ESC190H1S Winter 2022: Lab 2

TA evaluations in practicals: February 1st/2nd, 2022
Code submission due: February 3rd, 2022 at 11:59pm on Gradescope

Overview

The estimated completion time for this lab is 3 hours. Starter code is not provided for this lab. This is intentional so that you practice the lifecycle of writing, compiling, and running `.c` code. Starter code will be provided from lab 3 onwards.

It is assumed that you have the following prerequisite knowledge from lab 1:

- Basic structure of `.c` files
- Compiling C code with `gcc`
- Running terminal commands such as Valgrind
- Familiarity with the following terms: *compiler flags*, *executable*, *includes*

Learning outcomes:

- using pointers in C
- memory model in C
- dynamic memory allocation and management
- using Valgrind to detect memory malpractice

Additional Requirements

You **MUST NOT** add any `#include` statements other than the following permitted libraries:

- `stdio.h`
- `stdlib.h`

Violating this constraint does not guarantee your code will function when grading and could result in a grade of zero.

Do **NOT** add any additional print statements to your code other than what is indicated in the lab handout. Do not read from any filestreams, and do not write to any filestream other than `stdout`.

Submission Checklist

There are 7 tasks for you to complete in this lab. Submit only the files indicated below. Make sure the cAsInG, spelling, and whitespace (or lack thereof) in the filenames is identical to those listed below.

- `lab2_task1.c`
- `lab2_task2.c`
- `lab2_task3.c`
- `lab2_task4.c`
- `lab2_task5.c`
- `lab2_task6.c`
- `lab2_task7.c`

Please note, this lab has hidden test cases that are not visible to you on Gradescope that will be used for grading. Make sure to thoroughly test the functionality and memory management of your code yourself.

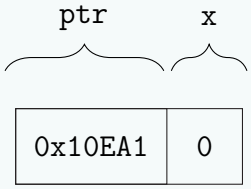
Pointers in C

Task 1 Create a `lab2_task1.c` file with the contents of the `main` function shown below. Compile and run the code. Remember to include `stdio.h`.

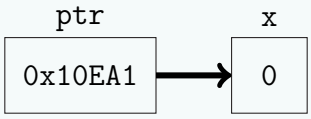
```
int x = 0;                                                    1
int* ptr = &x;                                                2
fprintf(stdout,                                               3
  "x contains the value %d at location %p\n", *ptr, ptr       4
);                                                            5
```

The `*` operator means different things in different contexts. At line 2, it is declaring the type of `ptr` to be a *pointer* to an `int`. At line 4, it is used as a *dereferencing* operator, giving us the value stored at the memory address of `ptr`.

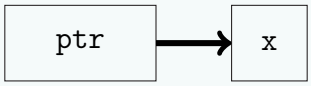
The memory model for the above code is shown below. Note that an arbitrary hex address is used for `ptr`, and will be different from what your program outputs.



An alternate representation that has the pointer *pointing* to the data stored at its address:



Or a simplified version:



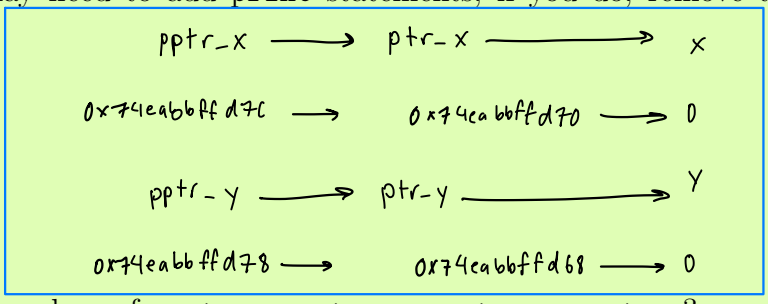
Task 2 Create a `lab2_task2.c` file with the contents of the `main` function shown below. Compile and run the code. Remember to include `stdio.h`.

```
int x = 0;
int y = x;

int* ptr_x = &x;
int* ptr_y = &y;
int** pptr_x = &ptr_x;
int** pptr_y = &ptr_y;

fprintf(stdout,
  "x contains the value %d at location %p\n", *ptr_x, ptr_x
);
fprintf(stdout,
  "y contains the value %d at location %p\n", *ptr_y, ptr_y
);
```

Draw the memory model for the above piece of code. Include `pptr_x` and `pptr_y` in your picture. Use the memory addresses you get from running the above code (you may need to add `print` statements, if you do, remove these before submitting).



