BYTE yPos, LPBYTE lpStr )
//显示数字
showNum ( BYTE xPos, BYTE yPos, int num )
//以 16 进制方式显示数字
showHexNum ( BYTE xPos, BYTE yPos, int num )
```

最终，我们用 DrawText (BYTE xPos, BYTE yPos, LPBYTE lpStr, ...)函数代替了原先所有的输出函数，程序得到了简化。就这样，兄弟们用得爽翻了。

3. 运行机制探索

通过第 2 节我们学会了可变参数表的使用方法，相信喜欢抛根问底的读者还不甘心，必然想知道如下问题：

（1）为什么按照第 2 节的做法就可以获得可变参数并对其进行操作？

（2）C/C++在底层究竟是依靠什么来对这一语法进行支持的，为什么其它语言就不能提供可变参数表呢？

我们带着这些疑问来一步步进行摸索。

3.1 调用机制反汇编

反汇编是研究语法深层特性的终极良策，先来看看 2.2 节例子中主函数进行 max（ 5, 5, 6 ,3 ,8 ,5)调用时的反汇编：

```
1. 004010C8   push        5
2. 004010CA   push        8
3. 004010CC   push        3
4. 004010CE   push        6
5. 004010D0   push        5
6. 004010D2   push        5
7. 004010D4   call        @ILT+5(max) (0040100a)
```

从上述反汇编代码中我们可以看出，C/C++函数调用的过程中：

第一步：将参数从右向左入栈（第 1～6 行）；

第二步：调用 call 指令进行跳转（第 7 行）。

这两步包含了深刻的含义，它说明 C/C++默认的调用方式为由调用者管理参数入栈的操作，且入栈的顺序为从右至左，这种调用方式称为_cdecl 调用。x86 系统的入栈方向为从高地址到低地址，故第 1 至 n 个参数被放在了地址递增的堆栈内。在被调用函数内部，读取这些堆栈的内容就可获得各个参数的值，让我们反汇编到 max 函数的内部：

```
int max ( int num, ...)
{
1. 00401020   push        ebp
2. 00401021   mov         ebp,esp
3. 00401023   sub         esp,50h
4. 00401026   push        ebx
5. 00401027   push        esi
6. 00401028   push        edi
7. 00401029   lea         edi,[ebp-50h]
8. 0040102C   mov         ecx,14h
9. 00401031   mov         eax,0CCCCCCCCh
10. 00401036   rep stos     dword ptr [edi]
      va_list ap;
      int m = -0x7FFFFFFF; /* 32 系统中最小的整数 */
11. 00401038   mov         dword ptr [ebp-8],80000001h
      va_start ( ap, num );
12. 0040103F   lea         eax,[ebp+0Ch]
13. 00401042   mov         dword ptr [ebp-4],eax
      for ( int i= 0; i< num; i++ )
14. 00401045   mov         dword ptr [ebp-0Ch],0
15. 0040104C   jmp         max+37h (00401057)
16. 0040104E   mov         ecx,dword ptr [ebp-0Ch]
17. 00401051   add         ecx,1
18. 00401054   mov         dword ptr [ebp-0Ch],ecx
19. 00401057   mov         edx,dword ptr [ebp-0Ch]
20. 0040105A   cmp         edx,dword ptr [ebp+8]
21. 0040105D   jge         max+61h (00401081)
```

