Calling conventions
for different C++ compilers and operating systems

By Agner Fog, www.agner.org

Contents

1 Introduction

This report describes differences between various C++ compilers that affect binary compatibility, such as data storage, function calling conventions, and name mangling. The function calling methods, name mangling schemes, etc. are described in detail for each compiler.

The purposes of publishing this information are:

- point out incompatibilities between compilers
- make new compilers compatible with old ones
- solve compatibility problems between DLLs produced by different compilers
- facilitate linking different programming languages together
- facilitate the making of assembly subroutines that are compatible with multiple compilers and operating systems
- solve compatibility problems for data stored in binary files
- facilitate debugging and disassembling of object files
- facilitate the construction of debugging and profiling tools
- facilitate the construction of object file conversion utilities
- provoke compiler vendors to use open standards
- inspire future standardization

Hardware platforms covered:

- x86 microprocessor 16 bit architecture and its extensions to 32 and 64 bits. This includes the Intel processors from 8086 to Pentium 4, Celeron and Xeon, as well as compatible processors from AMD and other vendors. (The 32 bit extension is called IA32. The 64 bit extension is called x86-64, x64, AMD64, or EM64T).

The IA64 architecture, which is implemented in Intel's Itanium processor, is not compatible with the x86 architecture, and is not covered in this report.

Operating systems covered:

- DOS, 16 bit
- Windows, 16 bit, 32 bit and 64 bit
- Linux, 32 bit and 64 bit
- FreeBSD, 32 bit and 64 bit

C++ compilers tested:

- Borland, 16 bit v. 3.0 and 5.0
- Microsoft, 16 bit, v. 8.0
- Watcom, 16 bit v. 1.2
- Borland 32 bit v. 5.0
- Microsoft, 32 bit, v. 9.0 and 13.10
- Gnu, 32 bit, v. 2.95, 3.3.3, and several other versions under Linux, FreeBSD and Windows
- Watcom, 32 bit, v. 1.2
- Symantec, 32 bit, v. 7.5
- Digital Mars, 32 bit, v. 8.3.8
- Intel, 32 bit for Windows and Linux, v. 8.1
- Microsoft, 64 bit, v. 14.00
- Gnu, 64 bit, v. 3.3.3 (Linux and FreeBSD)
- Intel, 64 bit for Windows and Linux, v. 8.1

This document provides information that is typically difficult to find. The documentation of calling conventions and binary interfaces of compilers and operating systems is often shamefully poor or completely absent. Name mangling schemes are rarely documented. For example, it is stated in Microsoft Knowledge Base that "Microsoft does not publish the algorithm its compilers use for name decoration because it may change in the future." (article number Q126845). However, the name mangling scheme does not seem to have changed much from the old 16-bit compiler to the newest 32-bit and 64-bit compilers.

As most of the information given here is based on my own experiments, it is obviously not authoritative, and is not guaranteed to be accurate or complete. This document tells how things are, not how they are supposed to be. Many details appear to be the haphazard consequences of how compilers happen to be implemented rather than results of careful planning. Calling "conventions" may not be the most appropriate term in this case, but it may be necessary to copy the quirks of existing compilers when full compatibility is desired.

I have no knowledge about whether any information provided here is protected by patents or other legal restrictions, but I have found no specific patent markings on the compilers.

I have gathered this information mainly by converting C++ code to assembly. All the compilers I have tested are capable of converting C++ to assembly, either directly or via object files. The reader is encouraged to do your own research, if necessary, to get additional information needed or to clarify any questions you may have. The easiest way of doing this research is to make the compiler convert a C++ test file to assembly. Other possible methods are to use object file dump utilities, disassembly utilities, or provoke error messages from a linker. If you find any errors in this document, please let me know.

Please note that I don't have the time and resources to help everybody with their programming problems. If you Email me with such questions, you will not get any answer. You may send your questions to appropriate internet forums instead.

2 The need for standardization

In the days of the old DOS operating system, it was often possible to combine development tools from different vendors with few compatibility problems. With 32-bit Windows, the situation has gone completely out of hand. Different compilers use different data representations, different function calling conventions, and different object file formats. While static link libraries have traditionally been considered compiler-specific, the introduction of dynamic link libraries (DLL's) has made the distribution of function libraries in binary form more common. Unfortunately, the standardization of data representation and calling conventions that would make DLL's compatible is still lacking.

In the Linux and BSD operating systems, there are fewer compatibility problems because the GNU development tools define a de facto standard, though not always fully documented. Modules compiled with Gnu C++ version 2.x and version 3.x are not mutually compatible.

Fortunately, there is a growing recognition of the need for standardization of application binary interfaces (ABI's). The ABI's for the new 64-bit operating systems are specified in much more detail than we have seen in older operating systems. However, these ABI's still lack specification of name mangling schemes and other details. Traditionally, compiler vendors have not published or standardized their name mangling schemes. A common excuse was that the object files would not be compatible anyway because of differences in data formats and calling conventions. Now that data formats and calling conventions are specified in the ABI's, there is no more excuse for not publishing and standardizing name mangling schemes as well. It is my hope that this document will be a contribution towards this end.

Compilers and other development tools is an area where *de facto* standards play an important role. For example, the C++ Windows compilers from Intel, Symantec and Digital Mars are all designed to be binary compatible with Microsoft's C++ compilers, despite the fact that Microsoft has never published the relevant standards. Likewise, Intel's C++ compilers for Linux are binary compatible with the Gnu C++ compilers. It is highly recommended that designers of development tools follow all available standards. Where no official standard exists, use an existing compiler for reference. Use the Microsoft compiler as a reference for Windows systems and the Gnu compiler as a reference for UNIX-like systems. For features that are not supported by these compilers, use the Intel compiler for reference. The calling conventions of these compilers may be considered de facto standards for Windows and UNIX platforms.

3 Data representation

Table 1. Data sizes

segment size	**16 bit**			**32 bit**						**64 bit**			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
bool	2	1	1	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1	1	1	1
wchar_t		2		2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2	2	2	2
int	2	2	2	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	4	4	4	8	8
__int64				8	8			8	8	8	8	8	8
enum	2	2	1	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8	8	8	8
long double	10	10	8	8	16	10	8	12	12	8	8	16	16
__m64				8	8				8				8
__m128				16	16				16	16	16		16
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	4	4	4										
function pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
data member pointer (min)	2	4	6	4	4	8	4	4	4	4	4	8	8
data member pointer (max)		4	6	12	12	8	12	4	4	12	12	8	8
member function pointer (min)	2	12	6	4	4	12	4	8	8	8	8	16	16
member function pointer (max)		12	6	16	16	12	16	8	8	24	24	16	16

Table 1 shows how many bytes of storage various objects use for different compilers.

Differences in data representation can cause problems when exchanging binary data files between programs, when exchanging data with a DLL compiled with a different compiler, and when porting C++ code that relies on a specific data format.

Bool

The type `bool` typically uses one byte of storage where all bits are significant. 0 indicates false and all other values indicate `true`. Most compilers will always store the value `true` as 1. However, none of the compilers I have tested take advantage of the fact that the only possible values are 0 and 1, even if the performance could be improved significantly by relying on the fact that other values are impossible. The ABI for 64 bit Linux/BSD specifies that other values than 0 and 1 are allowed only for function parameters and returns, not for memory objects. (The optimal convention would be to never allow other values than 0 and 1. This would make it possible to implement Boolean expressions without the use of expensive branch instructions except where the evaluation of the second operand of `&&` or `||` has side effects).

Integers

Signed integers are stored in 2-complement representation.

Floating point numbers

Floating point numbers are stored according to the IEEE-754 standard. The most significant bit of the mantissa is explicit (=1) in `long double` and implicit in `float` and `double`.

The x86 architecture specifies 10 bytes for `long double`. Some compilers don't support this precision, but store `long double` as `double`, using 8 bytes. Other compilers use more than 10 bytes for the sake of alignment. The extra bytes are unused, even if subsequent objects would fit into this unused space. The 32-bit Intel compiler for Windows has options for storing long double either as 8 bytes or 16 bytes.

Member pointers

A class data member pointer basically contains the offset of the member relative to the beginning of the object. A member function pointer basically contains the address of the member function.

Data member pointers and member function pointers may use extra storage in the general case in order to account for rare cases of multiple inheritance etc. The minimum value in table 1 applies to simple cases, the maximum value applies to the case where the compiler doesn't have any information about the class other than its name. Some compilers have options to cover this case in different ways. The extra information is stored in ways that are poorly documented and poorly standardized. The "Itanium C++ ABI" includes more detailed information about the representation of member pointers. This information may apply to other platforms as well. More information on the implementation of member pointers in different compilers can be found in "Member Function Pointers and the Fastest Possible C++ Delegates", by Don Clugston, www.codeproject.com/cpp/FastDelegate.asp

Borland compilers add an offset of 1 to data member pointers in order to distinguish a pointer to the first data member from a `NULL` pointer, represented by 0. The other compilers have no offset, but represent a `NULL` data member pointer by the value -1.

1 and 2-byte types in Gnu compiler

Gnu compilers always zero-extend or sign-extend function return values to 32 bits if the values are less than 32 bits in order to conform to a certain interpretation of the C standard. The 64 bit Gnu compiler sign-extends signed values to 32 bits rather than to 64 bits. The extension to 32 bits appears to be completely superfluous since the calling function will repeat the zero-extension or sign-extension operation if needed rather than relying on the higher bits being valid.

Arrays and strings

Arrays are stored as consecutive objects in memory. No information about the size of the array is included. Multidimensional arrays are stored with the last index as least significant. Arrays are passed to functions as pointers without copying. Strings are stored as arrays with a terminating element of 0.

Most programming languages other than C and C++ store arrays and strings in ways that include the size.

Composite objects

Objects of structures and classes are stored by placing the members consecutively in memory with possible padding for the sake of alignment. Additional information for virtual tables and runtime type identification may be added, as described in section 11.

4 Data alignment

Table 2. Alignment of static data

segment size	**16 bit**			**32 bit**						**64 bit**			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
1 byte char	1	1	1	1	4	1	1	1	4	1	4	1	4
2 byte int	2	2	2	4	4	2	2	2	4	4	4	2	4
4 byte int	2	2	4	4	4	4	4	4	4	4	4	4	4
8 byte int	2	2	8	8	8	4	8	8	8	8	8	8	8
float	2	2	4	4	4	4	4	4	4	4	4	4	4
double	2	2	8	8	8	4	8	8	8	8	8	8	8
long double	2	2	8	8	16	4	8	16	4	8	8	16	16
__m64					8				8				8
__m128					16				16		16		16
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	2	2	2										
big array	2	1-2	2-8	4-8	512	1-4	2-8	32	32	4-8	256	32	32
big structure	2	1	2	4	32	1	8	32	32	4	32	32	32

Table 2 shows the default alignment in bytes of static data. The alignment affects performance, but not compatibility.

Table 3. Alignment of structure members

segment size	**16 bit**			**32 bit**						**64 bit**			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
1 byte char	1	1	1	1	1	1	1	1	1	1	1	1	1
2 byte int	2	1	2	2	2	1	2	2	2	4	4	2	2
4 byte int	2	1	2	4	4	1	4	4	4	4	4	4	4
8 byte int	2	1	2	8	8	1	8	8	8	8	8	8	8
float	2	1	2	4	4	1	4	4	4	4	4	4	4
double	2	1	2	8	8	1	8	8	8	8	8	8	error
long double	2	1	2	8	16	1	8	16	16	8	8	16	16
__m64					8				8				8
__m128					16				16		16		16
pointer	2	1	2	4	4	1	4	4	4	8	8	8	8
far pointer	2	1	2										

Table 3 shows the alignment in bytes of data members of structures and classes. The compiler will insert unused bytes, as required, between members to obtain this alignment. The compiler will also insert unused bytes at the end of the structure so that the total size of the structure is a multiple of the alignment of the element that requires the highest alignment. Many compilers have options to change the default alignments.

