10,186,615 members (69,027 online)

Sign in

CODE PROJECT® For those who code

home articles quick answers discussions features community help

Search for articles, questions, tips

Articles » Languages » C / C++ Language » General

Next →

Article

Browse Code

Stats

Revisions

Alternatives

Comments & Discussions (46)

Calling Conventions Demystified

By **Nemanja Trifunovic**, 22 Sep 2001

Sign Up to vote

★ ★ ★ ★ ★ 4.88 (144 votes) Like 7

Tweet 4 1

About Article

Visual C++ calling conventions explained

Type	**Article**
Licence	**CPOL**
First Posted	**22 Sep 2001**
Views	**248,461**
Bookmarked	**155 times**

VC6 Win2K Visual-Studio Dev +

Print Email

Introduction

During the long, hard, but yet beautiful process of learning C++ programming for Windows, you have probably been curious about the strange specifiers that sometime appear in front of function declarations, like `__cdecl`, `__stdcall`, `__fastcall`, `WINAPI`, etc. After looking through MSDN, or some other reference, you probably found out that these specifiers specify the *calling conventions* for functions. In this article, I will try to explain different calling conventions used by Visual C++ (and probably other Windows C/C++ compilers). I emphasize that above mentioned specifiers are Microsoft-specific, and that you should not use them if you want to write portable code.

So, what are the calling conventions? When a function is called, the arguments are typically passed to it, and the return value is retrieved. A calling convention describes *how* the arguments are passed and values returned by functions. It also specifies how the function names are decorated. Is it really necessary to understand the calling conventions to write good C/C++ programs? Not at all. However, it may be helpful with debugging. Also, it is necessary for linking C/C++ with assembly code.

To understand this article, you will need to have some very basic knowledge of assembly programming.

No matter which calling convention is used, the following things will happen:

1. All arguments are widened to 4 bytes (on Win32, of course), and put into appropriate memory locations. These locations are typically on the stack, but may also be in registers; this is specified by calling conventions.
2. Program execution jumps to the address of the called function.
3. Inside the function, registers ESI, EDI, EBX, and EBP are saved on the stack. The part of code that performs these operations is called *function prolog* and usually is generated by the compiler.
4. The function-specific code is executed, and the return value is placed into the EAX register.
5. Registers ESI, EDI, EBX, and EBP are restored from the stack. The piece of code that does this is called *function epilog*, and as with the function prolog, in most cases the compiler generates it.
6. Arguments are removed from the stack. This operation is called *stack cleanup* and may be performed either inside the called function or by the caller, depending on the calling convention used.

As an example for the calling conventions (except for *this*), we are going to use a simple function:

⊟ Collapse | Copy Code

```
int sumExample (int a, int b)
{
    return a + b;
}
```

The call to this function will look like this:

⊟ Collapse | Copy Code

```
    int c = sum (2, 3);
```

For __cdecl, __stdcall, and __fastcall calling conventions, I compiled the example code as C (not C++). The function name decorations, mentioned later in the article, apply to the C decoration schema. C++ name decorations are beyond the scope of this article.

Top News

6 dirty secrets of the IT industry

Get the Insider News free each

C calling convention (__cdecl)

This convention is the default for C/C++ programs (compiler option /Gd). If a project is set to use some other calling convention, we can still declare a function to use __cdecl:

⊟ Collapse | Copy Code

```
int __cdecl sumExample (int a, int b);
```

The main characteristics of __cdecl calling convention are:

1. Arguments are passed from right to left, and placed on the stack.
2. Stack cleanup is performed by the caller.
3. Function name is decorated by prefixing it with an underscore character '_' .

Now, take a look at an example of a __cdecl call:

⊟ Collapse | Copy Code

```
; // push arguments to the stack, from right to left
push        3
push        2

; // call the function
call        _sumExample

; // cleanup the stack by adding the size of the arguments to ESP register
add         esp,8

; // copy the return value from EAX to a local variable (int c)
mov         dword ptr [c],eax
```

The called function is shown below:

⊟ Collapse | Copy Code

```
; // function prolog
  push        ebp
  mov         ebp,esp
  sub         esp,0C0h
  push        ebx
  push        esi
  push        edi
  lea         edi,[ebp-0C0h]
  mov         ecx,30h
  mov         eax,0CCCCCCCCh
  rep stos    dword ptr [edi]

; //    return a + b;
  mov         eax,dword ptr [a]
  add         eax,dword ptr [b]

; // function epilog
  pop         edi
  pop         esi
  pop         ebx
  mov         esp,ebp
  pop         ebp
  ret
```

Standard calling convention (__stdcall)

This convention is usually used to call Win32 API functions. In fact, WINAPI is nothing but another name for __stdcall:

⊟ Collapse | Copy Code

```
#define WINAPI __stdcall
```

We can explicitly declare a function to use the __stdcall convention:

⊟ Collapse | Copy Code

```
int __stdcall sumExample (int a, int b);
```

Also, we can use the compiler option */Gz* to specify __stdcall for all functions not explicitly declared with some other calling convention.

