WIKIPEDIA

Name mangling

In compiler construction, **name mangling** (also called **name decoration**) is a technique used to solve various problems caused by the need to resolve unique names for programming entities in many modern programming languages.

It provides a way of encoding additional information in the name of a function, structure, class or another datatype in order to pass more semantic information from the compilers to linkers.

The need arises where the language allows different entities to be named with the same identifier as long as they occupy a different namespace (where a namespace is typically defined by a module, class, or explicit *namespace* directive) or have different signatures (such as function overloading).

Any object code produced by compilers is usually linked with other pieces of object code (produced by the same or another compiler) by a type of program called a linker. The linker needs a great deal of information on each program entity. For example, to correctly link a function it needs its name, the number of arguments and their types, and so on.

Contents

Examples

C

Although name mangling is not generally required or used by languages that do not support function overloading (such as C and classic Pascal), they use it in some cases to provide additional information about a function. For example, compilers targeted at Microsoft Windows platforms support a variety of calling conventions, which determine the manner in which parameters are sent to subroutines and results returned. Because the different calling conventions are not compatible with one another, compilers mangle symbols with codes detailing which convention should be used to call the specific routine.

The mangling scheme was established by Microsoft, and has been informally followed by other compilers including Digital Mars, Borland, and GNU GCC, when compiling code for the Windows platforms. The scheme even applies to other languages, such as Pascal, D, Delphi, Fortran, and C#. This allows subroutines written in those languages to call, or be called by, existing Windows libraries using a calling convention different from their default.

When compiling the following C examples:

```
int _cdecl    f (int x) { return 0; }
int _stdcall  g (int y) { return 0; }
int _fastcall h (int z) { return 0; }
```

32 bit compilers emit, respectively:

```
_f
_g@4
@h@4
```

In the stdcall and fastcall mangling schemes, the function is encoded as _name@X and @name@X respectively, where **X** is the number of bytes, in decimal, of the argument(s) in the parameter list (including those passed in registers, for fastcall). In the case of cdecl, the function name is merely prefixed by an underscore.

The 64-bit convention on Windows (Microsoft C) has no leading underscore. This difference may in some rare cases lead to unresolved externals when porting such code to 64 bits. For example, Fortran code can use 'alias' to link against a C method by name as follows:

```
SUBROUTINE f()
!DEC$ ATTRIBUTES C, ALIAS:'_f' :: f
END SUBROUTINE
```

This will compile and link fine under 32 bits, but generate an unresolved external '_f' under 64 bits. One workaround for this is not to use 'alias' at all (in which the method names typically need to be capitalized in C and Fortran). Another is to use the BIND option:

```
SUBROUTINE f() BIND(C,NAME="f")
END SUBROUTINE
```

C++

C++ compilers are the most widespread users of name mangling. The first C++ compilers were implemented as translators to C source code, which would then be compiled by a C compiler to object code; because of this, symbol names had to conform to C identifier rules. Even later, with the emergence of compilers which produced machine code or assembly directly, the system's linker generally did not support C++ symbols, and mangling was still required.

The C++ language does not define a standard decoration scheme, so each compiler uses its own. C++ also has complex language features, such as classes, templates, namespaces, and operator overloading, that alter the meaning of specific symbols based on context or usage. Meta-data about these features can be disambiguated by mangling (decorating) the name of a symbol. Because the name-mangling systems for such features are not standardized across compilers, few linkers can link object code that was produced by different compilers.

Simple example

A single C++ translation unit might define two functions named `f()`:

```
int  f (void) { return 1; }
int  f (int)  { return 0; }
void g (void) { int i = f(), j = f(0); }
```

These are distinct functions, with no relation to each other apart from the name. The C++ compiler therefore will encode the type information in the symbol name, the result being something resembling:

```
int  __f_v (void) { return 1; }
int  __f_i (int)  { return 0; }
void __g_v (void) { int i = __f_v(), j = __f_i(0); }
```

Even though its name is unique, g() is still mangled: name mangling applies to **all** symbols.

