grotlee@gmail.com

In C++ source, what is the effect of extern "C"?

stackoverflow.com Updated Oct 28th, 2017

What exactly does putting `extern "C"` into C++ code do?

1041 For example:

```
extern "C" {
   void foo();
}
```

411

c++ c linkage name-mangling extern-c

share improve this question

edited May 30 '16 at 5:31
Jonathan Leffler
500k 76 579 919

asked Jun 25 '09 at 2:10
Litherum
5,977 3 14 2

4 I'd like to introduce you this article: http://www.agner.org/optimize/calling_conventions.pdf It tells you mu
8 more about calling convention and the difference between compilers. – Sam Liao Jun 25 '09 at 2:18

@Litherum On the top of my head, it is telling the compiler to compile that scope of code using C, giver you have a cross-compiler. Also, it means that you have a Cpp file where you have that `foo()` functic ha9u63ar Jun 27 '13 at 8:18

add a comment

12 Answers

active oldest

1056 extern "C" makes a function-name in C++ have 'C' linkage (compiler does not mangle the na that client C code can link to (i.e use) your function using a 'C' compatible header file that coı just the declaration of your function. Your function definition is contained in a binary format (tl was compiled by your C++ compiler) that the client 'C' linker will then link to using the 'C' nan

Since C++ has overloading of function names and C does not, the C++ compiler cannot just the function name as a unique id to link to, so it mangles the name by adding information abc arguments. A C compiler does not need to mangle the name since you can not overload func names in C. When you state that a function has extern "C" linkage in C++, the C++ compiler not add argument/parameter type information to the name used for linkage.

Just so you know, you can specify "C" linkage to each individual declaration/definition explici use a block to group a sequence of declarations/definitions to have a certain linkage:

```
extern "C" void foo(int);
extern "C"
{
   void g(char);
   int i;
}
```

If you care about the technicalities, they are listed in section 7.5 of the C++03 standard, here brief summary (with emphasis on extern "C"):

- extern "C" is a linkage-specification
- Every compiler is *required* to provide "C" linkage
- a linkage specification shall occur only in namespace scope
- ~~all function types, function names and variable names have a language linkage~~ **See Ric Comment:** Only function names and variable names with external linkage have a langu linkage
- two function types with distinct language linkages are distinct types even if otherwise ide
- linkage specs nest, inner one determines the final linkage
- extern "C" is ignored for class members
- at most one function with a particular name can have "C" linkage (regardless of namesp
- ~~extern "C" forces a function to have external linkage (cannot make it static)~~ **See Richarc comment:** 'static' inside 'extern "C"' is valid; an entity so declared has internal linkage, a does not have a language linkage
- Linkage from C++ to objects defined in other languages and to objects defined in C++ fr other languages is implementation-defined and language-dependent. Only where the ob layout strategies of two language implementations are similar enough can such linkage achieved

share improve this answer

edited May 23 at 12:02
Community ♦
1 1

answered Jun 25 '09 at 2:1
Faisal Vali
18.5k 6 34 4

1 C compiler does not use mangling which c++'s does. So if you want call a c interface from a c++ progra
4 have to clearly declared that the c interface as "extern c". – Sam Liao Jun 25 '09 at 2:28

4 @Faisal: do not try to link code built with different C++ compilers, even if the cross-references are all 'e
6 "C"'. There are often differences between the layouts of classes, or the mechanisms used to handle exceptions, or the mechanisms used to ensure variables are initialized before use, or other such differe plus you might need two separate C++ run-time support libraries (one for each compiler). – Jonathan L Jun 25 '09 at 3:24

5 @Leffler - thanks, you make good points. I did not mean to encourage using different C++ compilers by extern "C". Rather, I was hoping to suggest that if you are not writing something that would need to be l to by another C++ compiler, you probably don't need extern "C". – Faisal Vali Jun 25 '09 at 3:57

6 'extern "C" forces a function to have external linkage (cannot make it static)' is incorrect. 'static' inside 'e "C"' is valid; an entity so declared has internal linkage, and so does not have a language linkage. – Richard Smith Feb 14 '13 at 4:06

1 'all function types, function names and variable names have a language linkage' is also incorrect. Only
1 function names and variable names with external linkage have a language linkage. – Richard Smith Fe '13 at 4:07

show **12** more comments

Just wanted to add a bit of info, since I haven't seen it posted yet.