The main characteristics of __stdcall calling convention are:

1. Arguments are passed from right to left, and placed on the stack.
2. Stack cleanup is performed by the called function.
3. Function name is decorated by prepending an underscore character and appending a '@' character and the number of bytes of stack space required.

The example follows:

morning.

Related Videos

Related Articles

.NET file format - Signatures under the hood, Part 1 of 2

How to invoke C++ member operations from inline-assembler code segments

Using the _CDECL calling convention in C# (changing compiler service)

Compilers Demystified: Function Pointers in Visual Basic 6.0

P/Invoke Tutorial: Basics (Part 1)

WPFSpark: 1 of n: SprocketControl

Using Assembly routines in Visual C++

Using C calling convention callback functions in C# and VB - the easy way

Finite State Machine and Multithreading using .NET

Thread synchronization: Wait and Pulse demystified

Calling Conventions for SSI Test Modules

Globalization and localization demystified in ASP.NET 2.0

Some practices to write better C#/.NET code

Static Keyword Demystified

Late binding on native DLLs with C#

Switching threads

Case-Insensitive String Search

Step by Step: Calling C++ DLLs from VC++ and VB - Part 2

Accessing an API through a closed source C++ DLL

Calling DLLs in VB and C++

Related Research

The Essential Guide to Mobile App Testing

⊟ Collapse | Copy Code

```
; // push arguments to the stack, from right to left
  push        3    
  push        2    

; // call the function
  call        _sumExample@8

; // copy the return value from EAX to a local variable (int c)
  mov         dword ptr [c],eax
```

The function code is shown below:

⊟ Collapse | Copy Code

```
; // function prolog goes here (the same code as in the __cdecl example)

; //    return a + b;
  mov         eax,dword ptr [a] 
  add         eax,dword ptr [b] 

; // function epilog goes here (the same code as in the __cdecl example)

; // cleanup the stack and return
  ret         8
```

Because the stack is cleaned by the called function, the `__stdcall` calling convention creates smaller executables than `__cdecl`, in which the code for stack cleanup must be generated for each function call. On the other hand, functions with the variable number of arguments (like `printf()`) must use `__cdecl`, because only the caller knows the number of arguments in each function call; therefore only the caller can perform the stack cleanup.

Fast calling convention (__fastcall)

Fast calling convention indicates that the arguments should be placed in registers, rather than on the stack, whenever possible. This reduces the cost of a function call, because operations with registers are faster than with the stack.

We can explicitly declare a function to use the `__fastcall` convention as shown:

⊟ Collapse | Copy Code

```
int __fastcall sumExample (int a, int b);
```

We can also use the compiler option */Gr* to specify `__fastcall` for all functions not explicitly declared with some other calling convention.

The main characteristics of `__fastcall` calling convention are:

1. The first two function arguments that require 32 bits or less are placed into registers ECX and EDX. The rest of them are pushed on the stack from right to left.
2. Arguments are popped from the stack by the called function.
3. Function name is decorated by by prepending a '@' character and appending a '@' and the number of bytes (decimal) of space required by the arguments.

Note: Microsoft have reserved the right to change the registers for passing the arguments in future compiler versions.

Here goes an example:

⊟ Collapse | Copy Code

```
; // put the arguments in the registers EDX and ECX
  mov         edx,3 
  mov         ecx,2 

; // call the function
  call        @fastcallSum@8

; // copy the return value from EAX to a local variable (int c)
  mov         dword ptr [c],eax
```

Function code:

⊟ Collapse | Copy Code

```
; // function prolog

  push        ebp  
  mov         ebp,esp 
  sub         esp,0D8h 
  push        ebx  
  push        esi  
  push        edi  
  push        ecx  
  lea         edi,[ebp-0D8h] 
  mov         ecx,36h
  mov         eax,0CCCCCCCCh
  rep stos    dword ptr [edi]
  pop         ecx
  mov         dword ptr [ebp-14h],edx
  mov         dword ptr [ebp-8],ecx
; // return a + b;
  mov         eax,dword ptr [a]
  add         eax,dword ptr [b]
;// function epilog
  pop         edi
  pop         esi
  pop         ebx
  mov         esp,ebp
  pop         ebp
  ret
```

How fast is this calling convention, comparing to `__cdecl` and `__stdcall`? Find out for yourselves. Set the compiler option */Gr*, and compare the execution time. I didn't find `__fastcall` to be any faster than other calling conventons, but you may come to different conclusions.

