INTEGER PROMOTIONS AND CONVERSIONS
IN C

BY

`0xLeo` (github.com/0xleo)

JULY 2, 2019

DRAFT X.YY
MISSING: ...

Contents

1 Integer promotions and signed conversions in C

1.1 Integer sub-types and ranges

Integer promotion refers to when sub-types of `int`, such as `short` and `char` are implicitly converted to `int`. The table below shows the size of `int` and its sub-types for most 32-bit machines.

Types	Bits	Naming	Min	Max
`char (signed char)`	8	byte	-2^7	2^7-1
`unsigned char`	8	byte	0	2^8-1
`short (signed short)`	16	word	-2^{15}	$2^{15}-1$
`unsigned short`	16	word	0	$2^{16}-1$
`int (signed int)`	32	double word	-2^{31}	$2^{31}-1$
`unsigned short`	32	double word	0	$2^{32}-1$

Note that the sizes in the table are common among many systems but not universal. For example, OpenBSD systems use different numbers of bits.

1.2 Integer promotion example

As we'll see, this happens when we perform arithmetic operations on the sub-types. The second basic rule is that any operand which is sub-type of `int` is automatically converted to the type `int` , provided `int` is capable of representing all values of the operand's original type. If `int` is not sufficient, the operand is converted to `unsigned int`.

In the code below, the sub-expression `c1 * c2 = 400` is promoted to `int`. The division `c1 * c2 / c3` also yields an int (40). Since that fits in the `signed char` range of $[-128, 127]$ [1], we have no overflow so it can safely be cast back to `signed char`. Note that values such as `10, 100, '('` are also treated as `int`, therefore take 4 bytes, before being cast to `char` (1 byte).

Listing 1: char promotion to int. (src/char_to_int.c).

```
1  #include <stdio.h>
2  
3  char foo(char c1, char c2, char c3) {
4      char res = c1 * c2 / c3;
5      printf("%d, %d, %d\n",
6              res, sizeof(res), sizeof(c1*c2/c3));
7      return res;
8  }
9  
10 int main()
11 {
12     // ASCII '(' = decimal 40
13     foo(100, 4, '(');
14 }
```

The disassembly for line 4 shows clearly what happens. `char c1, c2, c3` are all treated as `int` and so is the result `char res = char c1, c2, c3`, which is stored in register `EAX` after the `idiv` instruction [2]. However, because `res` was declared as `char` type, we extract only its bottom 8 bits (`AL` sub-sub register of `EAX`) and store them back to a local variable.

```
; char res = c1 * c2 / c3;
movsx   edx, BYTE PTR [ebp-28]
movsx   eax, BYTE PTR [ebp-32]
imul    eax, edx
movsx   ecx, BYTE PTR [ebp-36]
cdq
```

[1] If it didn't fit in that range, we'd have signed overflow, which is undefined behaviour in C and wouldn't be able to determine the value of `res`. If, on the other hand, `res` was `unsigned char` and was assigned e.g. $256 \notin [0, 255]$, we'd have unsigned overflow. The compiler would map 256 to `256 mod UCHAR_MAX = 256 mod 256 = 0`, 257 to 1 etc.

[2] App **TODO!** describes in detail how instruction `idiv` works.

```
idiv    ecx
mov     BYTE PTR [ebp-9], al
```

2 Signed and unsigned conversions

2.1 Conversion golden rule

Another problem occurs when we mix `unsigned` with `signed` types, e.g. by adding them together. The general integer conversion rule, that holds for short, char, int, either signed or unsigned is:

> "In case of operands of different data types, one integer operand (and hence the result) is promoted to the type of other integer operand, if other integer operand can hold larger number."

If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type (`unsigned short`, `unsigned int`, etc.).

This rule applies whenever we perform arithmetic or logical operations (for both the left and right side operands), be it <, +, ==, etc.

2.2 Example of conversions

Below is a listing that demonstrates the principle. Note that `printf` was not used much as e.g. trying to print and unsigned integer as signed (%d) results in undefined behaviour.

