CS:APP Web Aside DATA:TMIN: Writing $TMin$ in C*

Randal E. Bryant
David R. O'Hallaron

December 29, 2014

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Third Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2016. In this document, all references beginning with "CS:APP3e " are to this book. More information about the book is available at* `csapp.cs.cmu.edu`*.*

1 The Situation

In Figure CS:APP3e-2.19 and in Problem CS:APP3e-2.21, we carefully wrote the value of $TMin_{32}$ as `-2147483647-1`. Why not simply write it as either `-2147483648` or `0x80000000`? Looking at the C header file `limits.h`, we see that they use a similar method as we have to write $TMin_{32}$ and $TMax_{32}$:

```
/* Minimum and maximum values a 'signed int' can hold.  */
#define INT_MAX   2147483647
#define INT_MIN   (-INT_MAX - 1)
```

Unfortunately, a curious interaction between the asymmetry of the two's complement representation and the conversion rules of C force us to write $TMin_{32}$ in this unusual way. Although understanding this issue requires us to delve into one of the murkier corners of the C language standards, it will help us appreciate some of the subtleties of integer data types and representations.

Consider the case of writing $TMin_{32}$ as `-2147483648` and compiling the code as a 32-bit program, using the data sizes shown in Figure CS:APP3e-2.9. When the compiler encounters a number of the form $-X$, it first determines the data type and value for X and then negates it. The value 2,147,483,648 is too large to

ISO C90		ISO C99	
Decimal	Hexadecimal	Decimal	Hexadecimal

`int`
`long`
`unsigned`
`unsigned long`
	`int`
	`unsigned`
	`long`
	`unsigned long`
		`int`
		`long`
		`long long`
			`int`
			`unsigned`
			`long`
			`unsigned long`
			`long long`
			`unsigned long long`

Figure 1: **Data types for representing integer constants.** According to the language version and format (decimal or hexadecimal), the data type for a constant is given by the first type in the appropriate list that can represent the value.

Word Size	ISO C90		ISO C99	
Expression	`-2147483648`	`0x80000000`	`-2147483648`	`0x80000000`
32	`unsigned`	`unsigned`	`long long`	`unsigned`
64	`long`	`unsigned`	`long`	`unsigned`

Figure 2: **Data types resulting from constant expressions for $TMin_{32}$.** According to the language version and format (decimal or hexadecimal), we can get three different data types for the two expressions, including cases where the value is positive.

represent as an `int`, since this value is one larger than $TMax_{32}$ (the asymmetry strikes!). The compiler tries to determine a data type that can represent this value properly. It proceeds down one of the lists shown for the decimal cases in Figure 1, depending on the language version.[1] For the case of ISO C90, it proceeds from `int` to `long` to `unsigned`, only then finding a data type that can represent the number 2,147,483,648. As we will see in CS:APP3e-2.3.3, values 2,147,483,648 and $-$2,147,483,648 have the same bit representations as 32-bit numbers, and so the resulting constant has data type `unsigned` and value `2147483648`. For the case of ISO C99, the compiler proceeds from `int` to `long` to `long long`, finally finding a data type that can represent the number 2,147,483,648. With 64 bits, we can uniquely represent both 2,147,483,648 and $-$2,147,483,648, and so the resulting constant has data type `long long` and value `-2147483648`.

When compiling hexadecimal constant `0x80000000` in a 32-bit program, the compiler proceeds in a similar fashion, following one of the lists for the hexadecimal cases in Figure 1. For both language versions, it first compares the number to $TMax_{32}$ (`0x7FFFFFFF`) and, since it is larger, decides that the value cannot be represented as an `int`. It next compares the number to $UMax_{32}$ (`0xFFFFFFFF`) and, since it is smaller, chooses an `unsigned` representation. It therefore yields a constant with data type `unsigned` and value `0x80000000` (or, equivalently, 2,147,483,648).

Things work a bit differently with a 64-bit program. For both language versions, the decimal form yields a constant with data type `long` (64-bits) and value $-$2,147,483,648, while the hexadecimal form yields a constant with type `unsigned` and value `0x80000000` (or, equivalently, 2,147,483,648).

[1] Data type `long long` is not covered in CS:APP3e . It was introduced in ISO C99 as a data type that is at least 64 bits long.

All of these variations can be summarized by the table shown in Figure 2. For the cases where the result has type `long` or `long long`, the constant is negative, but it is 64 bits long. For the cases where the result has type `unsigned`, the constant is positive and 32 bits long. These outcomes can be demonstrated by the following code:

```
int dpos32 = (-2147483648 > 0);
int hpos32 = (0x80000000  > 0);
```

These lines of code attempt to express $TMin_{32}$ as a decimal or hexadecimal constant and test whether the value is greater than zero. Depending on the compiler version and word size, we find that the value of `dpos32` can be either 1 or 0, indicating that the decimal constant can be either positive or nonnegative, while the value of `hpos32` is consistently 1, indicating that the hexadecimal constant is consistently positive. Our seemingly simple task of writing $TMin_{32}$ as a constant is more difficult than might be expected!