Differences in structure member alignment will cause incompatibility between different programs or modules accessing the same data and when data are stored in binary files.

The programmer can avoid such compatibility problems by ordering the structure members so that no unused bytes need to be inserted. Likewise, the padding at the end of the structure may be specified explicitly by inserting dummy members of the required size. The size of the virtual table pointer, if any, must be taken into account (see section 11).

5 Stack alignment

The stack pointer must be aligned by the stack word size at all times.

If higher alignment is needed in 32-bit mode, then you may use the methods described in Intel's application note AP 589 "Software Conventions for Streaming SIMD Extensions", "Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel® C/C++ Compiler", and "IA-32 Intel ® Architecture Optimization Reference Manual".

The 64 bit systems have higher alignment requirements. The stack word size is 8 bytes, but the stack must be aligned by 16 before any call instruction. Consequently, the value of the stack pointer is always 8 modulo 16 at the entry of a procedure. A procedure must subtract an odd multiple of 8 from the stack pointer before any call instruction. A procedure can rely on these rules when using XMM instructions that require 16-byte alignment. This applies to all 64 bit systems (Windows, Linux and FreeBSD).

6 Register usage

Table 4. Register usage

	16 bit DOS, Windows	32 bit Windows, Linux, FreeBSD	64 bit Windows	64 bit Linux, FreeBSD
scratch registers	AX, BX, CX, DX, ES, ST(0)-ST(7)	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7	RAX, RCX, RDX, R8-R11, ST(0)-ST(7), XMM0-XMM5, high half of XMM6-XMM15	RAX, RCX, RDX, RSI, RDI, R8-R11, ST(0)-ST(7), XMM0-XM15
callee-save registers	SI, DI, BP, DS	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12-R15, low half of XMM6-XMM15	RBX, RBP, R12-R15
registers for parameter transfer	see table 5	see table 5	RCX, RDX, R8, R9, XMM0-XMM3	RDI, RSI, RDX, RCX, R8, R9, XMM0-XMM7
registers for return	AX, DX, ST(0)	EAX, EDX, ST(0)	RAX, XMM0	RAX, RDX, XMM0, XMM1, ST(0), ST(1)

The rules for register usage depend on the operating system, as shown in table 4. Scratch registers are registers that can be used for temporary storage without restrictions (also called caller-save or volatile registers). Callee-save registers are registers that you have to save before using them and restore after using them (also called non-volatile registers). You can rely on these registers having the same value after a call as before the call. For example, a function using `EBP` may look like this

```
FunctionUsingEBP PROC NEAR
        push    ebp
        mov     ebp, esp
        sub     esp, 48
        ...
        mov     eax, [ebp+8]
        push    eax
        call    AnotherFunction
        mov     esp, ebp
        pop     ebp
        ret
FunctionUsingEBP ENDP
```

Here, `EBP` is saved on the stack in the beginning of the function and restored in the end. The code relies on `EBP` being unchanged after the call to `AnotherFunction`. `EAX` is also used, but doesn't have to be saved.

It is more efficient to use registers for transferring parameters to a function and for receiving the return value than to store these values on the stack. Some calling conventions use certain registers for parameter transfer, but the rules for which registers to use are compiler-specific in 16-bit and 32-bit systems. In 64-bit systems, the use of registers for parameter transfer is standardized. All systems use registers for return values if the returned object fits into the registers that are assigned for this purpose. See the next chapter for details.

Segment registers

You only have to care about segment registers in 16-bit mode. `DS` has to be saved and restored if you change it. `ES` can be changed freely. In DOS programs, `ES` can have any value. In 16-bit Windows, `ES` can only have values that are valid segment descriptors. It is not allowed to use `ES` for other purposes.

In 32-bit and 64-bit mode, it is not allowed to change any segment register, not even temporarily. `CS`, `DS`, `ES` and `SS` all point to the flat segment group. `FS` is used for a thread environment block in Windows and for thread specific data in Linux. `GS` is used for a processor control region in 64-bit Windows. It is unused but reserved in 32-bit Windows. It is probably unused in Linux.

Arithmetic flags

The rules for the arithmetic flags (zero flag, carry flag, etc.) are the same as for scratch registers. These flags need not be saved. Some programming languages (not C++) use the carry flag for Boolean returns.

Direction flag

The rules for the direction flag is the same in all systems. The direction flag is cleared by default. If the direction flag is set, then it must be cleared again before any call or return. Some compilers and subroutine libraries rely on the direction flag always being clear (Microsoft, Watcom, Digital Mars) while other systems use the double-safe strategy of always leaving the direction flag cleared, but not relying on receiving it cleared (Borland, Gnu).

There is a slight possibility that some programmers may have ignored the rule for the direction flag. Therefore, it may be wise to use the double-safe strategy and clear the direction flag before using it if your code will be linked together with modules from unreliable sources.

Interrupt flag

It is not allowed to turn off the interrupt flag in programs running in multi-user systems because that would make it possible to steal unlimited amounts of CPU time from other processes. It may be possible to turn off the interrupt flag in console mode programs running under Windows 98 and earlier operating systems without network. But since programs written for old operating systems are likely to be run under newer operating systems, it is reasonable to say that it is never possible to turn off the interrupt flag in application programs.

Floating point/MMX registers

The floating point registers `ST(0)`-`ST(7)` need not be saved. These registers must be emptied before any call or return, except for registers used for return values. The `MM0`-`MM7` registers, which are part of the `ST(0)`-`ST(7)`, can be used without saving, but these registers must be cleared by `EMMS` before any call or return, and before any code that uses `ST(0)`-`ST(7)`. The 64-bit Microsoft compiler doesn't use `ST(0)`-`ST(7)`.

Floating point control word and MXCSR register

The floating point control word and bit 6-15 of the `MXCSR` register must be saved and restored by any procedure that changes them, except for procedures that have the purpose of changing these.

Deviating from the conventions

It is possible to deviate from the register usage conventions in an isolated section of code as long as all interfaces to other parts of the code conform to the conventions. Any deviation from the conventions must be well documented. Deviations from good programming practice are justified only if a significant gain in speed can be obtained.

Microsoft 16-bit compiler

The 16-bit Microsoft compiler returns `float` and `double` through a static memory location pointed to by `AX`. `long double` is returned in `ST(0)`.

Watcom compiler

The Watcom compiler doesn't conform to the register usage conventions in table 4. The only scratch register is `EAX`. All other general purpose registers are callee-save, except for `EBX`, `ECX`, `EDX` when used for parameter transfer, and `ESI` when used for return pointer. (In 16-bit mode, `ES` is also a scratch register). It is possible to specify any other register usage by the use of pragmas in the Watcom compiler.

How many registers should be callee-save?

I have never seen a study of the optimal ratio of caller-save to callee-save registers. Scratch registers are preferred for temporary values that don't have to be saved across a function call. Functions that don't call other functions (leaf functions) and functions that have a low probability of calling other functions (effective leaf functions) will prefer to use scratch registers. If a function has more than one call to other functions and needs to store values across these function calls, then the function becomes simpler by using callee-save registers. If the called functions need to use the same registers, then there is no advantage in speed, but possibly in size. If the called functions can use other registers, then there is an advantage in speed as well. Since leaf functions are the most likely ones to be speed-critical, it is reasonable to have as many scratch registers as are typically needed in a leaf function. Functions that call other functions, on the other hand, are likely to have more variables and thus need more registers. Balancing these considerations, I would expect the optimal fraction of scratch registers to be between a half and two thirds for architectures that have few registers, and somewhat lower if there are plenty of registers.

Some compilers have capabilities for whole-program-optimization, and we can expect such features to become more common in the future. If the compiler has information about the register needs of both caller and callee at the same time, then it can allocate different registers to the two functions so that no registers need to be saved. In this case, the optimal convention is to define callee-save registers only for system functions, device drivers and library functions.

6.1 Can floating point registers be used in 64-bit Windows?

There is widespread confusion about whether 64-bit Windows allows the use of the floating point registers `st(0)-st(7)` and the `mm0 - mm7` registers that are aliased upon these. One technical document found at Microsoft's website says "x87/MMX registers are unavailable to Native Windows64 applications" (Rich Brunner: Technical Details Of Microsoft® Windows® For The AMD64 Platform, Dec. 2003). An AMD document says: "64-bit Microsoft Windows does not strongly support MMX and 3Dnow! instruction sets in the 64-bit native mode" (Porting and Optimizing Multimedia Codecs for AMD64 architecture on Microsoft® Windows®, July 21, 2004). However, a document in Microsoft's MSDN says: "A caller must also handle the following issues when calling a callee: [...] Legacy Floating-Point Support: The MMX and floating-point stack registers (MM0-MM7/ST0-ST7) are volatile. That is, these legacy floating-point stack registers do not have their state preserved across context switches" (MSDN: Kernel-Mode Driver Architecture: Windows DDK: Other Calling Convention Process Issues. Preliminary, June 14, 2004). It is apparent from the context of the MSDN document that "context switches" here means function calls, not task switches.

The Microsoft C++ compiler never uses these registers in 64-bit mode, and the Intel C++ compiler for 64-bit Windows issues error messages when attempts are made to enable long double support or to use the type `__m64`. Neither does the Intel compiler use these registers in situations where it would be optimal. There are rumors that these registers cannot be used in 64-bit Windows because they are not saved during task switches. These rumors appear to be false, however. My tests indicate that these registers are saved correctly during task switches and thread switches in 64-bit mode. Furthermore, I see no reason to not save these registers. If the floating point registers were not saved during a task switch, then they would have to be cleared for security reasons, and the time required for clearing these registers offsets the time saved by not saving these registers. The floating point registers must be supported when running legacy 32-bit programs.

Considering that the MSDN document is more authoritative (though preliminary) than the other documents; that this document contains the words "Legacy Floating-Point Support"; and that it is in agreement with my tests, my conclusion is that it is probably safe to use floating point and mmx registers in x64 Windows. It is likely that, at an early stage in the development of this operating system, it was planned not to support floating point registers, but that this decision was reversed when programmers found that it was simpler and more safe to save all registers (using `FXSAVE`) than to selectively save the supported registers one by one.

7 Function calling conventions

Table 5. Function calling conventions

segment word size	**calling convention, operating system, compiler**	**parameters in registers**	**parameter order on stack**	**stack cleanup by**	**comments**
16 bit	cdecl		C	caller	
	pascal		Pascal	function	
	fastcall Microsoft (non-member)	ax, dx, bx	Pascal	function	return pointer in bx
	fastcall Microsoft (member function)	ax, dx	Pascal	function	`this` on stack low address. return pointer in ax
	fastcall Borland	ax, dx, bx	Pascal	function	`this` on stack low address. return ptr on stack high addr.
	Watcom	ax, dx, bx, cx	C	function	return pointer in si
32 bit	cdecl		C	caller	
	stdcall		C	function	
	pascal		Pascal	function	
	Gnu		C	hybrid	
	fastcall Microsoft	ecx, edx	C	function	return pointer on stack if not member function
	fastcall Gnu	ecx, edx	C	function	
	fastcall Borland	eax, edx, ecx	Pascal	function	
	thiscall Microsoft	ecx	C	function	default for member functions
	Watcom	eax, edx, ebx, ecx	C	function	return pointer in esi
64 bit	Windows (Microsoft)	rcx/xmm0, rdx/xmm1, r8/xmm2, r9/xmm3	C	caller	Stack aligned by 16. Shadow space on stack for register parameters. The specified registers can only be used for parameter number 1, 2, 3 and 4, respectively.
	Linux, BSD (Gnu)	rdi, rsi, rdx, rcx, r8, r9, xmm0-7	C	caller	Stack aligned by 16. Red zone below stack.

