_stdcall 是 Pascal 程序的缺省调用方式，通常用于 Win32 Api 中，函数采用从右到左的压栈方式，自己在退出时清空堆栈。VC 将函数编译后会在函数名前面加上下划线前缀，在函数名后加上"@"和参数的字节数。

2、C 调用约定（即用__cdecl 关键字说明）按从右至左的顺序压参数入栈，由调用者把参数弹出栈。对于传送参数的内存栈是由调用者来维护的（正因为如此，实现可变参数的函数只能使用该调用约定）。另外，在函数名修饰约定方面也有所不同。

_cdecl 是 C 和 C＋＋程序的缺省调用方式。每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用_stdcall 函数的大。函数采用从右到左的压栈方式。VC 将函数编译后会在函数名前面加上下划线前缀。是 MFC 缺省调用约定。

3、__fastcall 调用约定是“人”如其名，它的主要特点就是快，因为它是通过寄存器来传送参数的（实际上，它用 ECX 和 EDX 传送前两个双字（DWORD）或更小的参数，剩下的参数仍旧自右向左压栈传送，被调用的函数在返回前清理传送参数的内存栈），在函数名修饰约定方面，它和前两者均不同。

_fastcall 方式的函数采用寄存器传递参数，VC 将函数编译后会在函数名前面加上"@"前缀，在函数名后加上"@"和参数的字节数。

4、thiscall 仅仅应用于“C++”成员函数。this 指针存放于 CX 寄存器，参数从右到左压。thiscall 不是关键词，因此不能被程序员指定。

5、naked call 采用 1-4 的调用约定时，如果必要的话，进入函数时编译器会产生代码来保存 ESI，EDI，EBX，EBP 寄存器，退出函数时则产生代码恢复这些寄存器的内容。naked call 不产生这样的代码。naked call 不是类型修饰符，故必须和_declspec 共同使用。

关键字 __stdcall、__cdecl 和__fastcall 可以直接加在要输出的函数前，也可以在编译环境的 Setting...\C/C++ \Code Generation 项选择。当加在输出函数前的关键字与编译环境中的选择不同时，直接加在输出函数前的关键字有效。它们对应的命令行参数分别为/Gz、/Gd 和/Gr。缺省状态为/Gd，即__cdecl。

要完全模仿 PASCAL 调用约定首先必须使用__stdcall 调用约定，至于函数名修饰约定，可以通过其它方法模仿。还有一个值得一提的是 WINAPI 宏，Windows.h 支持该宏，它可以将出函数翻译成适当的调用约定，在 WIN32 中，它被定义为__stdcall。使用 WINAPI 宏可以创建自己的 APIs。

2)名字修饰约定

1、修饰名(Decoration name)

“C”或者“C++”函数在内部（编译和链接）通过修饰名识别。修饰名是编译器在编译函数定义或者原型时生成的字符串。有些情况下使用函数的修饰名是必要的，如在模块定义文件里头指定输出“C++”重载函数、构造函数、析构函数，又如在汇编代码里调用“C””或“C++”函数等。

修饰名由函数名、类名、调用约定、返回类型、参数等共同决定。

2、名字修饰约定随调用约定和编译种类(C 或 C++)的不同而变化。函数名修饰约定随编译种类和调用约定的不同而不同，下面分别说明。

a、C 编译时函数名修饰约定规则：

__stdcall 调用约定在输出函数名前加上一个下划线前缀，后面加上一个“@”符号和其参数的字节数，格式为_functionname@number。

__cdecl 调用约定仅在输出函数名前加上一个下划线前缀，格式为_functionname。

__fastcall 调用约定在输出函数名前加上一个“@”符号，后面也是一个“@”符号和其参数的字节数，格式为@functionname@number。

它们均不改变输出函数名中的字符大小写，这和 PASCAL 调用约定不同，PASCAL 约定输出的函数名无任何修饰且全部大写。

b、C++编译时函数名修饰约定规则：

__stdcall 调用约定：
1、以“?”标识函数名的开始，后跟函数名；
2、函数名后面以“@@YG”标识参数表的开始，后跟参数表；
3、参数表以代号表示：
X--void，
D--char，
E--unsigned char，
F--short，
H--int，
I--unsigned int，
J--long，
K--unsigned long，
M--float，
N--double，
_N--bool，

