Extra Exercise Collection

Disclaimer: These exercises are in no way officially affiliated with the course. There is no guarantee of correctness, although I do my best to fix any mistakes. (If you spot an error, please let me know!)

Current version: October 7, 2025

Exercice 1 :

Using the Shell

(a) Give the bash command to create a new directory with name "foo" in the current directory.

(b) Give the bash command to create a new file called "foo.c" in the parent directory of the current directory.

(c) Give the bash command to print the contents of "foo.txt", which is located in your home directory.

(d) Give the bash command to run the executable "foo", which is located in the sub-directory "build" of the current directory.

(e) Give the bash command to compile "foo.c", located in the current directory, into the executable "foo" using gcc, with optimisation level 3 enabled.

Exercice 2 :

Basics of C: Answer the following questions with true/false.

(a) There is no way for the value of a variable declared within the scope of a function to persist between calls.

(b) An `int` will always be exactly 4 bytes in size.

(c) The standard stipulates that a `long` must always be larger in size than an `int`.

(d) The integer 0 evaluates to False.

(e) The integer 100 evaluates to True.

(f) The integer -1 evaluates to False.

(g) The value of the expression `foo++` is the same as the value of the expression `++foo`.

(h) Suppose we have `int foo[5]`. Then `foo[0]` is guaranteed to be located next to `foo[1]` in memory.

(i) Suppose we have `int foo[5]`. Then `foo[-1]` is a valid expression.

(j) In general, to store a string of length `k`, the smallest possible array we can define to do so is `char str[k]`.

Exercice 3 :

Operator Precedence: Correctly parenthesise the following expressions according to the rules of operator precedence. Assume `i, j` are some `int`s, `p` is some pointers to `int`s, and `f` is some function returning `int`. *Eg.* `3 * 4 + 2` $\equiv$ `(3 * 4) + 2`.

(a) `i >> j + 1 * i - j`

(b) `i == j & * p + 3`

(c) Challenge: `* p = i >> 3 << j - 2 * ! f () | 1`

Exercice 4 :

C Integers: Answer the following questions with true/false (T/F) or a short answer (A). For the integer lengths, assume standard values for a x86_64 Linux machine.

(a) Give the hexadecimal representation of the minimum value of an `unsigned int` (A).

(b) Give the hexadecimal representation of the minimum value of an `int` (A).

(c) Give the hexadecimal representation of the maximum value of an `int` (A).

(d) Give a way to compute `i - j` using only bitwise operators and `+` (A).

(e) When signed and unsigned values are mixed in an expression, signed values are implicitly casted to unsigned (T/F).

(f) Suppose we have `short s` defined as some negative number. `(int) s` will be positive, as the extra bits on the left will be filled with 0s (T/F).

(g) Suppose we have `int i` defined as some negative number. `(char) i` will always be negative too (T/F).

(h) `u << 3 + u` ≡ `u * 9` (T/F).

Exercice 5 :

Multiplication puzzles: Match the following series of shifts and adds to a multiplicative factor. Assume `u` is some `unsigned` and no overflow.

(a) `u << 2`

(b) `(u << 5) + (u << 3) + u`

(c) `((u << 10) >> 5) + (u << ~(-11))`

Exercice 6 :

The C Preprocessor

(a) Define a macro `SQUARE(x)` that multiplies its argument with itself.

(b) Define a macro `DEBUG_INT(x)` that takes a variable's name x and prints "DEBUG: x = *value of* x" if `DEBUG` is defined, and otherwise does nothing.

(c) Name two things that belong in a header (.h) and a source (.c) file, respectively.

(d) When are preprocessor macros beneficial? What are their downsides?

Exercice 7 :

Memory Layout Basics

(a) Fill in the blanks: The ____ begins at a high address and grows downwards, whereas the ____ begins at a low address and grows upwards.

(b) True/False: Virtual memory gives each process its own address space. Hence it is never possible that two processes access the same physical page.

(c) Fill in the blanks: In a little-endian machine, the least significant byte is stored at the ____ address.

(d) True/False: The stack will always start at the exact same address in memory.

(e) Outline what happens on the stack when a function ("the caller") calls another function ("the callee") and then when the callee returns.

