Worksheet 3: Pointers

Updated: 28th July, 2015

The objective of this practical is to explore the use of pointers, including referencing and dereferencing and pointers to functions.

Pre-lab Exercises

1. Pointer Problems

There is an error in each of the following. What is it?

(a)
```
int value = NULL;
```

(b)
```
int* pointer;
pointer = 42;
```

(c)
```
int* pointer;
*pointer = 42;
```

(d)
```
char ch;
char** pointer;
pointer = &ch;
```

(e)
```
double value = 42.0;
void* pointer = (void*)&value;
*pointer = 84.0;
```

(f)
```
void (*pointer)(int, int);
(*pointer)(5, 10);
```

(g)
```
int function1(void) {
    return 42.0;
}

void function2(void) {
    void (*pointer)(int);
    pointer = &function1;
    ...
}
```

2. Referencing and Dereferencing

Say you have these declarations:

```
int a = 2;
int b = 10;
int* x = NULL;
int* y = NULL;
int** s = NULL;
int** t = NULL;
```

What are the values of `a` and `b` after each of the following? In each case, draw a small diagram showing which variables point to which other variables. (Assume the variables are "reset" after each example.)

(a)
```
x = &a;
*x = b;
```

(b)
```
s = &x;
x = &a;
y = &b;
t = s;
**t = **t * *y;
```

(c)
```
int i;
x = &a;
y = &b;
for(i = 1; i <= *x; i++)
{
    *y = *y * 2;
}
```

(d)
```
x = &b;
y = &a;
t = &y;
*t = x;
if(**t > 5) {
    s = t;
}
else {
    s = &x;
}
**s = **s / *x;
```

(e)
```
x = (int*)malloc(sizeof(int));
y = (int*)malloc(sizeof(int));
*x = 8;
```

```
*y = 8;

if(x == y) {
    a = *x;
}

if(*x == *y) {
    b = *y;
}
```

(f)
```
s = (int**)malloc(sizeof(int*));
*s = (int*)malloc(sizeof(int));

t = s;
x = *t;
**t = b;

s = (int**)malloc(sizeof(int*));
*s = &a;
**s = *x;
```

3. Miscellaneous Questions

(a) Why does `scanf()` produce a segmentation fault when you pass it an `int` without the `&`?

(b) If you have an `int` *pointer*, why would it be wrong to put a `&` in front of it when passing it to `scanf()`?

(c) Without any other information, can you determine what type of value is stored at the address of a void pointer?

(d) Can an int pointer be equal to a char pointer?

Practical Exercises

1. Valgrind

Obtain a copy of `memerrors.c` and compile it. (Don't look at the source code yet.)

Run it as follows:

```
[user@pc]$ ./memerrors 1
[user@pc]$ ./memerrors 2
[user@pc]$ ./memerrors 3
[user@pc]$ ./memerrors 4
[user@pc]$ ./memerrors 5
```

What errors do you see, if any?

Now run it with valgrind, as follows:

```
[user@pc]$ valgrind ./memerrors 1
[user@pc]$ valgrind ./memerrors 2
[user@pc]$ valgrind ./memerrors 3
[user@pc]$ valgrind ./memerrors 4
[user@pc]$ valgrind ./memerrors 5
```

Valgrind should find a different error in each case. Use the valgrind output to identify the nature of each error.

Note: When valgrind detects a memory *leak*, it will advise you to "Rerun with --leak-check=full to see details of leaked memory":

```
[user@pc]$ valgrind --leak-check=full ./memerrors ...
```

This will give more information to help you find the leak. (However, it won't help for other types of memory errors.)

Finally, re-compile `memerrors.c` using the `-g` switch to turn on debugging information:

```
[user@pc]$ gcc -g memerrors.c -o memerrors
```

This will allow valgrind to give you the actual line number at which an error occurs. Re-run the above valgrind commands and identify the location of each error in `memerrors.c` (including function name and line number).

Note: When it finds an error, valgrind will output the *call stack*. This shows the function currently executing, the function that called it, the function that called that one, and so on. Source files and line numbers are also given (if you compiled with `-g`).

Note: Valgrind may also output a second call stack, showing where a block of memory was previously freed.

Open `memerrors.c` in an editor. Find the lines that valgrind identifies, and determine the coding error.

2. Passing by Reference

Note: The remainder of this practical concerns a single multi-file C program. As you go, you should construct appropriate header files and an appropriate Makefile.

Create a file called `order.c` (and its associated header file).

Inside, create a static function called `ascending2()` that takes two `int` *pointers* and returns `void`. The function should place the smaller of the two `int` values in the first memory location and the larger in the second.

In other words, `ascending2()` should swap its parameters if the first is larger than the second, but do nothing otherwise.

Write another function called `ascending3()` that takes three `int` pointers and returns void. Similarly to `ascending2()`, the function should place the smallest value into the first memory location, the median value into the second location and the largest value into the third.

Note: It may be useful to call `ascending2()` from `ascending3()`.

Create a third function called `descending3()`, similar to `ascending3()` but placing the `int` values in the reverse order.

Create a file called `numbers.c` containing a `main()` function. Include appropriate testing code for the functions you've just created.

3. Scanf with Pointers

Create a file called `user_input.c`.

Inside, write a function called `readInts()` that reads three `int`s and one `char` and exports them to the calling function.

The function should obtain these values from the user. The user should be asked to enter the three `int`s, one after another, and then enter either "A" or "D".

Note: Use "` %c`" (with a space before the `%`) in the scanf string to read a single character.

Modify your existing `main()` function from the previous question so that it:

1. calls `readInts()`,
2. passes the three `int`s by reference to `ascending3()`, and
3. outputs the result.

(For the moment, don't do anything with the input character.)

4. Pointers to Functions

Back inside `order.c`, define another function called `order()`. This function should take a single `char` as a parameter, and return a function pointer. When passed "A", return a pointer to `ascending3()`. For "D", return a pointer to `descending3()` instead. If the parameter is neither "A" nor "D", return `NULL`.

To help simplify your code, write a `typedef` declaration for the function pointer type, placing it in the appropriate header file.

Modify your `main()` function again to make use of `order()`. Rather than calling `ascending3()` directly, you should use the function pointer returned by `order()`.

Recap

Based on all of the above, your program should do the following:

1. Read three `int`s from the user.
2. Ask the user whether to arrange them in ascending or descending order (by entering either "A" or "D").
3. Output the numbers in the desired order.

You should have split your code into three .c files — `numbers.c`, `user_input.c` and `order.c` — with appropriate header files — `user_input.h` and `order.h`. You should also be using a Makefile to compile and link the program.

End of Worksheet