Converting a pointer into an integer

Asked 16 years, 2 months ago Modified 2 years, 10 months ago Viewed 237k times

121

I am trying to adapt an existing code to a 64 bit machine. The main problem is that in one function, the previous coder uses a void* argument that is converted into suitable type in the function itself. A short example:

```
void function(MESSAGE_ID id, void* param)
{
    if(id == FOO) {
        int real_param = (int)param;
        // ...
    }
}
```

Of course, on a 64 bit machine, I get the error:

```
error: cast from 'void*' to 'int' loses precision
```

I would like to correct this so that it still works on a 32 bit machine and as cleanly as possible. Any idea ?

c++ gcc casting 64-bit 32-bit

Share
Improve this question
Follow

edited Sep 30, 2008 at 15:09

asked Sep 30, 2008 at 13:43

PierreBdR
43.2k ● 10 ● 48 ● 66

5 I know this is digging up an old post, but it seems like the accepted answer is not quite correct. A concrete example of `size_t` not working is i386 segmented memory. Though a 32-bit machine, `sizeof` returns `2` for `size_t`. Alex answer below appears correct. Alex's answer and `uintptr_t` works just about everywhere and its now standard. It provides a C++11 treatment, and it even gives the C++03 header guards. – jww Sep 27, 2016 at 17:00

11 Answers

Sorted by: Highest score (default)

I'd say this is the modern C++ way:

```
#include <cstdint>
void *p;
auto i = reinterpret_cast<std::uintptr_t>(p);
```

EDIT:

The correct type to the the Integer

So the right way to store a pointer as an integer is to use the `uintptr_t` or `intptr_t` types. (See also in cppreference integer types for C99).

These types are defined in `<stdint.h>` for C99 and in the namespace `std` for C++11 in `<cstdint>` (see integer types for C++).

C++11 (and onwards) Version

```
#include <cstdint>
std::uintptr_t i;
```

C++03 Version

```
extern "C" {
#include <stdint.h>
}

uintptr_t i;
```

C99 Version

```
#include <stdint.h>
uintptr_t i;
```

The correct casting operator

In C there is only one cast and using the C cast in C++ is frowned upon (so don't use it in C++). In C++ there are different types of casts, but `reinterpret_cast` is the correct cast for this conversion (see also here).

C++11 Version

```
auto i = reinterpret_cast<std::uintptr_t>(p);
```

C++03 Version

```
uintptr_t i = reinterpret_cast<uintptr_t>(p);
```

C Version

```
uintptr_t i = (uintptr_t)p; // C Version
```

Related Questions

- What is uintptr_t data type

Share

Improve this answer

Follow

edited Feb 17, 2022 at 17:58

Olivia Stork

4,809 ● 5 ● 29 ● 40

answered Oct 27, 2014 at 11:07

Alexander Oh

25.5k ● 15 ● 78 ● 79

11 the only answer which properly mentions reinterpret_cast – plasmacel Jul 20, 2015 at 10:07

If you meant to include <cstdint>, you probably also want to use std::uintptr_t instead. – linleno Nov 20, 2015 at 19:42

1 @jww read: en.cppreference.com/w/cpp/language/static_cast my understanding here is that `static_cast` might convert the type or if it's a pointer might do pointer adjustments if the type needs it. `reinterpret_cast` is really just changing the type of the underlying memory pattern (no mutations). to clarify: `static_cast` does behave identical here. – Alexander Oh Nov 10, 2016 at 15:15

4 this should be marked as selected answer instead, as it provides all the details how to cast in **C and C++**. – HeavenHM Apr 26, 2018 at 14:45

1 @Pryftan It's fine to have a personal preference, but it's quite strange to "cringe" at something that the C++ language is moving towards. Modern C++ (11 onward) has a clear preference for type-aligned qualifiers. For example, you simply cannot align `*`, `&`, or `&&` with the name of a variadic packs since it comes before the `...`. Plus, of course, everything is increasingly left-ot-right with `auto`: `int *get_ptr()` becomes `auto get_ptr() -> int*`, `int *p = &q` becomes `auto p = &q`, etc. – Bitwize Jun 28, 2021 at 14:56

74

Use `intptr_t` and `uintptr_t`.

