DARK CORNERS OF C
ALL NOTES

By

0xLeo (github.com/0xleo)

April 24, 2021

DRAFT X.YY
Missing: . . .

Contents

1 Integer promotions and signed conversions in C

1.1 Mixing floats with ints

Consider the following program. What will be the output?

```
double d = 3.33;
if (d*3 < 10L)
    puts("true");
```

Confirming the maths, the output is true, but why? Which value gets converted to which type? Let's look at the main part of the clang disassembly:

```
movsd   xmm0, qword ptr [.LCPI0_0] # xmm0 = double 10
movsd   xmm1, qword ptr [.LCPI0_1] # xmm1 = double 3
movsd   xmm2, qword ptr [.LCPI0_2] # xmm2 = double 3.33
movsd   qword ptr [ebp - 16], xmm2
mulsd   xmm1, qword ptr [ebp - 16] # xmm1 = xmm1 * 3.33
ucomisd xmm0, xmm1
jbe     .LBB0_2 # if 10 below or equal to xmm1 jump to return (.LBB0_2)
<-- omitted -->
call    puts # if xmm0 > xmm1
```

So 10L gets converted to double, which makes sense as the other way would make operand d*3 lose its precision. A similar conversion occurs in the following snippet too.

```
int i = 1;
if (i + 1.5 < i + 2)
    puts("correct!");
```

In this case, In this case i gets stored at an int register (e.g. eax register) and when the compiler sees the addition with 1.5 it copies eax to a floating pointer register (e.g. xmm1 and performs the addition with both operands as doubles. The result of i+2 also in the end gets stored at a floating point register and the comparison between doubles is made, printing correct!.

According the C ISO [1], the following rules apply when an operator operating on double or float and another type [1]:

TAKEAWAY 1.1 (conversion between double/ float and another type – C ISO 2007, 6.3.1.8/1).

1. *First, if the type of either operand is `long double`, the other operand is converted, without change of type domain, to `long double`.*
2. *Otherwise, if the type of either operand is double, the other operand is converted, without change of type domain, to `double`.*
3. *Otherwise, if the type of either operand is `float`, the other operand is converted, without change of type domain, to `float`.*
   To sum up:

   long double > double > float > int types

1.2 Integer sub-types and ranges

Sometimes integer types and its their sub-types need to be converted from one to another. Such a conversion is called integer promotion, for example when a char gets converted to int. As a reminder, 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 int	32	double word	0	$2^{32}-1$

[1] We don't take into account imaginary numbers for simplicity.

Note that the sizes in the table are common among many systems but not universal. For example, OpenBSD systems use different numbers of bits. Another integer type, which is *not* a sub-type of `int` but a super-type of it, is `long`. It is guaranteed to be at least 32 bits. On Linux Intel architecture, which is used in these notes, its size is the following.

Types	Architecture	Bits	Naming	Min	Max
`long (signed long)`	Linux IA-32	32	quad word	-2^{31}	$2^{31}-1$
`unsigned long`	Linux IA-32	32	quad word	0	$2^{32}-1$
`long (signed long)`	Linux IA/Intel-64	64	quad word	-2^{63}	$2^{63}-1$
`unsigned long`	Linux IA/Intel-64	64	quad word	0	$2^{64}-1$

1.3 Integer promotion

Integer promotion occurs implicitly when we operate on integer sub-types or integers. If an `int` can represent all values of the original type, the value is converted to an `int` (and its value is preserved), otherwise it is converted to an `unsigned int`. Bear in mind integer promotion is not the same as integer conversion, which is studied in another section. The following examples make it clear.

EXAMPLE 1.1 (Integer promotion – signed chars). Let's take a look at what happens when we operate on integer sub-types together, for example two (signed) `char` variables.

```
char c1 = 100, c2 = 3;
if (c1*c2 > 299)
    puts("Now I know 1st grade maths!");
```

The snippet above prints `Now I know 1st grade maths!`, as it should. But what's the type of `c1`, `c2`, `c1*c2` and 299 when the comparison is made? The `gcc` disassembly shows that initially `c1` and `c2` are stored in byte (char)-wide local variables but have been copied to 4-byte registers as signed integers[2]. Next, they are multiplied as signed integers (`imul`) and then the comparison takes place:

```
mov     BYTE PTR [ebp-9], 100
mov     BYTE PTR [ebp-10], 3
movsx   edx, BYTE PTR [ebp-9]
movsx   eax, BYTE PTR [ebp-10]
imul    eax, edx
cmp     eax, 299
```

■

EXAMPLE 1.2 (Integer promotion – signed char and unsigned char). We introduce a subtle change to the previous example, defining one `signed char` and one `unsigned char` instead, so the snippet looks like:

```
char c1 = 100;
unsigned char c2 = 3;
if (c1*c2 > 299)
    puts("Now I know 1st grade maths!");
```

Then `c1` is zero-extended when copied to a 4-byte general register to store local variable `c2` as unsigned but the final result, i.e. `c1*c2` is treated as a `signed int`, as indicated by the `imul`:

```
movsx   eax, byte ptr [ebp - 5]
movzx   ecx, byte ptr [ebp - 6]
imul    eax, ecx
```

The compiler would prefer to represent expression `c1*c2` as `signed int` even if both `c1,c2` were defined `unsigned char`! ■

EXAMPLE 1.3 (Integer promotion – unsigned char and unsigned char). In this case, have define `c1` and `c2` as `unsigned char`. However, we assign -6 to one of them. What gets printed?

[2] `movsx` means move and sign-extend

```
unsigned char c1 = 100;
unsigned char c2 = -6;
printf("%d\n", c1*c2);
```

The gcc disassembly sheds some light.

```
mov     BYTE PTR [ebp-0xa],0x64
mov     BYTE PTR [ebp-0x9],0xfa
movzx   edx,BYTE PTR [ebp-0xa]
movzx   eax,BYTE PTR [ebp-0x9]
imul    eax,edx
```

So what actually gets stored in c2 (local variable [ebp-0x9]) is 0xfa = 250, therefore the compiler mapped -6 to the unsigned char range $[0,255]$[3], doing what the type of c2 defined. Although c1 and c2 are correctly represented by unsigned values (movzx instruction), the expression c1*c2 is once again treated as signed integer, as indicated by imul. ■

We've seen what happens when operations between int sub-types are performed. But what if two (signed) ints are operated and the result is not small enough to fit in the signed int range?

