Lab: Pointers
CSC 161, “Imperative Problem Solving and Data Structures”
Department of Computer Science · Grinnell College
March 4, 2016

Pairs

- Mattori Birnbaum and Jae Hong Shin
- Colin Greenman and Gemma Nash
- Josh Lavin and Dennis Chan
- Ella Nicolson and Lex Martin
- Zachary Susag and Faizaan Ali
- Yuyin Sun and Tyler Williams
- Lilly Webster and Erhaan Ahmad
- Sanjay Sudhir and Ying Zhang
- Tanner Tufto and Anna Blindermann
- Jong Hoon Bae and Sophie Gaschott
- Cory McCartan and Eli Salm

Saung Thuya will fill in for anyone who is absent.

Exercises

Exercise 000: Write, compile, and test a C program that declares and initializes five variables and then prints their addresses and values. The variables should be of types double, int, char, bool, and int * (“pointer to int”) respectively. You may choose the initializing expressions of the first four variables however you like, but the fifth variable should be initialized so as to make its value a pointer to the storage location for the second variable.

Note that the “placeholder” for any pointer value in printf’s format string is %p, which causes the pointer to be printed as a base-sixteen numeral.

Exercise 001: After examining the output of the program in the preceding exercise, draw a diagram showing how the variables in question are arranged in memory and how many bytes of storage are reserved for each one.

Exercise 010: Add another call to printf to your program so that it also displays the int value stored in the location to which your pointer variable points. (Use the “indirection” operator * to “reach through” the pointer to recover this value.)

Changes within a program that may seem unimportant to the programmer can change how the memory that the program uses is allocated and arranged. The compiler will try to arrange memory for optimal performance, by (for instance) locating variables that occupy two, four, or eight bytes so that their addresses are evenly divisible by 2, 4, or 8 respectively.

Exercise 011: Rearrange the declarations in your program so that the variable of type char is declared and initialized first, then recompile and execute it. Draw another diagram of the memory locations. How do the alignment requirements show up?

One implication of such alignment rules is that a program that appears to work correctly, but in fact overruns the end of an array and stores values into the region adjacent to the array, can suddenly stop working when a new variable declaration is added or the order in which variables are declared is changed, removing an “alignment gap” between the end of an array and the storage assigned to the adjacent variable.
Exercise 100: Add a declaration for a second pointer-to-int variable to the program you wrote for the preceding exercises, and initialize it by assigning it the value of the previously declared pointer value. Then, before the calls to `printf`, assign a new value to the storage location at the address specified by the pointer. (Again, you’ll use the `*` operator to reach through the pointer to the storage location at which you want to place the new value.) Also, add a call to `printf` that displays the int value at the new pointer’s target location.

Compile and run the revised program and summarize and explain the changes in its output.

Exercise 101: Add a declaration for a second variable of type `double` to your program and initialize it with a value different from the value you used for the first `double` variable. Then (above the definition of the `main` function) define a new function called `swap` that has the side effect of exchanging the contents of two storage locations, each of which holds a `double`, so that each location winds up containing the value that the other formerly contained. This side effect should persist after `swap` returns. (Hint: the parameters of the `swap` procedure will be of type `double *`, not simply of type `double`.)

Back in the `main` function, add a call to `swap` that exchanges the values of the two `double` variables. Also add a new `printf` statement to show the value and location of the new variable so that you can confirm that both halves of the swap were carried out correctly. Compile and run the revised program and summarize and explain the changes in the output.

Exercise 110: Write and test a C function that takes an array of signed `short` integers and the number of elements in that array, and returns (through two additional parameters, the type of which should be `int *`) the sum of the positive elements of the array and the sum of its negative elements, respectively.

As we’ll see in the next reading from the textbook, several “arithmetic” operations can be performed on pointers as well as on numbers. One of these operations is the “increment” operation symbolized by the double plus sign, `++`. When a pointer is incremented, the number of bytes used to store a value of the type that it points to is added to it, so that it now points to a location in memory that is adjacent to the one to which it previously pointed. If the pointer happens to be pointing to an element in an array, incrementing the pointer “advances” it so that it points the location of the next element at the next higher position in the array.

Exercise 111: The function that you wrote in the previous exercise probably traverses the given array by declaring and updating an explicit index variable to serve as a subscript into the array. Use a “pointer to `short`” variable to traverse the array instead: Initialize it to the address of the element at position 0, increment it between iterations of the loop, and re-enter the loop only if the pointer is still pointing to some element of the array after it has been incremented. (You can test this by making sure that the pointer’s value is strictly less than the address of the notional element `arr[size]`, which would (if it existed) follow all the actual array elements `arr[0]`, `...`, `arr[size - 1]`. The address of `arr[size]` exists and can be usefully computed even though there is no such element!)