Thiscall

`Thiscall` is the default calling convention for calling member functions of C++ classes (except for those with a variable number of arguments).

The main characteristics of `thiscall` calling convention are:

1. Arguments are passed from right to left, and placed on the stack. `this` is placed in ECX.
2. Stack cleanup is performed by the called function.

The example for this calling convention had to be a little different. First, the code is compiled as C++, and not C. Second, we have a struct with a member function, instead of a global function.

⊟ Collapse | Copy Code

```
struct CSum
{
    int sum ( int a, int b) {return a+b;}
};
```

The assembly code for the function call looks like this:

⊟ Collapse | Copy Code

```
  push        3
  push        2
  lea         ecx,[sumObj]
  call        ?sum@CSum@@QAEHHH@Z            ; CSum::sum
  mov         dword ptr [s4],eax
```

The function itself is given below:

⊟ Collapse | Copy Code

```
  push        ebp
  mov         ebp,esp
  sub         esp,0CCh
  push        ebx
  push        esi
  push        edi
  push        ecx
  lea         edi,[ebp-0CCh]
  mov         ecx,33h
  mov         eax,0CCCCCCCCh
  rep stos    dword ptr [edi]
  pop         ecx
  mov         dword ptr [ebp-8],ecx
  mov         eax,dword ptr [a]
  add         eax,dword ptr [b]
  pop         edi
  pop         esi
  pop         ebx
  mov         esp,ebp
  pop         ebp
  ret         8
```

Now, what happens if we have a member function with a variable number of arguments? In that case, `__cdecl` is used, and `this` is pushed onto the stack last.

Conclusion

To cut a long story short, we'll outline the main differences between the calling conventions:

- `__cdecl` is the default calling convention for C and C++ programs. The advantage of this calling convetion is that it allows functions with a variable number of arguments to be used. The disadvantage is that it creates larger executables.

- `__stdcall` is used to call Win32 API functions. It does not allow functions to have a variable number of arguments.
- `__fastcall` attempts to put arguments in registers, rather than on the stack, thus making function calls faster.
- `Thiscall` calling convention is the default calling convention used by C++ member functions that do not use variable arguments.

In most cases, this is all you'll ever need to know about the calling conventions.

License

About the Author

Nemanja Trifunovic

Software Developer (Senior) SAP
United States

Born in Kragujevac, Serbia. Now lives in Boston area with his wife and daughters.

Wrote his first program at the age of 13 on a Sinclair Spectrum, became a professional software developer after he graduated.

Very passionate about programming and software development in general.

Article Top

Comments and Discussions

You must **Sign In** to use this message board.

Search this forum Go

☑ Profile popups Spacing Relaxed Noise Very High Layout Normal Per page 10 Update

First Prev Next

Differences in Usage in C/C++	**rnjailamba**	**29-Oct-13 21:26**
My vote of 5	**DaveyM69**	**22-May-13 3:32**
Re: My vote of 5	Nemanja Trifunovic	22-May-13 4:54
Re: My vote of 5	DaveyM69	22-May-13 21:49
My vote of 5	**hbobenicio**	**22-Jan-13 7:21**
link different calling conventions	**Member 3020704**	**29-Nov-12 3:05**
My vote of 5	**victory2381**	**20-Sep-12 11:04**
My vote of 5	**Member 9416287**	**18-Sep-12 0:02**

Benifit a lot here!	**Vick Qi**	**27-Jul-12 7:04**
My vote of 4	**Member 4417071**	**18-Jun-12 22:43**

Last Visit: 31-Dec-99 18:00 Last Update: 4-Nov-13 10:53 Refresh 1 2 3 4 5 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink | Advertise | Privacy | Mobile
Web02 | 2.7.131030.2 | Last Updated 23 Sep 2001

Layout: fixed | fluid