Example 1.4 (When result doesn't fit in signed int.)**.** In this example, we have the maximum integer than can be represented by the 32-bit range, i.e. $2^{31}-1$. We want to double it and compare it to zero, which should of course be true. But we want the operation to be performed between unsigned integers, as signed cannot represent the result. If we simply compared:

In C, integers are signed by default.

```
2*i > 0
```

, then 2, i, 0 are all signed and we'd end up comparing -2 to 0. The way to make the compiler understand that the result should be performed in unsigned if it doesn't fit in signed would be to make at least of the operands unsigned, e.g. as follows:

```
#include <stdio.h>
#include <limits.h>

void main(int argc, char *argv[])
{
    signed int i = INT_MAX; // 2^31 - 1 for 32b systems
    if (2U*i > 0)
        printf("correct! unsigned result = %u" , 2U*i);
}
```

Then

```
correct! unsigned result = 4294967294
```

is printed (i.e. $2^{32}-2$, or 0xfffffffe), as it should. As a final note, notice again that the expression 2*i > 0 operates on unsigned int if either of 2 (e.g. 2U), i, or 0 is unsigned (0U). The next section explains such conversions. ■

The last 4 examples can be justified by the following C ISO standard [1].

> **Takeaway 1.2** (integer promotion definition – C ISO 2007, 6.3.1.1/2)**.** *If an* int *can represent all values of the original type or* int *sub-type, the value is converted to an* int*; otherwise, it is converted to an* unsigned int*. These are called* integer promotions*. All other types are unchanged by integer promotions.*

Integer promotions occur every time we operate with integer sub-types.

1.4 IEEE priority (rank) rules

When operating on both signed and unsigned, there are a few things that C takes into account to determine the result, in order of priority:

1. The width of each type (e.g. 256 for signed char and unsigned char etc. The width of each type can be mapped to the "rank" in the C ISO. Integer rank defines which type will be converted to what.

[3] When mapping from one range of equal width to another, the byte representation doesn't change, only the way we interpret the bytes does.

2. Whether the operands are signed or unsigned, and whether they're different or not.

Regarding the rank, the ISO[1] defines the following:

> **TAKEAWAY 1.3** (integer rank – C ISO 2007 6.3.1.1/2).
>
> ```
> rank(long long) > rank(long) > rank(int) > rank(short) > rank(char)
> ```
>
> *The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any. Rank is essentially the width a type can represent.*

Here are some typical cases. Try to guess what rules apply given the rank and the signedness. Following the examples, the rule from C ISO is quoted.

EXAMPLE 1.5 (Mixing smaller int sub-type with larger sub-type).

```
signed char sc = -1;
unsigned short ush = 0;
(sc + ush < ush)? puts("-1 < 0"): puts("-1 >= 0");
```

In this case, we operate on `int` sub-types `signed char` and `unsigned short`. First, + takes place. According to the promotions, `sc` and `ush` are both promoted to `int` before the addition, both obtaining the same rank and yielding `-1`. So we compare the latter result to `ush`, which is zero. `ush` is also promoted to `int` before the comparison, resulting in the condition printing `-1 < 0`. ■

EXAMPLE 1.6 (Mixing int sub-type with signed int). This is similar to the previous, except `si` does not have to be promoted to `int` as it already is – only `ssh` will, maintaining its value.

```
signed short ssh = -1;
int si = 0;
(ssh + si < si)? puts("-1 < 0"): puts("-1 >= 0");
```

The result is `-1 < 0`.

EXAMPLE 1.7 (mixing signed int with unsigned int). We have the same code with the previous example, except now we compare `ssh + si` to unsigned zero (`0U`).

```
signed short ssh = -1;
int si = 0;
(ssh + si < 0U)? puts("-1 < 0"): puts("-1 >= 0");
```

The result is `-1 > 0` so `ssh + si` must have been converted to `unsigned int`. Indeed, `printf("`
`n%u`
`n", ssh + si)` gives 4294967295, which is $2^{32}-1$, which is the upper limit of `unsigned int` for 32-bit systems. To summarise, the bottom to top order is:

```
ssh => int
(ssh + si) => int
(ssh + si < 0U) => ssh + si => unsigned int
```

It seems that if we have operate on two types with the same rank and different sign, the compiler prefers to convert `signed` to `unsigned`. ■

EXAMPLE 1.8 (Mixing long with int). Now we have two types with different rank – `unsigned long` and `int`.

```
int si = -1;
unsigned long ul = 0;
(si + ul < ul)? puts("[4]: -1 < 0"): puts("[4]: -1 >= 0");
```

On a 32-bit system, `unsigned long` and `int` have the same width of 4 bytes therefore the same rank, therefore the output would depend on signedness; we've seen that signed is preferred to be converted to unsigned when the rank is the same so the result is `-1 >= 0`.

On a 64-bit system, the result would also be `-1 <= 0`. The rank of `unsigned long` is same the rank of `long` and is greater than the rank of `int`. So `si` would be converted to `unsigned long` etc.

The disassembly shows how the comparison is implemented in the generated code. The relevant parts are the following, and especially the `jbe` instruction.

```
mov DWORD PTR [rbp-0xc],0xffffffff ; int si = -1
mov QWORD PTR [rbp-0x8],0x0        ; unsigned long ul = 0
mov eax,DWORD PTR [rbp-0xc]
movsxd rdx,eax                     ; eax = 0xffffffffffffffff
mov rax,QWORD PTR [rbp-0x8]        ; rax = ul = 0
add rax,rdx                        ; rax += si
cmp QWORD PTR [rbp-0x8],rax
jbe 0x1178 <main+63>               ; if (PTR[rbp-0x8] <= rax)...
```

The instruction `jbe` treats the comparison operands as unsigned.

