ComSSA teaches UCP: test 1 revision

3 September 2016

Questions in UCP tests and exams follow a predictable format, and each of the sections below represent a kind of question that you may be asked in an assessment, you may have seen in a practical, or may otherwise improve your understanding.

To better prepare you for your assessments, the questions that follow are often slightly more difficult than you may expect to encounter, but they will help hone your skills and ensure that you pay extra attention when answering the real questions.

Dave may have given Mark the unit this year, but Mark hasn't really changed any of the content (yet). Dave likes to write tricky questions with subtle pitfalls, many of which test your understanding of *entirely valid code* that will compile but either run incorrectly, run surprisingly, or be otherwise less than ideal. Watch out for these warning signs:

- Preprocessor macros defined
  - without parentheses around each use of a parameter
  - without parentheses around the resultant expression
  - with a semicolon at the end, when there shouldn't be
- Loops and other control structures in which
  - counters or other key variables are modified
  - pointers are modified while they're being used
- Unusual uses of control structures such as
  - `case` statements without `break` statements
  - `for` and `while` loops with `break` and `continue` statements
- Variables that are modified once and never used again
- When Dave mentions a "pointer to an array", he nearly always actually means a "pointer to the first element of an array", but pointers to entire arrays are also a real concept, even if they are used rarely in practice

The following are some other hints that may be useful for today:

- When a function is referred to in a manner such as `function(3)`, you can learn more about the function in section 3 of the Unix manual by typing `man 3 function`
- A semicolon on its own is called a "null statement" and it does nothing at all
- `x = y` is an expression that "returns" the new value of `x`
- `v = w = x = y;` is thus equivalent to `x = y; w = x; v = w;`
- The exact data type of an expression is what one would write to declare a variable for it, without the name of the variable itself; for example, the type of a pointer to a function with no formal parameters that returns an `int` is `int (*)(void)`
- The expressions `foo[i]` and `*(foo + i)` are equivalent, and either form can be used to access both fixed and `malloc(3)`'d arrays equally well
- Whenever the name of an array is used in any expression, the array is temporarily converted to a pointer to its first element in a process that is colloquially known as "array decay", except for the expressions `sizeof(array)` and `&array`
- The only two places where `[]` makes sense are in the first dimension of an array declaration, where the compiler fills in the size based on the initialiser, and in the first dimension of a function parameter, where it is syntactic sugar for `*`

Questions marked * are probably beyond the scope of the assessment.

1 Declarations, expressions, and data types

For each of the variables, expressions, and/or snippets of code below, explain their data types, meanings, and/or effects:

a) `[17065012@saeshell01p ~]$ ./foo &`

b) `alias gcc='gcc -g -ansi -Wall -pedantic'`

c) `#define E 2.71828183f`

d) `#define TRIANGLE(b, h) ((b) * (h) / 2)`

e) `#include <header.h>`

f) `#include "header.h"`

g)
```
#ifdef DEBUG
        #define debug(s) printf("%s\n", s)
#else
        #define debug(s)
#endif
```

h)
```
#ifndef HEADER_H
#define HEADER_H
...
#endif
```

i) `int *foo, bar;`

j) `double foo, bar = 42.0;`

k) `unsigned char foo;`

* l) `extern int foo;`

* m) `extern int foo();`

n) `int foo(void);`

o) `int foo(int, int, int);`

p) `int foo(int, int y, int z);`

q) `static int foo(int x) { return x * x; }`

r) `int **foo(void);`

s) `int *(*foo)(void);`

t) `int (**foo)(void);`

u) `typedef int *(*foo)(void);`

* v) `typedef int *(*foo(float (*)(float)))(void);`

w) `int **foo = (int **) malloc(sizeof(int *));`

x) `int *foo; ... if (!foo) ...`

y) `int *foo; ... if (!*foo) ...`

z) `argc; argv; argv[0]; argv[4];`

α) `int *foo = malloc(5 * sizeof(int)); foo; &foo; *foo;`

β) `int *foo = malloc(5 * sizeof(int)); foo[3]; &foo[3];`

γ) `int *foo = malloc(5 * sizeof(int)); foo + 3; *(foo + 3);`

δ) `int foo[5]; foo; &foo; *foo;`

ε) `int foo[5]; foo[3]; &foo[3];`

ζ) `int foo[5]; foo + 3; *(foo + 3);`

η) `int foo[3][2] = { { 4, 5 }, { 6, 7 }, { 8, 9 } };`

ϑ) `int foo[5][4]; foo; &foo; *foo;`

ι) `int foo[5][4]; foo[3]; &foo[3];`

κ) `int foo[5][4]; foo[3][1]; &foo[3][1];`

λ) `int foo[5][4]; foo + 3; *(foo + 3);`

μ) `int foo[5][4]; *(foo + 3) + 1; *(*(foo + 3) + 1);`

ν) `int *foo[5][4], **bar[5][4];`

ξ) `int (*foo)[5][4], (*bar(void))[5][4];`

* ο) `int *(*foo[5][4])(int bar, int);`

π) `int foo(float bar[], size_t length);`

ρ) `int foo(float bar[][COLS], size_t rows);`

σ) `int foo(float bar[ROWS][COLS]);`

τ) `char *foo = "Hello, world!";`

υ) `char foo[] = "Hello, world!";`

2 Discussion

a) Why should you put forward declarations in header files?

b) Which of the following should reside in a header file and why?