To ensure it is defined in a portable way, you can use code like this:

```
#if defined(__BORLANDC__)
    typedef unsigned char uint8_t;
    typedef __int64 int64_t;
    typedef unsigned long uintptr_t;
#elif defined(_MSC_VER)
    typedef unsigned char uint8_t;
    typedef __int64 int64_t;
```

```
#else
    #include <stdint.h>
#endif
```

Just place that in some .h file and include wherever you need it.

Alternatively, you can download Microsoft's version of the `stdint.h` file from here or use a portable one from here.

Share

Improve this answer

Follow

edited Aug 28, 2015 at 8:19

idmean

14.8k ● 9 ● 60 ● 88

answered Sep 30, 2008 at 13:45

Milan Babuškov

61k ● 49 ● 130 ● 180

See stackoverflow.com/questions/126279/… for info on how to get a stdint.h that works with MSVC (and possibly Borland). – Michael Burr Sep 30, 2008 at 14:55

3 Both links broken! – Antonio Mar 16, 2018 at 22:58

2 This answer is related to C but the language is tagged **C++** so it not the answer what i was looking for. – HeavenHM Apr 26, 2018 at 12:51

@HaSeeBMiR An appropriate fix is to switch to `<cstdint>`, or to download the appropriate `cstdint` if you download a `stdint.h`. – Justin Time - Reinstate Monica Nov 13, 2018 at 20:36

1 @HaSeeBMiR The only reason the answer is related to C instead of C++ is that it uses a C header instead of the equivalent C++ header. The C preprocessor is a part of C++, and `cstdint` is a part of the C++ standard, as are all of the type names defined there. It is indeed appropriate for the specified tags. ...I do disagree with defining the types manually, but it may be necessary when working with compilers that don't do so.
– Justin Time - Reinstate Monica Nov 14, 2018 at 23:00

42

'size_t' and 'ptrdiff_t' are required to match your architecture (whatever it is). Therefore, I think rather than using 'int', you should be able to use 'size_t', which on a 64 bit system should be a 64 bit type.

This discussion unsigned int vs size_t goes into a bit more detail.

Share

Improve this answer

Follow

edited May 23, 2017 at 12:26

Community Bot

1 ● 1

answered Sep 30, 2008 at 15:05

Richard Corden

21.7k ● 9 ● 61 ● 87

38 While size_t is usually large enough to hold a pointer, it's not necessarily the case. It would be better to locate a stdint.h header (if your compiler doesn't already have one) and use uintptr_t. – Michael Burr Sep 30, 2008 at 22:00

3 Unfortunately the only constraint on `size_t` is that it must hold the result of any `sizeof()`. This doesn't necessarily make it 64 bits on x64. see also – Antoine Nov 19, 2013

at 10:44

3 `size_t` *can* safely store the value of a non-member pointer. See en.cppreference.com/w/cpp/types/size_t. – AndyJost Apr 9, 2016 at 0:10

3 @AndyJost No it cannot. Even your own link confirms that. – yyny Apr 23, 2016 at 0:45

1 @YoYoYonnY: "On many platforms (an exception is systems with segmented addressing) std::size_t can safely store the value of any non-member pointer, in which case it is synonymous with std::uintptr_t." - whatever are you talking about? – slashmais Jun 12, 2018 at 8:06

17

Use `uintptr_t` as your integer type.

Share

Improve this answer

Follow

edited Aug 28, 2015 at 8:19

idmean

14.8k ● 9 ● 60 ● 88

answered Sep 30, 2008 at 13:46

moonshadow

88.9k ● 7 ● 86 ● 121

9

Several answers have pointed at `uintptr_t` and `#include <stdint.h>` as 'the' solution. That is, I suggest, part of the answer, but not the whole answer. You also need to look at where the function is called with the message ID of FOO.