Complex example

The mangled symbols in this example, in the comments below the respective identifier name, are those produced by the GNU GCC 3.x compilers:

```
namespace wikipedia
{
  class article
  {
  public:
    std::string format (void);
      /* = _ZN9wikipedia7article6formatEv */

    bool print_to (std::ostream&);
      /* = _ZN9wikipedia7article8print_toERSo */

    class wikilink
    {
    public:
      wikilink (std::string const& name);
        /* = _ZN9wikipedia7article8wikilinkC1ERKSs */
    };
  };
}
```

All mangled symbols begin with **_Z** (note that an underscore followed by a capital is a reserved identifier in C, so conflict with user identifiers is avoided); for nested names (including both namespaces and classes), this is followed by N, then a series of <length, id> pairs (the length being the length of the next identifier), and finally E. For example, wikipedia::article::format becomes

```
_ZN9Wikipedia7article6formatE
```

For functions, this is then followed by the type information; as format() is a void function, this is simply v; hence:

```
_ZN9Wikipedia7article6formatEv
```

For print_to, a standard type std::ostream (or more properly std::basic_ostream<char, char_traits<char> >) is used, which has the special alias So; a reference to this type is therefore RSo, with the complete name for the function being:

```
_ZN9Wikipedia7article8print_toERSo
```

How different compilers mangle the same functions

There isn't a standard scheme by which even trivial C++ identifiers are mangled, and consequently different compiler vendors (or even different versions of the same compiler, or the same compiler on different platforms) mangle public symbols in radically different (and thus totally incompatible) ways. Consider how different C++ compilers mangle the same functions:

Compiler	void h(int)	void h(int, char)	void h(void)
Intel C++ 8.0 for Linux	_Z1hi	_Z1hic	_Z1hv
HP aC++ A.05.55 IA-64	_Z1hi	_Z1hic	_Z1hv
IAR EWARM C++ 5.4 ARM	_Z1hi	_Z1hic	_Z1hv
GCC 3.*x* and higher	_Z1hi	_Z1hic	_Z1hv
Clang 1.*x* and higher[1]	_Z1hi	_Z1hic	_Z1hv
IAR EWARM C++ 7.4 ARM	_Z<number>hi	_Z<number>hic	_Z<number>hv
GCC 2.9*x*	h__Fi	h__Fic	h__Fv
HP aC++ A.03.45 PA-RISC	h__Fi	h__Fic	h__Fv
Microsoft Visual C++ v6-v10 (mangling details)	?h@@YAXH@Z	?h@@YAXHD@Z	?h@@YAXXZ
Digital Mars C++	?h@@YAXH@Z	?h@@YAXHD@Z	?h@@YAXXZ
Borland C++ v3.1	@h$qi	@h$qizc	@h$qv
OpenVMS C++ V6.5 (ARM mode)	H__XI	H__XIC	H__XV
OpenVMS C++ V6.5 (ANSI mode)		CXX$__7H__FIC26CDH77	CXX$__7H__FV2CB06E8
OpenVMS C++ X7.1 IA-64	CXX$_Z1HI2DSQ26A	CXX$_Z1HIC2NP3LI4	CXX$_Z1HV0BCA19V
SunPro CC	__1cBh6Fi_v_	__1cBh6Fic_v_	__1cBh6F_v_
Tru64 C++ V6.5 (ARM mode)	h__Xi	h__Xic	h__Xv
Tru64 C++ V6.5 (ANSI mode)	__7h__Fi	__7h__Fic	__7h__Fv
Watcom C++ 10.6	W?h$n(i)v	W?h$n(ia)v	W?h$n()v

Notes:

- The Compaq C++ compiler on OpenVMS VAX and Alpha (but not IA-64) and Tru64 has two name mangling schemes. The original, pre-standard scheme is known as ARM model, and is based on the name mangling described in the C++ Annotated Reference Manual (ARM). With the advent of new features in standard C++, particularly templates, the ARM scheme became more and more unsuitable — it could not encode certain function types, or produced identical mangled names for different functions. It was therefore replaced by the newer "ANSI" model, which supported all ANSI template features, but was not backwards compatible.

- On IA-64, a standard ABI exists (see external links), which defines (among other things) a standard name-mangling scheme, and which is used by all the IA-64 compilers. GNU GCC 3.*x*, in addition, has adopted the name mangling scheme defined in this standard for use on other, non-Intel platforms.
- The Visual Studio and Windows SDK include the program undname which prints the C-style function prototype for a given mangled name.
- On Microsoft Windows, the Intel compiler[2] and Clang[3] uses the Visual C++ name mangling for compatibility.
- For the IAR EWARM C++ 7.4 ARM compiler the best way to determine the name of a function is to compile with the assembler output turned on and to look at the output in the ".s" file thus generated.