   * a) Global variable declaration (not `extern`)

   * b) Global variable declaration (`extern`)

     c) Function declaration (not `static`)

     d) Function declaration (`static`)

     e) Function definition (not `static`)

     f) Function definition (`static`)

     g) `typedef` declaration

     h) `#include` directive

     i) `#include` guard

c) What is the return type of `main()` and why?

d) What is the return type of `malloc()` and why?

e) What is the parameter type of `free()` and why?

f) What is an L-value?

g) What kinds of memory errors can `valgrind` catch?

h) Why is it a good idea to set a pointer to `NULL` after freeing it?

i) If you write a `bash` alias to supply options like `-Wall` to `gcc`, do you still need to specify those options in a `CFLAGS` variable in a Makefile? Why or why not?

j) In the shell script snippet below, will changing the value of `foo` in the third line have any effect on the command line arguments of `bar`? Why or why not?

```
foo=hello
./bar $foo &
foo=goodbye
```

3 Pointer diagrams (no arrays)

Say you have the following declarations:

```
int x = 13;
int y = 42;
int *p = NULL;
int *q = NULL;
int **s = NULL;
int **t = NULL;
```

For each of the following code snippets, draw a diagram showing all pointer relationships created, and state the resulting values of `x` and `y`. Assume that the declarations "reset" between questions — in other words, the questions are independent of one another.

a)

```
int z;
t = &q;
q = &y;
p = &x;
z = x;
x = y;
y = z;
*q += **t - *p;
```

* b)

```
p = (int *) malloc(sizeof(int));
q = (int *) malloc(sizeof(int));
s = t = (int **) malloc(sizeof(int *));
*s = p;
*t = q;
**s = x;
**t = y;
x = **s + **t;
```

* c)

```
int **h, i;
p = &x;
q = &y;
s = &q;
t = &p;
for (i = 0; i < **s / 4; i++) {
        h = s;
        s = t;
        t = h;
}
**s *= 2;
```

4 Pointer diagrams (with arrays)

Say you have the following declarations:

```
int x = 13;
int y[] = { 1, 4, 9 };
int z[][2] = { 2, 4, 6, 8 };
int *p = NULL;
int *q[2] = { &z[0][2], &y[1] };
int **s;
int **t[2];
```

For each of the following code snippets, draw a diagram showing all pointer relationships created, and state the resulting value of `x`. Assume that the declarations "reset" between questions — in other words, the questions are independent of one another.

a)

```
s = &p;
p = &x;
t[0] = &q[1];
t[1] = &q[0];
s[0][0] = **t[0] + **t[1];
```

b)

```
*t = &p;
t[z[1][0] - 5] = q;
p = (int *) malloc(2 * sizeof(int));
*(p + 0) = y[sizeof(z) / sizeof(z[0])];
*(p + 1) = q[0] - *z;
q[0] = &x;
t[1][0][0] = t[0][0][1] * ***t;
```

5 Outputs of C programs

For each of the following C programs, determine what will be sent to `stdout`.

a)

```
#include <stdio.h>
#define ODD(number) (number % 2 == 1)

int main(void) {
        if (ODD(3 + 2))
                printf("odd\n");
        else
                printf("even\n");
        return 0;
}
```

b)

```
#include <stdio.h>

void foo(void *p) {
        * (int *) p *= 2;
        printf("%d\n", * (int *) p);
}

int main(void) {
        int i = 1, j = 1;
        do {
                i++;
                foo((void *) &j);
        } while (i <= 10);
        return 0;
}
```

* c)

```
#include <stdio.h>

double foo(unsigned int x) {
        if (x < 2)
                return 1;
        else
                return foo(x - 1) * (double) x;
}

int main(void) {
        printf("%.0f\n", foo(5));
        return 0;
}
```

6 Error inspection

Each of the following code snippets contains an error. Explain what is wrong. (You don't need to show how to fix it, but you can if it will assist your explanation.)

