Week 02 Tutorial Solutions

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
```

Answer:

Use these types when your needs toknow exactly how many bits used of a variable.

This is not specified for the standard integer types.

You are only guaranteed a minimum number of bit, e.g. an **int** is guaranteed to be at least 16 bits.

More details here

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?

Answer:

   a. `1 = 00000001 = 001 = 0x01`
   b. `8 = 00001000 = 010 = 0x08`
   c. `10 = 00001010 = 012 = 0x0A`
   d. `15 = 00001111 = 017 = 0x0F`
   e. `16 = 00010000 = 020 = 0x10`
   f. `100 = 01100100 = 144 = 0x64`
   g. `127 = 01111111 = 177 = 0x7F`
   h. `200 = 11001000 = 310 = 0xC8`

An easy way to do this in C would be to write a bunch of *printf(3)*s like

```
printf("%d = %3o = %2x\n", 100, 100, 100);
```

You can't print binary directly in C, but it's easy to take hexadecimal and convert each hex digit to 4 binary digits.

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.

Answer:

a. `a | b == 0xFFFF`
b. `a & b == 0x0000`
c. `a ^ b == 0xFFFF`
d. `a & ~b == 0x5555`
e. `c << 6 == 0x0040`
f. `a >> 4 == 0x0555`
g. `a & (b << 1) == 0x5554`
h. `b | c == 0xAAAB`
i. `a & ~c == 0x5554`

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

Answer:

a. `device = (READING | AS_BYTES | LOCKED);`
b. `device = (WRITING | AS_BLOCKS | LOCKED);`
c. `device = device | LOCKED`
d. `device = device & ~LOCKED`
e. `device = (device & ~READING) | WRITING`

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);
```

Answer:

strtol(3) is a standard library funcion that converts a nul-terminated string passed as its first argument to an integer value (it returns `long`).

The third parameter value indicates what radix or numeric base to use; e.g., 10 for decimal. A value of 0 means values starting with 0x are interpreted as hexadeximal, values starting with 0 are interpreted as octal, and otherwise decimal.

strtol(3)'s second argument, if it is not `NULL`, allows for error checking: it is set to point to the first character that couldn't be understood.

We also check the value will fit in `int16_t`, a 16-bit signed integer by using *assert(3)*.

`sixteen_out` needs to return a string allocated with (e.g.,) *malloc(3)*: we call *free(3)* to release that allocation

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.

Answer:

```
typedef unsigned int Word;

Word reverseBits(Word w) {
    Word ret = 0;
    for (unsigned int bit = 0; bit < 32; bit++) {
        Word wMask = 1u << (31 - bit);
        Word retMask = 1u << bit;
        if (w & wMask) {
            ret = ret | retMask;
        }
    }

    return ret;
}

// testing
int main(void) {
    Word w1 = 0x01234567;
    // 0000 => 0000 = 0
    // 0001 => 1000 = 8
    // 0010 => 0100 = 4
    // 0011 => 1100 = C
    // 0100 => 0010 = 2
    // 0101 => 1010 = A
    // 0110 => 0110 = 6
    // 0111 => 1110 = E
    assert(reverseBits(w1) == 0xE6A2C480);
    puts("All tests passed!");
    return 0;
}
```

And here's a version, courtesy of Dylan Brotherston, that does not assume that `unsigned int`s are 32 bits:

```
Word reverseBits(Word w) {
    Word ret = 0; // Reversed bit-string
    Word copy = 1; // Left moving mask
    Word past = 0; // Right moving mask

    for (past = ~(-1u >> 1); past; copy <<= 1, past >>= 1)
        // If the current bit of `w` is 1,
        // set the current bit of `ret` to 1
        if (w & copy) {
            ret |= past;
        }
    return ret;
}
```

The statement `past = ~(-1u >> 1)` works as follows: `past` is an unsigned variable, and assigning it -1 will underflow, setting all bits to 1. Next we right shift by one so the leading (leftmost) bit is 0 and all other bits are 1. Finally we bitwise negate so the leading (leftmost) bit is 1 and all other bits are 0. Ultimately, we end up with a `Word` where the leading bit is 1, regardless of the size of the `Word`.

Revision questions

The following questions are primarily intended for revision, either this week or later in session. Your tutor may still choose to cover some of these questions, time permitting.

20. 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?

Answer:

Program output:

```
0x7fff00000000
4
43
0x7fff00000004
23
```

The `n[0]++` changes the value by one, because `n` is an `int` variable.

The `p++` changes the value by four, because `p` is a pointer to an `int`, and addition of one to a pointer changes it to the point to the next element of the array.

