Week 01 Tutorial Answers

1. Class introduction (for everyone, starting with the tutor):

    - Please turn on your camera for the intro, if you can and you are comfortable with this. Having your webcam on is optional in online COMP1521 tut-labs, unless you have a cute pet in which case its required, but you only need show the pet.
    - What is your prefered name (what should we call you?)
    - What other courses are you doing this term
    - What parts of C from COMP1511/COMP1911 were the hardest to understand?
    - Do you know any good resources to help students who havve forgotten their C? For example:
        - https://learnxinyminutes.com/docs/c/

Answer:

Answered in class

2. 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();
    ...
}

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

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

```c
#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:

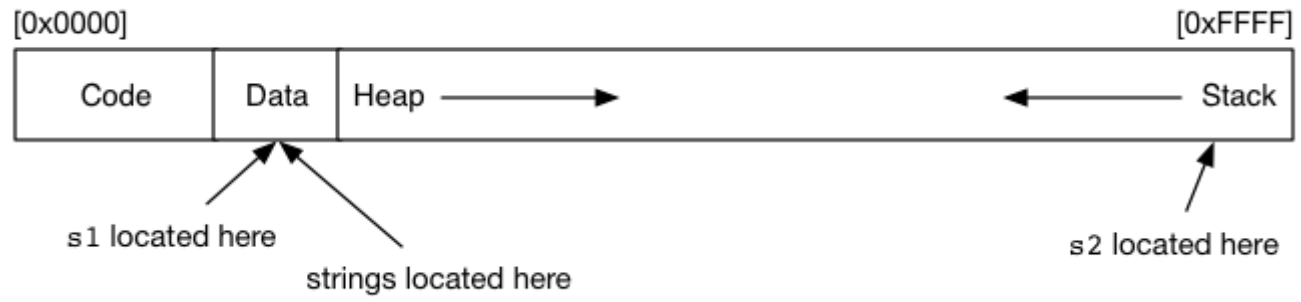
The `s1` variable is a global variable and would be accessible from any function in this `.c` file. It would also be accessible from other `.c` files that referenced it as an `extern`'d variable.

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.



4. C's sizeof operator is a prefix unary operator (precedes its 1 operand) - what are examples of other C unary operators?

Answer:

name	operator	C	note
unary minus	`-`	`int i = -5;`	
unary plus	`+`	`int j = +5;`	
Decrement	`--`	`int l = --i;`	prefix or postfix
Increment	`++`	`int k = ++j;`	prefix or postfix
Logical negation	`!`	`if (true == ! flase)`	
Bitwise negation	`~`	`int m = ~0;`	
Address of	`&`	`int *n = &m;`	
Indirection	`*`	`int o = *n;`	

5. Why is C's sizeof operator different to other C unary & binary operators?

Answer:

sizeof can be given a variable, value or a type as argument. The syntax to distinguish this is weird, the type is surrounded by brackets.

sizeof is strictly a compile time operator.

6. Discuss errors in this code:

```
struct node *a, *b, *c, *d;
a = NULL:
b = malloc(sizeof b);
c = malloc(sizeof struct node);
d = malloc(8);
c = a;
d.data = 42;
c->data = 42;
```

Answer:

`sizeof b` should be `sizeof *b`

`sizeof struct node` should be `sizeof (struct node)`

`malloc(8)` might be correct (depending on what struct ndoe is) but is definitely non-portable, struct node might be 8 bytes on a 32-bit OS and 12 or 16 bytes on a 64-bit OS

`d.data` is incorrect as d is not a struct its a pointer to a struct

`c->data` is illegal as c will be NULL when its executed

7. What is a pointer? How do pointers relate to other variables?

Answer:

Pointers are variables that refer (point) to another variable.

Typically they implement this by storing a memory address of the variable they refer to.

Given a pointer to a variable you can get its value and also assign to it.

8. Consider the following small C program:

```c
#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`

9. Consider the following pair of variables

```c
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.

10. Consider the following C program:

```c
#include <stdio.h>

int main(void)
{
    int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for (int i = 0; i < 10; i++) {
        printf("%d\n", nums[i]);
    }
    return 0;
}
```

This program uses a `for loop` to print each element in the array

Rewrite this program using a recursive function

Answer:

```c
#include <stdio.h>

void print_array(int nums[], int i)
{
    if (i == 10) return;
    printf("%d\n", nums[i]);
    print_array(nums, i + 1);
}

int main(void)
{
    int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    print_array(nums, 0);
    return 0;
}
```

11. What is a struct? What are the differences between structs and arrays?

Answer:

Arrays and struct are both compound data types, formed from other data types.

Array are homogeneous - formed from a single data type.

Structs can be heterogeneous - formed from a multiple data types.

Array element are accessed with integer array indexes.

Structs fields are accessed by name.

12. Define a struct that might store information about a pet.

    The information should include the pet's name, type of animal, age and weight.

    Create a variable of this type and assign information to it to represent an axolotl named "Fluffy" of age 7 that weighs 300grams.

Answer:

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

#define MAX_NAME_LENGTH 256
#define MAX_BREED_LENGTH 64

struct pet {
    char name[MAX_NAME_LENGTH];
    char breed[MAX_BREED_LENGTH];
    int age;
    int weight;
};

int main(void) {
    struct pet my_pet;

    strcpy(my_pet.name, "Fluffy");
    strcpy(my_pet.breed, "axolotl");
    my_pet.age = 7;
    my_pet.weight = 300;

    return 0;
}
```

13. Write a function that increases the age of fluffy by one and then increases its weight by the fraction of its age that has increased.
    The function is defined like this:
    `void age_fluffy(struct pet *my_pet);`

    e.g.: If fluffy goes from age 7 to age 8, it should end up weighing 8/7 times the amount it weighed before. You can store the weight as an int and ignore any fractions.