Consider this code and compilation:

```
$ cat kk.c
#include <stdio.h>
static void function(int n, void *p)
{
    unsigned long z = *(unsigned long *)p;
    printf("%d - %lu\n", n, z);
}

int main(void)
{
    function(1, 2);
    return(0);
}
$ rmk kk
        gcc -m64 -g -O -std=c99 -pedantic -Wall -Wshadow -Wpointer-arith \
            -Wcast-qual -Wstrict-prototypes -Wmissing-prototypes \
            -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE kk.c -o kk
kk.c: In function 'main':
kk.c:10: warning: passing argument 2 of 'func' makes pointer from integer
without a cast
$
```

You will observe that there is a problem at the calling location (in `main()`) — converting an integer to a pointer without a cast. You are going to need to analyze your `function()` in all its usages to see how values are passed to it. The code inside my `function()` would work if the calls were written:

```
unsigned long i = 0x2341;
function(1, &i);
```

Since yours are probably written differently, you need to review the points where the function is called to ensure that it makes sense to use the value as shown. Don't forget, you may be finding a latent bug.

Also, if you are going to format the value of the `void *` parameter (as converted), look carefully at the `<inttypes.h>` header (instead of `stdint.h` — `inttypes.h` provides the services of `stdint.h`, which is unusual, but the C99 standard says *[t]he header `<inttypes.h>` includes the header `<stdint.h>` and extends it with additional facilities provided by hosted implementations*) and use the PRIxxx macros in your format strings.

Also, my comments are strictly applicable to C rather than C++, but your code is in the subset of C++ that is portable between C and C++. The chances are fair to good that my comments apply.

Share
Improve this answer
Follow

edited Aug 31, 2014 at 0:08

answered Sep 30, 2008 at 14:20

Jonathan Leffler
752k ● 145 ● 946 ● 1.3k

3 I think you missed the point on my question. The code is storing the value of an integer in a pointer. And that part of the code is doing the opposite (e.g. extracting the value of the integer that was written *as* a pointer). – PierreBdR Oct 28, 2014 at 8:11

@PierreBdR Nevertheless he makes a very valid point. It's not always so simple as to look at code (including when compilers warn about it) that uses a signed int but is used for a size and think it okay to change it to unsigned. Unfortunately it's not always that simple. You have to look at each case explicitly unless you want to cause potential bugs - and subtle bugs at that. – Pryftan Nov 17, 2019 at 22:06

4

1. `#include <stdint.h>`
2. Use `uintptr_t` standard type defined in the included standard header file.

Share
Improve this answer
Follow

edited Dec 4, 2010 at 21:19

Igor Klimer
15.3k ● 3 ● 49 ● 57

answered Nov 10, 2010 at 11:23

Michael S.
51 ● 3

4

I came across this question while studying the source code of SQLite.

In the sqliteInt.h, there is a paragraph of code defined a macro convert between integer and pointer. The author made a very good statement first pointing out it should be a compiler dependent problem and then implemented the solution to account for most of the popular compilers out there.

```
#if defined(__PTRDIFF_TYPE__)  /* This case should work for GCC */
# define SQLITE_INT_TO_PTR(X)  ((void*)(__PTRDIFF_TYPE__)(X))
# define SQLITE_PTR_TO_INT(X)  ((int)(__PTRDIFF_TYPE__)(X))
#elif !defined(__GNUC__)       /* Works for compilers other than LLVM */
# define SQLITE_INT_TO_PTR(X)  ((void*)&((char*)0)[X])
# define SQLITE_PTR_TO_INT(X)  ((int)(((char*)X)-(char*)0))
#elif defined(HAVE_STDINT_H)   /* Use this case if we have ANSI headers */
# define SQLITE_INT_TO_PTR(X)  ((void*)(intptr_t)(X))
# define SQLITE_PTR_TO_INT(X)  ((int)(intptr_t)(X))
#else                          /* Generates a warning - but it always works
*/
# define SQLITE_INT_TO_PTR(X)  ((void*)(X))
# define SQLITE_PTR_TO_INT(X)  ((int)(X))
#endif
```

