


C PROGRAMMING
ALL NOTES

By

0xLeo (github.com/0xleo)

December 2, 2019

DRAFT X.YY
Missing: ...


Contents

1 Inline functions in C **2**
- 1.1 Why use them? . . . 2
- 1.2 Linkage issues – `static inline` vs `extern inline` . . . 2
  - 1.2.1 `static inline` . . . 3
- 1.3 `extern inline` . . . 5
  - 1.3.1 Force the GNU C compiler to inline a function . . . 7
- 1.4 Conclusion – When to use the `inline` keyword? . . . 8

2 Integer promotions and signed conversions in C **9**
- 2.1 Integer sub-types and ranges . . . 9
- 2.2 Integer promotion example . . . 9

3 Signed and unsigned conversions **10**
- 3.1 Conversion golden rule . . . 10
- 3.2 Example of conversions . . . 10

4 Operator precedence **13**
- 4.1 Associativity and precedence . . . 13
- 4.2 Precedence table . . . 13
  - 4.2.1 Application of using operators to write concise code – string manipulation . . . 17

A Appendices **19**
- A.1 ANSI C vs GNU C . . . 20
- A.2 `idiv` and `imul` instructions . . . 21
  - A.2.1 `imul` . . . 21
  - A.2.2 `idiv` . . . 22
  - A.2.3 The `cdq` instruction . . . 23
- A.3 Increment and decrement operators . . . 24
  - A.3.1 Pre vs post increment operator . . . 24
  - A.3.2 How much does the ++ increase the value? . . . 25
- A.4 Find the number of elements in array . . . 26

1 Inline functions in C

1.1 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 1.1.** `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).

```
1  inline int foo()
2  {
3      return 0xaa;
4  }
5
6  int main(int argc, char *argv[])
7  {
8      int ret;
9
10     ret = foo();
11     return 0;
12 }
```

```
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.

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

The C ISO, section 6.7.4[1], 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.*

[1] 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) [2] 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. ❞

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

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.

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

> **TAKEAWAY 1.2.** *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?

1.2.1 `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.

`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

> **TAKEAWAY 1.3.** `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.

Listing 2: `static inline` demonstration (src/inline_static.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[3]:

```
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
```

[2] translation unit (TU) = source file after it has been pre-processed - i.e. after all the `#ifdef`, `#define` etc. have been resolved.
[3] 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 1.4.** *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).

```
#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 1.5.** *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.*

1.3 `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 1.6.** *So the best practice when we want to make an inline function external is:*
>
> ```
> // .h file - regular definition
> void foo(void)
> {
>     ...
> }
>
> // .c caller file - declaration
> extern inline void foo(void);
> ...
> foo();
> ```

EXAMPLE 1.1. We have our simple regular function to inline in a header.

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

```
#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>
#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 1.2.

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
```

```
3
4 void foo_caller(void);
5
6 #endif /* FOO_CALLER_H */
```

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

```
1 #include <stdio.h>
2 #include "foo_caller.h"
3 #include "foo.h"
4
5 extern inline unsigned int foo(void);
6
7 void foo_caller(void)
8 {
9     printf("foo_caller called foo at 0x%x, ret = 0x%x\n", &foo, foo());
10 }
```

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).

```
1 #include <stdio.h>
2 #include "foo_caller.h"
3
4 extern inline unsigned int foo(void);
5
6 int main(int argc, char *argv[])
7 {
8     foo_caller();
9     printf("foo called from main at 0x%x, ret = 0x%x\n", &foo, foo());
10     return 0;
11 }
```

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.

1.3.1 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>
```

1.4 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.

2 Integer promotions and signed conversions in C

2.1 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.

2.2 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 $[-128,127]$ [4], 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 12: 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 [5]. 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
idiv    ecx
mov     BYTE PTR [ebp-9], al
```

[4] 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.

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

3 Signed and unsigned conversions

3.1 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.

3.2 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 13: Examples of signed and unsigned type mixing.

```
#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;
    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 c o m p l e m e n t. 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.

4 Operator precedence

4.1 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.

4.2 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[6]	→

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

> **TAKEAWAY 4.1.** *Comma operator returns the rightmost operand in the expression and evaluates the rest, rejecting their return value.*

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

EXAMPLE 4.1. 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.)

```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 [1].)

```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 [1]. 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 [1]. Will be evaluated as `(++var)++`. `++var` will yield 11, which is an unnamed value – unnamed values cannot be the operand of ++ [1].

```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` [1].

4.2.1 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 14: 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 15: String copy implementation (src/sstrlib/sstr.c).

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

Listing 16: Reverse a string implementation (src/sstrlib/sstr.c).

```c
void sstrrev(const char* pSrc, char* pDst)
{
    pSrc += sstrlen(pSrc) - 1;
    while (*pDst++ = *pSrc--);
}
```

Listing 17: Character to lowercase implementation (src/sstrlib/sstr.c).

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

Listing 18: Check palindrome implementation (src/sstrlib/sstr.c).

```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 19: Print string implementation (src/sstrlib/sstr.c).

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

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 20: `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 21: `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 22: `idiv` demonstration for signed division. (src/idiv_wrong2.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
12     ;;; Example 1 - zeroing edx before division
13     xor     edx, edx    ; clear out edx
```

```
14      mov     eax, -21
15      mov     ebx, 2
16      idiv    ebx     ; eax = edx:eax / ebx, edx = edx:eax % ebx
17      nop
18
19      ;;; Example 2 - sign extend eax
20      mov edx, 0xffffffff
21      mov eax, -21
22      mov ebx, 2      ; do we get eax = 5 and edx = 1?
23      idiv ebx
24      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 23: 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 = 0xfffffffea`. After line 22, `edx = 0xfffffffe` = −2 and `eax = 0xffffffbf` = −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 [7] (initially `i=0x42, j=0x41`)	Final[8] 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");
```

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

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

```
18
19      return 0;
20  }
```

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.

```
1  #include <stdio.h>
2
3  #define SIZE 5
4
5  int main(int argc, char *argv[])
6  {
7      int arr[SIZE] = {0, 1, 2, 3, 4};
8      short int sarr[SIZE] = {0, 1, 2, 3, 4};
9      int* p_arr = &arr[0]; // point to beginning - same as p_arr = arr
10     short int* p_sarr = &sarr[0];
11
12     printf("p_arr points to: 0x%x\n", p_arr);
13     printf("next, p_arr points to: 0x%x\n", ++p_arr);
14     printf("p_arr contains: %d\n", *p_arr);
15
16     printf("p_sarr points to: 0x%x\n", p_sarr);
17     printf("next, p_sarr points to: 0x%x\n", ++p_sarr);
18     printf("p_sarr contains: %d\n", *p_sarr);
19
20     return 0;
21 }
```

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

References

[1] *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).