(一)写一个简单的可变参数的C函数   
下面我们来探讨如何写一个简单的可变参数的C函数.写可变参数的   
C函数要在程序中用到以下这些宏:   
void va\_start( va\_list arg\_ptr, prev\_param );   
type va\_arg( va\_list arg\_ptr, type );   
void va\_end( va\_list arg\_ptr );   
va在这里是variable-argument(可变参数)的意思.   
这些宏定义在stdarg.h中,所以用到可变参数的程序应该包含这个   
头文件.下面我们写一个简单的可变参数的函数,改函数至少有一个整数   
参数,第二个参数也是整数,是可选的.函数只是打印这两个参数的值.   
void simple\_va\_fun(int i, ...)   
{   
va\_list arg\_ptr;   
int j=0;   
va\_start(arg\_ptr, i);   
j=va\_arg(arg\_ptr, int);   
va\_end(arg\_ptr);   
printf("%d %d\n", i, j);   
return;   
}   
我们可以在我们的头文件中这样声明我们的函数:   
extern void simple\_va\_fun(int i, ...);   
我们在程序中可以这样调用:   
simple\_va\_fun(100);   
simple\_va\_fun(100,200);   
从这个函数的实现可以看到,我们使用可变参数应该有以下步骤:   
1)首先在函数里定义一个va\_list型的变量,这里是arg\_ptr,这个变   
量是指向参数的指针.   
2)然后用va\_start宏初始化变量arg\_ptr,这个宏的第二个参数是第   
一个可变参数的前一个参数,是一个固定的参数.   
3)然后用va\_arg返回可变的参数,并赋值给整数j. va\_arg的第二个   
参数是你要返回的参数的类型,这里是int型.   
4)最后用va\_end宏结束可变参数的获取.然后你就可以在函数里使   
用第二个参数了.如果函数有多个可变参数的,依次调用va\_arg获   
取各个参数.   
如果我们用下面三种方法调用的话,都是合法的,但结果却不一样:   
1)simple\_va\_fun(100);   
结果是:100 -123456789(会变的值)   
2)simple\_va\_fun(100,200);   
结果是:100 200   
3)simple\_va\_fun(100,200,300);   
结果是:100 200   
我们看到第一种调用有错误,第二种调用正确,第三种调用尽管结果   
正确,但和我们函数最初的设计有冲突.下面一节我们探讨出现这些结果   
的原因和可变参数在编译器中是如何处理的.   
(二)可变参数在编译器中的处理   
我们知道va\_start,va\_arg,va\_end是在stdarg.h中被定义成宏的,   
由于1)硬件平台的不同 2)编译器的不同,所以定义的宏也有所不同,下   
面以VC++中stdarg.h里x86平台的宏定义摘录如下('\'号表示折行):   
typedef char \* va\_list;   
#define \_INTSIZEOF(n) \   
((sizeof(n)+sizeof(int)-1)&~(sizeof(int) - 1) )   
#define va\_start(ap,v) ( ap = (va\_list)&v + \_INTSIZEOF(v) )   
#define va\_arg(ap,t) \   
( \*(t \*)((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) )   
#define va\_end(ap) ( ap = (va\_list)0 )   
定义\_INTSIZEOF(n)主要是为了某些需要内存的对齐的系统.C语言的函   
数是从右向左压入堆栈的,图(1)是函数的参数在堆栈中的分布位置.我   
们看到va\_list被定义成char\*,有一些平台或[**操作系统**](http://lib.csdn.net/base/operatingsystem)定义为void\*.再   
看va\_start的定义,定义为&v+\_INTSIZEOF(v),而&v是固定参数在堆栈的   
地址,所以我们运行va\_start(ap, v)以后,ap指向第一个可变参数在堆   
栈的地址,如图:   
高地址|-----------------------------|   
|函数返回地址 |   
|-----------------------------|   
|....... |   
|-----------------------------|   
|第n个参数(第一个可变参数) |   
|-----------------------------|<--va\_start后ap指向   
|第n-1个参数(最后一个固定参数)|   
低地址|-----------------------------|<-- &v   
图( 1 )   
然后,我们用va\_arg()取得类型t的可变参数值,以上例为int型为例,我   
们看一下va\_arg取int型的返回值:   
j= ( \*(int\*)((ap += \_INTSIZEOF(int))-\_INTSIZEOF(int)) );   
首先ap+=sizeof(int),已经指向下一个参数的地址了.然后返回   
ap-sizeof(int)的int\*指针,这正是第一个可变参数在堆栈里的地址   
(图2).然后用\*取得这个地址的内容(参数值)赋给j.   
高地址|-----------------------------|   
|函数返回地址 |   
|-----------------------------|   
|....... |   
|-----------------------------|<--va\_arg后ap指向   
|第n个参数(第一个可变参数) |   
|-----------------------------|<--va\_start后ap指向   
|第n-1个参数(最后一个固定参数)|   
低地址|-----------------------------|<-- &v   
图( 2 )   
最后要说的是va\_end宏的意思,x86平台定义为ap=(char\*)0;使ap不再   
指向堆栈,而是跟NULL一样.有些直接定义为((void\*)0),这样编译器不   
会为va\_end产生代码,例如gcc在linux的x86平台就是这样定义的.   
在这里大家要注意一个问题:由于参数的地址用于va\_start宏,所   
以参数不能声明为寄存器变量或作为函数或数组类型.   
关于va\_start, va\_arg, va\_end的描述就是这些了,我们要注意的   
是不同的操作系统和硬件平台的定义有些不同,但原理却是相似的.   
(三)可变参数在编程中要注意的问题   
因为va\_start, va\_arg, va\_end等定义成宏,所以它显得很愚蠢,   
可变参数的类型和个数完全在该函数中由程序代码控制,它并不能智能   
地识别不同参数的个数和类型.   
有人会问:那么printf中不是实现了智能识别参数吗?那是因为函数   
printf是从固定参数format字符串来分析出参数的类型,再调用va\_arg   
的来获取可变参数的.也就是说,你想实现智能识别可变参数的话是要通   
过在自己的程序里作判断来实现的.   
另外有一个问题,因为编译器对可变参数的函数的原型检查不够严   
格,对编程查错不利.如果simple\_va\_fun()改为:   
void simple\_va\_fun(int i, ...)   
{   
va\_list arg\_ptr;   
char \*s=NULL;   
va\_start(arg\_ptr, i);   
s=va\_arg(arg\_ptr, char\*);   
va\_end(arg\_ptr);   
printf("%d %s\n", i, s);   
return;   
}   
可变参数为char\*型,当我们忘记用两个参数来调用该函数时,就会出现   
core dump(Unix) 或者页面非法的错误(window平台).但也有可能不出   
错,但错误却是难以发现,不利于我们写出高质量的程序.   
以下提一下va系列宏的兼容性.   
System V Unix把va\_start定义为只有一个参数的宏:   
va\_start(va\_list arg\_ptr);   
而ANSI C则定义为:   
va\_start(va\_list arg\_ptr, prev\_param);   
如果我们要用system V的定义,应该用vararg.h头文件中所定义的   
宏,ANSI C的宏跟system V的宏是不兼容的,我们一般都用ANSI C,所以   
用ANSI C的定义就够了,也便于程序的移植.   
小结:   
可变参数的函数原理其实很简单,而va系列是以宏定义来定义的,实   
现跟堆栈相关.我们写一个可变函数的C函数时,有利也有弊,所以在不必   
要的场合,我们无需用到可变参数.如果在C++里,我们应该利用C++的多   
态性来实现可变参数的功能,尽量避免用C语言的方式来实现.