....
PA--表示指针，后面的代号表明指针类型，如果相同类型的指针连续出现，以“0”代替，一个“0”代表一次重复；
4、参数表的第一项为该函数的返回值类型，其后依次为参数的数据类型,指针标识在其所指数据类型前；
5、参数表后以“@Z”标识整个名字的结束，如果该函数无参数，则以“Z”标识结束。

其格式为“?functionname@@YG*****@Z”或“?functionname@@YG*XZ”，例如
int Test1（char *var1,unsigned long）-----“?Test1@@YGHPADK@Z”
void Test2（） -----“?Test2@@YGXXZ”

__cdecl 调用约定：
规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YA”。

__fastcall 调用约定：
规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YI”。

VC++对函数的省缺声明是"__cedcl",将只能被 C/C++调用.

CB 在输出函数声明时使用 4 种修饰符号
//__cdecl
cb 的默认值，它会在输出函数名前加_，并保留此函数名不变，参数按照从右到左的顺序依次传递给栈，也可以写成_cdecl 和 cdecl 形式。
//__fastcall
她修饰的函数的参数将尽肯呢感地使用寄存器来处理，其函数名前加@，参数按照从左到右的顺序压栈；
//__pascal
它说明的函数名使用 Pascal 格式的命名约定。这时函数名全部大写。参数按照从左到右的顺序压栈；
//__stdcall
使用标准约定的函数名。函数名不会改变。使用__stdcall 修饰时。参数按照由右到左的顺序压栈，也可以是_stdcall；

```
TITLE   : __stdcall 与__cdecl 的区别
AUTHOR  : lionel@nkbbs.org
DATE    : 01/10/2005
CONTENT:
```

Visual C++ Compiler Options 可以指定的 Calling Convention 有 3 种：

/Gd /Gr /Gz

这三个参数决定了：

1. 函数参数以何种顺序入栈，右到左还是左到右。
2. 在函数运行完后，是调用函数还是被调用函数清理入栈的参数。
3. 在编译时函数名字是如何转换的。

下面我们分别详细介绍：

1. /Gd

这是编译器默认的转换模式，对一般函数使用 C 的函数调用转换方式__cdecl，

但是对于 C++ 成员函数和前面修饰了__stdcall __fastcall 的函数除外。

2. /Gr

对于一般函数使用__fastcall 函数调用转换方式，所有使用__fastcall 的函数

必须要有函数原形。但对于 C++ 成员函数和前面修饰了__cdecl __stdcall 的

函数除外。

3. /Gz

对于所有 C 函数使用__stdcall 函数调用转换方式，但对于可变参数的 C 函数以

及用__cdecl __fastcall 修饰过的函数和 C++ 成员函数除外。所有用__stdcall

修饰的函数必须有函数原形。

事实上，对于 x86 系统，C++ 成员函数的调用方式有点特别，将成员函数的 this

指针放入 ECX，所有函数参数从右向左入栈，被调用的成员函数负责清理入栈的

参数。对于可变参数的成员函数，始终使用__cdecl 的转换方式。

下面该进入主题，分别讲一下这三种函数调用转换方式有什么区别：

1. __cdecl

这是编译器默认的函数调用转换方式，它可以处理可变参数的函数调用。参数

的入栈顺序是从右向左。在函数运行结束后，由调用函数负责清理入栈的参数。

在编译时，在每个函数前面加上下划线(_)，没有函数名大小写的转换。即

```
_functionname
```

2. __fastcall

有一些函数调用的参数被放入ECX,EDX中，而其它参数从右向左入栈。被调用

函数在它将要返回时负责清理入栈的参数。在内嵌汇编语言的时候，需要注意

寄存器的使用，以免与编译器使用的产生冲突。函数名字的转换是：

```
@functionname@number
```

没有函数名大小写的转换，number表示函数参数的字节数。由于有一些参数不

需要入栈，所以这种转换方式会在一定程度上提高函数调用的速度。

3. __stdcall

函数参数从右向左入栈，被调用函数负责入栈参数的清理工作。函数名转换格

式如下：

```
_functionname@number
```

下面我们亲自写一个程序，看看各种不同的调用在编译后有什么区别，我们的被调

用函数如下：

```
int function(int a, int b)
{
```

```
    return a + b;
}

void main()
{
    function(10, 20);
}
```

