Ellie Makin (erm67)

C/C++ supervision work 1

Lecture 1

1. **What is the difference between `'a'` and `"a"`?**

`'a'` is a character, and has type `char`. It simply contains the ASCII value for the letter ‘a’. `"a"` is a string literal, which will be stored as a null-terminated `char` array.

2. **Will `char i, j; for(i=0; i<10,j<5; i++, j++) ;` terminate? If so, under what circumstances?**

This loop will always terminate:

Firstly, the comma in the test expression of the for loop means that the loop is actually equivalent to just `for (i=0; j<5; i++,j++);`.

Secondly, `j` is uninitialised in this code, so it will assume an indeterminate (supposedly) value within the range $[-128, 127]$, but whatever its initial value, it will eventually fail `j<5`, and the loop will exit.

3. **Write an implementation of bubble sort for a fixed array of integers. (An array of integers can be defined as `int i[] = {1,2,3,4};` the 2nd integer in an array can be printed using `printf("%d\n", i[1]);`.)**

```c
void bubbleSort(int a[], int length)
{
    int swapped = 1;    // C doesn't have bool!
    while (swapped)
    {
        swapped = 0;
        for (int j = 0; j < length-1; j++)
        {
            if (a[j] > a[j+1])
            {
                int temp_int = a[j+1]; a[j+1] = a[j]; a[j] = temp_int;
                swapped++;
            }
        }
    }
}
```

4. **Modify your answer to (3) to sort an array of characters into alphabetical order. (The 2nd character in a character array `i` can be printed using `printf("%c\n",i[1]);`.)**

```c
void bubbleSort(char a[], int length)
{
    int swapped = 1;
    while (swapped)
    {
        swapped = 0;
        for (int j = 0; j < length-1; j++)
        {
            if (a[j] > a[j+1])
            {
                char temp_char = a[j+1]; a[j+1] = a[j]; a[j] = temp_char;
                swapped++;
            }
        }
    }
}
```

Lecture 2

1. **Write a function definition which matches the declaration `int cntlower(char str[]);`. The implementation should return the number of lower-case letters in a string**

```c
int cntlower(char str[])
{
    int count = 0;
    int jChar = 0;
    char c;
    while (c = str[jChar], c != '\0')
    {
        if ('a' <= c && c <= 'z') {
            count++;
        }
        jChar++;
    }
    return count;
}
```

2. **Use function recursion to write an implementation of merge sort for a fixed array of integers; how much memory does your program use for a list of length n?**

```c
void subMergeSort(int a[], int b[], int start, int end)
{
    if (end - start <= 1)
        return;

    int split = (start + end)/2;
    subMergeSort(a, b, start, split); subMergeSort(a, b, split, end);

    // copy first half to temp array
    for (int j = start; j < split; j++)
    {
        b[j] = a[j];
```

```
    }

    int j0 = start, j1 = split, i = start;
    while (1)
    {
        if (j0 < split && j1 < end)
        {
            if (b[j0] < a[j1])
                a[i++] = b[j0++];
            else
                a[i++] = a[j1++];
        }
        else if (j0 < split)
            a[i++] = b[j0++];
        else if (j1 < end)
            a[i++] = a[j1++];
        else
            return;
    }
}

void mergeSort(int a[], int length)
{
    int b[length];    // temp array for merging
    subMergeSort(a, b, 0, length);
}
```

This implementation uses n extra memory to store the temporary array of size n. I could have made it $n/2$ but the indices are less convenient.

3. **Define a macro `SWAP(t,x,y)` that exchanges two arguments of type `t`**

```
#define SWAP(t, x, y) t swap_temp = x; x = y; y = swap_temp;
```

4. **Does your macro work as expected for `SWAP(int, v[i++], w[f(x)])`?**

No. In this case, since the preprocessor copies the text into two places, `i++` and `f(x)` are both executed twice, so `i` is doubly incremented, and if `f(x)` has any side effects they may be doubled up.

5. **Define a macro `SWAP(x,y)` that exchanges two arguments of the same integer type (e.g. `int` or `char`) *without using a temporary***

```
#define SWAP(x, y) {\
    x -= y;         \
    y += x;         \
    x = y - x;      \
}
```

6. **What is the effect of `SWAP(*p,*q)` when `p==q`?**

The value pointed to by `p` and `q` is set to zero in the first line of the macro, since they both dereference to the same variable. It's equivalent to `SWAP(a, a)` where `p = q = &a`.

Lectures 3 and 4

1. **If `p` is a pointer, what does `p[-2]` mean? When is this legal?**

`p[-2]` is equivalent to `*(p - 2)`. The exact address depends of `sizeof(*p)`. This yields undefined behaviour unless `p` already points to the middle of a defined block of memory like a struct or array, where we know that `p - 2` is still in that block.

2. **Write a string search function with a declaration of `const char *strfind(const char *needle, const char *hay)` which returns a pointer to first occurrence of `needle` in `hay` (and `NULL` otherwise). (You are not expected to implement Boyer-Moore algorithm but you might find it of general interest.)**

```
const char *strfind(const char *needle, const char *hay)
{
    const char *pStart = hay;
    while (*pStart != '\0')
    {
        int offset = 0;
        while (*(needle + offset) != '\0' && *(pStart + offset) != '\0')
        {
            if (*(needle + offset) != *(pStart + offset))
                break;

            offset++;
        }

        if (*(needle + offset) == '\0')
            return pStart;

        pStart++;
    }
    return NULL;
}
```

