ENSF 337: Programming Fundamentals
Lab 10 - Thursday 29, 2018

This lab contains material written by Dr. S. Norman and Dr. Moussavi for ENCM 339 in previous year

Note: This lab will not be marked. However, since questions similar to the exercises in this lab may appear in the final exam, you are strongly advised to complete this lab.

Solutions for this lab will be posted on the D2L on Wednesday Dec 5th

Objectives:

The objective of this lab is to help ENSF 337 students in understanding the concepts of:

- Pointer to pointers
- Command line arguments
- Recursive functions

Exercise A: Array of Pointers and Command Line Arguments

Read This First

Up to this point in our C and C++ programs, we have always used main functions without any arguments. In fact both languages, allow us to write programs that their main functions receives arguments. Here is the heading of such a function:

```
int main(int argc, char **argv){ ... }
```

Of course, we never call a main function from anywhere in our program, and we never pass arguments to it. A main function can only receive its arguments from command line. In other words, when a user runs the program from command-line, he/she can pass one or more arguments to the main. Good examples of this type of programs are many of the Linux and Unix commands such as: `cp, mv, gcc, g++`. For example, the following `cp` command receives the name of two files, to make file `f.dat` a copy of file `f.txt`:

```
cp f.txt f.dat
```

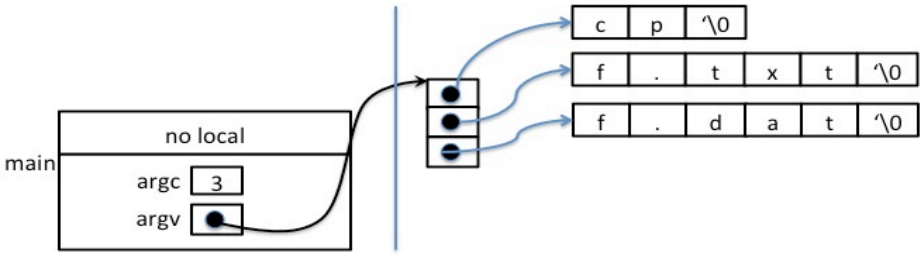
The first token in the above `cp` command is the program's executable file name, followed by two file names. This information will be accessible to the `main`. How does it work? Here is the answer:

To access the command-line arguments, a C/C++ main function can have arguments as follows:

```
#include <iostream>
using std::cerr;

int main(int argc, char **argv) {
    if(argc != 3) {
        cerr << "Usage: incorrect number of argument on the command line...\n"";
        exit(1);
    }

    // MORE CODE AS NEEDED.

    return 0;
}
```

Where, `argc` is the number of tokens on the command-line. In our `cp` command example, the value of `argc` should be always 3 (one for program name and two for the file names). If the value of `argc` is not 3, the program terminates after giving an appropriate message. In other words, this argument is mainly used for error-checking to make sure user has entered the right number of tokens on the command line. The delimiter to count for the number of tokens on the command-line is one or more spaces.

The second argument is a pointer-to-pointer, which points to an array of pointers. Each pointer in this array points to one of the string tokens on the command line. The following figure shows how `argv[0], argv[1],` and `argv[2]` point to the tokens on the command line:



The following code shows how you can simply access and display the command line strings in a C++ program:

```
cout << "The program name is: " << argv[0] << endl;
cout << "The first string on the command line is: " << argv[1] << endl;
cout << "The second string on the command line is: " << argv[2] << endl;
```

And, here is the output of this code segment for our `cp` command example:

```
The program name is: cp
The first string on the command line is: f.txt
The second string on the command line is: f.dat
```

The exact location of the memory allocated for command-line arguments depends on the underlying OS and the compiler, but for most of the C/C++ systems it is a special area on the stack that is not used for the activation records.

Read This Second

Since this part of exercise A deals with the command line entries by the user, you are **strongly** recommended to implement, compile and run the program on a Cygwin terminal.

You may be interested in knowing whether it is possible to pass command-line arguments to your program from inside an IDE environment such as Visual C++ or XCode. The answer is yes! It is possible.

Most of the commonly used IDEs provide some way of entering the command line arguments into C/C++ project. For example, in the newer versions of the XCode for the Mac computers, you can click on the 'Edit Scheme', under the 'Product' menu option, and enter the command line argument strings on the 'arguments' tab. A similar way is also available in Visual Studio: Right-click on the project, then select 'Properties' -> Debugging, and enter your arguments into the arguments box. If you choose to use this option, you should seek for more details by consulting the help options available on your target IDE, or search on the Internet.