The way of transferring parameters to a function is not always as well standardized as we would wish, as table 5 shows. In many cases it is possible to specify a particular calling convention in a C++ declaration, for example:

```
int __stdcall SomeFunction (float a);
```

In 16 bit and 32 bit mode, we have the same calling conventions in different operating systems, but some differences between different brands of compilers, especially for the `__fastcall` convention and for member functions. In 64 bit mode, the different operating systems use different calling conventions, but I would not expect differences between different compilers because all details are defined in the official ABIs.

The entries in table 5 need some explanations. Segment word size defines the hardware platform. 16 bit refers to DOS and Windows 3.x and earlier. 32 bit refers to Windows 95 and later, Linux and BSD for the 32-bit x86 processors. 64 bit refers to Windows, Linux and BSD for the x86-64 processor architecture (called AMD64 by AMD and EM64T by Intel).

Calling convention is the name of the calling convention. `__cdecl`, `__stdcall`, `__pascal` and `__fastcall` can be specified explicitly in C++ function declarations for compilers that support these conventions. `__cdecl` is the default for applications and static libraries. `__stdcall` is the default for system calls and DLLs in 32-bit Windows. `thiscall` is used by default in Microsoft compilers for member functions in 16 and 32 bit mode. In 64 bit mode, there is only one calling convention for each operating system (at the time of writing). Microsoft, Borland, Watcom and Gnu are brands of compilers. Intel compilers for Windows are compatible with Microsoft. Intel compilers for Linux are compatible with Gnu. Symantec and Digital Mars compilers are compatible with Microsoft.

Parameters in registers specifies which registers are used for transferring parameters. `ecx`, `edx` means that the first parameter goes into `ecx`, the second parameter goes into `edx`, and subsequent parameters are stored on the stack. Parameter types that do not fit into the registers are stored on the stack. In general, all integer types, bool, enum and pointers can be transferred in the general purpose registers. References are treated as identical to pointers in all respects. Arrays are transferred as pointers. Float and double types are transferred in XMM registers in 64 bit mode, otherwise on the stack. Long doubles, structures, classes and unions may be transferred on the stack or through pointers if they don't fit into registers. The rules for deciding whether an object is transferred in registers, on the stack, or through a pointer are explained below. Where no register is specified in table 5, all parameters go on the stack. Return parameters are returned in registers as specified in section 6. Composite objects are returned as specified below.

Parameter order on stack. The Pascal order means that the first parameter has the highest address on the stack and the last parameter has the lowest address, immediately above the return address. If parameters are put on the stack by push instructions then the first parameter is pushed first because the stack grows downwards. The C order is opposite: The first parameter has the lowest address, immediately above the return address, and the last parameter has the highest address. This method was introduced with the C language in order to make it possible to call a function with a variable number of parameters, such as `printf`.

Each parameter must take a whole number of stack entries. If a parameter is smaller than the segment word size then the rest of that stack entry is unused. Likewise, if a parameter is transferred in a register that is too big, then the rest of that register is unused.

If the type of a parameter is not specified explicitly because the function has no prototype or because it has varargs (`...`), then parameters of type `float` are converted to `double`, `char` and `short int` are converted to `int`.

Stack cleanup by. Specifies whether the stack space used by parameters is freed by the caller or by the called function. If n bytes of stack space is used for parameters and the called function has the responsibility for stack cleanup, then this function must return with a `ret n` instruction, otherwise `ret 0`. The 32-bit Gnu compiler uses a hybrid of these two methods: An object return pointer (see below) must be removed from the stack by the called function, all other parameters are removed by the caller.

If stack cleanup is the responsibility of the caller, and if speed is important, then it may be advantageous for the caller to leave the stack pointer where it is after the call and put parameters for a subsequent function call on the stack by `mov` instructions rather than by `push` instructions.

Further rules

Member functions (Applies to all C++ compilers and operating systems). All member functions receive a pointer to the object as an implicit parameter, known as `this` in C++. This pointer comes before the explicit parameters, usually as the first parameter. Constructors must return `this` in the return register.

Returning objects (Applies to all compilers and operating systems). Objects that do not fit into the return registers are returned to a storage space supplied by the caller. The caller must supply a return pointer as an implicit parameter to the called function if this is necessary. The same pointer is returned in the return register. The rules for deciding whether an object is returned in registers or through a return pointer are explained below.

A member function that returns an object can have two implicit parameters, a return pointer and a `this` pointer. In Microsoft compilers and 64 bit Windows, the `this` pointer is the first parameter, the return pointer is the second parameter, and all explicit parameters come thereafter. In Borland and Gnu compilers and in 64 bit Linux and BSD, the return pointer is the first parameter, the `this` pointer is the second parameter, and all explicit parameters come thereafter (this order is compatible with C).

64 bit Windows has more rules. The first parameter goes into `rcx` or `xmm0`; the second parameter goes into `rdx` or `xmm1`, etc. This means that if the first parameter is a float in `xmm0` and the second parameter is an integer, then the latter goes into `rdx`, while `rcx` is unused. The maximum number of parameters that can be transferred in registers is four in total, not four integers plus four floats. The caller must reserve stack space for all parameters that are transferred in registers. This shadow space is the location where the parameter would have been stored if it had been transferred on the stack. This is considered the "home address" of the parameter where the called function may store it in case the register is needed for something else. It is not possible to dispense with this rule because the shadow space is used quite often by functions compiled with the Microsoft compiler. Since the shadow space is owned by the called function, it is safe to use this storage space for any purpose by the called function. The caller does not need to put anything into the shadow space.

If the type of a parameter is not specified explicitly because the function has no prototype or because it has varargs (`...`), and a parameter of type `double` is passed in an XMM register (`float` is converted to `double`), then the corresponding integer register must contain the same value converted to `int`. This does not apply to parameters passed on the stack.

64 bit Linux and BSD. This system has six integer registers and eight XMM registers for parameter transfer. This means that a maximum of 14 parameters can be transferred in registers in 64 bit Linux and BSD, while 64 bit Windows allows only 4. There is no shadow space on the stack. Instead there is a "red zone" below the stack pointer that can be used for temporary storage. The red zone is the space from `[rsp-8]` to `[rsp-128]`. A function can rely on this space being untouched by interrupt and exception handlers. It is therefore safe to use this space for temporary storage as long as you don't do any `push` or `call` instructions. Everything stored in the red zone is destroyed by function calls.

If the type of a parameter is not specified explicitly because the function has no prototype or because it has varargs (`...`), then `rax` must indicate the number of XMM registers used for parameter transfer. Valid values are 0 - 8. A value or `rax` that is higher than the actual number of XMM registers used is allowed as long as it doesn't exceed 8.

`sysenter` calls use `r10` instead of `rcx` for parameter transfer and `rax` for function number.

7.1 Passing and returning objects

Table 6. Methods for passing structure, class and union objects

segment size	**16 bit**			**32 bit**						**64 bit**	
compiler	Microsoft	Borland	Watcom	Microsoft	Borland	Borland	Gnu v.3.x	Gnu v.3.x	Watcom	Windows/MS	Linux, BSD/Gnu
calling convention	all	all		all	default	fastcall	default	fastcall			
max number of integer registers used for transfer	0	0	2	0	0	1	0	1	1	1	2
max number of XMM registers used for transfer	0	0	0	0	0	0	0	0	0	0	2
simple structure, class or union	S	S	I	S	S	I	S	I	I	IZ	R
size not a power of 2	S	S	S	S	S	S	S	S	S	PI	R
contains mixed int and f.p.	S	S	S	S	S	S	S	S	S	IZ	R
contains long double	S	S	S	S	S	S	S	S	S	IZ	S
has member function	S	S	I	S	S	I	S	I	I	IZ	R
has constructor	S	S	I	S	S	S	S	I	I	IZ	R
has copy constructor	S	S	PI	S	S	S	PS	PI	PI	PI	PI
has destructor	S	S	PI	S	S	I	PS	PI	PI	IZ	PI
has virtual	S	S	PI	S	S	S	PS	PI	PI	PI	PI
has inheritance	S	S	I	S	S	I	S	I	I	IZ	R
has no data	S	S	I	S	S	I	S	I	I	IZ	S(R)

Symbols:

S:	Copy of entire object transferred on stack.
PI:	Temporary copy referenced by pointer in register. If no vacant register, use PS.
PS:	Temporary copy referenced by pointer on stack
I:	Entire object transferred in integer registers. Use S if not enough vacant registers.
IZ:	Entire object transferred in integer registers, zero-extended to register size. Use PS if no vacant register.
R:	Entire object is transferred in integer registers and/or XMM registers if the size is no bigger than 128 bits, otherwise on the stack. Each 64-bit part of the object is transferred in an XMM register if it contains only float or double, or in an integer register if it contains integer types or mixed integer and float. Two consecutive floats can be packed into the lower half of one XMM register. Consecutive doubles are not packed. No more than 64 bits of each XMM register is used. Use S if not enough vacant registers for the entire object.

There are several different methods to transfer a parameter to a function if the parameter is a structure, class or union object. A copy of the object is always made, and this copy is transferred to the called function either in registers, on the stack, or by a pointer, as specified in table 6. The symbols in the table specify which method to use. S takes preference over I and R. PI and PS take preference over all other passing methods.

As table 6 tells, an object cannot be transferred in registers if it is too big or too complex. For example, an object that has a copy constructor cannot be transferred in registers because the copy constructor needs an address of the object. The copy constructor is called by the caller, not the callee.

Objects passed on the stack are aligned by the stack word size, even if higher alignment would be desired. Objects passed by pointers are not aligned by any of the compilers studied, even if alignment is explicitly requested. The 64bit Windows ABI requires that objects passed by pointers be aligned by 16.

An array is not treated as an object but as a pointer, and no copy of the array is made, except if the array is wrapped into a structure, class or union.

The 64 bit compilers for Linux differ from the ABI (version 0.92) in the following respects: Objects with inheritance, member functions, or constructors can be passed in registers. Objects with copy constructor, destructor or virtual are passed by pointers rather than on the stack.

The Intel compilers for Windows are compatible with Microsoft. Intel compilers for Linux are compatible with Gnu with very few exceptions.