1.__cdecl

```
    _function
            push    ebp
            mov     ebp, esp
            mov     eax, [ebp+8]        ;参数 1
            add     eax, [ebp+C]        ;加上参数 2
            pop     ebp
            retn
    _main
            push    ebp
            mov     ebp, esp
            push    14h                 ;参数 2 入栈
            push    0Ah                 ;参数 1 入栈
            call    _function           ;调用函数
            add     esp, 8              ;修正栈
            xor     eax, eax
            pop     ebp
            retn
```

2.__fastcall

```
    @function@8
            push    ebp
            mov     ebp, esp            ;保存栈指针
            sub     esp, 8              ;多了两个局部变量
            mov     [ebp-8], edx        ;保存参数 2
            mov     [ebp-4], ecx        ;保存参数 1
            mov     eax, [ebp-4]        ;参数 1
            add     eax, [ebp-8]        ;加上参数 2
```

```
                mov     esp, ebp            ;修正栈
                pop     ebp
                retn
        _main
                push    ebp
                mov     ebp, esp
                mov     edx, 14h            ;参数 2 给 EDX
                mov     ecx, 0Ah            ;参数 1 给 ECX
                call    @function@8         ;调用函数
                xor     eax, eax
                pop     ebp
                retn
```

3.__stdcall

```
        _function@8
                push    ebp
                mov     ebp, esp
                mov     eax, [ebp]          ;参数 1
                add     eax, [ebp+C]        ;加上参数 2
                pop     ebp
                retn    8                   ;修复栈
        _main
                push    ebp
                mov     ebp, esp
                push    14h                 ;参数 2 入栈
                push    0Ah                 ;参数 1 入栈
                call    _function@8         ;函数调用
                xor     eax, eax
                pop     ebp
                retn
```

可见上述三种方法各有各的特点，而且_main 必须是__cdecl，一般 WIN32 的函数都是

__stdcall。而且在 Windef.h 中有如下的定义：

```
#define CALLBACK __stdcall
#define WINAPI   __stdcall
```

论调用约定

在 C 语言中，假设我们有这样的一个函数：

```
int function(int a,int b)
```

调用时只要用 result = function(1,2)这样的方式就可以使用这个函数。但是，当高级语言被编译成计算机可以识别的机器码时，有一个问题就凸现出来：在 CPU 中，计算机没有办法知道一个函数调用需要多少个、什么样的参数，也没有硬件可以保存这些参数。也就是说，计算机不知道怎么给这个函数传递参数，传递参数的工作必须由函数调用者和函数本身来协调。为此，计算机提供了一种被称为栈的数据结构来支持参数传递。

栈是一种先进后出的数据结构，栈有一个存储区、一个栈顶指针。栈顶指针指向堆栈中第一个可用的数据项（被称为栈顶）。用户可以在栈顶上方向栈中加入数据，这个操作被称为压栈(Push)，压栈以后，栈顶自动变成新加入数据项的位置，栈顶指针也随之修改。用户也可以从堆栈中取走栈顶，称为弹出栈(pop)，弹出栈后，栈顶下的一个元素变成栈顶，栈顶指针随之修改。

函数调用时，调用者依次把参数压栈，然后调用函数，函数被调用以后，在堆栈中取得数据，并进行计算。函数计算结束以后，或者调用者、或者函数本身修改堆栈，使堆栈恢复原装。

