int 0x80

Name mangling demystified

29/10/07

Fivos Kefallonitis

gorlist@int0x80.gr

http://www.int0x80.gr/papers/name_mangling.pdf

Copyright © Fivos Kefallonitis 2007

Contents

Introduction

This paper will discuss name mangling, also known as name decoration.

Name mangling is the process of generating unique names.
Name mangling is applied in several areas, although it's primarily used for generating unique (unambiguous) names for identifiers in a computer programming language.
Programming languages that deal with function/method overloading have compilers that implement some name mangling scheme.
Some examples are C++ (most people mean C++ when they address name mangling), C#, Java, Python, Objective C and Fortran.
Name mangling is highly compiler-dependent and it may also vary from one version to another of the same compiler.
The mangled name contains all the necessary information that the linker needs, such as linkage type, scope, calling convention, number of arguments, type of arguments etc.
Name mangling applies primarily to overloaded functions overloaded operators and variables.
Name mangling allows the linker to distinguish between the different overloaded functions that have the same identifier but different parameter lists.
Name demangling in programming context is the exact opposite, the process of changing the internal representation of identifiers back to their original source names.

Other examples of name mangling

Name mangling can be used to achieve "security through obscurity".

Name mangling is a very important aspect of code obfuscators.
An ideal obfuscator mangles namespaces, class names, method signatures fields and even string values of the binary/assembly without changing its functionality.

Name mangling is also used by operating systems.
- MSDOS only allows a maximum of 12 characters for filenames. It uses a 8.3 filename format (8

characters for filename, dot and 3 characters for the extension). Anything longer than 8.3 will be name mangled (truncated if you prefer) to 6 non-space characters, a tilde (~) and a number to ensure that the filename will be unique. E.g. "Program Files" becomes "PROGRA~1" and "My Documents" becomes "MYDOCU~1". Windows still have this for backwards compatibility with MSDOS.

- Filenames in *nix can contain colons and/or backslashes, but filenames in Windows cannot contain such characters (they are interpreted otherwise). When Windows tries to access a file on a *nix machine, it mangles the filename appropriately so that it does not contain those characters.

Mixing C and C++

Name mangling is not desirable when linking C modules with libraries or object files compiled with a C++ compiler. To prevent the C++ compiler from mangling the name of a function, you can apply the extern "C" linkage specifier to the declaration of the function or put the functions inside a extern "C" block.

Take a look at the following example:

```
[gorlist@waterdeep mangling]$ cat linkage.cpp
void one(int);
void two(char,char);

int main()
{
        one(1);
        two('a','b');
        return 0;
}
[gorlist@waterdeep mangling]$ g++ -c linkage.cpp
[gorlist@waterdeep mangling]$ nm linkage.o
         U _Z3onei
         U _Z3twocc
         U __gxx_personality_v0
00000000 T main
```

The C++ functions were mangled as expected.
Now let's try putting those functions in an 'extern "C"' block.

```
[gorlist@waterdeep mangling]$ cat linkage2.cpp
extern "C"
{
        void one(int);
        void two(char,char);
}
int main()
{
        one(1);
        two('a','b');
        return 0;
}
[gorlist@waterdeep mangling]$ g++ -c linkage2.cpp
[gorlist@waterdeep mangling]$ nm linkage2.o
         U __gxx_personality_v0
00000000 T main
         U one
         U two
```

extern "C" prevented the functions from getting mangled.

That way, a C (or any other language) can call those functions.
If you want to invoke functions written in C++ from another language, you must declare have the extern "C" linkage specifier. That way you can put them in a DLL or shared library.
Otherwise, the DLL/shared library would contain the mangled names and the functions would not get linked.

The same thing happens when you call libc functions from C++.

```
[gorlist@waterdeep mangling]$ cat cfunc.cpp
#include <stdio.h>
int main()
{
        puts("name mangling paper by gorlist");
        return 0;
}
```

puts() must remain intact in order to get linked.

So in order to achieve this, we have something like:

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

int puts (__const char *__s);

#ifdef __cplusplus
}
#endif
```

__cplusplus is a macro thas is defined whenever C++ code is compiled.
This is needed because extern "C" does not exist in C.
So when called from C++, it does not mangle functions inside the extern "C" block (libc functions).
If called from C, it disregards what's inside the #ifdefs.

We can see that by instructing the compiler to stop right after the preprocessing stage. That is done with the -E option.