Exercice 8 :

Pointers

(a) Explain what the unary `&` and `*` operators do.

(b) True/False: It hold that `sizeof (int) == sizeof (int *)`

(c) True/False: Suppose we have:

```
int main(void) {
    int x = 0;
    int *p = malloc(sizeof (int));
    return 0;
}
```

Then it holds that `&x > p`.

(d) True/False: In a byte addressable system, if we have `int a[5]; int *p=a`, then `a[1]` can be accessed via `*(p + sizeof (int))`.

Exercice 9 :

Cdecl: For the following C declarations, give their natural language counterpart and vice-versa.

(a) `struct foo *x[10]`

(b) `struct foo (*x)[10]`

(c) `int **(*x)(int, int)[10]`

(d) Declare x as an array 10 of pointer to function returning pointer to `int`.

(e) `(char*[]) x`

(f) `(char (*)[10]) x`

(g) `(int (*)(int *, char)[3]) x`

(h) Challenge: Declare x as function taking pointer to `int` and returning array 3 of array 10 of pointer to function taking pointer to pointer to `int` and returning pointer to `union foo`.

(i) Challenge++: `int* (*(*x[10])(int (*)(int *)))(int *)`

Exercice 10 :

Malloc Implementations and Metrics

(a) Outline the differences between implicit and explicit free lists.

(b) (True/False): With explicit free lists, boundary tags are no longer required.

(c) Explain the differences between first fit, next fit, and best fit policies.

(d) Consider the following sequence of `(m/c)alloc/free` calls. Compute the aggregate payload P_k and the minimum heap size M_k at points 1, 2, and 3.

```
#include <stdint.h>

int main(void) {
    int *p1 = malloc(1<<10);
    free(p1);
    // Point 1
    uint32_t *p2 = calloc(1<<8, sizeof(uint32_t));
    char *p3 = malloc(1<<10);
    // Point 2
    int64_t p4 = malloc(1<<10);
    free(p2);
    // Point 3
    free(p3);
    free(p4);
    return 0;
}
```

Exercice 11 :

x86_64 Assembly Warm Up: Answer the following questions with true/false.

(a) In x86_64, each instruction is exactly 8 bytes in size.

(b) Generally speaking, compiling a given program for a RISC architecture will result in more instructions than for a CISC architecture.

(c) A long word is 64 bits in size.

(d) `movq (%rax), (%rcx)` is a valid instruction.

(e) `movq 0x9(%rbx, %rcx), %rax` will move whatever is at memory location `%rbx + %rcx + 0x9` into `%rax`.

(f) `testl %eax, %ebx` sets the ZF if `%eax != %ebx`.

Exercice 12 :

Condition Codes: Recall the x86_64 condition codes zero flag (ZF), sign flag (SF), carry flag (CF), and overflow flag (OF). For each of the following conditional jumps, derive a predicate that is true if and only if the jump is taken. You may use the operators $\wedge, \vee, \oplus$ (xor), and $\neg$. *Hint:* Assume the previous instruction was `cmpq a, b` (i.e. `subq a, b` where the result is discarded) and recall the mnemonics are from the perspective of `b` relative to `a` (i.e. `jle` is taken if `b` is less than or equal to `a`).

(a) `je`

(b) `jb`

(c) `ja`

(d) `jl`

(e) `jge`

Exercice 13 :

Reading Assembly Code I: In the following, assume the below C function was compiled with different values of the constant `K` and varying optimisation levels. For each code fragment, state the value of `K`. *Hint:* Per calling convention, the first argument to a function is stored in `%rdi`, the second in `%rsi`, while the return value is stored in `%rax`.

```c
int foo(int a) {
    return K * a;
}
```

(a)

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    $2, %eax
        popq    %rbp
        ret
```

(b)

```
foo:
        leal    (%rdi,%rdi,2), %eax
        sall    $5, %eax
        ret
```

(c)

```
foo:
        xorl    %eax, %eax
        ret
```

Exercice 14 :

Reading Assembly Code II: Consider the following C function that computes the GCD of `a` and `b` using the Euclidean Algorithm. Which of the following x86_64 assembly code fragments correspond to it? *Hint:* More than one answer may be correct. Also recall the calling convention as in Exercise 13.