3. **If `p` is a pointer to a structure, write some C code which uses all the following code snippets: "`++p->i`", "`p++->i`", "`*p->i`", "`*p->i++`", "`(*p->i)++`" and "`*p++->i`"; describe the action of each code snippet.**

```
#include <stdio.h>

struct s {
    struct t **i;
};

struct t {
    struct s m;
    struct s n;
```

```
    struct s o;
};

int main()
{
    struct t e = {
        { NULL }
    };

    struct t *pe = &e;

    struct t f = {
        { &pe },
        { &pe },
        { &pe }
    };

    struct s *p = &f.m;

    printf("%p\n", ++p->i);     // incr p->i and return it
    printf("%p\n", p++->i);     // return p->i and incr p
    printf("%p\n", *p->i);      // return deref p->i
    printf("%p\n", *p->i++);    // return deref p->i and incr p->i
    printf("%p\n", (*p->i)++);  // return deref p->i and incr deref p->i
    printf("%p\n", *p++->i);    // return deref p->i and incr p

    return 0;
}
```

4. **Write a program `calc` which evaluates a reverse Polish expression given on the command line; for example `$ calc 2 3 4 + x` should print `14` We use 'x' for multiply since asterisk will be expanded by the shell into a file list.**

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

#define OPERATION(pStack, op) { \
    int arg2 = pStack->top;     \
    pStack = pStack->next;      \
    int arg1 = pStack->top;     \
    pStack->top = arg1 op arg2; \
}

struct stack {
    int top;
    struct stack *next;
};

int calc(char *strings[], int n)
{
    struct stack *ps = NULL;
    for (int j = 1; j < n; j++)
    {
        char firstChar = *strings[j];
        switch (firstChar)
        {
```

```
            case '+':
                OPERATION(ps, +)
                break;
            case '-':
                OPERATION(ps, -)
                break;
            case '/':
                OPERATION(ps, /)
                break;
            case 'x':
                OPERATION(ps, *)
                break;
            default:
            {
                struct stack* temp_stack = ps;
                ps = (struct stack *) malloc(sizeof(struct stack));
                ps->top = atoi(strings[j]);
                ps->next = temp_stack;
            }
        }
    }
    return ps->top;
}

int main(int argc, char *argv[])
{
    int result = calc(argv, argc);
    printf("%d\n", result);
}
```

5. **What is the value of `i` after executing each of the following on your laptop (and which might vary on a 2015 vintage Raspberry Pi?)**

a. `i = sizeof(char);`

`i = 1`

b. `i = sizeof(int);`

`i = 4`

c. `int a; i = sizeof a;`

`i = 4`

d. `char b[5]; i = sizeof(b);`

`i = 5`

e. `char *c=b; i = sizeof(c);`

`i = 8`

f. `struct {int d;char e;} s; i = sizeof s;`

`i = 8`

g. `void f(int j[5]) { i = sizeof j;}`

`i = 8`

h. `void f(int j[][10]) { i = sizeof j;}`

`i = 8`

b, c, e, f, g, h are all likely to be different on a Raspberry Pi.

6. **Write a program that adds up all of the characters in a large file, treating them as unsigned 8-bit quantities. The program should print the total.**

```c
#include <stdio.h>

#define BUFFER_SIZE 0xFFFF

int main()
{
    FILE *pFile;
    char buf[BUFFER_SIZE];

    pFile = fopen("example.txt", "r");

    char total = 0;
    while (1)
    {
        char *flag = fgets(buf, BUFFER_SIZE, pFile);
        if (flag == NULL)
        {
            break;
        }

        for (int jChar = 0; jChar < BUFFER_SIZE-1; jChar++)
        {
            if (buf[jChar] == '\0')
                break;
            else
                total += buf[jChar];
        }
    }

    fclose(pFile);

    printf("%d\n", total);

    return 0;
}
```

7. **If you used `fopen` or `read` for the previous exercise, do it again, but this time use `mmap`. Or vice versa. Compare the performance of the two programs (e.g. simply bracket it as follows `date; ./mprog; date` or `time ./a.out`). Is the performance a linear function of file**

size? What is the performance of your file system, in MByte per second? Does it make a difference if the program is run again just after it has been run on the same file or the file is accessed twice in the same run of a program?

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>

int main()
{
    int filedesc = open("example.txt", O_RDONLY);

    struct stat filestats;
    fstat(filedesc, &filestats);

    size_t filesize = filestats.st_size;

    char *region = mmap(
        NULL,
        filesize,
        PROT_READ,
        MAP_FILE | MAP_PRIVATE,
        filedesc,
        0
    );

    unsigned char total = 0;
    for (int jChar = 0; jChar < filesize; jChar++)
    {
        if (region[jChar] == '\0')
            break;
        else
            total += region[jChar];
    }

    close(filedesc);

    printf("%d\n", total);

    return 0;
}
```

On a 50,000,000 byte file of random characters, the results from `time` for each program is roughly as follows:

```
$ time ./fopen
119 // program output

real    0m0.190s
user    0m0.170s
sys     0m0.020s

$ time ./mmap
```

```
119 // program output

real    0m0.160s
user    0m0.155s
sys     0m0.007s
```

Doubling the file size gives a time slightly less than double the original time for `fopen`, and even less for `mmap`.

Using a script to run each program 3 times in succession, it seems that both programs run slightly faster after the first run, although moreso for `mmap` than `fopen`.

Repeating the main code body of the program twice seemed to have no effect other than doubling the execution time.

8. **Give alternative C code that does not use square brackets for each of the following expressions: `A[b]`, `b[A]`, `A[b, c]` and `A[b][c]`, Which form is suitable for a *jagged array*? [Hint: don't forget that C has the comma expression list.]**

```
A[b]    == *(A+b)
b[A]    == *(A+b)
A[b, c] == *(b, A+c)
A[b][c] == *(*(A+b)+c)  // this can be used for jagged arrays
```