===================================================

概述  
由于在C语言中没有函数重载,解决不定数目函数参数问题变得比较麻烦;即使采用C++,如果参数个数不能确定,也很难采用函数重载.对这种情况,有些人采用指针参数来解决问题.下面就c语言中处理不定参数数目的问题进行讨论.   
定义  
大家先看几宏.  
在VC++6.0的include有一个stdarg.h头文件,有如下几个宏定义:  
#define \_INTSIZEOF(n)   ((sizeof(n)+sizeof(int)-1)&~(sizeof(int) - 1) )   
#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\_list)0 )                            // 将指针置为无效  
如果对以上几个宏定义不理解,可以略过,接这看后面的内容.   
参数在堆栈中分布位置  
在进程中,堆栈地址是从高到低分配的.当执行一个函数的时候,将参数列表入栈,压入堆栈的高地址部分,然后入栈函数的返回地址,接着入栈函数的执行代码,这个入栈过程,堆栈地址不断递减,一些黑客就是在堆栈中修改函数返回地址,执行自己的代码来达到执行自己插入的代码段的目的.  
总之,函数在堆栈中的分布情况是:地址从高到低,依次是:函数参数列表,函数返回地址,函数执行代码段.  
堆栈中,各个函数的分布情况是倒序的.即最后一个参数在列表中地址最高部分,第一个参数在列表地址的最低部分.参数在堆栈中的分布情况如下:  
最后一个参数  
倒数第二个参数  
...  
第一个参数  
函数返回地址  
函数代码段   
示例代码  
void arg\_test(int i, ...);  
int main(int argc,char \*argv[])   
{  
int int\_size = \_INTSIZEOF(int);  
printf("int\_size=%d\n", int\_size);  
arg\_test(0, 4);