```
        {
            int t= va_arg (ap, int);
22. 0040105F    mov         eax,dword ptr [ebp-4]
23. 00401062    add         eax,4
24. 00401065    mov         dword ptr [ebp-4],eax
25. 00401068    mov         ecx,dword ptr [ebp-4]
26. 0040106B    mov         edx,dword ptr [ecx-4]
27. 0040106E    mov         dword ptr [t],edx
            if ( t > m )
28. 00401071    mov         eax,dword ptr [t]
29. 00401074    cmp         eax,dword ptr [ebp-8]
30. 00401077    jle         max+5Fh (0040107f)
                m = t;
31. 00401079    mov         ecx,dword ptr [t]
32. 0040107C    mov         dword ptr [ebp-8],ecx
        }
33. 0040107F    jmp         max+2Eh (0040104e)
        va_end (ap);
34. 00401081    mov         dword ptr [ebp-4],0
        return m;
35. 00401088    mov         eax,dword ptr [ebp-8]
    }
36. 0040108B    pop         edi
37. 0040108C    pop         esi
38. 0040108D    pop         ebx
39. 0040108E    mov         esp,ebp
40. 00401090    pop         ebp
41. 00401091    ret
```

分析上述反汇编代码，对于一个真正的程序员而言，将是一种很大的享受；而对于初学者，也将使其受益良多。所以请一定要赖着头皮认真研究，千万不要被吓倒！

行 1～10 进行执行函数内代码的准备工作，保存现场。第 2 行对堆栈进行移动；第 3 行则意味着 max 函数为其内部局部变量准备的堆栈空间为 50h 字节；第 11 行表示把变量 n 的内存空间安排在了函数内部局部栈底减 8 的位置（占用 4 个字节）。

第 12～13 行非常关键，对应着 va_start (ap, num)，这两行将第一个可变参数的地址赋值给了指针 ap。另外，从第 12 行可以看出 num 的地址为 ebp+0Ch；从第 13 行可以看出 ap 被分配在函数内部局部栈底减 4 的位置上（占用 4 个字节）。

第 22～27 行最为关键，对应着 va_arg (ap, int)。其中，22~24 行的作用为将 ap 指向下一可变参数（可变参数的地址间隔为 4 个字节，从 add eax,4 可以看出）；25~27 行则取当前可变参数的值赋给变量 t。这段反汇编很奇怪，它先移动可变参数指针，再在赋值指令里面回过头来取先前的参数值赋给 t（从 mov edx,dword ptr [ecx-4]语句可以看出）。Visual C++同学玩得有意思，不知道碰见同样的情况 Visual Basic 等其它同学怎么玩？

第 36～41 行恢复现场和堆栈地址，执行函数返回操作。

痛苦的反汇编之旅差不多结束了，看了这段反汇编我们总算弄明白了可变参数的存放位置以及它们被读取的方式，顿觉全省轻松！

3.2 特殊的调用约定

除此之外，我们需要了解 C/C++函数调用对参数占用空间的一些特殊约定，因为在_cdecl 调用协议中，有些变量类型是按照其它变量的尺寸入栈的。

例如，字符型变量将被自动扩展为一个字的空间，因为入栈操作针对的是一个字。

参数 n 实际占用的空间为((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))，这就是第 2.1 节_INTSIZEOF(v)宏的来历！

既然如此，2.1 节给出的 va_arg (list, mode)宏为什么玩这么大的飞机就很清楚了。这个问题就留个读者您来分析。

C/C++数组名与指针区别深层探索

作者： 宋宝华 **e-mail:**21cnbao@21cn.com

1. 引言

指针是 C/C++语言的特色，而数组名与指针有太多的相似，甚至很多时候，数组名可以作为指针使用。于是乎，很多程序设计者就被搞糊涂了。而许多的大学老师，他们在 C 语言的教学过程中也错误得给学生讲解：“数组名就是指针”。很幸运，我的大学老师就是其中之一。时至今日，我日复一日地进行着 C/C++项目的开发，而身边还一直充满这样的程序员，他们保留着“数组名就是指针”的误解。

想必这种误解的根源在于国内某著名的 C 程序设计教程。如果这篇文章能够纠正许多中国程序员对数组名和指针的误解，笔者就不甚欣慰了。借此文，笔者站在无数对知识如饥似渴的中国程序员之中，深深寄希望于国内的计算机图书编写者们，能以“深入探索”的思维方式和精益求精的认真态度来对待图书编写工作，但愿市面上多一些融入作者思考结晶的心血之作！

2. 魔幻数组名

请看程序（本文程序在 WIN32 平台下编译）：

```
1. #include <iostream.h>
2. int main(int argc, char* argv[])
3. {
4.   char str[10];
5.   char *pStr = str;
6.   cout << sizeof(str) << endl;
7.   cout << sizeof(pStr) << endl;
8.   return 0;
9. }
```

2.1 数组名不是指针

我们先来推翻“数组名就是指针”的说法，用反证法。

证明 数组名不是指针

假设：数组名是指针；

则：pStr 和 str 都是指针；

因为：在 WIN32 平台下，指针长度为 4；

所以：第 6 行和第 7 行的输出都应该为 4；

实际情况是：第 6 行输出 10，第 7 行输出 4；

所以：假设不成立，数组名不是指针

2.2 数组名神似指针

上面我们已经证明了数组名的确不是指针，但是我们再看看程序的第 5 行。该行程序将数组名直接赋值给指针，这显