Listing 2: Examples of signed and unsigned type mixing.

```c
#include <stdio.h>
#include <limits.h> // USHRT_MAX

int main(int argc, char *argv[])
{
    short int shi;
    unsigned int ui;
    signed int si;
    signed char sc;
    unsigned char uc;
    signed short ss;

    // Example 1
    si = -5;
    ui = 2;
    (si + ui <= 0) ? puts("[Ex1]: -5 + 2 <= 0") :
        puts("[Ex1]: -5 + 2 > 0");

    // Example 2
    shi = -5;
    uc = 2;
    (shi + uc < 0) ? puts("[Ex2]: -5 + 2 < 0") :
        puts("[Ex2]: -5 + 2 >= 0");

    // Example 3 (http://www.idryman.org/blog/2012/11/21/integer-promotion/)
    uc = 0xff;
    sc = 0xff;
    (sc == uc) ? puts("[Ex3]: equal"):
        printf("[Ex3]: signed = 0x%x, unsigned = 0x%x\n", sc, uc);

    // Example 4
    ss = -1;
    ui = UINT_MAX;
```

```c
    printf("[Ex4]: signed = 0x%x, unsigned = 0x%x\n", ss, ui);

    // Example 5 (https://pleasestopnamingvulnerabilities.com/integers.html)
    shi = -1;
    si = 1;
    (si > shi) ? puts("[Ex5]: 1 > -1") : puts("[Ex5]: 1 <= -1");

    // Example 6
    uc = 200;
    int i = uc + 100 > uc;
    printf("[Ex 6]: %d\n", i);

    // Example 7
    uc = 200;
    i = uc + (unsigned int)100 > uc;
    printf("[Ex 6]: %d\n", i);

    // Example 8 - unsigned overflow -> wraparound
    uc = -1;
    printf("unsigned char = %u\n", uc);
}
```

The output is:

```
[Ex1]: -5 + 2 > 0
[Ex2]: -5 + 2 < 0
[Ex3]: signed = 0xffffffff, unsigned = 0xff
[Ex4]: signed = 0xffffffff, unsigned = 0xffffffff
[Ex5]: 1 > -1
[Ex 6]: 1
[Ex 7]: 1
[Ex 8]: unsigned char = 255
```

Let's interpret the results.

Example 1. The summation operands are `signed int si` and `unsigned int ui`. Because the latter can express larger numbers, `si` is converted to unsigned integer by adding to it `UNSIGNED_INT_MAX + 1`. Therefore the result we compare against zero is a very larger number.

Example 2. Since `(signed) short` can hold larger values than `unsigned char`, `uc` is converted to `short`. Its value is the same as either type so we have no loss of information. Compiling for 32 bits, the disassembly would look essentially like as follows.

```
mov     word ptr [ebp - 6], -5
mov     byte ptr [ebp - 7], 2
movsx   ecx, word ptr [ebp - 6]
movzx   edx, byte ptr [ebp - 7]
add     ecx, edx
```

In the beginning, the values are represented by the sizes corresponding to their types but before the addition they have to be moved to 32 bit registers, hence be zero extended (`movzx`) or sign extended (`movsx`). The compiler prefers to directly move the data to the full registers instead of explicitly applying the integer conversion rule, which in this case would be converting them to short integers.

Example 3. In this example, the two `char`s are converted to a hex value of length 8, i.e. to `unsigned int` type. `sc` is *sign extended* (i.e. its leading one is propagated to the higher bits until it fits in 32 bits) and `uc` is *zero extended* (its leading zero is propagated).

Example 4. In this example, although numerically `ss` and `ui` are different, we convert them to `unsigned int` via the `printf` function. `ui` is already `0xffffffff` in hex therefore no extension is needed and `ss` is signed-extended to also represent `0xffffffff` in hex. The result of == would be `true`.

Example 5. Here, we have two signed operands. The one that can hold larger values is `signed int si`. Therefore `shi` is converted to that type (by sign extension) and it will again represent `-1`. Since `-1` fits in the new range, we have no loss of information.

Example 6. We have two operations – addition and comparison. Due to integer promotion rules, the intermediate result of `uc + 100` will be represented as an `int`. Next, we compare an `int` to an `unsigned char`. Therefore the latter type will be converted to the former. `uc` doesn't lose any information so we compare whether `300 > 200`.

Example 7. We have a similar comparison but add `unsigned int 100` to the `unsigned char` instead. The result of the addition will be represented as `unsigned int` by `300`.

Example 8. We convert the representation of `-1` from `unsigned char` to `unsigned int`. `-1` is represented as `0xff` (or 255) as `unsigned char`. Note that its bit don't change – they're still `1111 1111`, only its representation. In the `printf`, zero extension is performed so it doesn't lose any information.