On the other hand, consider defining `ul` as `long`. Then `si` wouldn't have to be converted to `unsigned long` (in fact not even to long as it's simply compared to zero, hence the `DWORD`!). We'd end up with:

```
mov DWORD PTR [rbp-0xc],0xffffffff
cmp DWORD PTR [rbp-0xc],0x0
jns 0x116b <main+50>
```

`cmp` compares `si` to zero. `jns` (Jump if Not Sign) checks its sign, i.e. its first bit, and branches accordingly. ■

EXAMPLE 1.9 (signed char to unsigned char).

```
char sc = -2;
unsigned char uc = 1;
(sc + uc == -1)? puts("[5] -1"): puts("[5] != -1");
```

In this case, both `sc` and `uc` are promoted to integer before the addition and the sign and values are preserved:

```
movsx  edx,BYTE PTR [rbp-0x2]
movsx  eax,BYTE PTR [rbp-0x1]
add    eax,edx
cmp    eax,0xffffffff
jne    0x117d <main+52>
```

`[5] -1` is printed. The result would also be true in the following cases:

```
(sc + uc == -1U)
(sc + uc == UINT_MAX)
(sc + uc == 0xffffffff)
```

■

EXAMPLE 1.10 (signed int to unsigned int). signed int si = -5; unsigned int ui = 2; (si + ui == -3)? puts("[6]: -5+2==-3"): puts("[6]: -5+2!=-3"); In this case, `si` and `ui` have the same rank but different sign. Because `int` can represent both, no conversion is necessary therefore `[6]: -5+2==-3` is printed. ■

The code for all examples is found in `src/int_conv/conversions.c`. The following C ISO extract applies to the above examples. Note that when we operate on signed and unsigned it is *not always* the case that signed will be converted to unsigned (paragraph 4).

> **TAKEAWAY 1.4** (C ISO 2007 – 6.3.1.8/1).
>
> 1. *If both operands have the same type, then no further conversion is needed.*
> 2. *Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.*
> 3. *Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.*
> 4. *Otherwise, 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, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.*
> 5. *Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the*

> *operand with signed integer type.*

Finally, if it's not clear yet, to convert a negative signed to unsigned we do the following:

```
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 compiler in C.

2 Inline functions in C

2.1 Why use them?

Functions in C can be declared as `inline` to *hint* (but not force) the compiler to optimise the speed of the code where they are used. Although many compilers know when to `inline` a function, it's a good practice to declare them in the source code.

Making a function `inline` means that instead of calling it, its body is copied by the compiler to the caller line. This eliminates the overhead of calling a function (creating stack space, arguments and local variables, and jumping to its definition, push variables to stack, pop etc.). It's a good practice for short functions that are called a few times in the code, otherwise it increases the code size (each call, one copy is added to the code).

> **Takeaway 2.1.** `inline` *is nothing but a hint to the compiler to try to replace a function call with its definition code wherever it's called.*

It may seem that inline functions are similar to macros. They are, but there are two key differences:

- Macros are expanded by the preprocessor before compilation and they *always* substitute the caller text with the body text.
- `inline` functions are type-checked but macros are not since macros are just text.

Let's create an inline function, call it and see what happens. If we try to compile the code below (without optimisations), i.e.

```
gcc inline_error.c -o inline_error
```

we get the linker error:

Listing 1: Attempting to declare an inline function (src/inline_error.c).

```c
inline int foo()
{
    return 0xaa;
}

int main(int argc, char *argv[])
{
    int ret;

    ret = foo();
    return 0;
}
```

```
inline_error.c:(.text+0x12): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

In this case, the compiler has chosen *not to* inline `foo`, searches its definition symbol and cannot find it. However, if we compile with optimisations, i.e.

```
gcc -O inline_error.c -o inline_error
```

, then everything will work. `foo` will be inlined and so the linker will not need the "regular" definition.

2.2 Linkage issues – `static inline` vs `extern inline`

The C ISO, section 6.7.4[4], defines the following regarding the linkage of inline functions.

> *Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply:*
>
> *If a function is declared with an inline function specifier, then it shall also be defined in the same translation unit.*

[4] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

> *If all of the file scope declarations for a function in a translation unit (TU)* [5] *include the inline function specifier without* `extern`*, then the definition in that TU is an inline definition. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another TU. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same TU. It is unspecified whether a call to the function uses the inline definition or the external definition.*

The key part of this specification is "an inline definition does not provide an external definition for the function, and does not forbid an external definition in another TU." When we defined function `foo()` in Listing 2 and used it, although it was in the same file, the call to that function is resolved by the linker not the compiler, because it is implicitly `extern`. But this definition alone does not provide an external definition of the function. That's how inline functions differ from regular ones. Also, note the part "it is unspecified whether a call to the function uses the inline definition or the external definition.". That means that if let's say, we have defined a function `inline int foo()`. When the function is called, it's up to the compiler to choose whether to inline it or not. If it chooses not to, it will call `int foo()`. But the symbol for `int foo()` is not defined, hence the error. If it chooses to inline it, it will of course find it and link it so no error.

When we inline a function, we want to have both the "regular" and inline def's available.

> **Takeaway 2.2.** *The definition of an inline function must be present in the TU where it is accessed.*

To resolve the missing definition behaviour it is recommended that linkage always be resolved by declaring them as `static inline` or `extern inline`. Which one is preferred though?

`inline` functions in `gcc` should be declared `static` or `extern`.

2.2.1 `static inline`

If the function is declared to be a `static inline` then, as before the compiler may choose to inline the function. In addition the compiler may emit a locally scoped version of the function in the object file if necessary. There can be one static version per object file, so you may end up with multiple definitions of the same function, so if the function is long this can be very space inefficient. The reason for this is that the compiler will generate the function definition (body) in every TU that calls the inline function.

> **Takeaway 2.3.** `static` *means "compile the function only with the current TU and then link it only with it".*

Listing 2 demonstrates a case where both the "regular" and inline definitions are needed. In this case, the compiler will inline the code to compute `x*x`. However, next, it will search for the address of the (regular) `square` function. In `square` was only inlined, the address would not be found as the definition symbol would not exist.