```
long foo(long a, long b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

(a)

```
foo:
        movq    %rsi, %rax
        cmpq    %rsi, %rdi
        jne     .L5
.L2:
        ret
.L3:
        subq    %rdi, %rax
.L4:
        cmpq    %rax, %rdi
        je      .L2
.L5:
        cmpq    %rax, %rdi
        jle     .L3
        subq    %rax, %rdi
        jmp     .L4
```

(b)

```
foo:
        movl    $1, %edx
        xorl    %eax, %eax
.L2:
        cmpq    %rsi, %rdx
        jg      .L5
        imulq   %rdi, %rdx
        incq    %rax
        jmp     .L2
.L5:
```

```
        decq    %rax
        ret
```

(c)

```
foo:
        movq    %rdi, %rax
        cmpq    %rsi, %rdi
        jge     .L4
.L3:
        movq    %rsi, %rcx
        addq    %rsi, %rax
        subq    %rdi, %rcx
        leaq    -1(%rcx), %rdx
        imulq   %rsi, %rdx
        movq    %rcx, %rsi
        addq    %rdx, %rax
        cmpq    %rcx, %rdi
        jl      .L3
        ret
.L4:
        ret
```

(d)

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        jmp     .L2
.L4:
        movq    -8(%rbp), %rax
        cmpq    -16(%rbp), %rax
        jle     .L3
        movq    -16(%rbp), %rax
        subq    %rax, -8(%rbp)
        jmp     .L2
.L3:
        movq    -8(%rbp), %rax
        subq    %rax, -16(%rbp)
.L2:
        movq    -8(%rbp), %rax
        cmpq    -16(%rbp), %rax
        jne     .L4
```

```
        movq    -8(%rbp), %rax
        popq    %rbp
        ret
```

Exercice 15 :

Compiling Basics: Answer the following questions with true/false.

(a) There exist functions that do not require a stack frame.

(b) Every function must end in a `ret` instruction.

(c) In C, nested arrays are stored in row-major ordering.

(d) A multi-level array of k dimensions requires the same amount of memory accesses to access as a nested array of k dimensions.

(e) `switch` statements are always implemented as a series of `if`/`else` statements.

Exercice 16 :

Struct Alignment: Sketch the memory layout and state the alignment requirement K of the following `struct` declarations

(a) `struct s {int i; char c; void *p;};`

(b) `struct s {char a[3]; char c; double d; int *p[1]};`

(c) `struct s {short s; struct {int i; char c} t[2]; union {int i; float f} u;};`

Exercice 17 :

Nested Arrays

(a) Let `T a[X]` be an array X of type T. Derive a formula for the address of `a[i]`. *Hint:* You may use the base address of `a` as $\&a$ and the size of `T` as $sizeof(T)$.

(b) Let `T a[X][Y]` be an array X of array Y of type T. Derive a formula for the address of `a[i][j]`. *Hint:* You may also declare `a` as `S a[X]` with `typedef T S[Y]`.

(c) Let `T a[K_1][K_2]...[K_n]` be an array K_1 of array K_2 ... of array K_n of T. Derive a formula for the address of `a[i_1][i_2]...[i_n]`.

Exercice 18 :

Reading `setjmp/longjmp`: What does the following code output?

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

static jmp_buf buf;

void foo(void) {
    printf("foo1\n");
    longjmp(buf, 2);
    printf("foo2\n");
```

```c
}

int main(void) {
    int rv;
    if ((rv = setjmp(buf)) == 0) {
        printf("main1\n");
        foo();
        printf("main2\n");
    }
    else if (rv == 1){
        printf("main3\n");
    }
    else {
        printf("main4\n");
    }
    return 0;
}
```

Exercice 19 :

Linker Quiz: Answer the following questions with true/false.

(a) Statically linked executables tend to produce larger binaries than dynamically linked executables.

(b) The linker will not link together two symbols of different sizes (assume `-fno-common`).

(c) All the operating system needs to do when starting execution of a statically linked binary is to copy its contents into memory.

(d) By marking the stack as non-executable, buffer overruns are no longer a concern.