```
[gorlist@waterdeep mangling]$ cat cfunc.cpp
#include <stdio.h>
int main()
{
        puts("name mangling paper by gorlist");
        return 0;
}
[gorlist@waterdeep mangling]$ g++ -E cfunc.cpp | grep puts
extern int fputs (__const char *__restrict __s, FILE *__restrict __stream);
extern int puts (__const char *__s);
extern int fputs_unlocked (__const char *__restrict __s,
 puts("name mangling paper by gorlist");
```

Hm, but where's extern "C"?

```
[gorlist@waterdeep mangling]$ g++ -E cfunc.cpp | grep 'extern "C"'
extern "C" {
extern "C" {
```

It's a block, so it can affect more than one functions.

Mangling

The C++ standard does not standardize name mangling.
Quoting the g++ FAQ:

```
"Compilers differ as to how objects are laid out, how multiple inheritance is implemented, how
virtual function calls are handled, and so on, so if the name mangling were made the same, your
programs would link against libraries provided from other compilers but then crash when run. For
this reason, the ARM[1] encourages compiler writers to make their name mangling different from
that of other compilers for the same platform. Incompatible libraries are then detected at link
time, rather than at run time."
```

[1]: Annotated C++ Reference Manual

I. GCC

GCC has adopted the Intel IA64 name mangling scheme defined in the Intel IA64 standard ABI for its name mangling (in all platforms) from GCC 3.x and later.

From the g++ FAQ:

```
"GNU C++ does not do name mangling in the same way as other C++ compilers.
This means that object files compiled with one compiler cannot be used with
another."
```

Builtin types encoding

```
  <builtin-type> ::= v  # void
                 ::= w  # wchar_t
                 ::= b  # bool
                 ::= c  # char
                 ::= a  # signed char
                 ::= h  # unsigned char
                 ::= s  # short
                 ::= t  # unsigned short
                 ::= i  # int
                 ::= j  # unsigned int
                 ::= l  # long
                 ::= m  # unsigned long
                 ::= x  # long long, __int64
                 ::= y  # unsigned long long, __int64
                 ::= n  # __int128
                 ::= o  # unsigned __int128
                 ::= f  # float
                 ::= d  # double
                 ::= e  # long double, __float80
                 ::= g  # __float128
                 ::= z  # ellipsis
                 ::= u <source-name>    # vendor extended type
```

Operator encoding

```
  <operator-name> ::= nw # new
                  ::= na       # new[]
                  ::= dl       # delete
                  ::= da       # delete[]
                  ::= ps       # + (unary)
                  ::= ng       # - (unary)
                  ::= ad       # & (unary)
                  ::= de       # * (unary)
                  ::= co       # ~
                  ::= pl       # +
                  ::= mi       # -
                  ::= ml        # *
                  ::= dv        # /
                  ::= rm        # %
                  ::= an        # &
                  ::= or        # |
                  ::= eo        # ^
                  ::= aS        # =
                  ::= pL        # +=
                  ::= mI        # -=
                  ::= mL        # *=
                  ::= dV        # /=
                  ::= rM        # %=
                  ::= aN        # &=
                  ::= oR        # |=
                  ::= eO        # ^=
                  ::= ls        # <<
                  ::= rs        # >>
                  ::= lS        # <<=
                  ::= rS        # >>=
                  ::= eq        # ==
                  ::= ne        # !=
                  ::= lt        # <
                  ::= gt        # >
                  ::= le        # <=
                  ::= ge        # >=
                  ::= nt        # !
                  ::= aa        # &&
                  ::= oo        # ||
                  ::= pp        # ++
                  ::= mm        # --
                  ::= cm        # ,
                  ::= pm        # ->*
                  ::= pt        # ->
                  ::= cl        # ()
                  ::= ix        # []
                  ::= qu        # ?
                  ::= st        # sizeof (a type)
                  ::= sz        # sizeof (an expression)
                  ::= cv <type> # (cast)
                  ::= v <digit> <source-name>   # vendor extended operator
```

Types encoding

```
  <type> ::= <CV-qualifiers> <type>
         ::= P <type>   # pointer-to
         ::= R <type>   # reference-to
         ::= O <type>     # rvalue reference-to (C++0x)
         ::= C <type>   # complex pair (C 2000)
         ::= G <type>   # imaginary (C 2000)
         ::= U <source-name> <type>     # vendor extended type qualifier
```

Let's try with an example and observe the output.
We'll examine its symbols with the nm(1) utility, which is part of GNU binutils.
It lists symbols from object files.