在参数传递中，有两个很重要的问题必须得到明确说明：

- 当参数个数多于一个时，按照什么顺序把参数压入堆栈
- 函数调用后，由谁来把堆栈恢复原装

在高级语言中，通过函数调用约定来说明这两个问题。常见的调用约定有：

- stdcall
- cdecl
- fastcall
- thiscall
- naked call

stdcall 调用约定

stdcall 很多时候被称为 pascal 调用约定，因为 pascal 是早期很常见的一种教学用计算机程序设计语言，其语法严谨，使用的函数调用约定就是 stdcall。在 Microsoft C++系列的 C/C++编译器中，常常用 PASCAL 宏来声明这个调用约定，类似的宏还有 WINAPI 和 CALLBACK。

stdcall 调用约定声明的语法为(以前文的那个函数为例）：

int __stdcall function(int a,int b)
stdcall 的调用约定意味着:1)参数从右向左压入堆栈,2)函数自身修改堆栈 3)函数名自动加前导的下划线，后面紧跟一个@符号，其后紧跟着参数的尺寸
以上述这个函数为例,参数 b 首先被压栈,然后是参数 a,函数调用 function(1,2)调用处翻译成汇编语言将变成:
push 2 第二个参数入栈 push 1 第一个参数入栈 call function 调用参数，注意此时自动把 cs:eip 入栈

而对于函数自身，则可以翻译为:
push ebp 保存 ebp 寄存器，该寄存器将用来保存堆栈的栈顶指针，可以在函数退出时恢复 mov ebp,esp 保存堆栈指针 mov eax,[ebp + 8H] 堆栈中 ebp 指向位置之前依次保存有 ebp,cs:eip,a,b,ebp +8 指向 a add eax,[ebp + 0CH] 堆栈中 ebp + 12 处保存了 b mov esp,ebp 恢复 esp pop ebp ret 8

而在编译时，这个函数的名字被翻译成_function@8
注意不同编译器会插入自己的汇编代码以提供编译的通用性,但是大体代码如此。其中在函数开始处保留 esp 到 ebp 中，在函数结束恢复是编译器常用的方法。从函数调用看，2 和 1 依次被 push 进堆栈，而在函数中又通过相对于 ebp(即刚进函数时的堆栈指针）的偏移量存取参数。函数结束后，ret 8 表示清理 8 个字节的堆栈，函数自己恢复了堆栈。

cdecl 调用约定

cdecl 调用约定又称为 C 调用约定,是 C 语言缺省的调用约定,它的定义语法是:
int function (int a ,int b) //不加修饰就是 C 调用约定 int __cdecl function(int a,int b)//明确指出 C 调用约定

在写本文时，出乎我的意料，发现 cdecl 调用约定的参数压栈顺序是和 stdcall 是一样的，参数首先由有向左压入堆栈。所不同的是，函数本身不清理堆栈，调用者负责清理堆栈。由于这种变化，C 调用约定允许函数的参数的个数是不固定的，这也是 C 语言的一大特色。对于前面的 function 函数，使用 cdecl 后的汇编码变成:
调用处 push 1 push 2 call function add esp,8 **注意：这里调用者在恢复堆栈 被调用函数_function 处** push ebp 保存 ebp 寄存器，该寄存器将用来保存堆栈的栈顶指针，可以在函数退出时恢复 mov ebp,esp 保存堆栈指针 mov eax,[ebp + 8H] 堆栈中 ebp 指向位置之前依次保存有 ebp,cs:eip,a,b,ebp +8 指向 a add eax,[ebp + 0CH] 堆栈中 ebp + 12 处保存了 b mov esp,ebp 恢复 esp pop ebp ret **注意，这里没有修改堆栈**

MSDN 中说，该修饰自动在函数名前加前导的下划线，因此函数名在符号表中被记录为_function，但是我在编译时似乎没有看到这种变化。
由于参数按照从右向左顺序压栈，因此最开始的参数在最接近栈顶的位置，因此当采用不定个数参数时，第一个参数在栈中的位置肯定能知道，只要不定的参数个数能够根据第一个后者后续的明确的参数确定下来，就可以使用不定参数，例如对于 CRT 中的 sprintf 函数，定义为：

```
int sprintf(char* buffer,const char* format,…)
```