Exercice 20 :

Identify the linker symbols: For the following C file, for each name state what kind of linker symbol is generated (global, external, local, none), and whether it is weak or strong where applicable. Assume `-fcommon`. *Challenge:* Additionally, state the exact type of each linker symbol (e.g. an external symbol generates a U in the symbol table). `man nm` provides a succinct overview of the various types.

```c
#include <stddef.h>
#include <stdlib.h>

#define BIAS (10)

extern void add_vec(int *dest, int *src, size_t len);
int sum_vec(int *vec, size_t len);

extern int *vec1;
```

```
int arr[25];
int *vec2 = arr;
static const size_t ARR_LEN = 25ul;

int sum() {
    static int iter = 0;

    int *res = calloc(ARR_LEN, sizeof(int));
    add_vec(res, vec1, sizeof(int));
    add_vec(res, vec2, sizeof(int));

    int sum = sum_vec(res, ARR_LEN);
    free(res);

    return sum + iter + BIAS;
}
```

Exercice 21 :

Floating Point Quiz: Answer the following questions with true/false.

(a) Every `int` can be exactly represented as a `double` on an x86_64 Linux machine.

(b) When converting a `float` to a `long`, the error will always be at most 1. (Assuming no special values)

(c) The exponent bias B is computed as $B = 2^e - 1$, where e is the number of exponent bits.

(d) In the case of a denormalised floating point number, the exponent $E = -B + 1$.

(e) Floating point numbers are generally evenly distributed.

(f) Round-to-even rounding is chosen for floating point numbers out of ease of implementation, despite it being statistically biased.

(g) Suppose we have a binary decimal number of the form $1.B \dots BGRX_1X_2\dots$, where the guard bit G is the LSB of our result. Let $S = \bigvee_{i=1}^{\infty} X_i$. We round the result iff $G = R = 1, S = 0$.

Exercice 22 :

Converting to FP: Give the bit representation of the following numbers when converted to floating point. The floating point format is half precision IEEE 754, that is 1 sign bit, 5 exponent bits, and 10 fraction bits.

(a) $5000 = 1001110001000_2$

(b) $-\frac{1}{3}$

(c) 1×10^{20}

Exercice 23 :

Determining key FP values: Give the value of the following descriptions in the format $\frac{a}{b} \times 2^c$

with a, b, c being base 10 integers. Assume the same format as in Exercise 22, i.e. 1 sign bit, 5 exponent bits, and 10 fraction bits.

(a) The largest denormalised number.

(b) The smallest positive normalised number.

(c) The largest normalised number.

Exercice 24 :

Optimisation Quiz: Answer the following questions with true/false.

(a) With the `-O3` flag enabled, the compiler might perform optimisations that slightly change the semantics of a program, with the benefit of improving performance.

(b) Among RAR, RAW, WAW, and WAR dependencies, RAW is the one that typically carries the greatest performance penalty.

(c) If a procedure is throughput bound, then its operations must execute sequentially.

(d) Much of the manual optimisation that was shown in the lecture is not necessary these days, as modern compilers will simply auto-vectorise code anyway.

(e) If functions didn't have side effects, the compiler could optimise around calls much more.

Exercice 25 :

Is the optimisation legal? For the following pairs of C snippets, argue whether they are semantically equivalent in **all** cases, i.e. a compiler would be allowed to rewrite the code in such a manner.

(a)

```c
void f_0(int *a, int *b, int *c, size_t len) {
    for (size_t i=0; i<len; i++) {
        b[i] += a[i];
        c[i] += a[i];
    }
}
/* ============================================= */
void f_1(int *a, int *b, int *c, size_t len) {
    size_t i;
    for (i=0; i<len-1; i+=2) {
        b[i] += a[i];
        b[i+1] += a[i+1];
        c[i] += a[i];
        c[i+1] += a[i+1];
    }
    for (; i<len; i++) {
        b[i] += a[i];
        c[i] += a[i];
    }
```

```
}
```

(b)

```
int f_0(int *a, size_t j, size_t k) {
    int res = 0;
    for (size_t i = 0; i < k; i++) {
        res += a[2 * i + j * k];
    }
    return res;
}
/* ============================================= */
int f_1(int *a, size_t j, size_t k) {
    int res = 0;
    size_t i;
    size_t offset = j * k;
    for (i = 0; i < k - 1; i += 2) {
        size_t inner = i << 1;
        res += a[inner + offset];
        res += a[inner + 2 + offset];
    }
    for (; i < k; i++) {
        res += a[(i << 1) + offset];
    }
    return res;
}
```