```
[gorlist@waterdeep mangling]$ cat example.cpp
#include <iostream>

int testfunc(char*, int, double, int, char, int*, float)
{
return 1;
}

int main()
{
        int x;
        int *pointer;
        int a=5;
        pointer=&a;
        x=testfunc("hello",3,2.2,9,'c',pointer,1.23);
        return 0;
}
[gorlist@waterdeep mangling]$ g++ -c example.cpp
[gorlist@waterdeep mangling]$ cat example.cpp
#include <iostream>

int testfunc(char*, int, double, int, char, int*, float)
{
        return 1;
}

int main()
{
        int x;
        int *pointer;
        int a=5;
        pointer=&a;
        x=testfunc("hello",3,2.2,9,'c',pointer,1.23);
        return 0;
}
[gorlist@waterdeep mangling]$ g++ -c example.cpp
[gorlist@waterdeep mangling]$ nm example.o
000000d4 t _GLOBAL__I__Z8testfuncPcidicPif
0000008e t _Z41__static_initialization_and_destruction_0ii
00000000 T _Z8testfuncPcidicPif
         U _ZNSt8ios_base4InitC1Ev
         U _ZNSt8ios_base4InitD1Ev
00000000 b _ZSt8__ioinit
         U __cxa_atexit
         U __dso_handle
         U __gxx_personality_v0
000000ec t __tcf_0
00000020 T main
[gorlist@waterdeep mangling]$ nm -C example.o
000000d4 t global constructors keyed to _Z8testfuncPcidicPif
0000008e t __static_initialization_and_destruction_0(int, int)
00000000 T testfunc(char*, int, double, int, char, int*, float)
         U std::ios_base::Init::Init()
         U std::ios_base::Init::~Init()
00000000 b std::__ioinit
         U __cxa_atexit
         U __dso_handle
         U __gxx_personality_v0
000000ec t __tcf_0
00000020 T main
```

Take a look at "_Z8testfuncPcidicPif", which is our testfunc function.
It starts with _Z which is a reserved identifier in C/C++.

From ISO/IEC 9899:1999 (or C99 Standard):

```
"All identifiers that begin with an underscore and either an uppercase letter or another
underscore are always reserved for any use."
```

Then the number 8, which is the number of characters of the function (testfunc).

Finally, it encodes the parameters.
Pc -> pointer to a char or char*
i -> integer
d -> double
i -> integer
c -> char

Pi -> pointer to an int or int*
f -> float

II. MSVC

Windows and Microsoft products tend to be complex.
Microsoft's compiler, cl.exe even does some (minimal) mangling on C function names as you will see.
Microsoft tends to favor the term 'name decoration' to 'name mangling'.
I will be using Microsoft Visual Studio 2005 Professional, CL.EXE Version 14
cl.exe is the C/C++ compiler that Visual Studio uses.
Note that MSVC and MSVC++ are the same thing.

C:

The source is the following:

```
#include <stdio.h>

void print1 (void)
{
       puts("I'm print1 (cdecl)");
}

int printalt1 (int a, char b)
{
       puts("I'm print_alt (cdecl)");
       return 0;
}

void __stdcall print2 (void)
{
       puts("I'm print2 (stdcall)");
}

int __stdcall printalt2 (int a, char b)
{
       puts("I'm print_alt2 (stdcall)");
       return 0;
}

void __fastcall print3 (void)
{
       puts("I'm print3 (fastcall)");
}

int __fastcall printalt3 (int a, char b)
{
       puts("I'm print_alt3 (fastcall)");
       return 0;
}

int main(void)
{
       int x,y,z;
       print1();
       print2();
       print3();
       x=printalt1(5,'A');
       y=printalt2(5,'A');
       z=printalt3(5,'A');
       return 0;
}
```

After compiling, I need a tool to examine the symbols in the .obj file (equivalent of .o on *nix platforms).

MSVC++ doesn't put the symbol and debugging info in the executable itself, so don't try to open the .exe to see the symbols. It will appear as if it was stripped.
More on that lies in the "Demangling" section.

There is Microsoft's dumpbin (equivalent of GNU's nm utility) and its "/symbols" switch.

You could also use nm from Cygwin:

```
$ nm c_mangle.obj
00000000 T @print3@0
00000000 T @printalt3@8
00000000 T _main
00000000 T _print1
00000000 T _print2@0
00000000 T _printalt1
00000000 T _printalt2@8


void print1 (void)                          ->     _print1
void __stdcall print2 (void)                ->     _print2@0
void __fastcall print3 (void)                      ->     @print3@0

int printalt1 (int a, char b)               ->      _printalt1
int __stdcall printalt2 (int a, char b)            ->     _printalt2@8
int __fastcall printalt3 (int a, char b)    ->     @printalt3@8
```

cl.exe uses different mangling schemes to distinguish between the different calling convention styles.

Mangling schemes:
The cdecl mangling scheme just prepends an underscore ("_") to the function name.
stdcall has the form of _name@x and fastcall has the form of @name@x .
X is the size of the argument(s) passed on the stack, in bytes.

We don't need to specify "__cdecl" at print1() and printalt1() because cdecl is the default calling convention used in C/C++Windows Console Applications.
The default calling convention for Win32 Applications (WinAPI) is stdcall.

But why "8" and not "5", since in the x86 platform char and int take up 1 and 4 bytes respectively?

Stuff pushed to the stack must have the arch word's size.
In the x86 platform that means 4 bytes at a time.

So if you look at the assembly (.asm file) generated by cl.exe (using the /Fa switches), you will notice that the char (1 byte) gets padded in order to get pushed in the stack.

```
push   65                                   ; 00000041H
```

The character 'A' has the decimal value of 65 in ASCII.

So 4 bytes for the padded char plus 4 for the int results in 8.

C++:

The source is the same as in the GCC example.

A decorated name for a C++ function contains the following information:

* The function name.
* The class that the function is a member of, if it is a member function. This may include the class that encloses the function's class, and so on.
* The namespace the function belongs to (if it is part of a namespace).
* The types of the function's parameters.
* The calling convention.
* The return type of the function.

I will be using Microsoft's dumpbin (equivalent of GNU's nm utility) and the "/symbols" switch.
I fire up a cmd.exe prompt and type:

```
C:\Documents and Settings\gorlist>dumpbin /symbols "C:\Documents and Setti
ngs\gorlist\My Documents\Visual Studio 2005\Projects\mangling\mangling\Relea
se\mangle.obj" > output.txt
```

Next, I open up output.txt and search for "testfunc".
Note that the output will be huge compared to GCC's and considerably larger than the C file.

Excerpt:

```
092 00000000 SECT31 notype ()    External     | ?testfunc@@YAHPADHNHDPAHM@Z (int __cdecl
testfunc(char *,int,double,int,char,int *,float))
```

Looks like that is our testfunc function.

You could also use nm from Cygwin:

```
$ nm mangle.obj | grep testfunc
00000000 T ?testfunc@@YAHPADHNHDPAHM@Z
```

Or a listing file (viewing the assembly).

The name decoration procedure performed by cl.exe is described in the following steps:

1. a prefix of '?'
2. the function's name (ignoring any class). Operators, constructors, and destructors are represented by two-character codes; see below.
3. If the function is not an operator, the separator '@' terminates the function name.
4. If the name is qualified by a class name:
   a. For each qualifying class name from inner to outer, the encoded class name followed by the separator '@'.
   b. the separator '@'
   c. the value "Q"

   else

   a. the separator '@'
   b. the value "Y"
5. the encoded function calling convention
6. the encoded signature of the function's return value
7. the encoded signature of the function's arguments
8. the '@' separator

9. a suffix of 'Z'

Type encoding

```
signed char    C
char           D
unsigned char  E
short          F
unsigned short G
int            H
unsigned int   I
long           J
unsigned long  K
float          M
double         N
long double    O
pointer        PA
array          Q
struct/class   V
void           X
```

Operators

```
myclass::myclass           ?0
myclass::~myclass          ?1
myclass::operator new      ?2
myclass::operator delete   ?3
myclass::operator=         ?4
myclass::operator+         ?H
myclass::operator++        ?E
```

Calling Conventions

```
 __cdecl -> "A"
 __stdcall -> "G"
 __fastcall -> "I"
```

?testfunc@@YAHPADHNHDPAHM@Z
int testfunc(char*, int, double, int, char, int*, float)

Explanation:

?
testfunc
@ (not an operator)
@Y (function is not a member function, meaning that it's not part of a class)
A (cdecl calling convention)
H (function returns int)

Parameter list:
PAD (pointer to a double or char*)
H (int)
N (double)
H (int)
D (char)
PAH (int *)
M (float)

As you can see, it's much less straightforward than GCC's method.

Demangling

I. GCC

Name mangling generated by g++ can be demangled by using one of the following:
* c++filt
* nm -C
* __cxa_demangle()

I'll demonstrate their usage and output with a small program that takes two hardcoded values, adds them up and displays the sum on stdout.