`static inline` means "We have to have this function. If you use it but don't inline it then make a static version of it in this TU." – Linus

Listing 2: `static inline` demonstration (src/inline_static.c).

```c
#include <stdio.h>

static inline unsigned int square(int x)
{
    return x*x;
}

int main(int argc, char *argv[])
{
    unsigned int i = square(5);
    printf("address = 0x%x, return = 0x%x\n", (unsigned int) &square, i);
    return 0;
}
```

Create the object file with optimisations[6]:

```
gcc -g -O -c inline_static.c -o inline_static.o
```

View the symbol table for the object file:

```
objdump -t -M intel inline_static.o
```

[5] translation unit (TU) = source file after it has been pre-processed - i.e. after all the `#ifdef`, `#define` etc. have been resolved.

[6] Code highlighted in grey indicates it has been entered in the command line.

```
inline_static.o:     file format elf32-i386
SYMBOL TABLE:
00000000 l    df *ABS* 00000000 inline_static.c
00000000 l    d  .text 00000000 .text
00000000 l    d  .data 00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l     F .text 00000008 square
<-- omitted -->
00000008 g     F .text 00000032 main
00000000         *UND* 00000000 __printf_chk
```

Next, observe how both the inlined body and the function call (at `printf`) co-exist in the executable. Create the executable with optimisations:

Functions are more likely to be inlined when the TU is compiled with optimisations.

```
gcc -g -O inline_static.c -o inline_static
```

View the disassembly:

```
gdb -q inline_static
```

```
Reading symbols from inline_static...done.
```

```
(gdb) disas square
```

```
Dump of assembler code for function square:
   0x0804842b <+0>: mov    eax,DWORD PTR [esp+0x4]
   0x0804842f <+4>: imul   eax,eax
   0x08048432 <+7>: ret
End of assembler dump.
```

```
(gdb) print &square
```

```
$1 = (int (*)(int)) 0x804842b <square>
```

```
(gdb) disas main
```

```
<-- omitted -->
   0x08048444 <+17>: push   0x19
   0x08048446 <+19>: push   0x804842b
   0x0804844b <+24>: push   0x80484f0
   0x08048450 <+29>: push   0x1
   0x08048452 <+31>: call   0x8048310 <__printf_chk@plt>
   0x08048457 <+36>: add    esp,0x10
<-- omitted -->
```

To print 5*5=`0x19`, the compiler directly pushes it in the stack instead of calling `square`, avoiding all the call overhead. At the same time, the function definition exists in the file since its address (0x804842b) had to be printed. The takeaway here is that:

> **TAKEAWAY 2.4.** *The compiler will generate function code for a `static inline` only if its address is used.*

The listing below demonstrates it.

Listing 3: In this case, function code for square will bbnot be generated. (src/inline_static_no_code.c).

```c
#include <stdio.h>

static inline unsigned int square(int x)
{
    return x*x;
}

int main(int argc, char *argv[])
{
    unsigned int i = square(5);
    printf("return = 0x%x\n", i);
    return 0;
}
```

If we compile it with optimisations and search for the symbol of the square function, nothing will be found. It is used solely as inlined. This can also be confirmed by gdb.

```
gcc -c -O -g inline_static_no_code.c -o inline_static_no_code.o
objdump -t -M intel inline_static_no_code.o | grep square
```

Takeaway 2.5. *Short, simple functions are OK to be defined as `static inline` in the TU that calls then as long as they don't generate too much bloat.*

2.3 extern inline

In C, all functions are extern by default, i.e. visible to other TUs, so for regular functions there's no need to use it.

Declaring a function as extern tells the compiler that the storage for this function is defined somewhere and if you haven't seen it's definition that's OK – it will be connected with the linker. Thus extends the visibility of a function (or variable). It is useful when an inline function is defined in a header. Then it can be declared extern in the .c file that wants to call it. The linker will link it to the one in the header and depending on whether the compiler has decided to optimise or not, it will use either the function call or the inline code.

Since the declaration can be done any number of times and definition can be done only once, we notice that declaration of a function can be added in several TUs. But the definition only exists in one TU and it might contain. And as the extern extends the visibility to the whole file, the function with extended visibility can be called anywhere in any TU provided the declaration of the function is known. This way we avoid defining a function with the same body again and again.

A good practice is to declare a function defined somewhere else as extern.

Takeaway 2.6. *So the best practice when we want to make an inline function external is:*

```c
// .h file - regular definition
void foo(void)
{
    ...
}

// .c caller file - declaration
extern inline void foo(void);
...
foo();
```

Example 2.1. We have our simple regular function to inline in a header.

Listing 4: Definition of `foo()` (src/foo.h).

```c
#ifndef FOO_H
#define FOO_H

#include <stdio.h>

unsigned int foo(void)
{
    return 0xaa;
}
#endif /* FOO_H */
```

We want our .c caller to see it and potentially inline it. As mentioned before, the way to do this is by adding `extern inline` in front of the declaration (Listing 5).

Listing 5: Telling the compiler to use the external def'n of `foo` (src/extern_call_foo.c).

```c
#include <stdio.h>
```

```c
#include "foo.h"

extern inline unsigned int foo(void);

int main(int argc, char *argv[])
{
    printf("address = 0x%x, ret = 0x%x\n", &foo, foo());
    return 0;
}
```

The following caller, although it does not explicitly declares `foo` as external, would also work since all functions in modern C are external by default. The code produced with or without `extern` is the same. However it's a good practice to use the `extern` keyword to make it clear.

Listing 6: Implicitly telling the compiler to use the external def'n of `foo` (src/extern_call_foo2.c).

```c
#include <stdio.h>
#include "foo.h"

inline unsigned int foo(void);

int main(int argc, char *argv[])
{
    printf("address = 0x%x, ret = 0x%x\n", &foo, foo());
    return 0;
}
```

Both of the last two listings work with or without the `-O` flag - i.e. the compiler is free to choose either the inline or regular version of `foo`. In this case, it will inline `foo()` with `-O`. For the address, it of course needs the full definition.

code	gcc	gcc -O
Listing 5	✔	✔
Listing 6	✔	✔

Example 2.2.