(c)

```
int f_0(int *a, size_t len) {
    int res = 0;
    for (int i = 0; i < len; i++) {
        res += a[i * 30];
    }
    return res;
}
/* ============================================= */
int f_1(int *a, size_t len) {
    int res = 0;
    for (int i = 0; i < len; i++) {
        res += a[i << 5 - i << 1];
    }
    return res;
}
```

(d)

```
float f_0(float *a, long len) {
    float res = 0;
    for (long i = 0; i < len; i++) {
        res += a[i];
    }
    for (long i= len - 1; i >= 0; i--) {
        res += a[i];
    }
    return res;
}
/* ============================================= */
float f_1(float *a, long len) {
    float res = 0;
    for (long i = 0; i < len; i++) {
        res += 2.0f * a[i];
    }
    return res;
}
```

Exercice 26 :

Cache Quiz: Answer the following questions with true/false.

(a) A **capacity** miss occurs when a cache is not associative enough to store all of the lines that map to the same set.

(b) A write-allocate policy is usually employed with write-back caches.

(c) A unified cache is accessible by all cores.

Exercice 27 :

Cache formulae: Derive the desired formulae.
Suppose a cache has lines of size 2^b bytes, with associativity 2^e, and 2^s sets in total. Assume a 32-bit address space.

(a) Give a formula for the total number of data bytes that can be stored in the cache.

(b) Give a formula for the number of tag, index, and offset bits.

(c) Give a formula for the set that a given address a will be placed into. *Hint:* You may use the >> operator from C in addition to regular arithmetic operators.

Exercice 28 :

Exceptions Quiz: Answer the following questions with true/false.

(a) Traps return to the same instruction that triggered them.

(b) Synchronous exceptions are typically handled by the process that triggered them.

(c) There exists exceptions which cannot be masked by the processor.

(d) PICs generally are permitted to reorder exceptions.

Exercice 29 :

Map the exception to its type: For each of the following exception triggering events, state whether it is synchronous or asynchronous, and whether it is a trap, fault, interrupt, or abort.

(a) Calling the `open` system call.

(b) Reading a (valid) memory location that is not paged.

(c) The arrival of data from the network card.

(d) Encountering an `INT 3` (breakpoint) instruction is x86_64.

(e) A memory parity error.

(f) Dereferencing a `NULL` pointer.

Exercice 30 :

Virtual Memory Quiz: Answer the following questions with true/false.

(a) The virtual address space must be larger than the physical address space.

(b) Linear page tables are too large to fit in typical main memory sizes.

Exercice 31 :

Cache + Virtual Memory Calculations: Assume a 32-bit virtual address space with a 16-bit physical address space. Assume page size to be 1KiB. Further, assume an 8-way VP cache of size 8KiB with 64 byte blocks.

(a) How many VPN, PPN, VPO, PPO bits are there? Where are they located?

(b) How many CT, CI, CO bits are there? Where are they located?

(c) Will homonyms be an issue here?

(d) How large would a linear page table be? Suppose a 2 byte PTE.

Exercice 32 :

Multiprocessing Quiz: Answer the following questions with true/false.

(a) We say that caches are **consistent**, if their values match.

(b) Snoopy caches require a write-through policy.

(c) In the MESI protocol, if a block is exclusively held, then a *BusRdX* must still be issued upon a write.

(d) x86_64 processors generally implement sequential memory consistency.

(e) False sharing cannot occur in a single processor system, even if multiple threads are running.

(f) The ABA problem cannot occur with pointer tagging.

Exercice 33 :

Devices Quiz: Answer the following questions with true/false.

(a) Each DMA-capable device gets its own virtual address space.

(b) In a producer/consumer ring **underrun** the consumer is consuming faster than the producer can produce.

(c) An advantage of programmed I/O compared to DMA is that caches cannot become inconsistent, as data always flows through the CPU.

(d) When talking about a transmission in the context of devices, we mean a transfer of data from OS to device.

(e) Descriptor rings are typically preferred to buffer rings.