197 You'll very often see code in C headers like so:

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

// all of your legacy C code here

#ifdef __cplusplus
}
#endif
```

What this accomplishes is that it allows you to use that C header file with your C++ code, bec
the macro "__cplusplus" will be defined. But you can *also* still use it with your legacy C code,
the macro is *NOT* defined, so it won't see the uniquely C++ construct.

Although, I have also seen C++ code such as:

```
extern "C" {
#include "legacy_C_header.h"
}
```

which I imagine accomplishes much the same thing.

Not sure which way is better, but I have seen both.

share improve this answer

answered Oct 21 '12 at 1:0

UncaAlby
2,277 1 9 10

9 There is a distinct difference. In case of the former, if you compile this file with normal gcc compiler it w
generate an object where the function name is not mangled. If you then link C and C++ objects with the
it will NOT find the functions. You will need to include those "legacy header" files with the extern keywo
your second code block. – Anne van Rossum Apr 12 '13 at 14:00

@Anne: The C++ compiler will look for unmangled names also, because it saw `extern "C"` in the he
It works great, used this technique many times. – Ben Voigt Jun 27 '14 at 5:34

8 @Anne: That's not right, the first one is fine as well. It's ignored by the C compiler, and has the same e
the second in C++. The compiler couldn't care less whether it encounters `extern "C"` before or after
includes the header. By the time it reaches the compiler, it's just one long stream of preprocessed text
anyway. – Ben Voigt Jun 30 '14 at 15:54

3 @Anne, no, I think you've been affected by some other error in the source, because what you are desc
is wrong. No version of `g++` got this wrong, for any target, at any time in the last 17 years at least. The
point of the first example is that it doesn't matter whether you use a C or C++ compiler, no name mangl
be done for the names in the `extern "C"` block. – Jonathan Wakely Jan 19 '16 at 20:45

1 "which one is better" - for sure, the first variant is better: It allows including the header directly, whithout
further requirements, both in C and C++ code. The second approach is a workaround for C headers the
author forgot the C++ guards (no problem, though, if these are added afterwards, nested extern "C"
declarations are accepteded...). – Aconcagua Aug 9 at 9:23

show **3** more comments

147

In every C++ program, all non-static functions are represented in the binary file as symbols. symbols are special text strings that uniquely identify a function in the program.

In C, the symbol name is the same as the function name. This is possible because in C no tw static functions can have the same name.

Because C++ allows overloading and has many features that C does not — like classes, me functions, exception specifications - it is not possible to simply use the function name as the name. To solve that, C++ uses so-called name mangling, which transforms the function nam all the necessary information (like the number and size of the arguments) into some weird-lo string which only the compiler knows about.

So if you specify a function to be extern C, the compiler doesn't performs name mangling wit and it can be directly accessed using its symbol name.

This comes handy while using `dlsym()` and `dlopen()` for calling such functions.

share improve this answer

answered Jun 25 '09 at 5:2

sud03r
9,853 14 60

add a comment

84

Let's **decompile the object file g++ generated** to see what goes on inside this implementat

Generate example

Input:

```
void f() {}
void g();

extern "C" {
    void ef() {}
    void eg();
}

/* Prevent g and eg from being optimized away. */
void h() { g(); eg(); }
```

Compile with GCC 4.8 Linux ELF output:

```
g++ -c a.cpp
```

Decompile the symbol table:

```
readelf -s a.o
```

The output contains:

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
  8: 0000000000000000     6 FUNC    GLOBAL DEFAULT    1 _Z1fv
  9: 0000000000000006     6 FUNC    GLOBAL DEFAULT    1 ef
 10: 000000000000000c    16 FUNC    GLOBAL DEFAULT    1 _Z1hv
 11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z1gv
 12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND eg
```

Interpretation

We see that:

- `ef` and `eg` were stored in symbols with the same name as in the code
- the other symbols were mangled. Let's unmangle them:

  ```
  $ c++filt _Z1fv
  f()
  $ c++filt _Z1hv
  h()
  $ c++filt _Z1gv
  g()
  ```

Conclusion: both of the following symbol types were *not* mangled:

- defined
- declared but undefined (`Ndx = UND`), to be provided at link or run time from another obje

So you will need `extern "C"` both when calling:

- C from C++: tell `g++` to expect unmangled symbols produced by `gcc`
- C++ from C: tell `g++` to generate unmangled symbols for `gcc` to use

Things that do not work in extern C

It becomes obvious that any C++ feature that requires name mangling will not wok inside `ex` `C` :

```
extern "C" {
    // Overloading.
    // error: declaration of C function ‘void f(int)’ conflicts with
    void f();
    void f(int i);

    // Templates.
    // error: template with C linkage
    template <class C> void f(C i) { }
}
```

share improve this answer

edited Aug 5 '16 at 9:07

answered May 29 '15 at 10

Ciro Santilli 刘晓波
四事件 法轮功
90.4k 16 362

add a comment

Not any C-header will compile with extern "C". When identifiers in a C-header conflict with C

20 keywords the C++ compiler will complain about this.