a)

```
double cube(double x) {
        return x * square(x);
}

double square(double x) {
        return x * x;
}
```

b)

```
void swap(int a, int b) {
        int t;
        t = a;
        a = b;
        b = t;
}
```

c)

```
#include <stdio.h>
#define ADD(x, y) (x + y);

void foo(void) {
        printf("%d\n", ADD(9, 4));
}
```

7 Effects of C functions

Describe the effect of the following functions, using examples as needed:

a)

```
typedef void (*callback)(void *);
void fun0(void *);
void fun1(void *);
void fun2(void *);
void fun3(void *);

callback foo(void) {
        int bar;
        scanf("%d", &bar);
        switch (bar) {
                case 1:
                        return &fun1;
                case 2:
                        return &fun2;
                case 3:
                        return &fun3;
                default:
                        return &fun0;
        }
}
```

b)

```
int foo(void) {
        char bar;
        int baz = 0;
        do {
                scanf("%c", &bar);
                if (bar >= '0' && bar <= '9')
                        baz = (baz * 16) + (bar - '0');
                if (bar >= 'A' && bar <= 'F')
                        baz = (baz * 16) + 10 + (bar - 'A');
                if (bar >= 'a' && bar <= 'f')
                        baz = (baz * 16) + 10 + (bar - 'a');
        } while (
                (bar >= '0' && bar <= '9') ||
                (bar >= 'A' && bar <= 'F') ||
                (bar >= 'a' && bar <= 'f')
        );
        return baz;
}
```

8 Makefiles and dependencies

Consider a chat client and server made up of the following C files:

Listing 1: client.c

```
#include "client.h"
#include "network.h"

...
```

Listing 2: server.c

```
#include "server.h"
#include "network.h"

...
```

Listing 3: network.c

```
#include "network.h"
#define PORT 6697

...
```

Listing 4: client.h

```
#include "interface.h"

...
```

Listing 5: server.h

```
#include <event.h>

...
```

Listing 6: network.h

```
#include <socket.h>

...
```

Listing 7: interface.c

```
#include "interface.h"

...
```

Listing 8: interface.h

```
#include <curses.h>

...
```

Listing 9: common.h

```
#include <stdlib.h>

#define VERSION 4.2.0
```

a) What files would serve as targets in a makefile, and why?

b) What are the dependencies for each target, and why?

c) If the file `interface.h` changes, which files would need to be recompiled, and how does make figure this out?

d) Produce a suitable makefile for this project. Make good use of makefile variables, and include a suitable "clean" rule. The executable files should be named `client` and `server`.

9 Export pointers and function pointers

You've been asked to help write code for a simple video game with an overhead two-dimensional display. The game consists of a grid of positions (x, y) where $1 \leq x \leq 80$ and $1 \leq y \leq 20$. The player's character may reside in any of the 1600 (x, y) pairs, except where there is an object in the way, and the character must "wrap around" to an opposite side when the player tries to move beyond the boundaries provided.

Your task is to write a function called `keyboard` that is called whenever the player presses a key to move their character. The key will be given to you in the form of a `char`, and will be `'h'`, `'j'`, `'k'`, or `'l'` to move left, down, up, or right respectively. Except for wrapping around an edge, either x or y should change by ± 1 when a player successfully moves in any direction.

You may assume that the following constants have already been written:

```
#define WIDTH 80
#define HEIGHT 20
```

Your function must match the following prototype:

```
void keyboard(
        char key,
        void (*location)(int *x, int *y),
        int (*occupied)(int x, int y),
        void (*move)(int x, int y)
);
```

You may assume that the three function pointer parameters will be supplied with valid pointers to functions that satisfy the behaviours below.

`location` exports the player character's current location using the pointers `x` and `y` supplied by the caller. `occupied` returns `1` if the space at (x, y) is occupied, and `0` if the space is not occupied. `move` unconditionally moves the player's character to the space at (x, y), without checking or validating anything about the location given.

If the player presses any key that wasn't mentioned above, your function should do nothing. Otherwise, your function should first check to see whether or not the player is trying to move beyond an edge. Based on that, your function should then check to see whether the target location is occupied. If not, your function should move the player's character to the target location.

10 Memory usage*

For each of the following code snippets, state how much additional memory would be used at runtime if you were to add them to a program. You may assume that none of the functions are being called "right now", so the stack is not affected, and that:

- the size of any `int` is 4 bytes
- the size of any pointer is 8 bytes
- the size of any function is 30 bytes

a)

```
extern int errno;
char *strerror(int errnum);

int foo(int bar) {
        if (some_syscall(bar) == -1) {
                fprintf(stderr, "%s\n", strerror(errno));
                return errno;
        }
        return 0;
}
```

b)

```
extern int foo;
int *bar;
int *baz(void);
int *qux(void) {
        ...
}
int (*bat)(void);
typedef int (*cat)(void);
```