Table 7. Methods for returning structure, class and union objects

segment size	**16 bit**			**32 bit**				**64 bit**	
compiler	Microsoft	Borland	Watcom	Microsoft	Borland	Watcom	Gnu	Windows/MS	Linux, BSD/Gnu
max number of integer registers used for return	0	2	2	2	1	1	2	1	2
max number of XMM registers used for return	0	0	0	0	0	0	0	0	2
max number of f.p. registers used for return	0	0	0	0	0	0	0(1)	0	1
simple structure, class or union	P	I	I	I	I	I	I	I	R
bigger than max registers	P	PF	PSI	PS	P	PSI	P	P	P
size not a power of 2	P	PF	PSI	PS	P	PSI	P	P	R,X
contains only f.p.	P	I	I	I	I	I	P(F1)	I	X
contains mixed int and f.p.	P	PF	PSI	I	P	PSI	I	I	R
contains mixed float and double	P	PF	PSI	PS	P	PSI	P	P	X
contains only long double	P	PF	PSI	I	P	PSI	P(F1)	I	F
contains mixed long double and other	P	PF	PSI	PS	P	PSI	P	P	P
has member function	P	I	I	I	I	I	P(I)	I	R
has inheritance	P	I	I	P	I	I	P(I)	P	R
has constructor	P	PF	I	P	P	I	P(I)	P	R
has copy constructor	P	PF	PAX	P	P	PAX	P	P	P
has destructor	P	PF	PAX	P	P	PAX	P	P	P
has virtual	P	PF	PAX	P	P	PAX	P	P	P
has no data	P	I	I	0	I	I	P(0)	0	I

Symbols:

I	Returned in integer registers
X	returned in XMM registers
F	returned in ST(0) register
F1	If one float, double or long double, use ST(0), otherwise I.
R	Entire object is returned in integer registers and/or XMM registers if the size is no bigger than 128 bits, otherwise on the stack. Each 64-bit part of the object is transferred in an XMM register if it contains only float or double, or in an integer register if it contains integer types or mixed integer and float. Two consecutive floats can be packed into the lower half of one XMM register. Consecutive doubles are not packed. No more than 64 bits of each XMM register is used. Use P if not enough vacant registers.
P	Pointer to temporary memory space passed to function. Pointer may be passed in register if fastcall or 64-bit mode, otherwise on stack. Same pointer is returned in `AX`, `EAX` or `RAX`.
PS	Pointer to temporary memory space passed to function. Pointer is passed on stack, even if fastcall. Same pointer is returned in `EAX`.
PF	Far pointer to temporary memory space passed on stack and returned in `DX:AX`.
PAX	Pointer to temporary memory space passed to function in `AX`/`EAX` and returned unchanged in `AX`/`EAX`.
PSI	Pointer to temporary memory space passed to function in `SI`/`ESI` and returned unchanged in `AX`/`EAX`.
0	Nothing passed or returned.

A structure, class or union object can be returned from a function in registers only if it is sufficiently small and not too complex. If the object is too complex or doesn't fit into the appropriate registers then the caller must supply storage space for the object and pass a pointer to this space as a parameter to the function. This pointer can be passed in a register or on the stack. The same pointer is returned by the function. The detailed rules are given in table 7.

P and PF take precedence over all other. PS takes precedence over all but P. PAX takes precedence over PSI, which takes precedence over I.

The storage space pointed to by return pointers is aligned by 16 in the 64 bit Gnu compiler and the 64 bit MS compiler. The 32 bit Microsoft compiler can align this space if explicitly requested.

The Gnu compiler version 2.x and some implementations of version 3.x differ, as indicated by the parentheses.

The Intel compilers for Windows are compatible with Microsoft. Intel compilers for Linux are compatible with Gnu with very few exceptions.

The 64 bit Gnu compiler differs from the ABI (version 0.92) by using only one floating point register for return.

7.2 Passing and returning SIMD types

Table 8. Methods for passing and returning SIMD data types

segment size	**32 bit**		**64 bit**	
operating system	**Windows**	**Linux**	**Windows**	**Linux**
`__m64` parameters transferred in registers	3	3	not supported	0
alignment of `__m64` parameters on stack	4	4		8
`__m64` returned in register	mm0	mm0		mm0

`__m128` parameters transferred in registers	3	3	transferred by pointer	8
alignment of `__m128` parameters on stack	16	16	16	16
`__m128` returned in register	xmm0	xmm0	xmm0	xmm0

The types `__m64` and `__m128` define the SIMD data types that fit into the 64-bit mmx registers and the 128-bit xmm registers, respectively. The Intel compiler supports these types as intrinsic types, and the Microsoft compiler has partial support for these types. There are special rules for passing and returning function parameters of the types `__m64` and `__m128`, as shown in table 8. Some compilers have options for either treating these types as intrinsic types, using the passing methods defined in table 8, or treating them as structures according to the rules in table 6 and table 7. The types `__m128i` and `__m128d` may or may not be treated as `__m128`.

Under 32-bit Windows and 32-bit Linux, the first three parameters of type `__m64` are transferred in registers mm0 - mm2. Any additional parameters of type `__m64` are transferred on the stack aligned by 4. Under 64-bit Linux, `__m64` parameters are passed on the stack. `__m64` is not supported in 64-bit Windows (see p. 10). Return values of type `__m64` are transferred in mm0 on all platforms where this type is supported.

Under 32-bit Windows and 32-bit Linux, the first three parameters of type `__m128` are transferred in registers xmm0 - xmm2. Any additional parameters of type `__m128` are transferred on the stack aligned by 16. This alignment is accomplished as follows: If there are more than three parameters of type `__m128` then the stack must be aligned by 16 before the call instruction. Consequently, the value of the stack pointer is 12 modulo 16 at the entry of the called function. The parameter space on the stack is padded, if necessary, to align the parameters of type `__m128` by 16. In 64-bit Windows, parameters of type `__m128` are passed by a pointer to an aligned copy. In 64-bit Linux, the first eight parameters of type `__m128` are transferred in registers xmm0 - xmm7. Any additional parameters of type `__m128` are transferred on the stack aligned by 16. If there are more than eight parameters of type `__m128` then the stack pointer must be aligned by 16 before the call instruction, and the stack space is padded, if necessary, to align the parameters. Return values of type `__m128` are transferred in xmm0 on all platforms.

Intel's application note AP 589 "Software Conventions for Streaming SIMD Extensions" specifies that the caller must provide shadow space on the stack where the first three `__m128` parameters would be stored if they were not transferred in registers. Since the Intel compiler does not do so, this document can be considered invalid.

8 Name mangling

Name mangling (also called name decoration) is a method used by C++ compilers to add additional information to the names of functions and objects in object files. This information is used by linkers when a function or object defined in one module is referenced from another module. Name mangling serves the following purposes:

1. make it possible for linkers to distinguish between different versions of overloaded functions

2. make it possible for linkers to check that objects and functions are declared in exactly the same way in all modules

3. make it possible for linkers to give complete information about the type of unresolved references in error messages

Name mangling was invented to fulfill purpose 1. The other purposes are secondary benefits not fully supported by all compilers.

The minimum information that must be supplied for a function is the name of the function and the types of all its parameters. Possible additional information includes the return type, calling convention, etc. All this information is coded into a single ASCII text string which looks cryptic to the human observer. The linker doesn't have to know what this code means in order to fulfill purpose 1 and 2. It only needs to check if strings are identical.

Each brand of C++ compiler uses its own name mangling scheme. Hitherto, there has been no need to standardize name mangling because the object files produced by different compilers were incompatible anyway for other reasons. However, since data representation, calling conventions, and other details are now being standardized to an increasing degree in the official ABI (Application Binary Interface) standards of new operating systems, there is no reason not to standardize name mangling schemes as well.

Unfortunately, few compiler vendors have cared to publish their name mangling schemes. This is the reason why I have studied the name mangling schemes of several different C++ compilers and published the detailed results here.

Compiler vendors typically use the same name mangling scheme on different hardware platforms. Though the information given here has been gathered by investigating compilers for the 16, 32 and 64 bit x86 platforms, it is likely to apply to other platforms as well, and possibly even other programming languages. Not all mangling schemes are covered in this report. There are other schemes used on other platforms, often resembling the schemes described here as Gnu1 and Gnu2.

The codes used for parameter types, calling conventions, etc. are given in the tables below. The complete syntax for each compiler is given in the following sections. The syntax is written in extended Bacchus Naur notation. For example

`<a> ::= <b>  |  [<c>]` $[< d >]_x^y$

means that syntax element `<a>` must consist of either syntax element `<b>` or zero or one instance of `<c>` followed by at least `x` and at most `y` instances of `<d>`. Spaces may be included in the syntax specification for the sake of readability, but the coded string cannot contain spaces.

Table 9. Type codes

type	Microsoft	Borland	Watcom	Gnu2	Gnu3
void	X	v	v	v	v
bool	_N	4bool	q	b	b
char	D	c	a	c	c
signed char	C	zc	c	Sc	a
unsigned char	E	uc	uc	Uc	h
short int	F	s	s	s	s
unsigned short int	G	us	us	Us	t
int	H	i	i	i	i
unsigned int	I	ui	ui	Ui	j
long int	J	l	l	l	l
unsigned long int	K	ul	ul	Ul	m
__int64	_J		z	x	x[1]
unsigned __int64	_K		uz	Ux	y[1]
wchar_t	G	b	w	w	w
float	M	f	b	f	f
double	N	d	d	d	d
long double	O, _T, _Z [2]	g	t	r	e
__complex__ float				Jf	Cf
__complex__ double				Jd	Cd
__m64	T__m64@@				y

__m128	U__m128@@				y
varargs ...	Z	e	e	e	z
const X	X	xX	xX	X	X
X *	PAX[3]	pX	pnX	PX	PX
X const *	PBX[3]	pxX[4]	pnxX[4]	PCX[4]	PKX[4]
X &	AAX[3]	rX	rnX	RX	RX
X const &	ABX[3]	rxX[4]	rnxX[4]	RCX[4]	RKX[4]
X[ ] (as global object)	PAX[3, 5]		[]X	PX	
X[][8] (as global object)	PAY07X[3,5,6]		[][8]X	PA7_X[7]	
X[][16][5] (as glob obj)	PAY1BA@4X[3,5,6]		[][16][5]X	PA15_A4_X[3]	
X[ ] (as function parameter)	QAX[3]	pX	pnX	PX	PX
const X[] (as function param.)	QBX[3]	xpX	pnxX[4]	PCX[4]	
X[][8] (as func-tion parameter)	QAY07X[3,6]	pa8$X	pn[8]X	PA7_X[3]	PA8_X
X[][16][5] (as function param.)	QAY1BA@4X[3,6]	pa16$a5$X	pn[16][5]X	PA15_A4_X[7]	PA16_A5_X
X near *	PAX[3]	pX	pnX		
X far *	PEX[3]	nX	pfX		
X huge *	PIX[3]	upX	phX		
X _seg *		urX			
X near &	AAX[3]	rX	rnX		
X far &	AEX[3]	mX	rfX		
X huge &	AIX[3]	umX	rhX		
union X	TX@@	<LX>[8]X	$X$$	G[9]<LX>[8]X	<LX>[8]X
struct X	UX@@	<LX>[8]X	$X$$	G[9]<LX>[8]X	<LX>[8]X
class X	VX@@	<LX>[8]X	$X$$	G[9]<LX>[8]X	<LX>[8]X
enum X	W4X@@	<LX>[8]X	$X$$	<LX>[8]X	<LX>[8]X
enum Y::X	W4X@Y@@	<LX+LY+1>[8]Y@X	$X$:Y$$	Q2<LY>Y<LX>[8]X	N<LY>Y<LX>[8]X
X (*Y)(W)[10]	P6AXW@Z[11]	pqW$X	pn(W)X[12]	PFW_X	PFXWE
X Y::*V[13]	PQY@@X[14]	M<LY>[8]YX	m$Y$$nX[12]	PO<LY>[8]Y_X	M<LY>[8]YX
X (Y::*V)(W)[15]	P8Y@@AEXW@Z[16]	M<LY>[8]YqW$X	m$Y$$n(W)X[12]	PM<LY>YFP<LY>YW_X	M<LY>[8]YFXWE

notes:

[1] In 64-bit Linux/BSD, use the code for long int or unsigned long int.