Handling of C symbols when linking from C++

The job of the common C++ idiom:

```
#ifdef __cplusplus
extern "C" {
#endif
    /* ... */
#ifdef __cplusplus
}
#endif
```

is to ensure that the symbols within are "unmangled" – that the compiler emits a binary file with their names undecorated, as a C compiler would do. As C language definitions are unmangled, the C++ compiler needs to avoid mangling references to these identifiers.

For example, the standard strings library, `<string.h>` usually contains something resembling:

```
#ifdef __cplusplus
extern "C" {
#endif

void *memset (void *, int, size_t);
char *strcat (char *, const char *);
int   strcmp (const char *, const char *);
char *strcpy (char *, const char *);

#ifdef __cplusplus
}
#endif
```

Thus, code such as:

```
if (strcmp(argv[1], "-x") == 0)
    strcpy(a, argv[2]);
```

```
else
  memset (a, 0, sizeof(a));
```

uses the correct, unmangled strcmp and memset. If the extern had not been used, the (SunPro) C++ compiler would produce code equivalent to:

```
if (__1cGstrcmp6Fpkc1_i_(argv[1], "-x") == 0)
  __1cGstrcpy6Fpcpkc_0_(a, argv[2]);
else
  __1cGmemset6FpviI_0_ (a, 0, sizeof(a));
```

Since those symbols do not exist in the C runtime library (*e.g.* libc), link errors would result.

Standardised name mangling in C++

Though it would seem that standardised name mangling in the C++ language would lead to greater interoperability between compiler implementations, such a standardization by itself would not suffice to guarantee C++ compiler interoperability and it might even create a false impression that interoperability is possible and safe when it isn't. Name mangling is only one of several application binary interface (ABI) details that need to be decided and observed by a C++ implementation. Other ABI aspects like exception handling, virtual table layout, structure and stack frame padding, *etc.* also cause differing C++ implementations to be incompatible. Further, requiring a particular form of mangling would cause issues for systems where implementation limits (e.g., length of symbols) dictate a particular mangling scheme. A standardised *requirement* for name mangling would also prevent an implementation where mangling was not required at all — for example, a linker which understood the C++ language.

The C++ standard therefore does not attempt to standardise name mangling. On the contrary, the *Annotated C++ Reference Manual* (also known as *ARM*, ISBN 0-201-51459-1, section 7.2.1c) actively encourages the use of different mangling schemes to prevent linking when other aspects of the ABI, such as exception handling and virtual table layout, are incompatible.

Nevertheless, as detailed in the section above, on some platforms[4] the full C++ ABI has been standardized, including name mangling.

Real-world effects of C++ name mangling

Because C++ symbols are routinely exported from DLL and shared object files, the name mangling scheme is not merely a compiler-internal matter. Different compilers (or different versions of the same compiler, in many cases) produce such binaries under different name decoration schemes, meaning that symbols are frequently unresolved if the compilers used to create the library and the program using it employed different schemes. For example, if a system with multiple C++ compilers installed (e.g., GNU GCC and the OS vendor's compiler) wished to install the Boost C++ Libraries, it would have to be compiled multiple times (once for GCC and once for the vendor compiler).

It is good for safety purposes that compilers producing incompatible object codes (codes based on different ABIs, regarding e.g., classes and exceptions) use different name mangling schemes. This guarantees that these incompatibilities are detected at the linking phase, not when executing the software (which could lead to obscure bugs and serious stability issues).

For this reason name decoration is an important aspect of any C++-related ABI.

Demangle via c++filt

```
$ c++filt _ZNK3MapI10StringName3RefI8GDScriptE10ComparatorIS0_E16DefaultAllocatorE3hasERKS0_
Map<StringName, Ref<GDScript>, Comparator<StringName>, DefaultAllocator>::has(StringName const&) const
```

Demangle via builtin GCC ABI

```
#include <stdio.h>
#include <stdlib.h>
#include <cxxabi.h>

int main() {
  const char *mangled_name = "_ZNK3MapI10StringName3RefI8GDScriptE10ComparatorIS0_E16DefaultAllocatorE3hasERKS0_";
  char *demangled_name;
  int status = -1;
  demangled_name = abi::__cxa_demangle(mangled_name, NULL, NULL, &status);
  printf("Demangled: %s\n", demangled_name);
  free(demangled_name);
  return 0;
}
```

Output:

```
Demangled: Map<StringName, Ref<GDScript>, Comparator<StringName>, DefaultAllocator>::has(StringName const&) const
```

Java

In Java, the **signature** of a method or a class contains its name and the types of its method arguments and return value where applicable. The format of signatures is documented, as the language, compiler, and .class file format were all designed together (and had object-orientation and universal interoperability in mind from the start).