Our experience shows, using the command line argument from inside some of the IDEs is not always straightforward and figuring out how exactly it works, can be sometimes time-consuming.

What to Do:

Download the file `lab10exe_A.cpp` from D2L. This is a simple program that uses the insertion-sort algorithm to sort an array of integer numbers. Now, you should take the following steps:

1. Read the given program carefully to understand what it does.

2. If you are working from ICT lab (which is recommended), compile the program from Cygwin terminal using the following command to create an executable file called sort:

   ```
   g++ -Wall lab10exe_A.cpp -o sort
   ```

3. Run the program using the following command:

   ```
   ./sort
   ```

   It should print the list of several integer numbers, followed by the same list, sorted in ascending order.

4. Now change the value of the local variable `sort_order` from 1 to 2, recompile the program, and run it again. Now it should sort the array in descending order.

5. Your task from this point is to modify the `main` function to have access to the command line arguments. We want to be able to run the program with the options of sorting the numbers either in ascending or descending order, based on the user's input on the command-line.

   Considering that the program's executable name is sort, users must be able to run the program from the command-line in one of the following formats:
   `./sort  -a`

   Or:
   `./sort  -d`

   In the first command the option `-a` (no spaces between dash and a) stands for the ascending, and the second command with the option `-d` stands for the descending order.

   Depending on the user's selection of one of these options, the program must sort an array accordingly.

   To be more precise, the main function in this program must check the following conditions:
   - If `argc > 2` the program should give the following message and terminate:
     `Usage: Too many arguments on the command line`

   - If `argc == 1`, (meaning that the user didn't enter any of the two options), the program should use ascending sort as its default order.

   - If `argv[1],` is NOT one of the following: `-a, -A, -d,` or `-D`, the program should give the following message and terminate:

     `Usage: Invalid entry for the command line option.`

   Then, the program should set the value of `sort_order` to:
   - `1`, if `argv[1]` points to: "`-a`" or "`-A`".
   - `2`, if `argv[1]` points to: "`-d`" or "`-D`".

Note: as you should recall from material discussed earlier in the course, you cannot compare C-strings by simply using relational operators. Therefore, you need to call C library function `strcmp`. You may refresh your memory regarding this function by reading your course notes, any of your textbooks, or an online sources such as cplusplus.com at: http://www.cplusplus.com/reference/cstring/strcmp.

Exercise B: Using Pointer to Pointers

In this exercise you will write code to complete the definition of a global function in a C++ program that uses array of pointers and pointer to pointers.

What to Do:

Download file `lab10exe_B.cpp` from D2L. This file contains a main function and two other global functions. The implementation of one of these functions called, `append_strings`, is incomplete and your task in this exercise is to read the given function interface comment and complete its definition. Here is the prototype of this function:

```
void append_strings (const char** string_array, int n, char** appended_string);
```

The first argument in this function is required to point to an existing array of c-strings with n element. The second argument is a pointer that holds the address of another pointer declared in the calling function (in this exercise the main function).

The function is expected to create a C-string dynamically, from existing C-strings in string_array. Please notice this function is not supposed to return anything -- it is a void function.

Exercise C: Understanding Recursion

Read This First:

Mark Allen Weis the author of *Efficient C Programming*, has listed four fundamental rules of recursion:

1. *Base cases.* There must always be at least one case where the function can do its job without making a recursive call.
2. *Always make progress.* When a recursive call is made, it must make progress toward a base case.
3. *Believe that recursive calls work.* You should be able to convince yourself that a recursive function is correct without tracing execution all the way down to the base case.
4. *Some forms of recursion are terribly wasteful.* Badly designed recursive algorithms can do an enormous amount of redundant work.

Rules 1 and 2 must both be satisfied in order to ensure that infinite recursion does not occur.

Rule 3 is important. If you are trying to design a recursive function to solve a problem, you should think about how solving the problem involves solving a simpler problem of the same kind; you should *not* have to visualize all the activation records that will be required when the function runs.

A recursive function for computing a sequence of numbers known as Fibonacci numbers is the classic example for Rule 4.

What to Do:

Download the file `reverse.cpp` from D2L. Compile it and run it to see what it does.