For example, I have seen the following code fail in a g++ :

```
extern "C" {
struct method {
    int virtual;
};
}
```

Kinda makes sense, but is something to keep in mind when porting C-code to C++.

share improve this answer

edited Jan 26 '15 at 22:22
Alexey Shmalko
2,841 1 8 29

answered Jan 9 '13 at 22:1
Sander Mertens
365 3 9

9 `extern "C"` means to use C linkage, as described by other answers. It doesn't mean to "compile the contents as C" or anything. `int virtual;` is invalid in C++ and specifying different linkage doesn't cl that. – M.M Jan 26 '15 at 22:26

add a comment

19 It changes the linkage of a function in such a way that the function is callable from C. In prac that means that the function name is not mangled.

share improve this answer

answered Jun 25 '09 at 2:1
Employed Russiar
104k 18 141

mangled or decorated what is the proper term? – ojblass Jun 25 '09 at 2:15

2 Mangled is the term generally used... Don't believe I've ever seen 'decorated' used with this meaning. – Matthew Scharley Jun 25 '09 at 2:17

add a comment

15 It informs the C++ compiler to look up the names of those functions in a C-style when linking because the names of functions compiled in C and C++ are different during the linking stage

share improve this answer

answered Jun 25 '09 at 2:1
Mark Rushakoff
158k 25 334

add a comment

11 extern "C" is meant to be recognized by a C++ compiler and to notify the compiler that the nc function is (or to be) compiled in C style. So that while linking, it link to the correct version of function from C.

share improve this answer

answered Apr 10 '12 at 9:4

Flami
423 5 7

add a comment

9

Most programming languages aren't built on-top of existing programming languages. C++ is on-top of C, and for that reason there are C++ keywords like `extern` which provide backwar compatibility with C.

Let's look at the following example:

```
#include <stdio.h>

// Two functions are defined with the same name
// but have different parameters

void printMe(int a) {
  printf("int: %i\n", a);
}

void printMe(char a) {
  printf("char: %c\n", a);
}

int main() {
  printMe("a");
  printMe(1);
  return 0;
}
```

A C compiler will throw an error when running this example, because the same function `pri` defined twice (even though they have different parameters `int a` vs `char a`).

> gcc -o printMe printMe.c && ./printMe;
> *1 error. PrintMe is defined more than once.*

However, a C++ compiler won't throw an error if the function name `printMe` is defined twice long as the parameters are different).

> g++ -o printMe printMe.c && ./printMe;

This is because a C++ compiler implicitly renames (mangles) functions based on their param In C, this feature was not supported. However, when C++ was built over C, the language wa designed to be object-oriented, and needed to support the ability to create different classes v methods (functions) of the same name, and to override methods (method overriding) based different parameters.

However, mangling C function names with a C++ compiler can cause errors in the linking ph that follows compilation. The linker is supposed to match function references/calls to functior names. But, if the C function references/calls in external files are not mangled as well, then t can't be linked to a mangled function name.

And therefore, the `extern` keyword tells the C++ compiler - "Let's avoid this whole linker me is not C++, so don't rename C function names".

share improve this answer edited Jul 8 at 23:11 answered Feb 12 at 1:50

tfmontague
4,621 1 22 2

add a comment

5

I used 'extern "C"' before for dll(dynamic link library) files to make etc. main() function "expor so it can be used later in another executable from dll. Maybe an example of where I used to can be useful.

DLL

```
#include <string.h>
#include <windows.h>

using namespace std;

#define DLL extern "C" __declspec(dllexport)
//I defined DLL for dllexport function
DLL main ()
{
    MessageBox(NULL,"Hi from DLL","DLL",MB_OK);
}
```

