C++: how to cast 2 bytes in an array to an unsigned short

Asked 16 years, 1 month ago Modified 11 years, 7 months ago Viewed 47k times

19

I have been working on a legacy C++ application and am definitely outside of my comfort-zone (a good thing). I was wondering if anyone out there would be so kind as to give me a few pointers (pun intended).

I need to cast 2 bytes in an unsigned char array to an unsigned short. The bytes are consecutive.

For an example of what I am trying to do:

I receive a string from a socket and place it in an unsigned char array. I can ignore the first byte and then the next 2 bytes should be converted to an unsigned char. This will be on windows only so there are no Big/Little Endian issues (that I am aware of).

Here is what I have now (not working obviously):

```
//packetBuffer is an unsigned char array containing the string "123456789" for
testing
//I need to convert bytes 2 and 3 into the short, 2 being the most significant
byte
//so I would expect to get 515 (2*256 + 3) instead all the code I have tried
gives me
//either errors or 2 (only converting one byte
unsigned short myShort;
myShort = static_cast<unsigned_short>(packetBuffer[1])
```

c++ pointers casting

Share

Improve this question

Follow

edited Nov 1, 2011 at 17:53

Jason Plank

2,336 ● 5 ● 32 ● 40

asked Nov 19, 2008 at 2:05

user38784

201 ● 1 ● 2 ● 4

now, what must this guy think about us c++ programmers. everyone has another "right" solution :D – Johannes Schaub - litb Nov 19, 2008 at 2:57

well he could be talking about the fact that we c++ ppl are a crafty bunch and we change the rules as we like. Mu ha haaa. – baash05 Nov 19, 2008 at 3:01

Does the input contain a string with values of '0'-'9', or does it contain bytes with values of 0-255? The docs say string, but it makes no sense to multiply by 256 in that case.

– Mark Ransom Nov 19, 2008 at 3:54

i suspect it contains the binary numbers 1 to 9. – Johannes Schaub - litb Nov 19, 2008 at 4:11

11 Answers

Sorted by: Highest score (default)

24

Well, you are widening the char into a short value. What you want is to interpret two bytes as an short. `static_cast` cannot cast from `unsigned char*` to `unsigned short*`. You have to cast to `void*`, then to `unsigned short*`:

```
unsigned short *p = static_cast<unsigned short*>(static_cast<void*>
(&packetBuffer[1]));
```

Now, you can dereference p and get the short value. But the problem with this approach is that you cast from unsigned char*, to void* and then to some different type. The Standard doesn't guarantee the address remains the same (and in addition, dereferencing that pointer would be undefined behavior). A better approach is to use bit-shifting, which will always work:

```
unsigned short p = (packetBuffer[1] << 8) | packetBuffer[2];
```

Share

Improve this answer

Follow

edited Nov 20, 2008 at 21:34

answered Nov 19, 2008 at 2:24

Johannes Schaub - litb

506k ● 131 ● 917 ● 1.2k

2 The shift part is the correct way to deal with this reliably across all hardware types. But the offsets are 0 and 1, not 1 and 2 - I'll edit momentarily. – Jonathan Leffler Nov 19, 2008 at 5:30

3 And this (and the other answers) assume an endian-ness -- big-endian, I think. – Jonathan Leffler Nov 19, 2008 at 5:31

2 Jonathan, your edit is wrong. he wanted to have 2 and 3 in it, not 1 and 2. – Johannes Schaub - litb Nov 20, 2008 at 21:34

4

This is probably well below what you care about, but keep in mind that you could easily get an unaligned access doing this. x86 is forgiving and the abort that the unaligned access causes will be caught internally and will end up with a copy and return of the value so your app won't know any different (though it's significantly slower than an aligned access). If, however, this code will run on a non-x86 (you don't mention the target platform, so I'm assuming x86 desktop Windows), then doing this will cause a processor data abort and you'll have to manually copy the data to an aligned address before trying to cast it.

In short, if you're going to be doing this access a lot, you might look at making adjustments to the code so as not to have unaligned reads and you'll see a perfromance benefit.