```
[gorlist@waterdeep mangling]$ cat add.cpp
#include <iostream>

int add(int, int);
int main()
{
        int a,b;
        int x;
        a=5;
        b=3;
        x=add(a,b);
        std::cout<<"The sum is "<<x<<"\n";
        return 0;
}

int add(int a, int b)
{
        return a+b;
}

[gorlist@waterdeep mangling]$ g++ -c add.cpp
[gorlist@waterdeep mangling]$ nm add.o
00000052 t _GLOBAL__I_main
00000000 T _Z3addii
0000000c t _Z41__static_initialization_and_destruction_0ii
         U _ZNSolsEi
         U _ZNSt8ios_base4InitC1Ev
         U _ZNSt8ios_base4InitD1Ev
         U _ZSt4cout
00000000 b _ZSt8__ioinit
         U _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
         U __cxa_atexit
         U __dso_handle
         U __gxx_personality_v0
0000006a t __tcf_0
0000007e T main
```

c++filt

From the c++filt(1) manpage:

```
"The C++ and  Java languages provide function overloading, which means
that you can write many functions with the same name, providing that
each function takes parameters of different types. In order to be able
to distinguish these similarly named functions C++ and Java encode them
into  a low-level assembler name which uniquely identifies each different version.
```

```
This process is known as mangling. The c++filt [1] program does the inverse mapping:
it decodes (demangles) low-level names into user-level names so that they can be        read."
```

"_Z3addii" is our add function.

You can use c++filt on just a symbol

```
[gorlist@waterdeep mangling]$ c++filt _Z3addii
add(int, int)
```

or you can pipe nm's output to it and have it demangle all the symbols.

```
[gorlist@waterdeep mangling]$ nm add.o | c++filt
00000052 t global constructors keyed to main
00000000 T add(int, int)
0000000c t __static_initialization_and_destruction_0(int, int)
         U std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
         U std::ios_base::Init::Init()
         U std::ios_base::Init::~Init()
         U std::cout
00000000 b std::__ioinit
         U std::basic_ostream<char, std::char_traits<char> >& std::operator<<
<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
         U __cxa_atexit
         U __dso_handle
         U __gxx_personality_v0
0000006a t __tcf_0
0000007e T main
```

nm -C

nm(1) also has a -C option which demangles the symbol names.

```
 [gorlist@waterdeep mangling]$ nm -C add.o
00000052 t global constructors keyed to main
00000000 T add(int, int)
0000000c t __static_initialization_and_destruction_0(int, int)
U std::ostream::operator<<(int)
U std::ios_base::Init::Init()
U std::ios_base::Init::~Init()
U std::cout
00000000 b std::__ioinit
U std::basic_ostream<char, std::char_traits<char> >& std::operator<<
<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char
const*)
U __cxa_atexit
U __dso_handle
U __gxx_personality_v0
0000006a t __tcf_0
0000007e T main
```

As you can see, the output is very similar to c++filt's.

__cxa_demangle()

The function prototype is:

```
namespace abi {
  extern "C" char* __cxa_demangle (const char* mangled_name,
                                   char* buf,
                                   size_t* n,
                                   int* status);
}
```

The GCC documentation states that the function is part of the cross-vendor C++ ABI (my guess would be that they mean Intel's ICC).

When status has the value of 0, it means the demangling was successful.

So let's test it with a simple program:

```
[gorlist@waterdeep mangling]$ cat cxa.cpp
#include <iostream>
#include <cxxabi.h>

int main()
{
        int     status=0;
        char   *demangled;
        char *mangled="_Z3addii";

        demangled = abi::__cxa_demangle(mangled, 0, 0, &status);
        std::cout <<mangled<<" => "<< demangled<<"\n";
        std::cout <<status<<"\n";
        free(demangled);

        mangled="_ZNSt8ios_base4InitC1Ev";
        demangled = abi::__cxa_demangle(mangled, 0, 0, &status);
        std::cout <<mangled<<" => "<< demangled<<"\n";
        std::cout <<status<<"\n";
        free(demangled);

        return 0;
}
[gorlist@waterdeep mangling]$ g++ cxa.cpp -o cxa.out
[gorlist@waterdeep mangling]$ ./cxa.out
_Z3addii => add(int, int)
0
_ZNSt8ios_base4InitC1Ev => std::ios_base::Init::Init()
0
```

It is written in C (hence the .h header), so you don't have to write C++ in order to demangle C+
+.

II. MSVC

Like in *nix, there are lots of ways to undecorate the symbols that cl.exe produces.

* dumpbin
* undname
* IDA
* nm (Cygwin)
* UnDecorateSymbolName (Win32 API)

dumpbin

dumpbin is also capable of undecorating the symbols (equivalent of nm -C) of C++ files.
It does not undecorate the (minimal) decoration of C files though.
The undecorated symbols can be seen between the set of parentheses.
There is also a GUI for dumpbin called dumpbinGUI.

undname

Microsoft's undname comes with MSVS and can process whole files (such as the output.txt) or single arguments.