    Show how this function can be called by passing the address of a struct variable to the function.

Answer:

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

#define MAX_NAME_LENGTH 256
#define MAX_BREED_LENGTH 64

struct pet {
    char name[MAX_NAME_LENGTH];
    char breed[MAX_BREED_LENGTH];
    int age;
    int weight;
};

void age_fluffy(struct pet *my_pet);

int main(void) {
    struct pet my_pet;

    strcpy(my_pet.name, "Fluffy");
    strcpy(my_pet.breed, "axolotl");
    my_pet.age = 7;
    my_pet.weight = 300;

    age_fluffy(&my_pet);
    printf("%d, %d\n", my_pet.age, my_pet.weight);

    return 0;
}

void age_fluffy(struct pet *my_pet) {
    double old_age = my_pet->age;
    my_pet->age = my_pet->age + 1;
    double weightMult = my_pet->age / old_age;
    my_pet->weight = my_pet->weight * weightMult;
}
```

14. Write a main function that takes command line input that fills out the fields of the pet struct. Remember that command line arguments are given to our main function as an array of strings, which means we'll need something to convert strings to numbers.

Answer:

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

#define MAX_NAME_LENGTH 256
#define MAX_BREED_LENGTH 64

struct pet {
    char name[MAX_NAME_LENGTH];
    char breed[MAX_BREED_LENGTH];
    int age;
    int weight;
};

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

    if(argc < 5) {
        printf("%s needs 4 arguments to populate a pet.\n", argv[0]);
        return 1;
    } else {
        struct pet my_pet;

        strcpy(my_pet.name, argv[1]);
        strcpy(my_pet.breed, argv[2]);
        my_pet.age = strtol(argv[3], NULL, 10);
        my_pet.weight = strtol(argv[4], NULL, 10);
    }
    return 0;
}
```

15. Consider the following C program:

```c
#include <stdio.h>

int main(void) {
    char str[10];
    str[0] = 'H';
    str[1] = 'i';
    printf("%s", str);
    return 0;
}
```

What will happen when the above program is compiled and executed?

Answer:

The above program will compile without errors . `printf`, like many C library functions expects strings to be null-terminated.

In other words `printf`, expects the array `str` to contain an element with value '\0' which marks the end of the sequence of characters to be printed.

`printf` will print `str[0]` ('H'), `str[1]` then examine `str[2]`.

Code produced by `dcc` will then stop with an error because `str[2]` is uninitialized.

The code with gcc will keep executing and printing element from `str` until it encounters one containing '\0'. Often `str[2]` will by chance contain '\0' and the program will work correctly.

Another common behaviour will be that the program prints some extra "random" characters.

It is also possible the program will index outside the array which would result in it stopping with an error if it was compiled with `dcc`.

If the program was compiled with gcc and uses indexes well outside the array it may be terminated by the operating system because of an illegal memory access.

16. How do you correct the program.

Answer:

```c
#include <stdio.h>

int main(void) {
    char str[10];
    str[0] = 'H';
    str[1] = 'i';
    str[2] = '\0';
    printf("%s", str);
    return 0;
}
```

17. 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:

```c
#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*).

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`*.

18. Consider the following (working) C code to trim whitespace from both ends of a string:

```c
// COMP1521 20T3 GDB debugging example

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

void trim(char *str);
char **tokenise(char *str, char *sep);
void freeTokens(char **toks);

int main(int argc, char **argv)
{
    if (argc != 2) exit(1);

    char *string = strdup(argv[1]);
    printf("Input: \"%s\"\n", string);
    trim(string);
    printf("Trimmed: \"%s\"\n", string);
    char **tokens = tokenise(string, " ");
    for (int i = 0; tokens[i] != NULL; i++)
        printf("tok[%d] = \"%s\"\n", i, tokens[i]);
    freeTokens(tokens);
    return 0;
}

// trim: remove leading/trailing spaces from a string
void trim(char *str)
{
    int first, last;
    first = 0;
    while (isspace(str[first])) first++;
    last  = strlen(str)-1;
    while (isspace(str[last])) last--;
    int i, j = 0;
    for (i = first; i <= last; i++) str[j++] = str[i];
    str[j] = '\0';
}

// tokenise: split a string around a set of separators
// create an array of separate strings
// final array element contains NULL
char **tokenise(char *str, char *sep)
{
    // temp copy of string, because strtok() mangles it
    char *tmp;
    // count tokens
    tmp = strdup(str);
    int n = 0;
    strtok(tmp, sep); n++;
    while (strtok(NULL, sep) != NULL) n++;
    free(tmp);
    // allocate array for argv strings
    char **strings = malloc((n+1)*sizeof(char *));
    assert(strings != NULL);
    // now tokenise and fill array
    tmp = strdup(str);
    char *next; int i = 0;
    next = strtok(tmp, sep);
    strings[i++] = strdup(next);
    while ((next = strtok(NULL,sep)) != NULL) strings[i++] = strdup(next);
    strings[i] = NULL;
    free(tmp);
    return strings;
}

// freeTokens: free memory associated with array of tokens
void freeTokens(char **toks)
{
    for (int i = 0; toks[i] != NULL; i++) free(toks[i]);
    free(toks);
}
```

You can grab a copy of this code as trim.c.

The part that you are required to write (i.e., would not be part of the supplied code) is highlighted in the code.

Change the code to make it incorrect. Run the code, to see what errors it produces, using this command:

```
gcc -std=gnu99 -Wall -Werror -g -o trim trim.c
./trim  "   a string  "
```

Then use GDB to identify the location where the code "goes wrong".

Answer:
CSE Debugging Guide