Share Improve this answer Follow

answered Nov 19, 2008 at 2:15

ctacke

67.2k ● 20 ● 98 ● 155

You don't have to copy; you can do the bit shift operations instead. – Jonathan Leffler Nov 19, 2008 at 5:31

@Jonathan: yes, but it's still requiring an assignment into another variable, which is a copy. – ctacke Nov 19, 2008 at 15:03

3

```
unsigned short myShort = *(unsigned short *)&packetBuffer[1];
```

Share Improve this answer Follow

answered Nov 19, 2008 at 2:28

PiNoYBoY82

1,648 ● 2 ● 14 ● 17

3

The bit shift above has a bug:

```
unsigned short p = (packetBuffer[1] << 8) | packetBuffer[2];
```

if `packetBuffer` is in bytes (8 bits wide) then the above shift can and will turn `packetBuffer` into a zero, leaving you with only `packetBuffer[2];`

Despite that this is still preferred to pointers. To avoid the above problem, I waste a few lines of code (other than quite-literal-zero-optimization) it results in the same machine code:

```
unsigned short p;
p = packetBuffer[1]; p <<= 8; p |= packetBuffer[2];
```

Or to save some clock cycles and not shift the bits off the end:

```
unsigned short p;
p = (((unsigned short)packetBuffer[1])<<8) | packetBuffer[2];
```

You have to be careful with pointers, the optimizer will bite you, as well as memory alignments and a long list of other problems. Yes, done right it is faster, done wrong the bug can linger for a long time and strike when least desired.

Say you were lazy and wanted to do some 16 bit math on an 8 bit array. (little endian)

```
unsigned short *s;
unsigned char b[10];

s=(unsigned short *)&b[0];

if(b[0]&7)
{
   *s = *s+8;
   *s &= ~7;
}

do_something_With(b);

*s=*s+8;

do_something_With(b);

*s=*s+8;

do_something_With(b);
```

There is no guarantee that a perfectly bug free compiler will create the code you expect. The byte array `b` sent to the `do_something_with()` function may never get modified by the `*s` operations. Nothing in the code above says that it should. If you don't optimize your code then you may never see this problem (until someone does optimize or changes compilers or compiler versions). If you use a debugger you may never see this problem (until it is too late).

The compiler doesn't see the connection between s and b, they are two completely separate items. The optimizer may choose not to write `*s` back to memory because it sees that `*s` has a number of operations so it can keep that value in a register and only save it to memory at the end (if ever).

There are three basic ways to fix the pointer problem above:

1. Declare `s` as volatile.
2. Use a union.
3. Use a function or functions whenever changing types.

Share

Improve this answer

Follow

edited Nov 1, 2011 at 17:55

Jason Plank

2,336 ● 5 ● 32 ● 40

answered Nov 20, 2008 at 23:39

old_timer

71.4k ● 9 ● 98 ● 174

1 it won't be turned into a zero. the char value is first converted into an int (promoted), then it is shifted. if both the left, and the right sides are char, then it will run into that problem – Johannes Schaub - litb Nov 21, 2008 at 12:28

"The operands shall be of integral or enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand." – Johannes Schaub - litb Nov 21, 2008 at 12:30

2

You should not cast a unsigned char pointer into an unsigned short pointer (for that matter cast from a pointer of smaller data type to a larger data type). This is because it is assumed that the address will be aligned correctly. A better approach is to shift the bytes into a real unsigned short object, or memcpy to a unsigned short array.

No doubt, you can adjust the compiler settings to get around this limitation, but this is a very subtle thing that will break in the future if the code gets passed around and reused.

Share Improve this answer Follow

answered Nov 19, 2008 at 2:18

sep

3,485 ● 4 ● 31 ● 33

2

Maybe this is a very late solution but i just want to share with you. When you want to convert primitives or other types you can use union. See below:

```
union CharToStruct {
    char charArray[2];
    unsigned short value;
};


short toShort(char* value){
    CharToStruct cs;
    cs.charArray[0] = value[1]; // most significant bit of short is not first
bit of char array
    cs.charArray[1] = value[0];
    return cs.value;
}
```

When you create an array with below hex values and call toShort function, you will get a short value with 3.

```
char array[2];
array[0] = 0x00;
array[1] = 0x03;
```

```
short i = toShort(array);
cout << i << endl; // or printf("%h", i);
```

Share Improve this answer Follow

answered May 21, 2013 at 14:50

ilkayaktas

188 ● 1 ● 7

1

static cast has a different syntax, plus you need to work with pointers, what you want to do is:

```
unsigned short *myShort = static_cast<unsigned short*>(&packetBuffer[1]);
```