```
C:\Documents and Settings\gorlist>undname.exe ?add@@YAHHH@Z
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation. All rights reserved.

Undecoration of :- "?add@@YAHHH@Z"
is :- "int __cdecl add(int,int)"
```

IDA

MSVS doesn't put the symbol and debugging info in the executable itself. That's why .pdb files exist.
PDB stands for Program Database. It stores a list of all symbols in a module with their addresses and possibly the name of the file and the line on which the symbol was declared.

Note: In IDA Evaluation version you cannot open .obj files. Just .exe files.
IDA is one of the programs that can load the .pdb file along with the executable and load the symbols for it.

MSVS generates no debugging files by default.
To generate the .pdb file, go to Project Properties->Configuration Properties->Linker->Generate Debug Info and select "Yes (/DEBUG)" from the listbox.

Compile and then drag and drop the executable (.exe) in IDA. If it finds a .pdb it will ask you if you want it loaded.
With the .pdb file loaded, all symbols are displayed in the Names window.
IDA has the capability of displaying demangled names.
The default is to display them as comments, but you can change that in Options->Demangled Names.

nm

By default, Cygwin's nm cannot demangle MSVC++ symbols.
However, there is a patch available in the Cygwin mailing list which enables a "--demangle=msvc" option in nm.
It's No.12 in the references.

Win32 API

The Win32 API offers a UnDecorateSymbolName function.

Code example:

```
#include <iostream>
#include <windows.h>
#include <Dbghelp.h>
// Uncomment the following line if you don't want to link manually
// #pragma comment(lib, "dbghelp.lib")
int main()
{
        DWORD  error;

/*
        HANDLE hProcess;
        DWORD  processId;
        SymSetOptions(SYMOPT_UNDNAME | SYMOPT_DEFERRED_LOADS);
        hProcess = GetCurrentProcess();

        if (SymInitialize(hProcess, NULL, TRUE))
        {
                puts("SymInitialize() success!");
        }
        else
        {
                error = GetLastError();
                printf("SymInitialize returned error : %d\n", error);
        }
*/

        char* mangled="?add@@YAHHH@Z";
        char demangled[100];
        DWORD undname_complete=UNDNAME_COMPLETE;

        if (UnDecorateSymbolName(mangled, demangled, sizeof(demangled), undname_complete))
        {
                // UnDecorateSymbolName returned success
                printf ("Mangled: %s\n", mangled);
                printf ("Demangled: %s\n", demangled);
        }
        else
        {
                // UnDecorateSymbolName failed
                error = GetLastError();
                printf("UnDecorateSymbolName() returned error %d\n", error);
        }

        return 0;
}
```

You will have to link manually with dgbhelp.lib .
Either use the #pragma preprocessor directive, or go to Project Properties->Configuration Properties->Linker->Input->Additional Dependencies and type dbghelp.lib .

MSDN suggests that you initialize the Symbol Handler before you use UnDecorateSymbolName, but on my machine it works without it as well (the code for initialization of the symbol handler is commented out).

Example output:

```
Mangled: ?add@@YAHHH@Z
Demangled: int __cdecl add(int,int)
```

Taking advantage of name mangling

Case I:
Demangling identifiers could reveal the function's prototype.
This could be used to discover special undocumented API functions that a company would want to keep as a secret.

This function is supposed to be used only by a partner of the company.
For example it could be a GPU-selling company having a contract with another company that sells

a GPU benchmark.
This function could be one that is that gets a higher score in the specific benchmark than the usual way of doing it (the documented functions).
This could be a dirty hack just to achieve better results (often cheat) in several programs and virtually beat the competition.
Such undocumented functions are used in the real world.
That's how hackers discover them and expose them to the public.

Case II:
You could replace a function in a C++ .o file (relocatable file) with a your custom C function.
The logic behind this is very simple.
As I mentioned earlier in this paper, C identifiers do not get mangled.
So, if you had a C function named exactly as the mangled C++ identifier in the .o file, you could link against it.
An example should clear it up:

```
[gorlist@waterdeep mangling]$ cat mangled.cpp
#include <iostream>

int x();

int main()
{
        int r;
        r=x();
        std::cout<<r<<"\n";
        return 0;
}

int x()
{
        return 5;
}
```

So this is our original cpp source.
Let's run it:

```
[gorlist@waterdeep mangling]$ g++ mangled.cpp -o mangled.out
[gorlist@waterdeep mangling]$ ./mangled.out
5
```

Good, so now let's produce the relocatable file out of it.

