Week 02 Tutorial Questions

1. When should the types in **stdint.h** be used:

```
#include <stdint.h>

                // range of values for type
                //             minimum              maximum
    int8_t   i1; //                -128                 127
    uint8_t  i2; //                   0                 255
    int16_t  i3; //              -32768               32767
    uint16_t i4; //                   0               65535
    int32_t  i5; //         -2147483648          2147483647
    uint32_t i6; //                   0          4294967295
    int64_t  i7; // -9223372036854775808  9223372036854775807
    uint64_t i8; //                   0 18446744073709551615
```

2. Show what the following decimal values look like in 8-bit binary, 3-digit octal, and 2-digit hexadecimal:

    a. `1`
    b. `8`
    c. `10`
    d. `15`
    e. `16`
    f. `100`
    g. `127`
    h. `200`

    How could I write a C program to answer this question?

3. Assume that we have the following 16-bit variables defined and initialised:

```
uint16_t a = 0x5555, b = 0xAAAA, c = 0x0001;
```

    What are the values of the following expressions:

    a. `a | b` (bitwise OR)
    b. `a & b` (bitwise AND)
    c. `a ^ b` (bitwise XOR)
    d. `a & ~b` (bitwise AND)
    e. `c << 6` (left shift)
    f. `a >> 4` (right shift)
    g. `a & (b << 1)`
    h. `b | c`
    i. `a & ~c`

    Give your answer in hexadecimal, but you might find it easier to convert to binary to work out the solution.

4. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

    The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

    Write C expressions to implement each of the following:

    a. mark the device as locked for reading bytes
    b. mark the device as locked for writing blocks
    c. set the device as locked, leaving other flags unchanged
    d. remove the lock on a device, leaving other flags unchanged
    e. switch a device to/from reading and writing, leaving other flags unchanged

5. Discuss the starting code for `sixteen_out`, one of this week's lab exercises. In particular, what does this code (from the provided `main`) do?

```
    long l = strtol(argv[arg], NULL, 0);
    assert(l >= INT16_MIN && l <= INT16_MAX);
    int16_t value = l;

    char *bits = sixteen_out(value);
    printf("%s\n", bits);

    free(bits);
```

6. Given the following type definition

```
typedef unsigned int Word;
```

Write a function

```
Word reverseBits(Word w);
```

... which reverses the order of the bits in the variable `w`.

For example: If `w == 0x01234567`, the underlying bit string looks like:

```
0000 0001 0010 0011 0100 0101 0110 0111
```

which, when reversed, looks like:

```
1110 0110 1010 0010 1100 0100 1000 0000
```

which is `0xE6A2C480` in hexadecimal.

Revision questions

The remaining tutorial questions are primarily intended for revision - either this week or later in session.

Your tutor may still choose to cover some of the questions time permitting.

7. Consider the following small C program:

```
#include <stdio.h>

int main(void) {
    int n[4] = { 42, 23, 11, 7 };
    int *p;

    p = &n[0];
    printf("%p\n", p); // prints 0x7fff00000000
    printf("%lu\n", sizeof (int)); // prints 4

    // what do these statements print ?
    n[0]++;
    printf("%d\n", *p);
    p++;
    printf("%p\n", p);
    printf("%d\n", *p);

    return 0;
}
```

Assume the variable `n` has address `0x7fff00000000`.

Assume `sizeof (int) == 4`.

What does the program print?

8. What is the output from the following program and how does it work? Try to work out the output *without* copy-paste-compile-execute.

```
#include <stdio.h>

int main(void) {
    char *str = "abc123\n";

    for (char *c = str; *c != '\0'; c++) {
        putchar(*c);
    }

    return 0;
}
```

9. Consider the following `struct` definition defining a type for points in a three-dimensional space:

```
typedef struct Coord {
    int x;
    int y;
    int z;
} Coord;
```

and the program fragment using `Coord` variables and pointers to them:

```
{
    Coord coords[10];
    Coord a = { .x = 5, .y = 6, .z = 7 };
    Coord b = { .x = 3, .y = 3, .z = 3 };
    Coord *p = &a;

    /*** A ***/
    (*p).x = 6;
    p->y++;
    p->z++;
    b = *p;
    /*** B ***/

}
```

a. Draw diagrams to show the state of the variables `a`, `b`, and `p`, at points `A` and `B`.

b. Why would a statement like `*p.x++;` be incorrect?

c. Write code to iterate over the `coords` array using just the pointer variable `p` the address of the end of the array, and setting each item in the array to `(0,0,0)`. Do not use an index variable.