由于所有的不定参数都可以通过 format 确定，因此使用不定个数的参数是没有问题的。

fastcall

fastcall 调用约定和 stdcall 类似，它意味着：

- 函数的第一个和第二个 DWORD 参数（或者尺寸更小的）通过 ecx 和 edx 传递，其他参数通过从右向左的顺序压栈
- 被调用函数清理堆栈
- 函数名修改规则同 stdcall

其声明语法为：int fastcall function(int a,int b)

thiscall

thiscall 是唯一一个不能明确指明的函数修饰，因为 thiscall 不是关键字。它是 C++类成员函数缺省的调用约定。由于成员函数调用还有一个 this 指针，因此必须特殊处理，thiscall 意味着：

- 参数从右向左入栈
- 如果参数个数确定，this 指针通过 ecx 传递给被调用者；如果参数个数不确定，this 指针在所有参数压栈后被压入堆栈。
- 对参数个数不定的，调用者清理堆栈，否则函数自己清理堆栈

为了说明这个调用约定，定义如下类和使用代码：

```
class A { public:    int function1(int a,int b);    int function2(int a,...);}; int A::function1 (int a,int b) {    return a+b;} #include  int A::function2(int a,…) {    va_list ap;    va_start(ap,a);    int i;    int result = 0;    for(i = 0 ; i < a ; i ++)    {      result += va_arg(ap,int);    }    return result; } void callee() {    A a;    a.function1 (1,2);    a.function2(3,1,2,3); }
```

callee 函数被翻译成汇编后就变成：

```
 //函数 function1 调用 0401C1D push 2 00401C1F push 1 00401C21 lea ecx,[ebp-8] 00401C24 call function1 注意，这里 this 没有被入栈 //函数
```

function2 调用 00401C29 push 3 00401C2B push 2 00401C2D push 1 00401C2F push 3 00401C31 lea eax,[ebp-8] 这里引入 this 指针 00401C34 push eax 00401C35 call function2 00401C3A add esp,14h

可见，对于参数个数固定情况下，它类似于 stdcall，不定时则类似 cdecl

naked call

这是一个很少见的调用约定，一般程序设计者建议不要使用。编译器不会给这种函数增加初始化和清理代码，更特殊的是，你不能用 return 返回返回值，只能用插入汇编返回结果。这一般用于实模式驱动程序设计，假设定义一个求和的加法程序，可以定义为：

```
__declspec(naked) int add(int a,int b) { __asm mov eax,a __asm add eax,b __asm ret }
```

注意，这个函数没有显式的 return 返回值，返回通过修改 eax 寄存器实现，而且连退出函数的 ret 指令都必须显式插入。上面代码被翻译成汇编以后变成：

```
 mov eax,[ebp+8] add eax,[ebp+12] ret 8
```

注意这个修饰是和__stdcall 及 cdecl 结合使用的，前面是它和 cdecl 结合使用的代码，对于和 stdcall 结合的代码，则变成：

```
__declspec(naked) int __stdcall function(int a,int b) { __asm mov eax,a __asm add eax,b __asm ret 8 //注意后面的 8 }
```

至于这种函数被调用，则和普通的 cdecl 及 stdcall 调用函数一致。

函数调用约定导致的常见问题

如果定义的约定和使用的约定不一致，则将导致堆栈被破坏，导致严重问题，下面是两种常见的问题：

1. 函数原型声明和函数体定义不一致
2. DLL 导入函数时声明了不同的函数约定

以后者为例，假设我们在 dll 种声明了一种函数为：

```
__declspec(dllexport) int func(int a,int b);//注意，这里没有 stdcall，使用的是 cdecl
```

使用时代码为：

```
 typedef int (*WINAPI DLLFUNC)func(int a,int b); hLib = LoadLibrary(...); DLLFUNC func = (DLLFUNC)GetProcAddress(...)//这里修改了调用约定 result = func(1,2);//导致错误
```

由于调用者没有理解 WINAPI 的含义错误的增加了这个修饰，上述代码必然导致堆栈被破坏，MFC 在编译时插入的 checkesp 函数将告诉你，堆栈被破坏了。