```
[gorlist@waterdeep mangling]$ g++ -c mangled.cpp
[gorlist@waterdeep mangling]$ nm mangled.o
00000050 t _GLOBAL__I_main
00000000 T _Z1xv
0000000a t _Z41__static_initialization_and_destruction_0ii
         U _ZNSolsEPFRSoS_E
         U _ZNSolsEi
         U _ZNSt8ios_base4InitC1Ev
         U _ZNSt8ios_base4InitD1Ev
         U _ZSt4cout
         U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
00000000 b _ZSt8__ioinit
         U __cxa_atexit
         U __dso_handle
         U __gxx_personality_v0
00000068 t __tcf_0
0000007c T main
```

First column is the symbol's value, second is the symbol's type and third is the symbol name.

Look for "_Z1xv" at the second line. That's the mangled identifier for our x function.

We'll use nm's -C option to demangle the symbols.

```
[gorlist@waterdeep mangling]$ nm -C mangled.o
00000050 t global constructors keyed to main
00000000 T x()
0000000a t __static_initialization_and_destruction_0(int, int)
         U std::ostream::operator<<(std::ostream& (*)(std::ostream&))
         U std::ostream::operator<<(int)
         U std::ios_base::Init::Init()
         U std::ios_base::Init::~Init()
         U std::cout
         U std::basic_ostream<char, std::char_traits<char> >& std::endl<char,
std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
00000000 b std::__ioinit
         U __cxa_atexit
         U __dso_handle
         U __gxx_personality_v0
00000068 t __tcf_0
0000007c T main
```

It should make more sense now.

I'll make a new C source file, named custom_func.c .

```
[gorlist@waterdeep mangling]$ cat custom_func.c
int _Z1xv()
{
        return 1;
}
[gorlist@waterdeep mangling]$ gcc -c custom_func.c
[gorlist@waterdeep mangling]$ nm custom_func.o
00000000 T _Z1xv
```

I'll produce an executable file out of those two relocatable files.

```
[gorlist@waterdeep mangling]$ g++ custom_func.o mangled.o -o final.out
mangled.o: In function `x()':
mangled.cpp:(.text+0x0): multiple definition of `x()'
custom_func.o:custom_func.c:(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

The linker is complaining that there are two functions with the same name.
ld(1) has a –allow-multiple-definition option which allows that.

```
[gorlist@waterdeep mangling]$ g++ -Wl,--allow-multiple-definition custom_func.o mangled.o -o
final.out
[gorlist@waterdeep mangling]$ ./final.out
1
```

As you can see our custom function in custom_func.c replaced the original one in mangled.cpp.

Case III:
In the previous case I demonstrated how you could replace a function in a relocatable file.
In this one, I will show you how to replace a C++ operator (function would be fine too) in the final executable file with your own custom function.
This can be achieved through the use of the LD_PRELOAD environment variable.
From the Linux ld.so(8) manpage:

```
A whitespace-separated list of additional,  user-specified,  ELF
shared  libraries  to  be loaded before all others.  This can be
```

```
       used  to  selectively  override  functions   in   other   shared
       libraries.
```

Here's our C++ source:

```
[gorlist@waterdeep mangling]$ cat original.cpp
int main()
{
        int* a = new int;
}
```

Like in the previous example:

```
[gorlist@waterdeep mangling]$ g++ original.cpp -o test.out
[gorlist@waterdeep mangling]$ ./test.out
```

Just like we expected, it does nothing.

```
[gorlist@waterdeep mangling]$ g++ -c original.cpp
[gorlist@waterdeep mangling]$ nm original.o
         U _Znwj
         U __gxx_personality_v0
00000000 T main
```

This is our custom C function.

```
[gorlist@waterdeep mangling]$ cat inject.c
#include <stdio.h>

void _Znwj()
{
        puts("Success!");
}
```

Now I'll turn this C source into a DSO (dynamically shared object):

```
[gorlist@waterdeep mangling]$ gcc inject.c -shared -fPIC -o inject.so
[gorlist@waterdeep mangling]$ file inject.so
inject.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not stripped
```

Ok, now we'll set the LD_PRELOAD environment variable and execute the executable (test.out) we built earlier from original.cpp.

```
[gorlist@waterdeep mangling]$ LD_PRELOAD=./inject.so ./test.out
Success!
```

Source Code

Name mangling is handled by the gcc/cp/mangle.c source file.
You can see it online via the CVS (I have a link in the "Further Reading" section) or you can always download the full source code.