And here is a quote of the comment for more details:

```
/*
** The following macros are used to cast pointers to integers and
** integers to pointers.  The way you do this varies from one compiler
** to the next, so we have developed the following set of #if statements
** to generate appropriate macros for a wide range of compilers.
**
** The correct "ANSI" way to do this is to use the intptr_t type.
** Unfortunately, that typedef is not available on all compilers, or
** if it is available, it requires an #include of specific headers
** that vary from one machine to the next.
**
** Ticket #3860:  The llvm-gcc-4.2 compiler from Apple chokes on
** the ((void*)&((char*)0)[X]) construct.  But MSVC chokes on ((void*)(X)).
** So we have to define the macros in different ways depending on the
** compiler.
*/
```

Credit goes to the committers.

Share Improve this answer Follow

answered Mar 25, 2017 at 17:11

B.Mr.W.
19.6k ● 35 ● 119 ● 182

3

The best thing to do is to avoid converting from pointer type to non-pointer types. However, this is clearly not possible in your case.

As everyone said, the uintptr_t is what you should use.

This link has good info about converting to 64-bit code.

There is also a good discussion of this on comp.std.c

Share Improve this answer Follow

answered Sep 30, 2008 at 14:05

Benoit
38.9k • 24 • 85 • 117

0

I think the "meaning" of void* in this case is a generic handle. It is not a pointer to a value, it is the value itself. (This just happens to be how void* is used by C and C++ programmers.)

If it is holding an integer value, it had better be within integer range!

Here is easy rendering to integer:

```
int x = (char*)p - (char*)0;
```

It should only give a warning.

Share

Improve this answer

Follow

edited Nov 10, 2011 at 6:47

Mat
207k • 41 • 400 • 415

answered Nov 10, 2011 at 6:21

Craig Hicks
41 • 1 • 1

0

Since `uintptr_t` is not guaranteed to be there in C++/C++11, if this is a one way conversion you can consider `uintmax_t`, always defined in `<cstdint>`.

```
auto real_param = reinterpret_cast<uintmax_t>(param);
```

To play safe, one could add anywhere in the code an assertion:

```
static_assert(sizeof (uintmax_t) >= sizeof (void *) ,
              "No suitable integer type for conversion from pointer type");
```

Share

Improve this answer

edited Mar 19, 2018 at 16:40

answered Mar 16, 2018 at 23:04

Antonio
20.2k • 14 • 106 • 214

Follow

1 If you do not have uintptr_t, then uintmax_t is not an answer either: there is no garanty you can store the value of a pointer in it! There might be no integer type that does that.
– PierreBdR Mar 18, 2018 at 11:48

0

With C++11, For what it's worth, suppose you don't have any headers, then define:

```
template<bool B, class T, class F> struct cond { typedef T type; };
template<class T, class F> struct cond<false, T, F> { typedef F type;};
static constexpr unsigned int PS = sizeof (void *);

using uintptr_type = typename cond<
   PS==sizeof(unsigned short), unsigned short ,
      typename cond<
          PS==sizeof(unsigned int), unsigned int,
              typename cond<
                  PS==sizeof(unsigned long), unsigned long, unsigned long
long>::type>::type>::type;
```

After that you can do the following:

```
static uintptr_type ptr_to_int(const void *pointer) {
    return reinterpret_cast<uintptr_type>(pointer);
}
static void *int_to_ptr(uintptr_type integer) {
    return reinterpret_cast<void *>(integer);
}
```

Share Improve this answer Follow

answered Nov 24, 2021 at 7:24

Tomer Shalev
59 • 5