[2] The implementation of long double in Microsoft compilers has the same precision as double (64 bits) and uses the code O (capital letter O). A different symbol is used when 80 bits precision is supported. The Intel compiler uses the code _T when 80 bits is used and O when 64 bits is used. The Symantec/Digital Mars compiler uses the code _Z for 80 bits.

[3]. The second symbol (A, B etc.) is the storage class of the target, according to table 10.

[4]. The letter before X is the storage class of the target, according to table 10.

[5]. Replace P by _O in 64 bit mode for Microsoft compiler. See page 22 for a comment.

[6]. After QAY follows: the number of dimensions minus 1, then each dimension except the first one. These numbers are all coded in the way described in table 17 page 25.

[7]. Each dimension N, except the first one, is coded as the decimal number N-1.

[8]. <LX> = length of name X, <LY> = length of name Y, as decimal numbers.

[9]. The G prefix is only used when a union, struct or class appears as a function parameter. It is not used with pointers or references to these or when the type appears as a template argument. The G comes before Q2 if the name has a qualifying namespace.

[10]. Y is a pointer to a function with argument type W and return type X. In the code columns, X and W represent the codes for types X and W.

[11]. Replace 6 with 7 if far. A represents the calling convention, using table 15. It may be followed by a return type modifier code from table 12.

[12]. Insert any return type modifier or target modifier (table 12) before the return type, and any member function access code (table 14) before (.

[13]. V is a pointer to a data member of class Y of type X. In the code columns, X and Y represent the codes for types X and Y, <LY> represents the length of the name Y.

[14]. Q qualifies the target. Replace with R if const, S if volatile, T if const volatile.

[15]. `V` is a pointer to a function member of class `Y` with argument type `W` and return type `X`. In the code columns, `X` ,`Y` and `W` represent the codes for types `X`, `Y` and `W`, `<LY>` represents the length of the name `Y`.
[16]. Replace `8` with `9` if far. `A` represents a member function access code, using table 14. It is omitted in 16-bit mode. `E` represents the calling convention, using table 15. It may be followed by a return type modifier code from table 12.

Table 10. Storage class codes

storage class	Microsoft	Borland	Watcom	Gnu2	Gnu3
(default)	A		n		
near	A		n		
const	B	x	nx	C	K
volatile	C	w	ny	V	V
const volatile	D	xw	nyx	CV	VK
far	E		f		
const far	F		fx		
volatile far	G		fy		
const volatile far	H		fyx		
huge	I		h		

Example: `const int a;`

Table 11. Function distance codes

calling distance	Microsoft	Borland	Watcom	Gnu2	Gnu3
near	Y or Q		n		
far	Z or R		f		

Example: `void far Function1 (int x);`

Table 12. Storage class codes for return

storage or call type	Microsoft	Borland	Watcom	Gnu2	Gnu3
default	?A				
const	?B		x		
volatile	?C		y		
const volatile	?D		yx		

Example: `const int Function2 (int x);`

Table 13. Member function modifier codes

storage or call type	Microsoft	Borland	Watcom	Gnu2	Gnu3
default	Q (far: R)				
static	S (far: T)				
virtual	U (far: V)				

Example: `virtual int Class1::MemberFunction3 (int x);`

Table 14. Member function access codes

storage or call type	Microsoft	Borland	Watcom	Gnu2	Gnu3
default	A				
const	B	x	.x		
volatile	C	w	.y		
const volatile	D	xw	.yx		

Example: `int Class1::MemberFunction4 (int x) const;`

Table 15. Function calling convention codes

calling convention	Microsoft	Borland	Watcom	Gnu2	Gnu3
__cdecl	A[17]				
__pascal	C	(uppercase)			
__fortran	C	qf			
__thiscall	E				
__stdcall	G	qs		@<size>	
__fastcall	I[17]	qr			
interrupt	A	qi			

Example: `int __stdcall Function5 (int x);`

notes:

[17]. In 64-bit mode, the only possible calling convention is `fastcall`, which is coded as `A`.

Table 16. Operator name codes

operator	Microsoft	Borland	Watcom	Gnu2	Gnu3
constructor X	?0	$bctr	$ct	_	C1,C2
destructor ~X	?1	$bdtr	$dt	_$	D1
operator []	?A	$bsubs	$od	__vc_	ix
operator ()	?R	$bcall	$op	__cl_	cl
operator ->	?C	$barow	$oe	__rf_	pt
operator X++	?E	$binc	$og	__pp_	pp
operator X--	?F	$bdec	$oh	__mm_	mm
operator new	?2	$bnew	$nw	__nw_ [18]	nw
operator new[]	?_U	$bnwa	$na	__vn_ [18]	na
operator delete	?3	$bdele	$dl	__dl_ [18]	dl
operator delete[]	?_V	$bdla	$da	__vd_ [18]	da
operator *X	?D	$bind	$of	__ml_	de
operator &X	?I	$badr	$ok	__ad_	ad
operator +X	?H	$badd	$oj	__pl_	ps
operator -X	?G	$bsub	$oi	__mi_	ng
operator !	?7	$bnot	$oc	__nt_	nt
operator ~	?S	$bcmp	$oq	__co_	co
operator ->*	?J	$barwm	$ol	__rm_	pm
operator X * Y	?D	$bmul	$of	__ml_	ml
operator /	?K	$bdiv	$om	__dv_	dv
operator %	?L	$bmod	$on	__md_	rm
operator X + Y	?H	$badd	$oj	__pl_	pl
operator X - Y	?G	$bsub	$oi	__mi_	mi
operator <<	?6	$blsh	$ob	__ls_	ls
operator >>	?5	$brsh	$oa	__rs_	rs
operator <	?M	$blss	$rc	__lt_	lt
operator >	?O	$bgtr	$re	__gt_	gt
operator <=	?N	$bleq	$rd	__le_	le
operator >=	?P	$bgeq	$rf	__ge_	ge
operator ==	?8	$beql	$ra	__eq_	eq
operator !=	?9	$bneq	$rb	__ne_	ne
operator X & Y	?I	$band	$ok	__ad_	an
operator |	?U	$bor	$os	__or_	or
operator ^	?T	$bxor	$or	__er_	eo
operator &&	?V	$bland	$ot	__aa_	aa
operator ||	?W	$blor	$ou	__oo_	oo
operator =	?4	$basg	$aa	__as_	aS
operator *=	?X	$brmul	$ab	__aml_	mL
operator /=	?_0	$brdiv	$ae	__adv_	dV
operator %=	?_1	$brmod	$af	__amd_	rM
operator +=	?Y	$brplu	$ac	__apl_	pL
operator -=	?Z	$brmin	$ad	__ami_	mI
operator <<=	?_3	$brlsh	$ah	__als_	lS
operator >>=	?_2	$brrsh	$ag	__ars_	rS
operator &=	?_4	$brand	$ai	__aad_	aN
operator |=	?_5	$bror	$aj	__aor_	oR
operator ^=	?_6	$brxor	$ak	__aer_	eO
operator ,	?Q	$bcoma	$oo	__cm_	cm
operator TYPE()	?B	$o<L>TYPE[19]	$cv	__op<L>TYPE_ [19]	cv<L>TYPE[19]
virtual table	?_7				

Example: `Class1 & operator + (Class1 & a, Class1 & b);`

notes:

[18]. If the operator is not a class member and there are no extra parameters, i.e. if the built-in operator is replaced, then the full mangled name is replaced by `___builtin_new`, `___builtin_vec_new`, `___builtin_delete`, or `___builtin_vec_delete`.

[19]. `<L>` is the length of the name of `TYPE`.

8.1 Microsoft name mangling

Microsoft compilers use a name mangling syntax that includes all information needed to check that an object or function is declared in exactly the same way in all modules (except for array sizes). It is also designed to be as short as possible, while allowing case-insensitive linking. The code is unambiguous so that the complete C++ declaration of an object or function can be recovered from a mangled name.

The public mangled name of a global object is composed according to the following syntax:

`<public name> ::= ? <name> @` $[<\text{namespace}>@]_0^\infty$ `@ 3 <type> <storage class>`

The mangled name of a static class member object is:

`<public name> ::= ? <name> @` $[<\text{class name}>@]_1^\infty$ `@ 2 <type> <storage class>`

`<name>` is the case sensitive C++ name of the object.

`<namespace>` is any namespace surrounding the object.

`<class name>` is the class the object belongs to or a namespace. Class names and namespaces are treated as equivalent. In case of nested classes or namespaces, the innermost class or namespace comes first.

`<type>` is the code for object type, taken from table 9.

`<storage class>` is any storage modifier, taken from table 10. The default is A. This code is replaced by `Q1@` for member pointers and member function pointers, regardless of storage class.

Examples:
```
int alpha;
```
is coded as
```
?alpha@@3HA
```

```
char beta[6] = "Hello";
```
is coded as
```
?beta@@3PADA
```

```
double Class1::gamma[10][5];
```
is coded as
```
?gamma@Class1@@2PAY04NA
```

Note that global arrays are coded as pointers (`P`) while arrays as function parameters are coded as arrays (`Q`). This should have been opposite, since arrays as function parameters are equivalent to pointers, while global arrays are not. Apparently, this illogical coding has been retained in all 16 and 32 bit compilers for the sake of compatibility with legacy code. In 64 bit mode, the code for global arrays has been changed to _O in order to prevent an array in one module to be confused with a pointer with the same name in another module.

The mangled name of a global function is composed according to the following syntax:

`<public name> ::= ? <function name> @` $[<\text{namespace}>@]_0^\infty$ `@ <near far>`
`<calling conv> [<stor ret>] <return type>` $[<\text{parameter type}>]_1^\infty$ `<term> Z`

`<near far>` is `Y` for near, `Z` for far. Far calls are only possible in 16-bit mode.

`<calling conv>` is the calling convention, taken from table 15. The default is `A`.

`<stor ret>` defines the storage class of the return, using the codes in table 12. It is omitted for simple types if the storage class is not const or volatile. It is always included if the return type is a structure, class or union.

`<return type>` is the type returned by the function, taken from table 9.

`<parameter type>` is the type of each function parameter, taken from table 9.

`<term>` is `@` except if the parameter list is `void` (`X`) or ends with `...` (`Z`). In these cases, the `@` is omitted because the list is sure to end here.

Example:
`void Function1 (int a, int * b);`
is coded as
`?Function1@@YAXHPAH@Z`

The mangled name of a class member function is composed according to the following syntax:

`<public name> ::= ? <function name> @` $[<\text{class name}>@]_1^\infty$ `@ <statvirt nf>`

`<const vol> <calling conv> [<stor ret>] <return type>` $[<\text{parameter type}>]_1^\infty$ `<term> Z`

`<statvirt nf>` defines the near/far distance of the call as well as member function modifiers in table 13. The code is `Q` for near, `R` for far, `S` for static, `T` for far static, `U` for virtual, `V` for far virtual. Far calls are only possible in 16-bit mode.

`<const vol>` is a member function access code from table 14. The default is `A`.

The default calling convention for member functions in 16 bit mode is `C` (`__pascal`), in 32 bit mode it is `E` (`thiscall`). In 64 bit mode the only possible calling convention is `A`.

Example:
`int Class1::MemberFunction(int a, int * b);`
is coded in 32-bit mode as
`?MemberFunction@Class1@@QAEHHPAH@Z`

Constructors, destructors, operators and member operators are coded in the same way as functions, by replacing `<function name>@` with the operator name taken from table 16. The return type of constructors and destructors is replaced with `@`.

