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,该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) )     //内存对齐 作用

#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )   // 初始化 ap 的
值，这时 ap 指向 参数 v 的下一个 参数

#define va_arg(ap,t) \
( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )       //返回 ap 指向的数据 并将
ap 指针移向 下一个 参数

#define va_end(ap) ( ap = (va_list)0 )   // 将 ap 的值 置 空
```

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.