Creating unique names for inner and anonymous classes

The scope of anonymous classes is confined to their parent class, so the compiler must produce a "qualified" public name for the inner class, to avoid conflict where other classes with the same name (inner or not) exist in the same namespace. Similarly, anonymous classes must have "fake" public names generated for them (as the concept of anonymous classes only exists in the compiler, not the runtime). So, compiling the following java program

```
public class foo {
  class bar {
    public int x;
  }

  public void zark () {
    Object f = new Object () {
      public String toString() {
        return "hello";
      }
    };
  }
}
```

will produce three **.class** files:

- foo.class, containing the main (outer) class *foo*
- foo$bar.class, containing the named inner class *foo.bar*
- foo$1.class, containing the anonymous inner class (local to method *foo.zark*)

All of these class names are valid (as $ symbols are permitted in the JVM specification) and these names are "safe" for the compiler to generate, as the Java language definition prohibits $ symbols in normal java class definitions.

Name resolution in Java is further complicated at runtime, as fully qualified class names are unique only inside a specific classloader instance. Classloaders are ordered hierarchically and each Thread in the JVM has a so-called context class loader, so in cases where two different classloader instances contain classes with the same name, the system first tries to load the class using the root (or system) classloader and then goes down the hierarchy to the context class loader.

Java Native Interface

Java's native method support allows Java language programs to call out to programs written in another language (generally either C or C++). There are two name-resolution concerns here, neither of which is implemented in a particularly standard manner:

- Java to native name translation
- normal C++ name mangling

Python

In Python, mangling is used for "private" class members which are designated as such by giving them a name with two leading underscores and no more than one trailing underscore. For example, __thing will be mangled, as will ___thing and __thing_, but __thing__ and __thing___ will not. Python's runtime does not restrict access to such members, the mangling only prevents name collisions if a derived class defines a member with the same name.

On encountering name mangled attributes, Python transforms these names by a single underscore and the name of the enclosing class, for example:

```
>>> class Test(object):
...     def __mangled_name(self):
...         pass
...     def normal_name(self):
...         pass
>>> t = Test()
>>> [attr for attr in dir(t) if 'name' in attr]
['_Test__mangled_name', 'normal_name']
```

Pascal

Borland's Turbo Pascal / Delphi range

To avoid name mangling in Pascal, use:

```
exports
  myFunc name 'myFunc',
  myProc name 'myProc';
```

Free Pascal

Free Pascal supports function and operator overloading, thus it also uses name mangling to support these features. On the other hand, Free Pascal is capable of calling symbols defined in external modules created with another language and exporting its own symbols to be called by another language. For further information, consult Chapter 6.2 (http://www.freepascal.org/docs-html/prog/progse21.html) and Chapter 7.1 (http://www.freepascal.org/docs-html/prog/progse27.html) of Free Pascal Programmer's Guide (http://www.freepascal.org/docs-html/prog/prog.html).

Fortran

Name mangling is also necessary in Fortran compilers, originally because the language is case insensitive. Further mangling requirements were imposed later in the evolution of the language because of the addition of modules and other features in the Fortran 90 standard. The case mangling, especially, is a common issue that must be dealt with in order to call Fortran libraries (such as LAPACK) from other languages (such as C).

Because of the case insensitivity, the name of a subroutine or function "FOO" must be converted to a canonical case and format by the Fortran compiler so that it will be linked in the same way regardless of case. Different compilers have implemented this in various ways, and no standardization has occurred. The AIX and HP-UX Fortran compilers convert all identifiers to lower case ("foo"), while the Cray Unicos Fortran compilers converted identifiers all upper case ("FOO"). The GNU g77 compiler converts identifiers to lower case plus an underscore ("foo_"), except that identifiers already containing an underscore ("FOO_BAR") have two underscores appended ("foo_bar__"), following a convention established by f2c. Many other compilers, including SGI's IRIX compilers, GNU Fortran, and Intel's Fortran compiler (except on Microsoft Windows), convert all identifiers to lower case plus an underscore ("foo_" and "foo_bar_"). On Microsoft Windows, the Intel Fortran compiler defaults to uppercase without an underscore.[5]