Virtual tables are coded as `??_7`$[<\text{class name}>@]_1^\infty$ `@6B@`

Template functions and template classes are coded by replacing `<function name>` or `<class name>` by `?$ <name> @` $[<\text{template parameter}>]_1^\infty$
where `<name>` is the name of the templated function or class. If the template parameter is a typename or class then `<template parameter>` is a type as defined in table 9. If the template parameter is a constant, then
`<template parameter>  ::=  $0 <integer>`
where `<integer>` is coded as explained in table 17 below.

Abbreviations for repeated names and parameter types

The name mangling scheme includes two means of shortening mangled names that would contain the same name or type more than once. The first method involves repeated types, the second method involves repeated names.

Abbreviation of repeated types. This method applies to type declarations in a function parameter list or function pointer parameter list. Only types that need more than one character for its code are included in this scheme. This includes pointers, references, arrays, `bool`, `__int64`, `struct`, `class`, `union`, and `enum` parameters. The first such parameter in a parameter list is assigned the number `0`, the second such parameter is assigned the number `1`, and so forth. Simple types that are encoded with a single letter are not assigned a number. Any repeated instance of a type with an assigned number in the parameter list is replaced by the number of the first instance. The maximum number is `9`. If the number would exceed `9` then the repeated instance must use the full declaration. The return type is not included in the type abbreviation scheme.

Example:

```
bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);
```

is coded as

```
?ExampleFunction@@YA_NPAHHH0_N1PA_N@Z
```

Here, the return type bool is coded as `_N`. `int*a` is coded as `PAH`, which is assigned the number `0`. `int b` is coded as `H`, and is not assigned a number because it is coded as a single letter. `int c` is also coded as `H` because single letter codes are not abbreviated. `int*d` has the same type as `a`, and is abbreviated to the number `0`. `bool e` is coded as `_N` and is assigned the number `1`. The previous instance of `_N` was a return type, so it cannot be copied. `bool f` has the same type as `e` and is replaced by the number `1`. The `bool` in `bool*g` is not abbreviated because sub-expressions cannot be abbreviated.

If the parameter list contains a function pointer, then the parameter types inside the function pointer type declaration are included in the abbreviation scheme, both as sources that can be assigned numbers and as targets that can be abbreviated. The return type in the function pointer type declaration is not included. If the return type of a function declaration is a function pointer, then the parameters, but not the return type, of this function pointer declaration are included in the type abbreviation scheme of the whole function declaration.

Example:

```
typedef int * (* FunctionPointer) (int * a, int * b);
FunctionPointer WeirdFunction(FunctionPointer x,FunctionPointer y,int*z);
```

is coded as

```
?WeirdFunction@@YAP6APAHPAH0@ZP6APAH00@Z10@Z
```

Here, the code for `int *` is `PAH`, and the code for `FunctionPointer` without abbreviation would be `P6APAHPAHPAH@Z`. The first occurrence of `FunctionPointer` is in the return type of `WeirdFunction`. Within this occurrence, the first occurrence of `PAH` is the return type which is excluded from the abbreviation scheme. The second occurrence of `PAH`, representing `int*a`, is assigned the number `0`. The third occurrence of `PAH`, representing `int*b`, is replaced by `0`. The second occurrence of `FunctionPointer` represents parameter `x`. Within this, the first occurrence of `PAH` is not abbreviated because it represents the return type of `FunctionPointer`. The next two occurrences of `PAH`, representing `a` and `b` in `x`, are both replaced by the `0` that has already been assigned. The entire sequence representing parameter `x` is thus `P6APAH00@Z`. This sequence is assigned the number `1`. `FunctionPointer y` is simply reduced to `1`, and `int*z` is reduced to `0`.

Abbreviation of repeated names. This method applies to any name that appears inside a declaration, such as structures, classes, unions, enums, and namespaces. If any such name occurs more than once in a mangled name, then all but the first occurrence will be replaced by a number, no matter how short the name is. The number will represent a copy of the name, but not its context or meaning. A name can be copied even if the different occurrences of the name have different meanings (because of namespace or class scope qualifications). The algorithm is as follows: First eliminate any repeated types using the first abbreviation method. Any names that have been eliminated by the type abbreviation method need no further consideration. Then assign numbers to the first occurrence of each name. The first name, which is usually the function's name (except for constructors, destructors, operators and template functions), is assigned the number `0`, the second name is `1`, and so forth. Each repeated name is then eliminated by replacing `<name>@` with the number. If the number would be higher than `9` then the name cannot be eliminated.

Example:
```
Class1 * SomeFunction (Class1 * a, Class2 * b, Class2 * c, Class2 & d);
```
is coded as
```
?SomeFunction@@YAPAVClass1@@PAV1@PAVClass2@@1AAV2@@Z
```
Here, parameters `b` and `c` have the same type, so `Class2 * c` is reduced by the first method, and simply becomes a `1`. The last parameter `Class2 & d` cannot be reduced by the type abbreviation method because `b` and `d` have different types. Neither can the double occurrence of the name `Class1`, because the type abbreviation method doesn't apply to return types. The name abbreviation method now assigns the numbers `SomeFunction` = `0`, `Class1` = `1`, `Class2` = `2`. Now parameter `a` can be changed from `PAVClass1@@` to `PAV1@`, and parameter `d` is changed from `AAVClass2@@` to `AAV2@`.

Templated names and template parameters are isolated from the numbering of names. This means that a name inside a template argument can only be eliminated if there are multiple occurrences of this name within the same templated name. Likewise, templated names are isolated from each other, even if they are identical. In case of nested templates, each sub-template has its own isolated number sequence.

Example:
```
void Class1::MyTemplateFunction<Class1> (Class1*);
```
will be coded as
```
??$MyTemplateFunction@VClass1@@@Class1@@QAEXPAV0@@Z
```
Here, the templated name `MyTemplateFunction<Class1>` is coded as `?$MyTemplateFunction@VClass1@@`. This template has its own number sequence (`MyTemplateFunction` = `0`, `VClass1` = `1`), which is isolated from the rest. The first name in the rest of the code is the representation of the scope `Class1::` coded as `Class1@`. This occurrence of `Class1` gets the number `0`. The parameter `Class1*`, which was first coded as `PAVClass1@@` is now changed to `PAV0@`, where the `0` refers to the name in `Class1::`, not the name in `<Class1>`.

Coding of numbers

Numbers within mangled names are needed for array dimensions, array sizes, and template parameters. These numbers are coded according to the algorithm in table 17. It appears that this algorithm was designed to make the coding as short as possible, rather than making it human readable.

Table 17. Microsoft number encoding

range for N	coding
1 ≤ N ≤ 10	(N - 1) as a decimal number
N > 10	code N as a hexadecimal number without leading zeroes, replace the hexadecimal digits 0 - F by the letters A - P, end with a @
N = 0	A@
N < 0	? followed by (- N) coded as above

8.2 Borland name mangling

The name of a global object without class or namespace qualifiers is not mangled, except for un underscore prefix:

```
<public name> ::= _ <name>
```

A global object with class or namespace qualifiers is coded as

`<public name> ::=` $[@ < \text{class name} >]_1^{\infty}$ `@ <name>`

where `<class name>` is a class or namespace. In case of nested classes or namespaces, the outermost comes first.

Functions, member functions, constructors, destructors and operators are all coded according to the following syntax:

`<public name> ::= [<template prefix>]` $[@ < \text{class name} >]_0^{\infty}$ `@ <name> $ [<const vol>]` `q [<calling convention>]` $[< \text{parameter type} >]_1^{\infty}$

`<template prefix>` is `@` if the function is a template function or member of a template class, otherwise nothing.

`<const vol>` is a member function access code from table 14. It is omitted by default.

`<calling convention>` defines the calling convention as given in table 15. It is omitted by default. There is no distinction between near and far calling.

`<parameter type>` defines each function parameter, using the codes in table 9. The return parameter is not coded.

For constructors, destructors and operators, replace `<name>` by an operator name from table 16.

Template functions have no special encoding other than the `@` prefix, as the template parameters are implied by the function parameter types. The Borland compilers I have tested only support such cases of template functions where the template parameters can be inferred from the function parameters.

Template class member functions and member objects are coded by replacing `<class name>` by

`% <name>` $[\$t < \text{type code} > \mid \$ii\$ < \text{value} >]_1^{\infty}$ `%`

Global objects of a template class are not mangled if in the global namespace.

Virtual tables are encoded as
`@@<class name>@3`

Type codes that appear more than once in the `<parameter type>` list of a function are abbreviated if the type is a pointer, reference, array, structure, class, union, enum or bool, but not if it is a simple type with one or two-letter code. All parameters are assigned a number, beginning with 1. The code `t1` repeats the first parameter, `t2` repeats the second parameter, `ta` repeats the 10'th parameter, `tz` repeats parameter number 35. Further parameters cannot be copied. If the parameter list contains a function pointer, then the list of parameters for the target function has its own isolated number sequence, so that type codes within the parameter list of the target function can be abbreviated, but not the return type of the target function. There is no method for abbreviating repeated names that are not part of identical parameter types.

Examples:
```
char beta[6] = "Hello";
```
is coded as
```
_beta
```

`double Class1::gamma[10][5];`
is coded as
`@Class1@gamma`

`bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);`
is coded as
`@ExampleFunction$qpiiit14boolt5p4bool`

`void TemplateClass<float>::MemberFunction (TemplateClass<float>*);`
is coded as
`@@%TemplateClass$tf%@MemberFunction$qp18%TemplateClass$tf%`

8.3 Watcom name mangling

The public mangled name of a global object is composed according to the following syntax:

`<public name> ::= W? <name> $` $[:<\text{namespace}>\$]_0^\infty$ `<storage class> <type>`

where `<storage class>` is taken from table 10 and `<type>` is taken from table 9. `<namespace>` can be any namespace or class qualifier, the innermost first.

Examples:
`int alpha;`
is coded as
`W?alpha$ni`

`char beta[6] = "Hello";`
is coded as
`W?Beta$npna`

`double Class1::gamma[10][5];`
is coded as
`W?gamma$:Class1$n[][5]d`

Functions, member functions, constructors, destructors and operators are coded as follows:

`<public name> ::= W? <function name> $` $[:<\text{class name}>\$]_1^\infty$ `<near far> [<const vol>]`
`(` $[<\text{parameter type}>]_0^\infty$ `) [<stor ret>] <return type>`

`<class name>` is a class name or namespace. In case of nested classes or namespaces, the innermost comes first.

`<near far>` is `n` for near or `f` for far. Far calls are only possible in 16 bit mode.

`<const vol>` is a member access code from table 14. It is omitted by default.

`<parameter type>` defines the type of each function parameter according to table 9. No `<parameter type>` is included if the parameter list is `(void)`.

`<stor ret>` is a return type storage class from table 12. It is omitted by default.

`<return type>` defines the return type according to table 9.

Example:
`bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);`
is coded as
`W?ExampleFunction$n(pniiipniqqpnq)q`

Constructors, destructors and operators are coded by replacing `<function name>$` with a name from table 16. The return type of constructors and destructors is replaced by _.

Names that appear more than once in a mangled code are reduced by replacing all but the first occurrence of a name by a reference to the first occurrence. The first occurrence of each name is assigned a number, starting with `0`. A repeated occurrence of a name is then abbreviated by replacing `<name>$` by one of the numbers `0` - `9`. No replacement is possible if a higher number would be needed. A name can be replaced even if the repeated occurrence has a different meaning or context. There is no method for abbreviating repeated types.

Virtual tables have the funny code
```
<public name> ::=  W?$Wvf <nl> o4: <class name> $$nx[]pn()v
```
where `<nl>` is the length of the class name + 4, coded as a two-digit base 36 number with digits `0-9`, `a-z`.