Practice Problem 1:

Consider the following code:

```
int dtmin32  = -2147483648;
int dpos32a = (dtmin32 > 0);
int htmin32  = 0x80000000;
int hpos32a = (htmin32 > 0);
```

We compile this code as both 32-bit and 64-bit programs on a machine using two's complement representations of integers, and we try it for both language standards ISO-C90 and ISO-C99. In all cases, we consistently get value 0 for both `dpos32a` and `hpos32a`, and further tests verify that `dtmin32` and `htmin32` indeed equal $TMin_{32}$. Explain why this code does not have the compiler and language sensitivities we saw for the earlier code example.

2 Implications

For many programs, the ambiguities caused by different word sizes and language standards would not affect program behavior (for example, see Problem 1.) Nonetheless, we can now appreciate why the convention of writing $TMin_{32}$ as `-2147483647-1` yields a more desirable result. Since `2147483647` is the value of $TMax_{32}$, it can be represented as an `int`, and hence there is no need to invoke the conversion rules of Figure 1.

Practice Problem 2:

Suppose we try to write $TMin_{32}$ as `-0x7FFFFFFF-1`. Would the C compiler generate a constant of type `int` for both 32- and 64-bit programs and for both versions of the C language standard? Explain.

Practice Problem 3:

You wish to write a succinct expression for $TMin_w$, where w is the number of bits in data type `long int`. Since the size of this data type varies depending on the machine and the compiler settings (see

Figures CS:APP3e-2.9 and CS:APP3e-2.10), you decide to make use of the `sizeof` operation, so that the expression will yield $TMin_w$ as long as w is a multiple of 8. You also use a trick, to be covered in Section CS:APP3e-2.3.6, that shifting a number left by 3 is the same as multiplying it by 8.

Your first attempt at this code is:

```
/* WARNING: This code is buggy */
/* Shift 1 over by 8*sizeof(long) - 1 */
1L  << sizeof(long)<<3 - 1
```

You compile your code as a 32-bit program and find that the expression evaluates to 64.

A. Explain why this happened.

B. What value would the expression yield for a 64-bit program?

C. Make minimal modifications to the expression so that it evaluates correctly.

Practice Problem 4:

Suppose we try to write the value of $TMin_{64}$ as decimal and hexadecimal constants. Fill in the following table using the rules shown in Figure 1 to determine what type the resulting value should be. You may find some cases where the rules do not define a valid representation for the constant. Indicate such cases with the entry "*undefined*."

Word Size	C Version	`-9223372036854775808`	`0x8000000000000000`
32	C90		
32	C99		
64	C90		
64	C99		

Solutions to Practice Problems

Problem 1 Solution: [Pg. 3]

In making the assignment to integer variables `dtmin32` and `htmin32`, we implicitly cast the value to a 32-bit, two's complement integer. This yields the value $-2{,}147{,}483{,}648$ regardless of whether or not the constant value is signed or unsigned, or whether it is 32 or 64 bits.

Problem 2 Solution: [Pg. 3]

Yes, this would work as expected regardless of word size and language standard. Since `0x7FFFFFFF` is equal to $TMax_{32}$, it will represent this value with data type `int`. The resulting expression therefore has data type `int`.

Problem 3 Solution: [Pg. 3]

This is a classic example of failing to consider the operator precedence rules in C. As mentioned in Section CS:APP3e-2.1.9, addition and subtraction have higher precedence than shifting, and shifting associates to the left.

A. Consider the case where data type `long` requires 4 bytes. Then the expression is equivalent to `1 << 4 << 3 - 1`, which evaluates as `(1 << 4) << 2`, yielding 64.

B. When `long` requires 8 bytes, we would have `1 << 8 << 3 - 1` which evaluates as `(1 << 8) << 2`, yielding 1024.

C. The problem can be fixed with just one set of parentheses:

```
/* Shift 1 over by 8*sizeof(long) - 1 */
1L  << (sizeof(long)<<3) - 1
```

We could also replace `sizeof(long)<<3` by `8*sizeof(long)`, and the higher precedence of multiplication would ensure correct expression evaluation. In fact, this would make the code more readable, and the resulting machine-level code would be identical.

Problem 4 Solution: [Pg. 4]

This problem uncovers some quirky aspects about the rules shown in Figure 1:

Word Size	C Version	`-9223372036854775808`	`0x8000000000000000`
32	C90	*undefined*	*undefined*
32	C99	*undefined*	`unsigned long long`
64	C90	`unsigned long`	`unsigned long`
64	C99	*undefined*	`unsigned long`

Here are explanations of the entries:

32 / C90: None of the data types in C90 can represent $TMin_{64}$, and so neither the decimal nor the hexadecimal expression yields a valid constant.

32 / C99: The decimal value `9223372036854775808` cannot be represented with any of the data types, including `long long`, and so this also does not yield a valid constant. On the other hand, the hexadecimal value `0x8000000000000000` can be represented with data type `unsigned long long`.

64 / C90: Both the decimal value `9223372036854775808` and the hexadecimal value `0x8000000000000000` can be represented as `unsigned long`.

64 / C99: The decimal value `9223372036854775808` cannot be represented with any of the data types, and so this also does not yield a valid constant. On the other hand, the hexadecimal value `0x8000000000000000` can be represented with data type `unsigned long`.