CSCI-1200 Data Structures — Fall 2012
Lab 3 — Pointers, Arrays, and the Stack

This lab explores the use of pointer arithmetic, allocation of single value and array variables on the stack, passing arguments by reference and by value, and the C calling convention. Please have your notes from Lecture 5 available for this lab.

Checkpoint 1

Write a function `compute_squares` that takes 3 arguments: two C-style arrays (*not* vectors), a and b, of unsigned integers, and an unsigned integer, n, representing the size of each of the arrays. The function should square each element in the first array, a, and write each result into the corresponding slot in the second array, b. You may not use the subscripting operator (`a[i]`) in writing this function; instead, practice using pointer arithmetic. Also, write a main function and a couple of test cases with output to the screen to verify that your function is working correctly.

To complete this checkpoint: Show a TA your function, the test cases, and the corresponding output.

Checkpoint 2

What will happen if the length of the arrays is not the same as n? What will happen if n is too small? If n is too big? What if the a array is bigger than the b array? Or vice versa? How might the order that the variables were declared in the `main` function impact the situation? First think about all of these questions and draw pencil & paper pictures of the memory. Jot down your hypotheses before testing.

Now let's print out the contents of memory and see what's going on. In the file "`lab3_code.cpp`" we provide the definition of the function `print_stack` that will help us see how variables and arrays are allocated on the stack. For example, the code on the left will result in output similar to that shown on the right (the exact memory addresses will vary). *NOTE: In order to accommodate 32-bit and 64-bit operating systems, the code uses the type* `uintptr_t`*. On a 32 bit OS/compiler, this will be a standard 4 byte unsigned integer and on a 64 bit OS/compiler, this will be a 8 byte unsigned integer type. You should use this type instead of* `int` *throughout this lab (edit your checkpoint 1 code).*

```
cout << "size of uintptr_t: "
     << sizeof(uintptr_t) << endl;
uintptr_t x = 72;
uintptr_t a[5] = {10, 11, 12, 13, 14};
uintptr_t *y = &x;
uintptr_t z = 98;
cout << "x address: " << &x << endl;
cout << "a address: " << &a << endl;
cout << "y address: " << &y << endl;
cout << "z address: " << &z << endl;
uintptr_t* bottom_address =
  min(&x,min(&a[0],min(&a[4],min(&y,&z))));
uintptr_t* top_address =
    max(&x,max(&a[0],max(&a[4],max(&y,&z))));
print_stack(bottom_address,top_address);
```

```
size of uintptr_t: 4
x address: 0xbfbfe95c
a address: 0xbfbfe930
y address: 0xbfbfe92c
z address: 0xbfbfe928
----------------------------------------
location: 0xbfbfe95c  VALUE:   72
location: 0xbfbfe958  garbage?
location: 0xbfbfe954  garbage?
location: 0xbfbfe950  garbage?
location: 0xbfbfe94c  garbage?
location: 0xbfbfe948  garbage?
location: 0xbfbfe944  garbage?
location: 0xbfbfe940  VALUE:   14
location: 0xbfbfe93c  VALUE:   13
location: 0xbfbfe938  VALUE:   12
location: 0xbfbfe934  VALUE:   11
location: 0xbfbfe930  VALUE:   10
location: 0xbfbfe92c  POINTER: 0xbfbfe95c
location: 0xbfbfe928  VALUE:   98
----------------------------------------
```

Typically the local variables will be allocated on the stack in order (note that the stack on x86 architectures is in descending order). You can see the elements of the array, but since the first element of the array is stored in the smallest memory location the array looks upside down. Also you might see extra space between the variables due to temporary variables or padding inserted by the compiler to improve alignment. This uninitialized extra space may be correctly labeled as "garbage" by the `print_stack` function (as shown above), or it might contain old data values or addresses that appear to be legal.

Using the `print_stack` command before and after a call to your `compute_squares` function should help you understand how the compiler is organizing the memory for your local variables and function arguments. (Be sure to switch `compute_squares` to use `uintptr_t`s instead of `int`s.) First try this on a correct test case to make sure you can correctly interpret the stack data. Then try it on one of the bad cases described at the beginning of this checkpoint that has incorrect behavior. Study the stack data and make sure you understand how the memory error occurs.

NOTE: Do not compile with optimizations enabled. By default `g++` *does not use optimizations.*

To complete this checkpoint: Show a TA the output of both your correct and incorrect test cases and describe how the `print_stack` output corresponds with your predicted behavior.