Template classes are coded by replacing `<class name>$` by
`<name>$::` $[1n < \text{type} > \mid 0 < \text{number} >]_1^\infty$
where `<type>` is a template type parameter, and `<number>` is a template integer parameter, coded as a base-32 number with digits `0-9`, `a-v`, followed by a suffix `z` if positive or zero, and `y` if negative. Template functions have no special encoding as the template parameters are implied by the function parameter types. The Watcom compilers I have tested only support such cases of template functions where the template parameters can be inferred from the function parameters.

8.4 Gnu2 name mangling

This mangling scheme is used in Gnu C++ version 2.x.x under several operating systems (Linux, FreeBSD, Windows). Later versions of Gnu C++ use a different scheme described in the next section.

The Gnu2 mangling scheme is a dialect of the scheme used by cfront, one of the oldest C++ tools. Variants of this scheme are widely used in UNIX systems (See: J. R. Levine: Linkers and Loaders. Morgan Kaufmann Publishers, 2000).

The type of a global object is not coded, only class or namespace qualifiers, if any:

```
<public name> ::= [_ <qualifiers list> <list term> ] <name>
```

where

`<qualifiers list> ::= [<qualifiers count>]` $[< \text{name length} >< \text{class name} >]_1^\infty$

```
<list term> ::= . | $
```

`<qualifiers count>` is the number of class or namespace qualifiers. It is omitted if the count is 1. It is `Q<number>` if the number is `2` - `9`. It is `Q_<number>_` if the number is more than `9`. All numbers are decimal. Some versions use `.` as list terminator (Red Hat), other versions use `$` (FreeBSD, Cygwin, Mingw32). The namespace `std` is ignored.

```
char beta[6] = "Hello";
```
is coded as
```
beta
```

```
char Namespace1::beta[6];
```
is coded as
`_10Namespace1.beta` or `_10Namespace1$beta`

Functions and member functions are coded as follows

`<public name> ::= <name> _ _ [ <qualifiers list> | F ]` $[<\text{parameter type}>]_1^\infty$

The `<qualifiers list>` is replaced with an `F` if there are no class or namespace qualifiers.

`<parameter type>` is the type of each function parameter, as defined by table 9. The return type and function modifiers are not included.

Types that occur more than once in the parameter list can be repeated according to the following rules. Each parameter is assigned a number, beginning with `0`. All parameters are numbered, regardless of whether they are identical to a previous parameter. A repeated occurrence of a parameter is replaced by a reference to the first occurrence if it is a pointer, reference, array or other non-simple type. `bool` is also treated as a non-simple type, while `long double` and `unsigned __int64` are treated as simple types. A repeated occurrence of a non-simple parameter is replaced by `T <first occur>` where `<first occur>` is the number assigned to the first occurrence. If the number is bigger than `9` then `<first occur>` is followed by an `_`. A sequence of identical types can be replaced by `N <count> <first occur>` where `<count>` is the number of identical parameter types to replace and `<first occur>` is the number assigned to the first occurrence. Both `<count>` and `<first occur>` are followed by an `_` if bigger than `9`. For obscure reasons, the compiler uses the `T` replacement rather than the `N` replacement for the first parameter in a sequence of identical parameters if the preceding parameter is not the first occurrence of the same type. There is no method for abbreviating repeated names that are not part of identical parameter types.

Example:
```
bool ExampleFunction (int*a, int b, int c, int*d, bool e, bool f, bool*g);
```
is coded as
```
ExampleFunction__FPiiiT0bT4Pb
```

Constructors, destructors and operators are coded in the same way as functions with `<name>_` replaced by an operator name from table 16.

Template functions are coded as follows

`<public name> ::= <name> _ _ H <numtp>` $[Z<\text{type parameter}> \mid <\text{type}><\text{value}>]_{\text{numtp}}^{\text{numtp}}$

`_ [ <qualifiers list> ]` $[<\text{parameter type}> \mid X<\text{temp.par.num}>1]_1^\infty$ `_ <return type>`

where `<numtp>` is the number of template parameters, `<type parameter>` is a template type parameter, `<value>` is a template constant parameter of type `<type>`. `<parameter type>` in the list of function parameters is replaced by `X <temp. par. num.> 1` if, and only if, it is explicitly declared as the same type as a template type parameter. `<temp. par. num.>` is the number of the template parameter referred to, starting at `0`. The return type is included only for template functions.

Template classes are coded by replacing `<class name length> <class name>` by `t <name length> <name> <numtp>` $[Z<\text{type parameter}> \mid <\text{type}><\text{value}>]_{\text{numtp}}^{\text{numtp}}$

Under Windows, all public names get an additional underscore prefix, for example
```
_ExampleFunction__FPiiiT0bT4Pb
```

The _ prefix is used only under Windows. It is omitted in Linux and FreeBSD, except possibly if the old a.out object file format is used.

8.5 Gnu3 name mangling

This mangling scheme is used in Gnu C++ version 3.x.x under several operating systems (Linux, FreeBSD, Windows) and on several platforms. It is described in "Itanium C++ ABI".

Earlier versions of Gnu C++ use a different scheme described in the previous section.

The name of a global object without class or namespace qualifiers is not decorated in any way:

```
<public name> ::= <name>
```

A global object with class or namespace qualifiers is coded as

```
<public name> ::= _Z <qualified name>
```

where

`<qualified name> ::= N` $[<\text{simple name}>]_2^\infty$ `E`

```
<simple name> ::= <name length> <name>
```

where `<name length>` is the length of each name as a decimal number. In case of nested classes or namespaces, the outermost comes first. The object name comes last.

Examples:
```
char beta[6] = "Hello";
```
is coded as
```
beta
```

```
char Namespace1::beta[6];
```
is coded as
```
_ZN10Namespace14betaE
```

There are special abbreviations if the outermost namespace is `std`. If `std` is the only qualifier, use

```
<qualified name> ::= St <simple name>
```

If there are more qualifiers, use

`<qualified name> ::= N St` $[<\text{simple name}>]_2^\infty$ `E`

If `std` is not the outermost qualifier, then it is treated as any other qualifier, i.e. coded as `3std`.

Functions, member functions, constructors, destructors and operators are all coded according to the following syntax:

`<public name> ::= _Z <simple or qualified name>` $[<\text{parameter type}>]_1^\infty$

```
<simple or qualified name> ::= <simple name> | <qualified name> | <operator name>
```

`<operator name>` is an operator name from table 16. Any classes or namespaces come first in `<qualified name>` and the function name comes last. The abovementioned abbreviations for `std` apply.

Example:
```
bool Example1Function (int a, int * b, bool c, bool d, bool * e);
```

is coded as
`_Z16Example1FunctioniPibbPb`

Virtual tables are coded as

```
<public name> ::= _ZTV <simple or qualified name>
```

Template functions and template classes are coded by replacing `<simple name>` by

`<simple name> I` $[< \text{template parameter} >]_1^\infty$ `E`

```
<template parameter> ::= <type>  |  L <type> <value> E
```

where the first option is for template type parameters and the second option for template constant parameters.

The return type of a template function is included as the first type in the parameter type list. If the function has no parameters, then the code for void (`v`) is omitted. This is different from non-template functions, where the return parameter is omitted and the void is included.

There is a method for abbreviating repeated names and types. This abbreviation scheme does not distinguish between names and types. The first occurrence of each name or non-simple type is assigned a symbol in the following sequence:

```
S_, S0_, S1_, ... S9_, SA_, SB_, ... SZ_, S10_, S11_, ...
```

These abbreviation symbols are assigned, in the order of occurrence, to the first occurrence of each name of structures, classes, unions, enums and namespaces, but not to the name of the function or object itself. The abbreviation symbols are also assigned to all non-simple types occurring anywhere in the mangled name. A non-simple type is any type that needs more than one character for its encoding, according to table 9. This scheme also assigns abbreviation symbols to non-simple types that form part of the declaration of a more complex type. For example, the type `Class1**` gets three abbreviation symbols, for `Class1`, `Class1*`, and `Class1**`, respectively. All but the first occurrence of each name or type is replaced by the abbreviation symbol, even if the abbreviation symbol is longer than the original code. If the same name has more than one meaning because of different class or namespace qualifiers, then the occurrences with different meanings are treated as different names.

Template type parameters are included in this abbreviation scheme. A repeated occurrence of a type in `<template parameter>` is abbreviated by e.g. `S0_`. If a function parameter is explicitly declared as the same type as a template parameter, then the first occurrence is replaced by `T_`, `T0_`, etc., where `T_` refers to the first template parameter. A repeated occurrence of the template parameter in the function parameter list is abbreviated using the `S_` scheme.

Example:
```
bool Example2Function (int a, int * b, Class1 & c, Class1 d, Class1 & e);
```
is coded as
`_Z16Example2FunctioniPiR6Class1S0_S1_`

Under Windows, all public names get an additional underscore prefix, for example
`__Z16Example2FunctioniPiR6Class1S0_S1_`

8.6 Intel name mangling for Windows

Intel compilers for 32 bit and 64 bit Windows use the same name mangling scheme as Microsoft with very few exceptions. I have found the following differences:

- in 64-bit mode, global arrays are coded in the same way as in 32-bit mode.
- in 64-bit mode, references, pointers and arrays as function parameters have an extra `E` before the target storage class code. Global references and pointers, but not arrays, also have an additional `E` before their own storage class code. This may be a bug. These `E`'s are not generated by the Microsoft compiler.

8.7 Intel name mangling for Linux

Intel compilers for 32 bit and 64 bit Linux use the same name mangling scheme as Gnu 3.x.

8.8 Symantec and Digital Mars name mangling

The Symantec and Digital Mars C++ compilers use the same name mangling scheme as Microsoft with very few exceptions. I have found the following differences:

- `long double` has 80 bits precision and is coded as `_Z` in Symantec/Digital Mars compilers. In Microsoft compilers, long double has 64 bits precision (same as double) and is coded as O. Intel compilers use `_T` for 80 bits precision.
- The type `wchar_t` is coded as `_Y`, while Microsoft compilers use the code for `unsigned short int` (`G`).
- The method for abbreviating types (page 24) applies to `bool` (`_N`), but not to other two-character codes (`_J`, `_K`, `_Y`, `_Z`). It does apply to pointers and references to such types.
- The coding of member function pointers do not have the member function access code and return type modifier code. This may be an obsolete syntax, since it is also missing in 16-bit Microsoft compilers.
- Global arrays have the code `Q` while arrays as function parameters are coded as pointers (`P`) in Symantec and Digital Mars compilers. This is more correct than the coding generated by Microsoft compilers, as explained on page 22.

8.9 Turning off name mangling with extern "C"

Table 18. Function name prefixes with extern "C" declaration

Call type	Microsoft 16 bit	Microsoft 32 bit	Microsoft 64 bit	Dig. Mars	Borland 16, 32	Watcom 16, 32	Gnu2 Windows 32	Gnu3 Windows 32	Gnu2, Gnu3 Linux, BSD 32	Gnu3 Linux, BSD 64
`(default)`	`_`	`_`		`_`	`_`		`_`	`_`		
`__cdecl`	`_`	`_`		`_`	`_`	`_`	`_`	`_`		
`__stdcall`	`_`	`_`		`_`		`_`	`_`	`_`		
`__fastcall`		`@`		`_`	`@`		`_`	`@`		
`pascal`					`UC`	`UC`				

__fortran					_	UC				

Explanation:

_	name has underscore prefix
@	name has @ prefix
UC	entire name converted to upper case

Table 19. Function name postfixes with extern "C" declaration

Call type	Microsoft 16 bit	Microsoft 32 bit	Microsoft 64 bit	Dig. Mars	Borland 16, 32	Watcom 16, 32	Gnu2 Windows 32	Gnu3 Windows 32	Gnu2, Gnu3 Linux, BSD 32	Gnu3 Linux, BSD 64
(default)						_				
__cdecl										
__stdcall		@S		@S		@S	@S	@S		
__fastcall		@S				_		@S		
pascal										
__fortran										

Explanation:

_	underscore appended after name
@S	name followed by @, followed by the combined size of all parameters expressed as the number of bytes pushed on the stack as a decimal number. For `__fastcall`, register parameters are included by the size they would have if they were transferred on the stack.