Read the definition of `print_backwards`. Does it satisfy the first two fundamental rules of recursion?

The program passes through `point one` several times. Draw a memory diagram for `point one` when `s` is pointing at the `'I'` in `"AEIOU"`. Show the stack and the string constant `"AEIOU"`.

Hint: Do not attempt to work backward from the point in time when `s` is pointing at the `'I'` in `"AEIOU"`. Instead, work forward from the first call to `print_backwards`. Start with a blank piece of paper. Draw the activation record for `main`

at the bottom, and draw the string constant "AEIOU" in the static storage area. Then trace through the program, repeatedly updating your diagram. Stop when you pass through point one and s points to the 'I' in "AEIOU".

Submit your diagram as part of your in-lab report.

Exercise D: A simple problem in writing recursive code

This is an in-lab exercise

Download file lab10exe_D.cpp from D2L. Compile and run the program to understand how simple function sum_of_array works. Now your job is to replace the given implementation of of function sum_of_array with a recursive function (there may not be any loops in your function definition).

Exercise E: A slightly harder example

Write a function definition for the following function interface:

```
int strictly_increasing(const int *a, int n);
// REQUIRES
//   n > 0, and elements a[0] ... a[n-1] exist.
// PROMISES
//   If n == 1, return value is 1.
//   If n > 1, return value is 1 if
//      a[0] < a[1] < ... < a[n-1]
//   and 0 otherwise.
```

Your solution must rely on recursion - there may not be any loops in your function definition.

Exercise F: Largest element in an array

Make a copy of the file largest.cpp.

Read the definition of largest_element. Does it satisfy the first two fundamental rules of recursion?

Here is the reasoning behind the recursive design of largest_element:

- The largest value in a sub-array with only one element is simply the value of that element.
- If a sub-array has two or more elements, you can find the largest value by splitting the sub-array into two smaller sub-arrays, finding the largest value in each smaller sub-array, and then comparing the two maximums.

Trace through the execution of the program by hand. Make a memory diagram for point one when begin is 1 and past_end is 2.

Hint: Because largest_element makes *two* recursive calls, tracing its execution is not a simple matter of working down to the base case and then returning to the original call. The base case will be reached several times--once with begin == 0 and past_end == 1, once with begin == 1 and past_end == 2, and so on.

Exercise G: Raising a number to an integer power

Download the file power.cpp. Build an executable. Run the program, using large integers such as 100, 200, and 300 as values for n.

(By the way, unlike `pow1` and `pow2`, the library function `pow` can be used with a second argument that is not an integer.)

Note the following facts:

- The global variable `mult_count` has nothing to do with generating the result of the exponentiation computation; it is used only to compare the efficiency of `pow1` with that of `pow2`.
- `pow2` is much more efficient than the simpler `pow1`: many fewer multiplications are required by `pow2`.
- The four different methods give slightly different answers. This reflects the approximate nature of floating-point computation.

The function `pow2` is based on the formula

$$x^n = \begin{cases} 1 & x = 0 \\ x & x = 1 \\ (x^{n/2})^2 & x > 1,\ x \text{ is even} \\ x\,(x^{(n-1)/2})^2 & x > 1,\ x \text{ is odd} \end{cases}$$

Progress toward a base case is rapid: each recursive call cuts the exponent roughly in half.
Suppose the value of `k` entered by the user is 97.

- Draw a memory diagram for `point one` when the argument `n` has a value of 24. The third fundamental rule of recursion will be useful here--since you know what the return value of `pow2` is supposed to be, you don't have to trace execution all the way down to the base case.
- With what arguments does `pow2` get called? You can check your answer by putting a `cout` statement above the `switch` statement in the function definition and re-compiling the program.

Exercise H: A class with recursive member functions

Member functions of C++ classes can be recursive. A typical use of recursion is to create a recursive `private` function to act as a helper for some other member function.

Download files `lab10exe_H.cpp, OLList2.cpp and OLList2.h`. Read the files--you will see that this code presents a new variation of the `OLList` class.

Find `point one` within the definition of `OLList::destroy_sublist`. Make a memory diagram for `point one` when `sublist->item` has a value of 250. (Make up and use a suitable graphical notation for showing deleted nodes in the free store.)

Now implement `copy` and `copy_sublist` so that the calls to the `OLList` copy constructor in `main` do the correct thing. You must implement `copy_sublist` recursively--no loops are allowed.