Identifiers in Fortran 90 modules must be further mangled, because the same procedure name may occur in different modules. Since the Fortran 2003 Standard requires that module procedure names not conflict with other external symbols,[6] compilers tend to use the module name and the procedure name, with a distinct marker in between. For example, in the following module

```
module m
contains
  integer function five()
    five = 5
  end function five
end module m
```

The name of the function will be mangled as `__m_MOD_five` (e.g., GNU Fortran), `m_MP_five_` (e.g., Intel's ifort), `m.five_` (e.g., Oracle's sun95), etc. Since Fortran does not allow overloading the name of a procedure, but uses generic interface blocks and generic type-bound procedures instead, the mangled names do not need to incorporate clues about the arguments.

The Fortran 2003 BIND option overrides any name mangling done by the compiler, as shown above.

Rust

Function names are mangled by default in Rust. However, this can be disabled by the `#[no_mangle]` function attribute. This attribute can be used to export functions to C, C++, or Objective-C.[7] Additionally, along with the `#[start]` function attribute or the `#[no_main]` crate attribute, it allows the user to define a C-style entry point for the program.[8] Starting from Rust 1.9, Rust uses a C++-style name-mangling scheme (which partly follows the Itanium C++ ABI).[9]

Objective-C

Essentially two forms of method exist in Objective-C, the class ("static") method, and the instance method. A method declaration in Objective-C is of the following form

```
+ (return-type) name0:parameter0 name1:parameter1 ...
– (return-type) name0:parameter0 name1:parameter1 ...
```

Class methods are signified by +, instance methods use -. A typical class method declaration may then look like:

```
+ (id) initWithX: (int) number andY: (int) number;
+ (id) new;
```

with instance methods looking like

```
- (id) value;
- (id) setValue: (id) new_value;
```

Each of these method declarations have a specific internal representation. When compiled, each method is named according to the following scheme for class methods:

```
_c_Class_name0_name1_ ...
```

and this for instance methods:

```
_i_Class_name0_name1_ ...
```

The colons in the Objective-C syntax are translated to underscores. So, the Objective-C class method `+ (id) initWithX: (int) number andY: (int) number;`, if belonging to the `Point` class would translate as `_c_Point_initWithX_andY_`, and the instance method (belonging to the same class) `- (id) value;` would translate to `_i_Point_value`.

Each of the methods of a class are labeled in this way. However, in order to look up a method that a class may respond to would be tedious if all methods are represented in this fashion. Each of the methods is assigned a unique symbol (such as an integer). Such a symbol is known as a *selector*. In Objective-C, one can manage selectors directly — they have a specific type in Objective-C — `SEL`.

During compilation, a table is built that maps the textual representation (such as `_i_Point_value`) to selectors (which are given a type `SEL`). Managing selectors is more efficient than manipulating the textual representation of a method. Note that a selector only matches a method's name, not the class it belongs to — different classes can have different implementations of a method with the same name. Because of this, implementations of a method are given a specific identifier too — these are known as implementation pointers, and are given a type also, `IMP`.

Message sends are encoded by the compiler as calls to the `id objc_msgSend (id receiver, SEL selector, ...)` function, or one of its cousins, where `receiver` is the receiver of the message, and `SEL` determines the method to call. Each class has its own table that maps selectors to their implementations — the implementation pointer specifies where in memory the actual implementation of the method resides. There are separate tables for class and instance methods. Apart from being stored in the `SEL` to `IMP` lookup tables, the functions are essentially anonymous.

The `SEL` value for a selector does not vary between classes. This enables polymorphism.

The Objective-C runtime maintains information about the argument and return types of methods. However, this information is not part of the name of the method, and can vary from class to class.

Since Objective-C does not support namespaces, there is no need for mangling of class names (that do appear as symbols in generated binaries).

Swift

Swift keeps metadata about functions (and more) in the mangled symbols referring to them. This metadata includes the function's name, attributes, module name, parameter types, return type, and more. For example:

The mangled name for a method func calculate(x: int) -> int of a MyClass class in module test is **_TFC4test7MyClass9calculatefS0_FT1xSi_Si**. The components and their meanings are as follows:[10]

_T: The prefix for all Swift symbols. Everything will start with this.

F: Non-curried function.

C: Function of a class. (method)

4test: The module name, with a prefixed length.