The `extern "C"` attribute on a C++ function turns off name mangling so that the public or external name becomes compatible with the C language. This can be useful for solving problems with incompatible name mangling schemes. In 16 and 32 bit DOS and Windows systems, however, there is still some name decoration. The public and external names get the prefixes shown in table 18 and the postfixes shown in table 19. This may cause compatibility problems for `__stdcall` and `__fastcall` functions.

The `extern "C"` attribute is only allowed for functions that can be coded in C. Hence, overloaded functions and member functions cannot have the `extern "C"` attribute. When compatibility with all compilers is desired, you may give all functions the `extern "C"` attribute, replace overloaded functions by functions with different names, and replace member functions by `friend` functions.

External functions with the `__declspec(dllimport)` attribute have prefix `__imp_` in all compilers except Borland.

Functions with the names `main` and `WinMain` always have `extern "C"` coding. In addition, some compilers give `WinMain` the `__stdcall` attribute by default.

8.10 Conclusion

Various characteristics of the different name mangling schemes are compared in table 20.

Table 20. Comparison of name mangling schemes

	Microsoft	Borland	Watcom	Gnu2	Gnu3
unambigous and reversible	yes	yes	yes	no	no
includes type of global objects	yes	no	yes	no	no
includes storage class	yes	no	yes	no	no
includes function return type	yes	no	yes	no	no
includes calling convention	yes	yes	no	(no)	(no)

includes function modifiers	yes	few	some	no	no
compact	yes	somewhat	yes	somewhat	yes
allows case insensitive linking	yes	no	no	no	no
human readable	no	yes	yes	yes	yes
non-C characters used	$ @ ?	$ @ %	$?()[]:.	$ or .	none

The Gnu2 and Gnu3 schemes use only characters that are valid in C names in most or all cases. The reason for this is that these schemes have their origin in tools that convert from C++ to C, so the mangled names must be valid C names. This has the disadvantage that the mangled name cannot unambiguously be translated back to the original C++ declaration. The mangled name of a C++ function could in principle be the unmangled name of a variable. This disadvantage is avoided if the mangled code contains characters that are not valid in C++ names. On the other hand, the character set should be restricted to characters that can be generated by common assemblers in order to allow compilation through assembly or linking with assembly language modules. The characters `$ ? @` are allowed in Microsoft and Borland assemblers. The Gnu assembler allows `.` and `$`. The Watcom assembler allows all characters in public symbols.

We will prefer that a name mangling scheme is complete, consistent and compact. It should also be relatively easy for humans to interpret the code, though this requirement conflicts with the desire for compactness. The Microsoft and Gnu3 schemes are the ones that have the most consistent syntax.

9 Exception handling and stack unwinding

An exception or a `longjmp` can lead to a process where functions are exited without the normal return being executed. Objects that go out of scope by this process may have destructors that need to be called. In order to find all objects that need to have their destructors called, the system must unwind the stack to trace backwards through consecutive function calls.

If a function has any local objects with destructors and if an exception or `longjmp` can occur inside the function or any of its child functions, then this function must support stack unwinding. The method of stack unwinding is different for different compilers. Usually, this process uses stack frames based on `BP/EBP/RBP`. The function prolog must save the old value of the frame pointer and save the value of the stack pointer in the frame pointer register:

```
_FunctionWithFramePointer PROC NEAR
        PUSH    EBP
        MOV     EBP, ESP
        ...
        MOV     ESP, EBP
        POP     EBP
        RET
_FunctionWithFramePointer ENDP
```

Additional information about destructors to be called may be provided either by the function itself or by data in a static data segment designed for only this purpose.

If a structured exception or `longjmp` can happen inside a function or any of its child functions, then it is not allowed to use `BP/EBP/RBP` for any other purpose than a frame pointer.

10 Initialization and termination functions

A C++ module may contain global objects with constructors that must be called before `main` is called, and destructors to be called after `main` has returned. There may be other initialization and termination tasks to perform, too. For this purpose, the compiler provides a list of pointers to initialization functions and termination functions. These lists of function pointers are stored in separate data segments designed for only this purpose. These lists may contain additional information about the priority or order in which the initialization and termination functions should be called. The names of these segments and the format of these tables are different for different compilers.

11 Virtual tables and runtime type identification

A class with virtual member functions always has a virtual table. This is a table of member function pointers used for finding the right version of a polymorphous function. Each instance of the class has a pointer to the virtual table. Borland, Microsoft and Gnu version 3.x compilers place the pointer to the virtual table at the beginning of the object, while Watcom and Gnu version 2.x compilers place it at the end of the object. This affects the offset of all data members of the class so that member functions may be incompatible between compilers.

Information for runtime type identification is usually stored in connection with the virtual table.

The "Itanium C++ ABI" includes more detailed information about the representation of virtual tables and runtime type identification. This information may apply to other platforms as well.

12 Communal data

Communal data are data that may occur identically in more than one module, but the final executable should contain only one instance of this redundant information. The linker will check that all instances are identical and store only one instance in the executable file. Communal data are used for virtual tables, template functions, and possibly for global data.

You cannot expect communal data produced by different compilers to be identical or to be identified in the same way in the object files. Assemblers may not support communal data.

13 Object file formats

Borland, Watcom, Symantec, Digital Mars and 16-bit Microsoft compilers use the OMF format for object files (also called Microsoft 8086 relocatable). Microsoft and Gnu compilers for 32-bit Windows use MS Windows COFF format (also called PE-i386). The Gnu compiler under Linux, BSD, and similar systems prefers ELF format (ELF32-i386). Gnu tools running under Linux and BSD usually accept several other formats, not including the formats used under Windows. Gnu tools running under Windows accept MS COFF and ELF formats.

Compilers for x64 Linux and BSD use ELF64-x86-64 format. Compilers for x64 Windows use a modification of MS COFF format called PE32+.

It is sometimes possible to convert object files from one format to another if appropriate tools can be found. The Microsoft command-line linker can convert from OMF to MS COFF. The objcopy utility in the Gnu binutils package running under Windows can convert from MS COFF to ELF.

Some compilers have features for optimizing code across function calls and modules (whole program optimization). These compilers use intermediate files containing partially compiled code. The format for these intermediate files is not standardized. It is not even guaranteed to be compatible between different versions of the same compiler. The intermediate file format is therefore not suitable for distributing function libraries. It would be useful to have a standardized object file format that includes information about which registers each function modifies, in order to optimize register allocation. Such a file format has not been implemented for the platforms I have studied.

14 Debug information

Compilers differ in the way they store debugging information and information for profilers in object files and executable files. Thus, it may not be possible to use the same debugger for all compilers. The information stored includes names of source files, line numbers, and variable names. I have not studied differences in the way this information is stored.

15 Data endian-ness

All systems based on 16, 32 or 64 bit x86 microprocessors use little-endian data storage, i.e. the least significant byte of a multi-byte data unit is stored at the lowest address. Some other microprocessor platforms use big-endian data storage. This can give rise to compatibility problems when exchanging binary data files between platforms, and when porting C++ programs that explicitly address part of a data object, such as the sign bit of a floating point number.

16 Predefined macros

Most C++ compilers have a predefined macro containing the version number of the compiler. Programmers can use preprocessing directives to check for the existence of these macros in order to detect which compiler the program is compiled on and thereby fix problems with incompatible compilers.

Table 21. Compiler version predefined macros

Compiler	predefined macro
Borland	`__BORLANDC__`
Digital Mars	`__DMC__`
Gnu	`__GNUC__`
Intel	`__INTEL_COMPILER`
Microsoft	`_MSC_VER`
Symantec	`__SYMANTECC__`
Watcom	`__WATCOMC__`

17 Available C++ Compilers

17.1 Microsoft

Microsoft Visual C++ comes in several different versions. The professional edition, which is relatively expensive, includes integrated development environment and many tools including debugger and profiler. An assembler is available for free as a plug-in. A student edition of

Visual C++ comes with various C++ textbooks, available from university bookstores. Command line versions of the 16, 32 and 64-bit C++ compilers and assemblers can be acquired very cheaply by ordering the Windows server 2003 driver development kit or the Platform Software Development Kit (SDK) from www.microsoft.com.

17.2 Borland

Borland C++ compilers and development tools are available in several different versions. A command line version can be downloaded for free from www.borland.com.

17.3 Watcom

Watcom C++ is no longer sold commercially, but it has been continued as an open source project. The Watcom compiler is available from www.openwatcom.org, including integrated development environment, debugger, profiler, assembler, disassembler, and other tools. The old commercial version is compatible with the new open source version.

Users of this compiler should be aware that register usage and calling conventions differ from other compilers. You must fix these problems by using #pragma's when defining or calling DLL functions and when combining with tools from other vendors.

17.4 Gnu

Gnu C++ is an open source compiler that comes with all distributions of Linux and FreeBSD. Windows versions are available from www.cygwin.com and www.mingw.org. Also available is an assembler (GAS) which uses the terrible AT&T syntax.

17.5 Digital Mars

Digital Mars C++ compiler is a continuation of Symantec C++, available from www.digitalmars.com. The compiler package is cheap, and a command line version is available for free. Both Symantec C++ and Digital Mars C++ are binary compatible with Microsoft C++ in most respects, including calling conventions and name mangling.

17.6 Intel

The 32-bit and 64-bit Intel C++ compilers for Windows and Linux are available in commercial versions and time-limited trial versions. The Intel C++ compilers for Linux are also available as free versions for non-commercial applications.

18 Literature

18.1 ABIs for Unix, Linux, BSD, etc.

At the time of writing (Nov 2004), there is no clear indication of which ABI documents apply to which platforms, and where to look for the latest authoritative version of each document. The reader is advised to do a Google search for the titles. I have been informed, that the "Itanium C++ ABI" applies to other hardware platforms than the Itanium, except for a few processor-specific details, though this fact appears to be unofficial and not all x86 compilers conform to this ABI. The "Itanium C++ ABI" contains valuable information about the representation of member pointers, virtual tables, runtime type identification and name mangling, not found anywhere else. This information may apply to 32-bit and 64-bit x86 platforms as well.

- SYSTEM V. APPLICATION BINARY INTERFACE. Edition 4.1, 1996. www.caldera.com

- Itanium C++ ABI. Revision 1.75. Draft, Oct 2004. www.codesourcery.com

Linux and FreeBSD, 32 bits:

- SYSTEM V. APPLICATION BINARY INTERFACE. Intel386 Architecture Processor Supplement. Fourth Edition.

Linux and FreeBSD, 64 bits:

- System V Application Binary Interface. AMD64 Architecture Processor Supplement. Draft Version 0.92, 2004.

18.2 ABIs for Windows

Windows, 32 bits:

- C++ Language Reference: Calling Conventions. msdn.microsoft.com, 2004.

Windows, 64 bits:

- Calling Convention for AMD 64-Bit Environments. msdn.microsoft.com, 2004.