In this example, we show how to call the *same* external inline function (address and body) in multiple `.c` files.

When we want to inline a function, its body must be present in the header where it's define As usual, the function we want to inline is `foo`. That's a rare case where we define a function in the header itself as regular functions as declared in `func.h` but defined in `func.c`.

We don't normally define functions in headers but functions we will later declare as `extern inline` are an exception.

Listing 7: Definition of `foo()` that we want to inline (src/foo.h).

```c
#ifndef FOO_H
#define FOO_H

#include <stdio.h>

unsigned int foo(void)
{
    return 0xaa;
}
#endif /* FOO_H */
```

Let's say that we have a function `foo_caller` that marks `foo` as inline, calls it, and prints its address and return declared in `foo_caller.h` and defined in `foo_caller.c`.

Listing 8: Declaration of `foo_caller()` (src/foo_caller.h).

```c
#ifndef FOO_CALLER_H
#define FOO_CALLER_H

void foo_caller(void);
```

```

#endif /* FOO_CALLER_H */
```

Listing 9: Declaration of `foo_caller()` (src/foo_caller.c).

```
#include <stdio.h>
#include "foo_caller.h"
#include "foo.h"

extern inline unsigned int foo(void);

void foo_caller(void)
{
    printf("foo_caller called foo at 0x%x, ret = 0x%x\n", &foo, foo());
}
```

Finally, we have the main function that will mark `foo` as `extern inline` (i.e. tells the compiler its definition is found elsewhere), and call it directly and through `foo_caller`. There's one important detail to note when including `foo`'s header. The header contains its definition. `foo_caller.c` includes `foo.h`, therefore contains one definition of `foo`. `main.c` includes `foo_caller.h`, therefore already contains one definition of `foo`. If we include `foo.h` in `main`, we'll end up with a multiple definition error emitted by the linker. This wouldn't be a problem with regular functions, as they only contain the declaration in the header and a function can be declared infinite times, but it is a problem when a function is defined in the header, e.g. a function we want to inline. In this case, the programmer must manually make sure to include the header only once!

When we define a function in a header, its header must only be included once in the main!

Listing 10: `main` function calling `foo` through two different files (src/main_ext_foo.c).

```
#include <stdio.h>
#include "foo_caller.h"

extern inline unsigned int foo(void);

int main(int argc, char *argv[])
{
    foo_caller();
    printf("foo called from main at 0x%x, ret = 0x%x\n", &foo, foo());
    return 0;
}
```

This compiles either without or without optimisations and the output is:

```
foo_caller called foo at 0x400526, ret = 0xaa
foo called from main at 0x400526, ret = 0xaa
```

Therefore both `main` and `foo_caller` use the same definition. The compiler is free to choose the inline or regular version of `foo` depending on the optimisation flag. For the address, it will of course always use the regular version as it needs the definition. We can do the usual checks with `gdb` and `objdumb` to confirm the disassembly looks as expected. Some final notes regarding this example.

We have been talking about `extern` functions, but `extern` variables also behave the same way.

2.3.1 Force the GNU C compiler to inline a function

In GNU C, we can force inlining of a function by setting its so-called attribute.

In GNU C (and C++), we can use function attributes to specify certain function properties that may help the compiler optimise calls or check code more carefully for correctness. Function attributes are introduced by the `__attribute__` keyword *in the declaration* of a function, followed by an attribute specification enclosed in double parentheses.

We can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following one attribute specification with another.

To get to the point, the particular attribute to force inlining is `always_inline`. According to `gcc` docs:

> ❝ *Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function even if no optimization level was specified.* ❞

Therefore we can force inlining, e.g. for a `static` function, as follows:

```
static void foo(void)
{
    // ...
}

static inline void foo(void)  __attribute__ ((always_inline));
```

We will experiment with the usual `foo` function:

Listing 11: Force `foo` to be inlined, with or without optimisations (src/force_inline.c).

```
#include <stdio.h>

static unsigned int foo(void)
{
    return 0xaa;
}

static inline unsigned int foo(void) __attribute__ ((always_inline));

int main(int argc, char *argv[])
{
    int i = foo();
    printf("ret = 0x%x\n", i);
    return 0;
}
```

Compiling without optimisations and debugging:

```
gcc -g force_inline.c -o force_inline
gdb force_inline
(gdb) set disassembly-flavor intel
(gdb) disas main
```

We see that the call to `printf`, which prints the return of `foo` is disassembled to the following snippet, which shows that our call has been inlined.

```
0x08048426 <+17>: mov    eax,0xaa
0x0804842b <+22>: mov    DWORD PTR [ebp-0xc],eax
0x0804842e <+25>: sub    esp,0x8
0x08048431 <+28>: push   DWORD PTR [ebp-0xc]
0x08048434 <+31>: push   0x80484d0
0x08048439 <+36>: call   0x80482e0 <printf@plt>
```

2.4 Conclusion – When to use the `inline` keyword?

`static inline` works in both ISO C and GNU C (see A.1), it's natural that people ended up settling for that and seeing that it appeared to work without giving errors. So `static inline` gives portability, although it may result in code bloat.

With the exception of tight loops and trivial functions, inlining is the sort of optimisation that should usually be used only when a performance bottleneck has been discovered through profiling. People suggest that:

- Don't use `inline` unless you know what they do and all of the implications.
- Choosing to use the inline code or not doing carries no guarantees but may improve performance.
- "Premature optimisation is the root of all evil." – D. Knuth.

3 Operator precedence

3.1 Associativity and precedence

Associativity and precedence defined how operations are evaluated when there are multiple in a line.

- Associativity defines the order in which operations of the same precedence are evaluated in an expression. It can be left to right (→) or right to left (←).
- Precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others.

3.2 Precedence table

If more than one operators are involved in an expression, C language has a predefined rule of priority for the operators. This rule of priority of operators is called operator precedence.