10. Consider the following pair of variables

```
int  x;  // a variable located at address 1000 with initial value 0
int *p;  // a variable located at address 2000 with initial value 0
```

If each of the following statements is executed in turn, starting from the above state, show the value of both variables after each statement:

a. `p = &x;`

b. `x = 5;`

c. `*p = 3;`

d. `x = (int)p;`

e. `x = (int)&p;`

f. `p = NULL;`

g. `*p = 1;`

If any of the statements would trigger an error, state what the error would be.

11. Consider the following C program skeleton:

```
int  a;
char b[100];

int fun1() { int c, d; ... }

double e;

int fun2() { int f; static int ff; ... fun1() ... }

unsigned int g;

int main(void) { char h[10]; int i; ... fun2() ... }
```

Now consider what happens during the execution of this program and answer the following:

a. Which variables are accessible from within `main()`?

b. Which variables are accessible from within `fun2()`?

c. Which variables are accessible from within `fun1()`?

d. Which variables are removed when `fun1()` returns?

e. Which variables are removed when `fun2()` returns?

e. Which variables are removed when `fun2()` returns?

f. How long does the variable `f` exist during program execution?

g. How long does the variable `g` exist during program execution?

12. Explain the differences between the properties of the variables `s1` and `s2` in the following program fragment:

```
#include <stdio.h>

char *s1 = "abc";

int main(void) {
    char *s2 = "def";
    // ...
}
```

Where is each variable located in memory? Where are the strings located?

13. How does the C library function

```
void *realloc(void *ptr, size_t size);
```

differ from

```
void *malloc(size_t size);
```

14. If the following program is in a file called `prog.c`:

```
#define LIFE 42
#define VAL random() % 20

#define sq(x) (x * x)
#define woof(y) (LIFE + y)

int main(void) {
    char s[LIFE];
    int i = woof(5);
    i = VAL;
    return (sq(i) > LIFE) ? 1 : 0;
}
```

... then what will be the output of the following command:

```
$ gcc -E prog.c
```

You can ignore the additional directives inserted by the C pre-processor.

15. What is the effect of each of the `static` declarations in the following program fragment:

```
#include <stdio.h>

static int x1;
...

static int f(int n) {
    static int x2 = 0;
    ...
}
```

16. What is the difference in meaning between the following pairs (a/b and c/d) of groups of C statements:

a.
```
if (x == 0) {
    printf ("zero\n");
}
```

b.
```
if (x == 0)
    printf ("zero\n");
```

c.
```
if (x == 0) {
    printf ("zero\n");
    printf ("after\n");
}
```

d.
```
if (x == 0)
    printf ("zero\n");
    printf ("after\n");
```

17. C functions have a number of different ways of dealing with errors:

- terminating the program entirely (rare)
- setting the system global variable `errno`

- returning a value that indicates an error (e.g., `NULL`, `EOF`)
- setting a returning parameter to an error value

They might even use some combination of the above.

Think about how the following code might behave for each of the inputs below. What is the final value for each variable?

```
int n, a, b, c;
n = scanf("%d %d %d", &a, &b, &c);
```

Inputs:

a. `42 64 999`
b. `42 64.4 999`
c. `42 64 hello`
d. `42 hello there`
e. `hello there`

18. Consider a function `get_int()` which aims to read standard input and return an integer determined by a sequence of digit characters read from input. Think about different function interfaces you might define to deal with input that is not a sequence of digits, or that is a very long sequence of digits.

19. For each of the following commands, describe what kind of output would be produced:

    a. `gcc -E x.c`
    b. `gcc -S x.c`
    c. `gcc -c x.c`
    d. `gcc x.c`

Use the following simple C code as an example:

```
#include <stdio.h>
#define N 10

int main(void) {
    char str[N] = { 'H', 'i', '\0' };
    printf("%s\n", str);
    return 0;
}
```

COMP1521 20T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au
CRICOS Provider 00098G