Each array element is four bytes, because `sizeof (int) == 4`

21. 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;
}
```

Answer:

Program output:

```
$ ./t1q3
abc123
```

The program works by starting the pointer `c` at the first char in the string, and then advancing it char-by-char along the string until `c` reaches the location containing the (implicit) `'\0'`. For each value of `c`, it prints the character being pointed to.

22. 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:

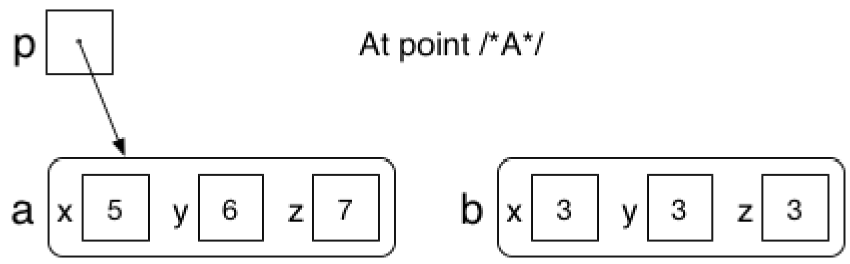
```
{
    Coord coords[10];
    Coord a = { .x = 5, .y = 6, .z = 7 };
    Coord b = { .x = 3, .y = 3, .z = 3 };
    Coord *p = &a;

    /*** 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`.

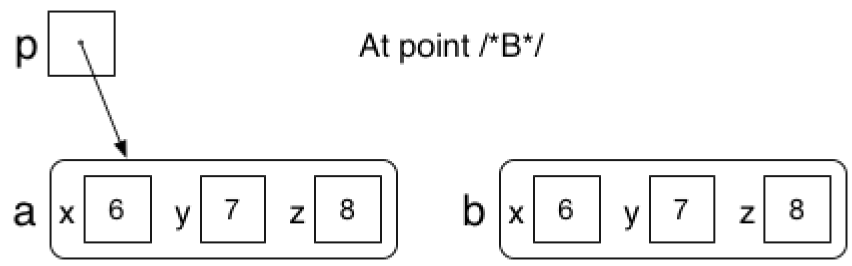
b. Why would a statement like `*p.x++;` be incorrect?

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.

Answer:



a.



b. The expression `*p.x++` is interpreted as `*((p.x)++)`, which takes the `x` coordinate of `p`, increments it, and then tries to use that as a pointer.

c.
```
for (p = &coords[0]; p <= &coords[9]; p++) {
    *p = { .x = 0, .y = 0, .z = 0 };
    // ... or ...
    p->x = 0;
    p->y = 0;
    p->z = 0;
}
```

23. 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.

Answer:

Starting with `x == 0` and `p == 0`:

```
a.  p = &x;       #  x == 0, p == 1000
b.  x = 5;        #  x == 5, p == 1000
c.  *p = 3;       #  x == 3, p == 1000
d.  x = (int)p;   #  x == 1000, p = 1000
e.  x = (int)&p;  #  x == 2000, p = 1000
f.  p = NULL;     #  x = 2000, p = NULL
g.  *p = 1;       #  error, dereference NULL pointer
```

Note that `NULL` is generally represented by a zero value. Note also that statements (d) and (e) are things that you are extremely unlikely to do.

24. Consider the following C program skeleton:

```c
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?

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

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

Answer:

a. All globals and all of `main`'s locals: `a, b, e, g, h, i`

b. All globals defined before `fun2`, and its own locals: `a, b, e, f, ff`

c. All globals defined before `fun1`, and its own locals: `a, b, c, d`

d. All of `fun1`'s local variables: `c, d`

e. All of `fun2`'s non-static local variables: `f`

f. The variable `f` exists only while `fun2` is "executing" (including during the call to `fun1` from inside `fun2`)

g. The variable `g` exists for the entire duration of program execution

25. 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?

Answer:

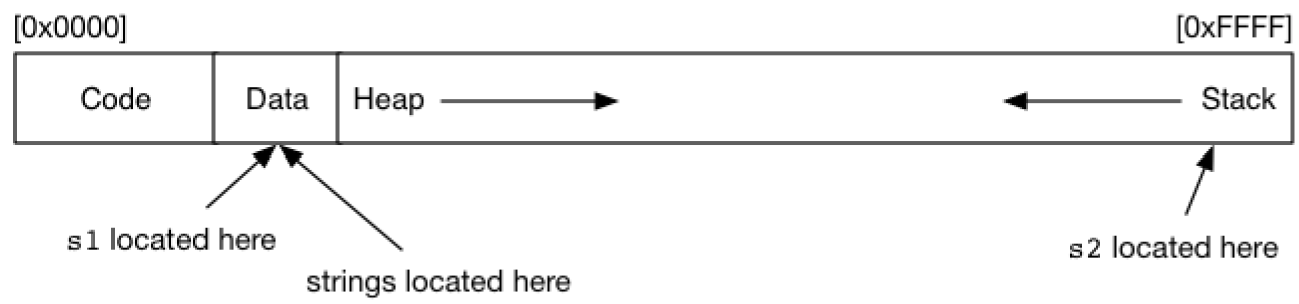
The `s1` variable is a global variable and would be accessible from any function in this `.c` file. It would also be accessible from other `.c` files that referenced it as an `extern`'d variable.

C implementations typically store global variables in the data segment (region of memory).

The `s2` variable is a local variable, and is only accessible within the `main()` function.

C implementations typically store local variables on the stack, in a stack frame created for function — in this case, for `main()`.

C implementations typically place string literals such as `"abc"` in the text segment with the program's code.



26. How does the C library function

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

differ from

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

Answer:

malloc allocates size bytes and returns a pointer to the allocated memory.

realloc changes the size of the memory block pointed to by **ptr** to be **size** bytes.

If possible **realloc** will extend/shrink the memory block in place and return **ptr**

If this is not possible: **realloc** will:

- allocate a new memory block
- copy over the contents of the old memory block
- free the old memory block
- return a pointer to the new memory block

Hence **realloc** can be conveniently use to grow an array.

27. 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.

Answer:

```
int main(void) {
    char s[42];
    int i = (42 + 5);
    i = random() % 20;
    return ((i * i) > 42) ? 1 : 0;
}
```

28. 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;
    ...
}
```

Answer:

In C, the `static` keyword is a *storage-class specifier*: it specifies variable locations and function visibility.

The `static` specifier on the `x1` variable has the effect of making it inaccessible from any other `.c` file. It is still available in all of the functions below the declaration, and will be stored in the `data` region of memory, with all of the other global variables.

The `static` specifier on the `f()` function has a similar effect on the visibility of the function. It is accessible from any function below the declaration, but is inaccessible from functions defined before `f()`, and from in other `.c` files.

The `static` specifier on the `x2` variable affects the *lifetime* of the variable, rather than its visibility. The variable is only visible within the `f()` function, but, unlike other local variables, it persists between invocations of the `f()` function.

It has the same life time as a global variable but it is only visible in the block it is defined.

29. 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");
```

Answer:

There is no difference between the first two (a and b). If an `if` controls just a single statement, the braces are optional but recommended

There is a difference between the second pair (c and d). In (c), `after` is only printed if the variable `x` has the value zero. In (d), `after` is always printed, regardless of the value of the variable `x`. The indentation in (d) is misleading, and makes it look like the second `printf` is controlled by the `if`.

We recommend always using braces:

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

30. 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`

Answer:

a. `n==3, a==42, b==64, c==999`
b. `n==2, a==42, b==64, c undefined`
c. `n==2, a==42, b==64, c undefined`
d. `n==1, a==42, b undefined, c undefined`
e. `n==0, a undefined, b undefined, c undefined`

31. 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.

Answer:

A typical implementation of a function like `get_int()` would first skip leading spaces and then start reading digit characters. It could then read digit characters until it found a non-digit. In both cases, it would also need to check that it had not reached the end of the input.

The only point at which an error would be detected would be after skipping spaces. This could be implemented as:

```
while ((ch = getchar()) != EOF && isspace(ch)) {}
if (!isdigit(ch)) {
    // ... we have an error ...
}
```

A simple (but crude) way to handle the error would be to use *assert(3)*, rather than the `if` test; for example:

```
assert(isdigit(ch));
```

If the `get_int()` function interface looks like

```
int get_int(void);
```

... then this *assert(3)* is probably the only solution.

However, we could define the `get_int()` interface as:

```
int get_int(int *);
```

so that the function places the value read into a variable whose address is passed as a parameter; this is analogous to the way `scanf()` works. The return value would not be the value read, but rather a Boolean to indicate whether or not an integer value was successfully read from input.

32. 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;
}
```

Answer:

a. `gcc -E x.c`

   Executes the C pre-processor, and writes modified C code to `stdout` containing the contents of all `#include`'d files and replacing all `#define`'d symbols.

b. `gcc -S x.c`

   Produces a file `x.s` containing the assembly code generated by the compiler for the C code in `x.c`. Clearly, architecture dependent.

c. `gcc -c x.c`

   Produces a file `x.o` containing relocatable machine code for the C code in `x.c`. Also architecture dependent. This is not a complete program, even if it has a `main()` function: it needs to be combined with the code for the library functions (by the linker *ld(1)*).

d. `gcc x.c`

   Produces an executable file called `a.out`, containing all of the machine code needed to run the code from `x.c` on the target machine architecture. The name `a.out` can be overridden by specifying a flag `-o` *`filename`*.