EXE

```
#include <string.h>
#include <windows.h>

using namespace std;

typedef LPVOID (WINAPI*Function)();//make a placeholder for function from dll
Function mainDLLFunc;//make a variable for function placeholder

int main()
{
    char winDir[MAX_PATH];//will hold path of above dll
    GetCurrentDirectory(sizeof(winDir),winDir);//dll is in same dir as exe
    strcat(winDir,"\\exmple.dll");//concentrate dll name with path
    HINSTANCE DLL = LoadLibrary(winDir);//load example dll
    if(DLL==NULL)
    {
        FreeLibrary((HMODULE)DLL);//if load fails exit
        return 0;
    }
    mainDLLFunc=(Function)GetProcAddress((HMODULE)DLL, "main");
    //defined variable is used to assign a function from dll
    //GetProcAddress is used to locate function with pre defined extern name "DLL"
    //and matcing function name
    if(mainDLLFunc==NULL)
    {
        FreeLibrary((HMODULE)DLL);//if it fails exit
        return 0;
    }
    mainDLLFunc();//run exported function
    FreeLibrary((HMODULE)DLL);
}
```

```
}
```

share improve this answer

answered Jul 1 '14 at 15:43

SturmCoder
648 6 13

1 Bogus. `extern "C"` and `__declspec(dllexport)` are unrelated. The former controls symbol deco the latter is responsible for creating an export entry. You can export a symbol using C++ name decorati as well. Besides completely missing the point of this question, there are other mistakes in the code san well. For one, `main` exported from your DLL doesn't declare a return value. Or calling convention, for matter. When importing, you attribute a random calling convention (`WINAPI`), and use the wrong symb 32-bit builds (should be `_main` or `_main@0`). Sorry, -1. – IInspectable Sep 7 '16 at 8:28

Actualy that is just an working example where I used it before to export function, and it should be used get idea how it can be used. I didn't said it is necessary to use it this way. Also, DLL main () declared in of type DLL which is -> (extern "C" __declspec(dllexport)) type not int or string, defined above and it do return anything like void, I named it "main", but it can be named whatever and it should be not confuse "main" of type int in executable itself. However this compiles and works great with many other functions with parameters and return also. – SturmCoder May 25 at 9:17

That only repeated, that you don't know, what you are doing, but doing it this way appears to work for y some undisclosed list of target platforms. You didn't address the issues I raised in my previous comme is still a down-vote, due to being wildly wrong (there's more, that didn't fit in a single comment). – IInspe May 25 at 9:26

I didn't said anywhere that I'm know what I am doing,I just said that I used extern "C" before in this way is works for me on windows only cause I tried it on that platform only,you can see "#include <windows.h>".WINAPI is just a macro that evaluates to stdcall to prevent corrupting the stack where ca and callee need to agree on a calling convention i learned that here too and it is used in placeholder fo exported function "typedef LPVOID (WINAPI*Function)();" to prevent stack corruption on run. Maybe yo right but I actualy don't care at all about your down-vote,thrust me ;) – SturmCoder May 25 at 10:28

Posting an answer on Stack Overflow kind of implies, that you know what you are doing. This is expect for your attempt *"to prevent stack corruption on run"*: Your function signature specifies a return value of `void*` , but your implementation doesn't return anything. That'll fly really well... – IInspectable May 25 10:40

show **4** more comments

3 `extern "C"` is a linkage specification which is used to **call C functions** in the **Cpp source f** We can **call C functions, write Variables, & include headers**. Function is declared in exte entity & it is defined outside. Syntax is

Type 1:

```
extern "language" function-prototype
```

Type 2:

```
extern "language"
{
     function-prototype
};
```

eg:

```
#include<iostream>
```

```
using namespace std;

extern "C"
{
     #include<stdio.h>     // Include C Header
     int n;                // Declare a Variable
     void func(int,int);   // Declare a function (function prototype)
}

int main()
{
    func(int a, int b);   // Calling function . . .
    return 0;
}

// Function definition . . .
void func(int m, int n)
{
    //
    //
}
```

share improve this answer

answered Nov 17 '15 at 12:
Yogeesh H T
1,115 10 11

add a comment

1

When mixing C and C++ (i.e., a. calling C function from C++; and b. calling C++ function fror the C++ name mangling causes linking problems. Technically speaking, this issue happens c when the callee functions have been already compiled into binary (most likely, a *.a library fil using the corresponding compiler.

So we need to use extern "C" to disable the name mangling in C++.

share improve this answer

edited Aug 7 at 7:54
Bhargav Rao ♦
26.5k 16 71 102

answered Jul 6 at 4:04
Trombe
81 8

add a comment

protected by 2501 Mar 9 at 7:41

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had removed, posting an answer now requires 10 reputation on this site (the association bonus does not count

Would you like to answer one of these unanswered questions instead?

Not the answer you're looking for? Browse other questions tagged c++ c linkage name-mangli extern-c or ask your own question.