arg\_cnt(4,1,2,3,4);  
return 0;  
}  
void arg\_test(int i, ...)  
{  
int j=0;   
va\_list arg\_ptr;

va\_start(arg\_ptr, i);   
printf("&i = %p\n", &i);//打印参数i在堆栈中的地址  
printf("arg\_ptr = %p\n", arg\_ptr);  
//打印va\_start之后arg\_ptr地址,  
//应该比参数i的地址高sizeof(int)个字节  
//这时arg\_ptr指向下一个参数的地址

j=\*((int \*)arg\_ptr);  
printf("%d %d\n", i, j);   
j=va\_arg(arg\_ptr, int);   
printf("arg\_ptr = %p\n", arg\_ptr);  
//打印va\_arg后arg\_ptr的地址  
//应该比调用va\_arg前高sizeof(int)个字节  
//这时arg\_ptr指向下一个参数的地址  
va\_end(arg\_ptr);   
printf("%d %d\n", i, j);   
}   
代码说明:  
int int\_size = \_INTSIZEOF(int);得到int类型所占字节数  
va\_start(arg\_ptr, i); 得到第一个可变参数地址,根据定义(va\_list)&v得到起始参数的地址, 再加上\_INTSIZEOF(v) ,就是其实参数下一个参数的地址,即第一个可变参数地址.  
j=va\_arg(arg\_ptr, int); 得到第一个参参数的值,并且arg\_ptr指针上移一个\_INTSIZEOF(int),即指向下一个可变参数的地址.  
va\_end(arg\_ptr);置空arg\_ptr,即arg\_ptr=0;  
总结:读取可变参数的过程其实就是堆栈中,使用指针,遍历堆栈段中的参数列表,从低地址到高地址一个一个地把参数内容读出来的过程.   
在编程中应该注意的问题和解决办法  
虽然可以通过在堆栈中遍历参数列表来读出所有的可变参数,但是由于不知道可变参数有多少个,什么时候应该结束遍历,如果在堆栈中遍历太多,那么很可能读取一些无效的数据.  
解决办法:a.可以在第一个起始参数中指定参数个数,那么就可以在循环还中读取所有的可变参数;b.定义一个结束标记,在调用函数的时候,在最后一个参数中传递这个标记,这样在遍历可变参数的时候,可以根据这个标记结束可变参数的遍历;  
下面是一段示例代码:  
//第一个参数定义可选参数个数,用于循环取初参数内容  
void arg\_cnt(int cnt, ...);  
int main(int argc,char \*argv[])   
{  
int int\_size = \_INTSIZEOF(int);  
printf("int\_size=%d\n", int\_size);  
arg\_cnt(4,1,2,3,4);  
return 0;  
}  
void arg\_cnt(int cnt, ...)  
{  
int value=0;   
int i=0;  
int arg\_cnt=cnt;   
va\_list arg\_ptr;   
va\_start(arg\_ptr, cnt);   
for(i = 0; i < cnt; i++)  
{  
   value = va\_arg(arg\_ptr,int);  
   printf("value%d=%d\n", i+1, value);  
}  
}

虽然可以根据上面两个办法解决读取参数个数的问题,但是如果参数类型都是不定的,该怎么办,如果不知道参数的类型,即使读到了参数也没有办法进行处理.解决办法:可以自定义一些可能出现的参数类型,这样在可变参数列表中,可以可变参数列表中的那类型,然后根据类型,读取可变参数值,并进行准确地转换.传递参数的时候可以这样传递:参数数目,可变参数类型1,可变参数值1,可变参数类型2,可变参数值2,....  
这里给出一个完整的例子:  
#include <stdio.h>  
#include <stdarg.h>  
const int INT\_TYPE   = 100000;  
const int STR\_TYPE   = 100001;  
const int CHAR\_TYPE   = 100002;  
const int LONG\_TYPE   = 100003;  
const int FLOAT\_TYPE = 100004;  
const int DOUBLE\_TYPE = 100005;  
//第一个参数定义可选参数个数,用于循环取初参数内容  
//可变参数采用arg\_type,arg\_value...的形式传递,以处理不同的可变参数类型  
void arg\_type(int cnt, ...);  
//第一个参数定义可选参数个数,用于循环取初参数内容  
void arg\_cnt(int cnt, ...);  
//[**测试**](http://lib.csdn.net/base/softwaretest)va\_start,va\_arg的使用方法,函数参数在堆栈中的地址分布情况  
void arg\_test(int i, ...);  
int main(int argc,char \*argv[])   
{  
int int\_size = \_INTSIZEOF(int);  
printf("int\_size=%d\n", int\_size);  
arg\_test(0, 4);