Regarding the last example, in general, to convert a negative signed to signed we do the following loop:

```
while (number < 0) {
    number += MAX_UNSIGNED_INT + 1
}
```

This does not change the binary representation of the number – only the way it's interpreted. In binary, negative numbers are represented by `2's complement`. For example, on a 4-bit machine, we have the signed

```
-2 = 1110b
```

Adding `MAX_UNSIGNED_INT = 16` does not change the bits of the number. Using the magnitude representation instead of 2's complement, we have

```
-2 + MAX_UNSIGNED_INT = 14 = 1110b
```

These are were basics of how integers are handled by the machine in C.

A Appendices

A.1 `idiv` and `imul` instructions

`imul` and `idiv` instructions are used in assembly to perform multiplication or division with signed integers. `mul` and `div` are their respective unsigned instructions. We'll be using Intel IA-32 instructions for convenience.

A.1.1 `imul`

The IMUL instruction takes one, two or three operands. It can be used for byte, word or dword operation. IMUL only works with signed numbers. The result is the correct sign to suit the signs of the multiplicand and the multiplier, therefore the if necessary (e.g. negative) is *sign extended* following the 2's complement rules. The size of the result maybe up to twice of the input size. Therefore when using a user-specified register as the destination (table below), the result is truncated to the register size and it's up to the user to prevent information loss. For 32-bit architectures, the result may be represented with up to 64 bits. Finally, remember that

- 32-bit signed range represents numbers $[-2^{31}, 2^{31}-1]$,
- 32-bit unsigned range represents numbers $[0, 2^{32}-1]$.

Listing 3: `imul` simple demonstration. (src/imul_only.asm).

```
; imul_only.asm
;
; assemble: nasm -f elf -g -F stabs imul_only.asm
; link:     ld -o imul_only imul_only.o -melf_i386

SECTION .data        ; data section
    ; dd = double word (32 bits)
    val1:   dd 10, 10           ; 10 = line end
    val2:   dd -10, 10          ; 10 = line end

SECTION .text        ; code section
global _start        ; make label available to linker
_start:              ; standard  nasm  entry point
    ;;; Example 1 - one operand
    mov     ecx, 2
    mov     eax, [val1]
    ; edx:eax = eax * ecx
    imul    ecx     ; stores the 64-bit result in (high:low) EDX:EAX

    ;;; Example 2 - one operand
    xor     eax, eax ; Clear eax register
    mov     ecx, 2
    mov     eax, [val2]
    ; edx:eax = eax * ecx
    imul    ecx     ; result edx:eax < 0 so sign extended

    ;;; Example 3 - two operands
    mov     eax, 2 ; Clear eax register
    ; eax = eax * [val2]
    imul    eax, [val2]

    ;;; Example 4 - three operands
    ; imul r, r/m32, const_value
    ; eax = [val2] * 3
    imul    eax, [val2], 3
    nop
```

Note that when the result in EDX:EAX is negative, the whole double register is sign extended, to 64 bits, e.g. when we obtain -20, EDX:EAX stores `0xffffffff:0xffffffec`.

Syntax	Description	Types
imul src	EDX:EAX = EAX * src	src: r/m32
imul dst, src	dst = src * dst	dst: r32, src:r32/m32
imul dst, src1, src2	dst = src1 * src2	dst: r32, src1: r32/m32, src2: val32

A.1.2 idiv

Assuming 32-bit architecture, `idiv src` performs signed division. It divides the 64-bit register pair `edx:eax` registers by the source operand `src` (divisor). It and stores the result in the the pair `edx:eax`. It stores the quotient in `eax` and the remainder in `edx`. Non-integral results are truncated (chopped) towards 0.

Syntax	Description	Types
idiv src	EDX = EDX:EAX % src, EAX = EDX:EAX / src	src: r/m32

However, we need to be careful before using `idiv`. Check out the following example.

At line 18, `edx = 0x20 = 32`. We pollute eax with `eax = 11 = 0xb` and want to divide by `ebx = 2` so the program will try to divide `edx:eax = 0x200000000b` by 2 and store the quotient `0x200000000b/2 = 68719476741` in eax. However, $68719476741 > 2^{32}$ so it cannot fit – the program will receive a `SIGFPE` (arithmetic exception) signal by the kernel and exit.

Always m sure that is zero be `idiv` (or `div`).