Rank	Operator	Type of operation	Associativity
1	()	Parentheses or function call	➡
1	[]	Brackets or array subscript	➡
1	.	Dot or member selection operator	➡
1	->	Arrow operator	➡
1	++ or --	Postfix increment/ decrement	➡
2	++ or --	Prefix increment/ decrement	⬅
2	+ or -	Unary plus or minus	⬅
2	!, ~	not operator, bitwise complement	⬅
2	(type), e.g. (double) 2	type cast	⬅
2	*	Indirection or dereference	⬅
2	&	address of	⬅
2	sizeof	Determine size in bytes	⬅
3	* . %	Multiplication, division, modulus	➡
4	+ -	Addition and subtraction	➡
5	« »	Bitwise left shift and right shift	➡
6	< <=	relational less/ less than or equal to	➡
6	> >=	relational greater/ greater than or equal to	➡
7	== !=	relational equal or not equal to	➡
8	&&	bitwise AND	➡
9	^	bitwise XOR	➡
10	|	bitwise OR	➡
11	&&	Logical AND	➡
12	||	Logical OR	➡
13	?:	Ternary operator	⬅
14	=	Assignment	⬅
14	+= -=	Add/ subtract and assign	⬅
14	*= /=	Multiply/ divide and assign	⬅
14	%= &=	Modulus and assign/ bitwise AND and assign	⬅
14	^= |=	Bitwise XOR/ bitwise OR and assign	⬅
14	«= »=	Shift left/ shift right and assign	⬅
15	,	comma operator[7]	➡

One in the table above that is not used often is the comma. (,).

> **Takeaway 3.1.** *Comma operator returns the rightmost operand in the expression and evaluates the rest, rejecting their return value.*

[7] https://stackoverflow.com/a/52558

EXAMPLE 3.1. The left column shows some examples of the `,` operator and the right their output.

It is no way implied that the following are good code practices, they simply test the understanding of operators!

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    i = 1, 2, 3;
    printf("%d\n", i);
}
```

```
1
```

(= operation has highest precedence, therefore `i = 1` gets evaluated first. Then, `, 2, 3` gets evaluated, which does nothing.)

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    i = (1, 2, 3);
    printf("%d\n", i);
}
```

```
1
```

(`()` have the highest priority, forcing what's inside them to get evaluated first, i.e. `1,2,3` to evaluate 3 (L to R). 3 gets assigned to `i`.)

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 1, 2, 3;
    printf("%d\n", i);
}
```

(= has the highest priority, defining `i` as `1`. Since the `int` type is multiplicative, `int 2` and `int 3` will also be attempted to be declared – compilation error.)

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("%d bytes, %d bytes\n",
        sizeof((double) (1,2,3)),
        sizeof((int) (1.0, 2.0)));
}
```

```
8 bytes, 4 bytes
```

(Outer parens first. Then inner parens, evaluating `3` and `2.0` respectively. Then type casts, evaluating `3.0` and `2` respectively. `sizeof` operates on each outer paren, returning 8 and 4 respectively.)

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0, j = 1, k = 2;
    int *p_i = &i, p_j = &j, p_k = &k;
    printf("%d %d %d\n", *p_i, *p_j, *
   p_k);
}
```

(compilation error – `int` is multiplicative but dereference (`*`) is not. Pointer `p_i` will be defined properly but `p_j, p_k` are just `int`. The correct way would be `*p_j = &j, *p_k = &k`.)

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int x = 0, y = 0;
    int i = (1, x++, ++y);
    printf("%d %d %d\n",
        i, x++, ++y);
    printf("%d %d %d\n",
        i = 1337, x, y);
}
```

```
1 1 2
1337 2 2
```

(For explanation of the pre/post increment, see A.3.)

add double loop example here... how long does it run?

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i, j = (printf("Hello?\n"),
            1337);
    printf("world!\n%d, %d\n",
            i, j);
}
```

```
Hello?
world!
0, 1337
```

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    int arr[5] = {1, 2, 3, 4, 5};
    int *p_arr = arr; // same as &arr
    [0]

    // pr(++post) > pr(++pre) = pre(*
   deref))
    // ++p_arr; *p_arr (L to R) => 2
    printf("(1) %d\n", *++p_arr);
    // p_arr++, then *p_arr => 3
    printf("(2) %d\n", *p_arr++); // *
    // ++(*p_arr) => arr[2]++
    printf("(3) %d\n", ++*p_arr);
    // ++(*(p_arr++))
    printf("(4) %d\n", ++*p_arr++);
    for (i = 0; i < sizeof(arr)/sizeof(
   arr[0]);
            ++i)
        printf("%d ", arr[i]);
    return 0;
}
```

```
(1) 2, 0xf9933074
(2) 2, 0xf9933078
(3) 4, 0xf9933078
(4) 5, 0xf993307c
1 2 5 4 5
```

(1) Reading L to R, we evaluate `*(++p_arr)`. `p_arr` initially points at the 0-th element, so we print 2.

(2) Post-inc has the highest priority, however evaluates AFTER the whole expression is evaluated, therefore we compute `*p_arr` = 2, then increment the pointer.

(3) Equal priority, so we read L to R and evaluate ++'s operand first. As we shifted the pointer before, result is `++(*p_arr)` = 3+1.

(4) Evaluated as `++(*p_arr++)` = `*p_arr++; ++p_arr`

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;
    int arr[5] = {1, 2, 3, 4, 5};

    printf("%d ", arr[++i]);
    printf("%d ", arr[i]);
    printf("%d ", arr[i++]);
    printf("%d \n", arr[i]);
    return 0;
}
```

```
\; \\
2 2 2 3
```

(Pre-increment always happens before the expressions have been evaluated and post after – see A.3.)

```c
#include <stdio.h>

void main(){
   int intVar = 20, x;
   x = ++intVar, intVar++, ++intVar;
   printf("Value of intVar = \%d, x =
    \%d", intVar, x);
}
```

```
Value of intVar=23, x=21
```

(Since = operator has more precedence than , = operator will be evaluated first. Here, `x = ++intVar, intVar++, ++intVar` so `x = ++intVar` will be evaluated, assigning 21 to `x`. Then comma operator will be evaluated [2].)