arg\_cnt(4,1,2,3,4);  
arg\_type(2, INT\_TYPE, 222, STR\_TYPE, "ok,hello world!");  
return 0;  
}  
void arg\_test(int i, ...)  
{  
int j=0;   
va\_list arg\_ptr;

va\_start(arg\_ptr, i);   
printf("&i = %p\n", &i);//打印参数i在堆栈中的地址  
printf("arg\_ptr = %p\n", arg\_ptr);  
//打印va\_start之后arg\_ptr地址,  
//应该比参数i的地址高sizeof(int)个字节  
//这时arg\_ptr指向下一个参数的地址

j=\*((int \*)arg\_ptr);  
printf("%d %d\n", i, j);   
j=va\_arg(arg\_ptr, int);   
printf("arg\_ptr = %p\n", arg\_ptr);  
//打印va\_arg后arg\_ptr的地址  
//应该比调用va\_arg前高sizeof(int)个字节  
//这时arg\_ptr指向下一个参数的地址  
va\_end(arg\_ptr);   
printf("%d %d\n", i, j);   
}  
void arg\_cnt(int cnt, ...)  
{  
int value=0;   
int i=0;  
int arg\_cnt=cnt;   
va\_list arg\_ptr;   
va\_start(arg\_ptr, cnt);   
for(i = 0; i < cnt; i++)  
{  
   value = va\_arg(arg\_ptr,int);  
   printf("value%d=%d\n", i+1, value);  
}  
}  
void arg\_type(int cnt, ...)  
{  
int arg\_type = 0;  
int int\_value=0;   
int i=0;  
int arg\_cnt=cnt;   
char \*str\_value = NULL;  
va\_list arg\_ptr;   
va\_start(arg\_ptr, cnt);   
for(i = 0; i < cnt; i++)  
{  
   arg\_type = va\_arg(arg\_ptr,int);  
   switch(arg\_type)  
   {  
   case INT\_TYPE:  
    int\_value = va\_arg(arg\_ptr,int);  
    printf("value%d=%d\n", i+1, int\_value);  
    break;  
   case STR\_TYPE:  
    str\_value = va\_arg(arg\_ptr,char\*);  
    printf("value%d=%d\n", i+1, str\_value);  
    break;  
   default:  
    break;  
   }  
}  
}   
=======================================================================

有关VA\_LIST的用法:

VA\_LIST 是在C语言中解决变参问题的一组宏

VA\_LIST的用法：        
       （1）首先在函数里定义一具VA\_LIST型的变量，这个变量是指向参数的指针  
      （2）然后用VA\_START宏初始化变量刚定义的VA\_LIST变量，这个宏的第二个参数是第一个可变参数的前一个参数，是一个固定的参数。  
       （3）然后用VA\_ARG返回可变的参数，VA\_ARG的第二个参数是你要返回的参数的类型。  
       （4）最后用VA\_END宏结束可变参数的获取。然后你就可以在函数里使用第二个参数了。如果函数有多个可变参数的，依次调用VA\_ARG获取各个参数。

VA\_LIST在编译器中的处理：

(1)在运行VA\_START(ap,v)以后，ap指向第一个可变参数在堆栈的地址。  
（2）VA\_ARG()取得类型t的可变参数值，在这步操作中首先apt = sizeof(t类型)，让ap指向下一个参数的地址。然后返回ap-sizeof(t类型)的t类型\*指针，这正是第一个可变参数在堆栈里的地址。然后用\*取得这个地址的内容。  
（3）VA\_END(),X86平台定义为ap = ((char\*)0)，使ap不再指向堆栈，而是跟NULL一样，有些直接定义为((void\*)0)，这样编译器不会为VA\_END产生代码，例如gcc在Linux的X86平台就是这样定义的。

