Lab 4 – Pointer basics

Lab targets: pointers in C, parameter passing, understanding the function stack in C.

Exercise: *parameter passing*
The following code is incorrect. Explain why?

```
typedef struct {
        short day, month;
        unsigned year;
} Date;

void DateStruct( int day, int month, int year, Date *date) {
        Date dummy;
        dummy.day = (short)day;
        dummy.month = (short)month;
        dummy.year = (unsigned)year;
        date = &dummy;
}

int main( void ) {
        int day, month, year;
        Date d;
        printf("\nGive day, month, year:");
        scanf("%d %d %d", &day, &month, &year);
        DateStruct( day, month, year, &d);
        printf("\ndate struct values: %d-%d-%d", d.day, d.month, d.year);
        return 0;
}
```

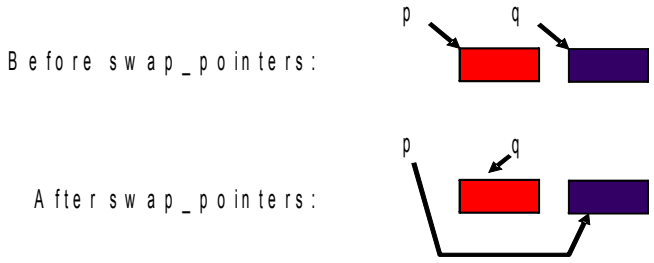
And what if we rewrite the code such that the function DateStruct returns a pointer to Date?

```
typedef struct {
        short day, month;
        unsigned year;
} Date;

void f( void ) {
        int x, y, z;
        printf("%d %d %d\n", x, y, z );
}

Date * DateStruct( int day, int month, int year ) {
        Date dummy;
        dummy.day = (short)day;
        dummy.month = (short)month;
        dummy.year = (unsigned)year;
        return &dummy;
}

int main( void ) {
        int day, month, year;
        Date *d;
        printf("\nGive day, month, year:");
        scanf("%d %d %d", &day, &month, &year);
        d = DateStruct( day, month, year );
        //f();
        printf("\ndate struct values: %d-%d-%d", d->day, d->month, d->year);
        return 0;
}
```

Exercise: *parameter passing*

Implement the function 'swap_pointers'. This functions takes two arguments of type void pointer and has no return value. The functions 'swaps' the two pointers as illustrated below.

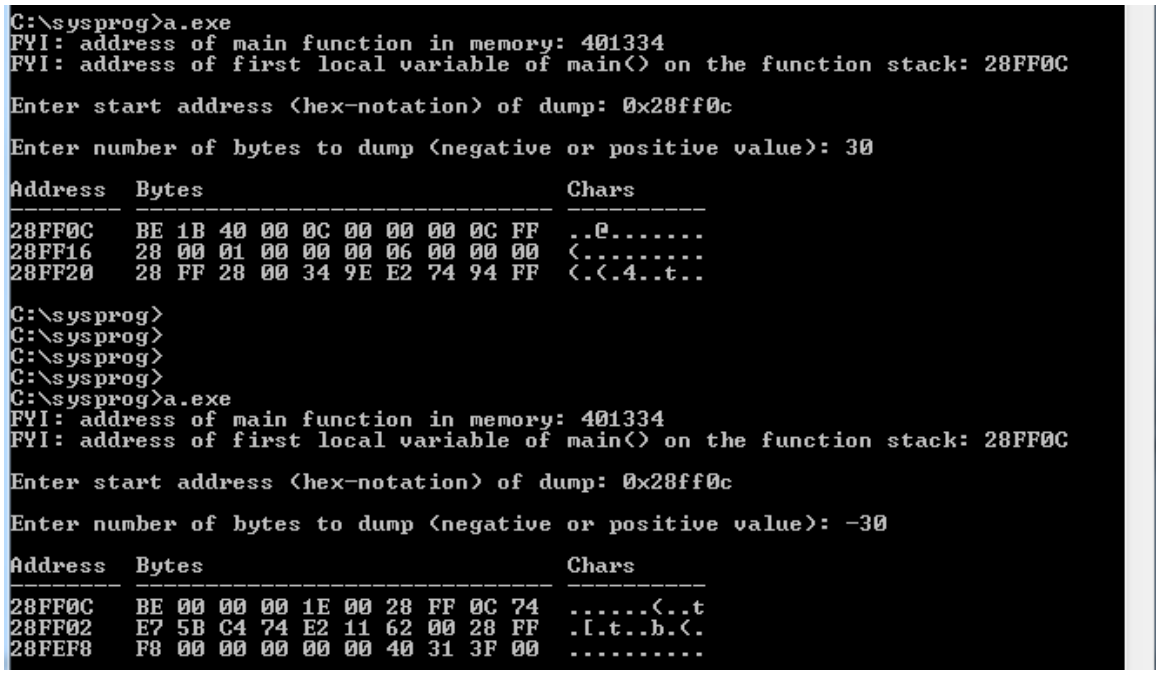
```
int a = 1;
int b = 2;
// for testing we use pointers to integers
int *p = &a;
int *q = &b;

printf("address of p = % and q = %p\n", q, p);
        // prints p = &a and q = &b

swap_pointers( ?p , ?q );
        // find the correct parameter passing of pointer p and q
```




```
printf("address of p = %p and q = %p\n", p, q);
        // prints p = &b and q = &a
```

Exercise: *pointers and memory access*

Implement a small but powerful tool (memdump) to print pieces of computer memory. The tool needs a starting memory address and the memory size (= number of bytes to print) as input. The tool then prints a memory dump (each byte is printed in hexadecimal and as characters) as is illustrated in the following screen shot:



The memdump tool prints a matrix on screen consisting of 3 columns and a number of rows depending on the memory size input parameter. The first column contains the address of the first byte printed in the second column. The second column prints a sequence of bytes in hexadecimal format, starting from the address mentioned in the first column. The number of bytes that will be printed on 1 row should be set as a constant in the program. In the above example this constant is set to 10. The third column prints again all bytes from the second column but this time as characters. A '.' should be printed if a byte is not printable. Of course, it is possible that the last row is not a 'full' row anymore; spaces can be used to stuff that row.

The memory size input parameter can be a positive or negative number. The screen shot above illustrates what happens in both situations.

When the memdump tool is started, it prints the memory address of the main function and the memory address of the first local variable of the main function. This information is only to help the user to input an interesting memory address, but is strictly spoken not needed to work with this tool.

Extension to this exercise: *command-line arguments*

Extend the previous exercise such that the start address and memory size input can be provided as command-line arguments to the memdump tool. E.g., the command:

> memdump 0x28ff1C -30

should print the 30 bytes before the start address 0x28ff1C.

Of course, when command-line argumnts are provided, the memdump tool should not ask the user anymore to input the start address and memory size paramters.

Exercise: *function stack*

You can use the 'memdump' tool from the previous exercise to solve this exercise.

Write code to obtain more information on the organization of the function stack in memory, i.e. find answers on questions such as:

- Does the function stack grow ‘up’ or ‘down’ in memory?
- In which sequence are function arguments stored on the system stack (left to right or right to left)?
- What’s the size on the stack of an array argument?

You can do this by calling functions with or without arguments, and with or without local variables. Store values in these variables that can be easily recognized, e.g.:

```
int a = 0xFFFFFFFF; // assuming 4 byte int
int b = 0xEEEEEEEE;
int c = 0xDDDDDDDD;
```

[After your own investigation, read http://unixwiz.net/techtips/win32-callconv-asm.html *to reveal the details.]*