Share Improve this answer Follow

answered Nov 19, 2008 at 2:08

arul

14.1k ● 1 ● 59 ● 78

This is wrong! It won't compile. Although I wouldn't recommend it, at least reinterpret_cast is a better deal. – sep Nov 19, 2008 at 2:26

indeed, static_cast can only cast the reverse of [what a standard implicit conversion can, exclusive conversion from a derived to one of its virtual base classes] unsigned short * p; unsigned char * c = p; won't work – Johannes Schaub - litb Nov 19, 2008 at 2:47

Watch out for alignment problems. – Loki Astari Nov 19, 2008 at 3:44

0

Did nobody see the input was a string!

```
/* If it is a string as explicitly stated in the question.
 */
int byte1 = packetBuffer[1] - '0'; // convert 1st byte from char to number.
int byte2 = packetBuffer[2] - '0';

unsigned short result = (byte1 * 256) + byte2;

/* Alternatively if is an array of bytes.
 */
int byte1 = packetBuffer[1];
int byte2 = packetBuffer[2];

unsigned short result = (byte1 * 256) + byte2;
```

This also avoids the problems with alignment that most of the other solutions may have on certain platforms. Note A short is at least two bytes. Most systems will give you a memory error if you try and de-reference a short pointer that is not 2 byte aligned (or whatever the sizeof(short) on your system is)!

Share

Improve this answer

Follow

edited Nov 19, 2008 at 5:32

answered Nov 19, 2008 at 3:41

Loki Astari

264k ● 86 ● 342 ● 571

It's not a string - and the bytes are not necessarily representing digits in the code set.
– Jonathan Leffler Nov 19, 2008 at 5:26

1 I quote: 'packetBuffer is an unsigned char array containing the string "123456789"'
– Loki Astari Nov 19, 2008 at 5:30

1 I quote: 'I receive a string from a socket and place it in an unsigned char array' – Loki Astari Nov 19, 2008 at 5:35

OK - it is a string; it is weirder than I realized; sorry. – Jonathan Leffler Nov 21, 2008 at 1:15

0

```
char packetBuffer[] = {1, 2, 3};
unsigned short myShort = * reinterpret_cast<unsigned short*>(&packetBuffer[1]);
```

I (had to) do this all the time. big endian is an obvious problem. What really will get you is incorrect data when the machine dislike misaligned reads! (and write).

you may want to write a test cast and an assert to see if it reads properly. So when ran on a big endian machine or more importantly a machine that dislikes misaligned reads an assert error will occur instead of a weird hard to trace 'bug' ;)

Share Improve this answer Follow

answered Nov 19, 2008 at 5:59

user34537

0

On windows you can use:

```
unsigned short i = MAKEWORD(lowbyte,hibyte);
```

Share Improve this answer Follow

answered Nov 21, 2008 at 11:33

Richard

753 ● 2 ● 6 ● 12

0

I realize this is an old thread, and I can't say that I tried every suggestion made here. I'm just making my self comfortable with mfc, and I was looking for a way to convert a uint to two bytes, and back again at the other end of a socket.

There are alot of bit shifting examples you can find on the net, but none of them seemed to actually work. Alot of the examples seem overly complicated; I mean we're just talking about grabbing 2 bytes out of a uint, sending them over the wire, and plugging them back into a uint at the other end, right?

This is the solution I finally came up with:

```
class ByteConverter
{
public:
 static void uIntToBytes(unsigned int theUint, char* bytes)
  {
   unsigned int tInt = theUint;

   void *uintConverter = &tInt;
   char *theBytes = (char*)uintConverter;

   bytes[0] = theBytes[0];
   bytes[1] = theBytes[1];
  }
 static unsigned int bytesToUint(char *bytes)
  {
   unsigned theUint = 0;

   void *uintConverter = &theUint;
   char *thebytes = (char*)uintConverter;

   thebytes[0] = bytes[0];
   thebytes[1] = bytes[1];

   return theUint;
  }
};
```

Used like this:

```
unsigned int theUint;
char bytes[2];
CString msg;

ByteConverter::uIntToBytes(65000,bytes);
theUint = ByteConverter::bytesToUint(bytes);

msg.Format(_T("theUint = %d"), theUint);
AfxMessageBox(msg, MB_ICONINFORMATION | MB_OK);
```

Hope this helps someone out.

Share Improve this answer Follow

answered Jan 23, 2010 at 20:37

Pete

1