要注意的是：由于参数的地址用于VA\_START宏，所以参数不能声明为寄存器变量，或作为函数或数组类型。

使用VA\_LIST应该注意的问题：  
   （1）因为va\_start, va\_arg, va\_end等定义成宏,所以它显得很愚蠢,可变参数的类型和个数完全在该函数中由程序代码控制,它并不能智能地识别不同参数的个数和类型. 也就是说,你想实现智能识别可变参数的话是要通过在自己的程序里作判断来实现的.  
    （2）另外有一个问题,因为编译器对可变参数的函数的原型检查不够严格,对编程查错不利.不利于我们写出高质量的代码。  
小结：可变参数的函数原理其实很简单，而VA系列是以宏定义来定义的，实现跟堆栈相关。我们写一个可变函数的C函数时，有利也有弊,所以在不必要的 场合，我们无需用到可变参数，如果在C++里，我们应该利用C++多态性来实现可变参数的功能，尽量避免用C语言的方式来实现。

==========================================================================

变长参数应用举例:

先得声明一个变长参数的变量va\_list list  
在使用前要先用va\_start(list, last\_param)对list进行初始化，last\_param为最右边的已知参数，表示list  
从last\_param的下一个参数开始  
va\_arg(list, 类型)  
最后不要忘了用va\_end(list)

eg1:  
#include<iostream>  
#include<iomanip>  
#include<stdarg.h>

using namespace std;

double average(int, ...);

int main()  
{  
    double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

    cout << setiosflags([**iOS**](http://lib.csdn.net/base/ios)::fixed | ios::showpoint)  
        << setprecision(1) << "w = " << w << "\nx = " << x  
        << "\ny = " << y << "\nz = " << z << endl;

    cout << average(2, w, x) << endl;  
    cout << average(3, w, x, y) << endl;  
    cout << average(4, w, x, y, z) << endl;

    return 0;  
}

double average(int i, ...)  
{  
    double total = 0;  
    va\_list ap;

    va\_start(ap, i);

    for(int j = 1; j <= i; j++)  
    {  
        total += va\_arg(ap, double);  
    }

    va\_end( ap );  
    return total/i;  
}

eg2:  
#include<iostream.h>  
#include <stdlib.h>   
#include <stdarg.h>  
void error(const char\*format...);  
void main()  
{  
    int a;  
    char c='d';  
    char s[100];  
    error("Enter a string:");      //输入一个字符串  
    cin>>s;  
    error("Enter an integer:");    //输入一整数  
    cin>>a;  
    error("%s\n%d\n%c\n",s,a,c);   //打印输出

}  
void error(const char\*format...)    //实现像printf函数一样的打印输出功能  
{  
    int i;  
    int j=0;  
    va\_list ap;  
    va\_start(ap,format);  
    for(i=0;\*(format+i)!=0;)  
    {  
        int in;  
        char\* pc;  
        char d;  
        if(\*(format+i)=='%')  
        {  
            switch(\*(format+i+1))  
            {  
            case'd':in=va\_arg(ap,int);cout<<in;i=i+2;break;  
            case's':pc=va\_arg(ap,char\*);cout<<pc;i=i+2;break;  
            case'c':d=va\_arg(ap,char);cout<<d;i=i+2;break;  
            default:cout<<'%';i=i+1;break;  
            }  
        }  
        else  
        {  
            cout<<\*(format+i);  
            i++;  
        }

    }  
}

================================================================

C++变长参数函数的用法

书上说,当无法列出传递函数的所有实参的类型和数目时,可用省略号指定参数表  
(...)

如:void foo(...);  
     void foo(parm\_list,...);  
void foo(...)  
{  
    //...  
}  
调用:foo(a,b,c);

就是不懂,把a,b,c的值传进函数里面后,用什么变量来接收???如果不能接收,(...)岂不是没意义?  
还有就是不明白  
int printf(const char\*...);  
printf("hello,&s\n",userName);

这个c的输出函数是怎么用(...)实现的.

首先函数体中声明一个va\_list，然后用va\_start函数来获取参数列表中的参数，使用完毕后调用va\_end()结束。像这段代码：   
void TestFun(char\* pszDest, int DestLen, const char\* pszFormat, ...)   
{   
va\_list args;   
va\_start(args, pszFormat);   
\_vsnprintf(pszDest, DestLen, pszFormat, args);   
va\_end(args);   
}

===========================================================

va\_list的用法       
还记得printf函数调用的时候那个“...”吗？就是可以输入任意的参数。现在你用va\_list也可以实现类似的函数声明，printf就是这样做的。

va\_list args;                                                 //声明变量   
va\_start(args, before);                               //开始解析。args指向before后面的参数   
参数类型 var = va\_arg(args, 参数类型);     //取下一个参数并返回。args指向下一个参数   
va\_end(args);