Excerpt from the GCC 4.2 mangle.c source:

```
/* TOP_LEVEL is true, if this is being called at outermost level of
  mangling. It should be false when mangling a decl appearing in an
  expression within some other mangling.

  <mangled-name>      ::= _Z <encoding>  */

static void
write_mangled_name (const tree decl, bool top_level)
```

```
{
  MANGLE_TRACE_TREE ("mangled-name", decl);

  if (/* The names of `extern "C"' functions are not mangled.  */
      DECL_EXTERN_C_FUNCTION_P (decl)
      /* But overloaded operator names *are* mangled.  */
      && !DECL_OVERLOADED_OPERATOR_P (decl))
    {
    unmangled_name:;

      if (top_level)
       write_string (IDENTIFIER_POINTER (DECL_NAME (decl)));
      else
       {
         /* The standard notes: "The <encoding> of an extern "C"
            function is treated like global-scope data, i.e. as its
            <source-name> without a type."  We cannot write
            overloaded operators that way though, because it contains
            characters invalid in assembler.  */
         if (abi_version_at_least (2))
           write_string ("_Z");
         else
           G.need_abi_warning = true;
         write_source_name (DECL_NAME (decl));
       }
    }
  else if (TREE_CODE (decl) == VAR_DECL
          /* The names of global variables aren't mangled.  */
          && (CP_DECL_CONTEXT (decl) == global_namespace
              /* And neither are `extern "C"' variables.  */
              || DECL_EXTERN_C_P (decl)))
    {
      if (top_level || abi_version_at_least (2))
       goto unmangled_name;
      else
       {
         G.need_abi_warning = true;
         goto mangled_name;
       }
    }
  else
    {
    mangled_name:;
      write_string ("_Z");
      write_encoding (decl);
      if (DECL_LANG_SPECIFIC (decl)
         && (DECL_MAYBE_IN_CHARGE_DESTRUCTOR_P (decl)
             || DECL_MAYBE_IN_CHARGE_CONSTRUCTOR_P (decl)))
       /* We need a distinct mangled name for these entities, but
          we should never actually output it.  So, we append some
          characters the assembler won't like.  */
       write_string (" *INTERNAL* ");
    }
}
```

References & Further Reading

1. http://www.codesourcery.com/cxx-abi/abi.html#demangler
2. http://gcc.gnu.org/viewcvs/branches/gcc-4_2-branch/gcc/cp/mangle.c?revision=127961&view=markup
3. http://www.codesourcery.com/cxx-abi/abi-examples.html#mangling
4. http://www.kegel.com/mangle.html
5. http://en.wikipedia.org/wiki/Microsoft_Visual_C++_Name_Mangling
6. http://theory.uwinnipeg.ca/gnu/gcc/gxxint_15.html
7. http://msdn2.microsoft.com/en-us/library/ms681400.aspx

8. http://msdn2.microsoft.com/en-us/library/ms680585.aspx
9. http://en.wikipedia.org/wiki/X86_calling_conventions
10. http://msdn2.microsoft.com/en-us/library/ms680585.aspx
11. http://www.agner.org/optimize/calling_conventions.pdf
12. http://sourceware.org/ml/binutils/2004-06/msg00413.html
13. http://root.cern.ch/root/roottalk/roottalk00/1249.html
14. http://en.wikipedia.org/wiki/WinDbg
15. http://en.wikipedia.org/wiki/Program_database
16. http://www.microsoft.com/whdc/devtools/debugging/default.mspx
17. http://msdn2.microsoft.com/en-us/library/t2k2877b(VS.80).aspx
18. http://msdn2.microsoft.com/en-us/library/367y26c6(VS.80).aspx
19. http://en.wikipedia.org/wiki/Microsoft_Visual_C++_Name_Mangling
20. http://www.datarescue.com/idabase/idadoc/611.htm
21. http://www.cheztabor.com/dumpbinGUI/index.htm
22. http://msdn2.microsoft.com/en-us/library/c1h23y6c(VS.80).aspx
23. http://support.microsoft.com/kb/177429
24. The manpages of the programs used