Listing 4: idiv demonstration for unsigned division. (src/idiv_wrong1.asm).

```
1  ; idiv_wrong1.asm
2  ;
3  ; assemble: nasm -f elf -g -F stabs idiv_wrong1.asm
4  ; link:     ld -o idiv_wrong1 idiv_wrong1.o -melf_i386
5
6  SECTION .data          ; data section
7
8  SECTION .text          ; code section
9  global _start          ; make label available to linker
10 _start:                ; standard  nasm  entry point
11     ;;; Example 1
12     xor     edx, edx    ; clear out edx
13     mov     eax, 21
14     mov     ebx, 2
15     idiv    ebx        ; eax = edx:eax / ebx, edx = edx:eax % ebx
16
17     ;;; Example 2 - forget to clear out edx before idiv
18     mov edx, 0x20
19     mov eax, 11
20     mov ebx, 2         ; do we get eax = 5 and edx = 1?
21     idiv ebx
22     nop
```

Let's examine what happens when we divide `EDX:EAX` by a negative number.

Listing 5: idiv demonstration for signed division. (src/idiv_wrong2.asm).

```
; idiv_wrong1.asm
;
; assemble: nasm -f elf -g -F stabs idiv_wrong1.asm
; link:     ld -o idiv_wrong1 idiv_wrong1.o -melf_i386

SECTION .data          ; data section

SECTION .text          ; code section
global _start          ; make label available to linker
_start:                ; standard  nasm  entry point

    ;;; Example 1 - zeroing edx before division
```

```
    xor      edx, edx    ; clear out edx
    mov      eax, -21
    mov      ebx, 2
    idiv     ebx      ; eax = edx:eax / ebx, edx = edx:eax % ebx
    nop

    ;;; Example 2 - sign extend eax
    mov edx, 0xffffffff
    mov eax, -21
    mov ebx, 2       ; do we get eax = 5 and edx = 1?
    idiv ebx
    nop
```

In the first example, we attempt to divide −21 by 2 so we move −21 to eax, which is represented in hex as `eax = 0xffffffeb`. `edx` is zero so `idiv` will try to define the positive number (leading 0) in `edx:eax = 00000000:ffffffeb = 4294967275`. As a result, 4294967275 will be divided by 2, writing `4294967275 div 2` = 2147483637 to eax and `4294967275 mod 2 = 1` to `edx`.

To get the value right, we need to ensure the whole divident (`edx:eax`) is negative. This is done by sign extending `edx` into `eax`, i.e. set `edx = 0xffffffff` is `eax < 0`. Example 2 correctly performs the division, writing `0xffffffff` (-1) to edx and `0xfffffff6` (-10) to eax.

The next section describes an instruction that can generalise this zero/sign extension before `idiv`.

A.1.3 The `cdq` instruction

`cdq` converts the doubleword (32 bits) in `EAX` into a quadword in `EDX:EAX` by sign-extending `EAX` into `EDX` (i.e. each bit of EDX is filled with the most significant bit of EAX).

For example, if `EAX` contained `0x7FFFFFFF` we'd get 0 in `EDX`, since the most significant bit of `EAX` is clear. But if we had `EAX = 0x80000000` we'd get `EDX = 0xFFFFFFFF` since the most significant bit of `EAX` is set. The point of `cdq` is to set up `EDX` prior to a division by a 32-bit operand, since the dividend is `EDX:EAX`.

The program below demonstrates the instruction.

Listing 6: Chaining cdg with `idiv` to avoid potential arithmetic errors due to sign. (src/idiv_correct.asm).

```
1 ; idiv_correct.asm
2 ;
3 ; assemble: nasm -f elf -g -F stabs idiv_correct.asm
4 ; link:     ld -o idiv_correct idiv_correct.o -melf_i386
5
6 SECTION .data        ; data section
7
8 SECTION .text        ; code section
9 global _start        ; make label available to linker
10 _start:              ; standard  nasm  entry point
11
12     ;;; Example 1 - zero extend eax
13     mov      eax, 22
14     mov      ebx, 4
15     cdq
16     idiv     ebx
17
18     ;;; Example 2 - sign extend eax
19     mov      eax, -22
20     mov      ebx, 4
21     cdq
22     idiv     ebx
23     nop
```

After line 16, `edx = 0x2` and `eax = 0x5`. After line 21, `edx = 0xffffffff` and `eax = 0xffffffea`. After line 22, `edx = 0xfffffffe = -2` and `eax = 0xffffffbf = -5`.