```c
#include <stdio.h>


void main()
{

    // (1)
    int x;
    x = (printf("AA") || printf("BB"));
    printf("%d\n", x);

    // (2)
    x = (printf("AA") && printf("BB"));
    printf("%d\n", x);

    // (3)
    x = printf("1") && printf("3") ||
        printf("3") && printf("7");
    printf("\n%d\n", x);

}
```

```
AA1
AABB1
13
1
```

Adapted from [2]. Keep in mind that `printf` returns the number of characters to be printed.

(1) A logical OR (`A || B`) condition checks whether A or B is true and as soon as one of them is true, exits. `printf("AA")` is true and returns `(int) true = 1` to `x`. In the end, prints `AA1`. (2) `A && B` checks whether both A and B are true and if so returns `true`, therefore runs the two `printf` commands and returns `2 && 2 = true` to `x`. (3) Similar to others, but evaluation stops when the OR (||), is hit, which is when the compiler understands the expression is true and returns `1` to `x`, therefore `13` and `1` are printed.

```c


void main()
{
    char var = 10;
    printf("var is = %d", ++var++);
}
```

```
lvalue required as increment operand
```

Question adapted from [2]. Will be evaluated as `(++var)++`. `++var` will yield `11`, which is an unnamed value – unnamed values cannot be the operand of ++ [2].

```c
#include <stdio.h>

void main()
{
   int a = 3, b = 2;
   a = a == b == 0;
   printf("%d,%d\n", a, b);
}
```

```
1, 2
```

= has the highest priority so the expression is evaluated as `a = (a == b == 0)`. Then L to R as `a =((a == b) == 0)`, i.e. `a = (0 == 0)`, i.e. `a = 1` [2].

3.2.1 Application of using operators to write concise code – string manipulation

A basic string library has been written to demonstrate how operators can be used for denser code. By using them, less buffer variables are needed. Remember that in C, strings are terminated by `'0'`, hence the conditions in the code. If precedence is correctly understood, then the code is easy to read too. Below are its functions. Prototypes are found in `sstr.h` and implementations at `sstr.c`.

Listing 12: String length implementation (src/sstrlib/sstr.c).

```c
unsigned int sstrlen(const char* pSrc)
{
    const char* start = pSrc;
    while (*++pSrc);
    return pSrc - start;
}
```

The following diagram shows how `sstrlen("abcd")` works assuming `pSrc` points to an imaginary address `0x800`. First, we increment the pointer and then compare its value to 0.

```
+--+--+--+--+--+
|a |b |c |d |\0| (0x8001)
+--+--+--+--+--+
    |
    v
    True
+--+--+--+--+--+
|a |b |c |d |\0| (0x8002)
+--+--+--+--+--+
       |
       v
       True
+--+--+--+--+--+
|a |b |c |d |\0| (0x8003)
+--+--+--+--+--+
          |
          v
          True
+--+--+--+--+--+
|a |b |c |d |\0| (0x8004)
+--+--+--+--+--+
             |
             v
             False ------> 0x8004 - 0x8000
```

Listing 13: String copy implementation (src/sstrlib/sstr.c).

```c
void sstrCpy(const char* pSrc, char* pDst)
{
    while (*pDst++ = *pSrc++);
}
```

Listing 14: Reverse a string implementation (src/sstrlib/sstr.c).

```c
void sstrrev(const char* pSrc, char* pDst)
```

```
{
    pSrc += sstrlen(pSrc) - 1;
    while (*pDst++ = *pSrc--);
}
```

Listing 15: Character to lowercase implementation (src/sstrlib/sstr.c).

```
char sstrLower(const char c)
{
    return c + 32; // ASCII table
}
```

Listing 16: Check palindrome implementation (src/sstrlib/sstr.c).

```
unsigned int sstrPalin(const char* pSrc)
{
    int len = sstrlen(pSrc);
    // empty string or one letter
    if (len == 1 || len == 0)
        return 1;
    const char *start = pSrc;
    const char *end = pSrc + len - 1;
    while (end-- >= start++)
        if (*end != *start)
            return 0;
    return 1;
}
```

Listing 17: Print string implementation (src/sstrlib/sstr.c).

```
void sstrPrint(char* pSrc)
{
    while (*pSrc)
    {
        printf("%c", *pSrc);
        pSrc++;
    }
    printf("\n");
}
```

4 Common optimisations

4.1 Caching the end of the for loop

Often, it may feel natural to write a `for` loop by calling a function in the end conditions, e.g. when operating on a string (below is pseudocode):

```
for (i = 0; i < strlen(str); ++i)
    // do something with the string
```

However this calls the `strlen` function *n* times, where *n* is the length of `str`. This may be a fast function, however for more costly functions and large loop the overhead may add up a lot. Therefore caching the length in a variable would remove the function call overhead:

```
len = strlen(str)
for (i = 0; i < len; ++i)
    // do something with the string
```

The neat way of doing this in C is by computing the length at the beginning step of the loop, i.e. the part where `i` is initialised. The following program illustrates this, except that instead of the `strlen` function we have `foo`, which simply stalls the execution by 1 second:

Listing 18: Caching the end of loop value (src/optimisations/loop_cache.c).

```c
#include <stdio.h>
#include <unistd.h> // sleep()


int foo() {
    sleep(1);
    return 1337;
}


int main(int argc, char *argv[])
{
    // Don't do this
    //for (int i = 0; i < foo(); i++)
    // Do this instead
    for (int len = foo(), i = 0; i < len; i++)
        puts(".");
    return 0;
}
```

Doing the loop body work with:

```
for (int len = foo(), i = 0; i < len; ++i)
```

makes the program take at least 1337 more seconds, even if we compile with the `-O2` flag in gcc!

References

[1] *C ISO/IEC 9899:TC3*. ANSII & ISO, Sep. 2007, pp. 43–45. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[2] *C operators - aptitude questions & answers*, includehelp.com. [Online]. Available: https://www.includehelp.com/c/operators-aptitude-questions-and-answers.aspx (visited on 11/28/2019).

A Appendices

A.1 ANSI C vs GNU C

In the main text, we have used the terms “ISO C, ANSI C” and “GNU C”. They mean different things.

- GNU C: GNU is a unix like operating system (www.gnu.org) & somewhere GNU’s project needs C programming language based on ANSI C standard. GNU use GCC (GNU Compiler Collection) compiler to compile the code. It has C library function which defines system calls such as malloc, calloc, exit...etc
- ANSI C is a standardised version of the C language. As with all such standards it was intended to promote compatibility between different compilers which tended to treat some things a little differently.
- standard specified in the ANSI X3.159-1989 document became known as ANSI C, but it was soon superseded as it was adopted as an international standard, ISO/IEC 9899:1990.

A.2 `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.2.1 `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 19: `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.2.2 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 make sure that `eax` is zero before `idiv` (or `div`).

Listing 20: `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 21: `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.2.3 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 22: 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 = 0xfffffffbf = -5.

A.3 Increment and decrement operators

A.3.1 Pre vs post increment operator

In C, pre-increment (`++i`) and post-increment (`i++`) work slightly differently. Pre-decrement (`-i`) and post-decrement (`i-`) also work in a similar manner. Pre-increment means that the variable is incremented and the incremented value is returned. Post-increment means that the variable is returned as its original value and then incremented. The way to remember them is:

- *pre*→ *first* increment, then evaluate,
- *post*→ increment *after* evaluating.

The following table summarises the differences. In the equivalent assembly code, we can see that

- in the first case (`j=i++`), `eax` register stores the initial value of local variable `i=0x42`. Next, `edx=eax+1=0x43`. Finally, `edx` is copied to `i` and `eax` is coped to `j` so `i=0x43, j=0x42`.
- In the second case (`j=++i`), `eax` stores again `i=0x42`. `eax` gets incremented. Then, it gets copied to both local variables `i` and `j` so `i=0x42, j=0x42`.

Table 1: Post vs pre-increment differences and generated code.

Operation	How it appears	Pseudocode	Assembly code [8] (initially `i=0x42, j=0x41`)	Final[9] values (initially `i=0x42`)

Post-increment	`j=i++`	`j=i`
		`i++`
			`mov DWORD PTR [ebp-0x10],0x42`
			`mov DWORD PTR [ebp-0xc],0x41`
			`mov eax,DWORD PTR [ebp-0x10]`
			`lea edx,[eax+0x1]`
			`mov DWORD PTR [ebp-0x10],edx`
			`mov DWORD PTR [ebp-0xc],eax`
				`i=0x43, j=0x42`

Pre-increment	`j=++i`	`i++`
		`j=i`
			`mov DWORD PTR [ebp-0x10],0x42`
			`mov DWORD PTR [ebp-0xc],0x41`
			`mov eax,DWORD PTR [ebp-0x10]`
			`add eax,0x1`
			`mov DWORD PTR [ebp-0x10],eax`
			`mov DWORD PTR [ebp-0xc],eax`
				`i=0x43, j=0x43`

Example A.1. In the following snippet, pre (post) increment evaluate before (after) the array indexing. Notice how the increment operator writes to its "lvalue" (either index `i` or array `arr`) each time.

```c
#include <stdio.h>

#define SIZE 5

int main(int argc, char *argv[])
{
    int arr[SIZE] = {0, 1, 2, 4, 5};
    int i = 0;

    printf("%d, ", arr[i++]);
    printf("%d, ", arr[i]);
    printf("%d, ", arr[++i]);
    printf("%d, ", arr[i]++);
    printf("%d\n", ++arr[i]);
    for (i = 0; i < SIZE; ++i)
        printf("arr[%i] = %d, ", i, arr[i]);
    printf("\n");

    return 0;
```

[8] Instruction `lea edx, [eax+0x1]` achieves the same as incrementing `eax` and moving it to `edx`.

[9] `j` of course doesn't need to be initialised but it was added just for clarity in the disassembly.

```
}
```

The output is:

```
0, 1, 2, 2, 4
arr[0] = 0, arr[1] = 1, arr[2] = 4, arr[3] = 4, arr[4] = 5,
```

A.3.2 How much does the ++ increase the value?

When we have an integers, increment operator increases the value by one. For floats/ doubles it also works the same. What if we have a pointer that points to some address and increment its value?

Assume we have an array of `int` called `arr` and a pointer `p_arr` pointing to its first element. Assume an `int` takes 4 bytes. Then it would be natural to want to move from `arr[0]` to `arr[1]`, which are 4 bytes away. So incrementing the pointer would make sense only if it was incremented by 4 (bytes). In general, here's what happens when we increment a pointer of type T.

figure for explanation

> **Takeaway A.1.** *When we increment a T*, it moves `sizeof(T)` bytes. It doesn't make sense to move any other value as then the pointer would point to incomplete data.*

The following example confirms it.

Example A.2.

```c
#include <stdio.h>

#define SIZE 5

int main(int argc, char *argv[])
{
    int arr[SIZE] = {0, 1, 2, 3, 4};
    short int sarr[SIZE] = {0, 1, 2, 3, 4};
    int* p_arr = &arr[0]; // point to beginning - same as p_arr = arr
    short int* p_sarr = &sarr[0];

    printf("p_arr points to: 0x%x\n", p_arr);
    printf("next, p_arr points to: 0x%x\n", ++p_arr);
    printf("p_arr contains: %d\n", *p_arr);

    printf("p_sarr points to: 0x%x\n", p_sarr);
    printf("next, p_sarr points to: 0x%x\n", ++p_sarr);
    printf("p_sarr contains: %d\n", *p_sarr);

    return 0;
}
```

The output is:

```
p_arr points to: 0xbfc8ce78
next, p_arr points to: 0xbfc8ce7c
p_arr contains: 1
p_sarr points to: 0xbfc8ce6e
next, p_sarr points to: 0xbfc8ce70
p_sarr contains: 1
```

In this system, `int` takes 4 bytes and `short int` 2. Hence we move from `0xbfc8ce78` to `0xbfc8ce78+4` in the first case and from `0xbfc8ce6e` to `0xbfc8ce6e+2` in the second.

A.4 Find the number of elements in array

https://stackoverflow.com/questions/27518251/how-does-sizeof-know-the-size-of-array
https://stackoverflow.com/questions/671790/how-does-sizeofarray-work