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 STL 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 function is incorrectly used and 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.

[http://www.cs.rpi.edu/academics/courses/spring14/ds/labs/03_pointers/print_stack.cpp](http://www.cs.rpi.edu/academics/courses/spring14/ds/labs/03_pointers/print_stack.cpp)

Now let’s print out the contents of memory and see what’s going on. The provided function `print_stack` that will help us see how variables and arrays are allocated on the stack. Sample output is shown on the next page (the exact memory addresses will vary).

```cpp
std::cout << "size of uintptr_t: " << sizeof(uintptr_t) << std::endl;
uintptr_t x = 72;
uintptr_t a[5] = {10, 11, 12, 13, 14};
uintptr_t *y = &x;
uintptr_t z = 98;
std::cout << "x address: " << &x << std::endl;
std::cout << "a address: " << &a << std::endl;
std::cout << "y address: " << &y << std::endl;
std::cout << "z address: " << &z << std::endl;

// label the addresses you want to examine on the stack
label_stack(&x,"x");
label_stack(&a[0],"a[0]");
label_stack(&a[4],"a[4]");
label_stack((uintptr_t*)&y,"y");
label_stack(&z,"z");

// print the range of the stack containing these addresses
print_stack();
```

**NOTE:** In order to accommodate 32-bit and 64-bit operating systems, the code uses the type `uintptr_t` in places of `int` and all pointers. 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 substitute this type instead of `int` throughout this lab (edit your checkpoint 1 code).
size of uintptr_t: 8
x address: 0x7fff5fbff800
a address: 0x7fff5fbff770
y address: 0x7fff5fbff7f8
z address: 0x7fff5fbff7f0

-----------------------------------------
location: 0x7fff5fbff828 garbage?
location: 0x7fff5fbff820 garbage?
location: 0x7fff5fbff818 garbage?
location: 0x7fff5fbff808 garbage?
x location: 0x7fff5fbff800 VALUE: 72
y location: 0x7fff5fbff7f8 POINTER: 0x7fff5fbff800
z location: 0x7fff5fbff7f0 VALUE: 98
location: 0x7fff5fbff7e8 garbage?
location: 0x7fff5fbff7e0 garbage?
location: 0x7fff5fbff7d8 garbage?
location: 0x7fff5fbff7d0 garbage?
location: 0x7fff5fbff7c8 garbage?
location: 0x7fff5fbff7c0 garbage?
location: 0x7fff5fbff7b8 garbage?
location: 0x7fff5fbff7b0 garbage?
location: 0x7fff5fbff7a8 garbage?
location: 0x7fff5fbff7a0 garbage?
location: 0x7fff5fbff798 garbage?
a[4] location: 0x7fff5fbff790 VALUE: 14
location: 0x7fff5fbff788 VALUE: 13
location: 0x7fff5fbff780 VALUE: 12
location: 0x7fff5fbff778 VALUE: 11
a[0] location: 0x7fff5fbff770 VALUE: 10
location: 0x7fff5fbff768 garbage?
location: 0x7fff5fbff760 garbage?
location: 0x7fff5fbff758 garbage?
location: 0x7fff5fbff750 garbage?
location: 0x7fff5fbff748 garbage?

The local variables (x, y, z, and a) are allocated on the stack in some order (the compiler has flexibility to re-arrange things a bit). 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 extra space may be labeled as “garbage?” or it might contain old data values or addresses that appear to be legal and useful.

Note: A mix of different types of data are stored within the stack. Our toy print_stack for lab assumes that the data in the stack is an integer if the value is small (+/- 1000) or a pointer if the number is “nearby” the memory locations labeled as interesting with label_stack. All other values are marked “garbage?”.

Now, use the print_stack command before and after the call to your compute_squares function to help you understand how the compiler is organizing the memory for your local variables and function arguments. You’ll need to switch compute_squares to use uintptr_t instead of int.) First try this on a correct test case to make sure you can correctly interpret the stack data. Then, try it on several of the different incorrect usage cases described at the beginning of this checkpoint. Study the stack data and make sure you understand how the memory error occurs. Make sure to exaggerate the errors so that memory is misused or clobbered and correct program behavior is disrupted.
NOTE: Do not compile with optimizations enabled. By default g++ does not use optimizations. You may also need to disable the memory debugging features of your IDE (use Cygwin & g++ for this lab if you are unsure how to disable memory debugging in your IDE).

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.

Checkpoint 3

For the last checkpoint, grab your Time class header file and implementation code from Lab 2. If that’s not handy, any simple C++ class using integers will do. NOTE: Switch all ints in your class to uintptr_t.

First determine the memory footprint in bytes of a single instance of the Time class using the command sizeof(Time). The return value of this expression is an integer representing the total number of bytes and should correspond to the sum of the sizes of the member variables of the class (or larger if the compiler inserts extra space to improve byte alignment).

Next, write a simple function named change_times that takes in as arguments two parameters t1 and t2 of type Time, one by reference and one by value. Within the function use setHour, setMinute, and setSecond to modify these inputs (e.g., add an hour and a half to t1 and t2). Now within the main function, create two local variables of type Time with different initial times (use distinctive numbers so you can easily see them on the stack). Then call the change_times function with these variables.

What happens when you pass a Time object by value (a.k.a. “by copy”) vs. by reference? We can see the difference and the extra memory use for copying by examining the stack closely. Also, we can see how the data for pass by reference variable has two labels. Inside of change_times, use the print_stack function to print the relevant portion of the stack showing the “stack frames” for both functions (main and change_times), before and after the edits. In addition to the stack output, also print the addresses of the Time variables in both functions. Looking at this output, identify the stack memory location of the local variables in the main function, and the arguments that were passed to change_times. Experiment with the stack by adding local variables to the main function and to the helper functions and watch how the variables are allocated within the proper frame.

NOTE: The label_stack function expect a memory address of type uintptr_t*, so if you try to compile this code:

```
Time foo(1,2,3);
label_stack(&foo,"foo");
```

You’ll see errors like “cannot convert `Time*` to `uintptr_t*`”. Both types are really just pointers (to memory locations) so the conversion is simple and safe in this case. To fix this use an explicit cast to override the compiler check.

```
label_stack((uintptr_t*)&foo,"foo");
```

Ask a TA if you have questions about the use of casting.

To complete this checkpoint: Show the TA your output and describe how the stack memory data matches your understanding of the differences between pass by reference and pass by value. Ask the TAs questions about the the C/C++ calling convention and the implementation of the print_stack function.