7MyClass: The class name the function belongs to, again, with a prefixed length.

9calculate: The function name.

f: The function attribute. In this case it's 'f', which is a normal function.

S0: Designates the type of the first parameter (namely the class instance) as the first in the type stack (here MyClass is not nested and thus has index 0).

_FT: This begins the type list for the parameter tuple of the function.

1x: External name of first parameter of the function.

Si: Indicates builtin Swift type Swift.Int for the first parameter.

_Si: The return type; again Swift.Int.

See also

- Application programming interface
- Application Binary Interface
- Calling convention
- Comparison of application virtual machines
- Foreign function interface
- Java Native Interface
- Language binding
- Stropping
- SWIG

References

1. *Clang - Features and Goals: GCC Compatibility* (http://clang.llvm.org/features.html#gcccompat), 15 April

2013

2. http://software.intel.com/en-us/forums/showthread.php?t=56817
3. "MSVC compatibility" (http://clang.llvm.org/docs/MSVCCompatibility.html#abi-features). Retrieved 13 May 2016.
4. "Itanium C++ ABI, Section 5.1 External Names (a.k.a. Mangling)" (https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling). Retrieved 16 May 2016.
5. *User and Reference Guide for the Intel Fortran Compiler 15.0*, Intel Corporation (2014), Summary of Mixed-Language Issues (https://software.intel.com/en-us/node/525345). Accessed 17 Nov. 2014.
6. https://software.intel.com/en-us/node/510637
7. "Foreign Function Interface # Calling Rust code from C" (https://doc.rust-lang.org/book/ffi.html#calling-rust-code-from-c). *Rust Manual*. rust-lang.org. Retrieved 13 May 2016.
8. "No stdlib" (https://doc.rust-lang.org/book/no-stdlib.html). *Rust Manual*. rust-lang.org. Retrieved 13 May 2016.
9. "rust/symbol_names.rs at 76affa5d6f5d1b8c3afcd4e0c6bbaee1fb0daeb4 · rust-lang/rust · GitHub" (https://github.com/rust-lang/rust/blob/76affa5d6f5d1b8c3afcd4e0c6bbaee1fb0daeb4/src/librustc_trans/back/symbol_names.rs#L358). Retrieved 13 May 2016.
10. Swift Name Mangling (https://mikeash.com/pyblog/friday-qa-2014-08-15-swift-name-mangling.html)

External links

- Linux Itanium ABI for C++ (https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling), including name mangling scheme.
- Macintosh C/C++ ABI Standard Specification (https://developer.apple.com/tools/mpw-tools/compilers/docs/abi_spec.pdf)
- c++filt (http://sources.redhat.com/binutils/docs-2.15/binutils/c--filt.html) — filter to demangle encoded C++ symbols for GNU/Intel compilers
- undname (http://msdn2.microsoft.com/en-us/library/5x49w699.aspx) — msvc tool to demangle names.
- demangler.com (http://demangler.com) — An online tool for demangling GCC and MSVC C++ symbols
- The Objective-C Runtime System (https://developer.apple.com/legacy/mac/library/documentation/Cocoa/Conceptual/OOPandObjC1/Articles/ocRuntimeSystem.html#//apple_ref/doc/uid/TP40005191-CH9-CJBBBCHG) — From Apple's *The Objective-C Programming Language 1.0 (https://developer.apple.com/legacy/mac/library/documentation/Cocoa/Conceptual/OOPandObjC1/Introduction/introObjectiveC.html)*
- Calling conventions for different C++ compilers (http://www.agner.org/optimize/calling_conventions.pdf) by Agner Fog contains detailed description of name mangling schemes for various x86 and x64 C++ compilers (pp. 24–42 in 2011-06-08 version)
- C++ Name Mangling/Demangling (http://www.kegel.com/mangle.html#operators) Quite detailed explanation of Visual C++ compiler name mangling scheme
- PHP UnDecorateSymbolName (http://sourceforge.net/projects/php-ms-demangle/) a php script that demangles Microsoft Visual C's function names.
- Mixing C and C++ Code (http://www.parashift.com/c++-faq-lite/mixing-c-and-cpp.html)
- Symbol management – 'Linkers and Loaders' by John R. Levine (http://www.iecc.com/linker

/linker05.html)
- Name mangling demystified by Fivos Kefallonitis (http://www.int0x80.gr/papers/name_mangling.pdf)