What are the values of `*pptr_x`, `*pptr_y`, `**pptr_x`, `**pptr_y`?

How could you make `x` and `y` share the same location in memory?

y = &x

Pointers to Static Arrays

Static arrays are fixed in size. Once a static array is allocated, it stays the same size forever. If you need more space for your data, you will need to create a *new* static array and transfer all the elements over one by one.

Task 3 Create a `lab2_task3.c` file with the contents of the `main` function shown below. Compile and run the code. Remember to include `stdio.h`.

```
int y[2] = {1, 5};
int x[3];

for (int i = 0; i < 3; i++){
   x[i] = y[i];
}

x[2] = 9;

fprintf(stdout,
 "Addresses of x 0: %p, 1: %p, 2: %p\n",
  x, x + 1, x + 2
);
fprintf(stdout,
  "Values of x through dereferencing 0: %d, 1: %d, 2: %d\n",
   *x, *(x + 1), *(x + 2)
);
fprintf(stdout,
  "Values of x through indexing 0: %d, 1: %d, 2: %d\n",
  x[0], x[1], x[2]
);
fprintf(stdout,
  "Indexing out of range: %d\n", x[3]
);
```

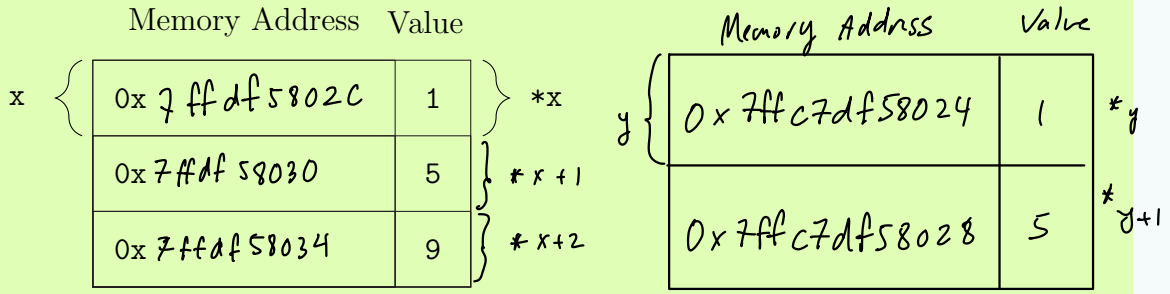
What happens when you attempt to access an element out of range of the array allocated?

you access garbage values or seg fault

Attempt to extend the size of the array by doing `y[2] = 9`. What happens?

nothing

Complete the following diagram with the memory addresses you get from running the above code. Add the memory blocks for `y` to the diagram. You may wish to add `print` statements to help draw your diagram.



Pointers to Dynamic Arrays

If we want to have arrays of data that can change size (similar to a Python list), we need to use *dynamic* memory allocation. This is achieved with the `malloc(...)`, `calloc(...)`, and `realloc(...)` functions from `stdlib.h`. For any dynamic memory we consume, we are responsible for freeing it back to the OS with `free(...)`.

Here is an overview some memory-related code terminology:

- Forgetting to free memory you allocated is called a *memory leak*. This usually does not result in a program error or termination.
- Attempting to access memory you are no longer allocated may cause a *segmentation fault*. However, this does not always happen - you may end up just reading garbage data.
- Reading data you are not allocated is an *invalid read*.

Task 4 Create a `lab2_task4.c` file with the contents of the `main` function shown below. Compile and run the code. Remember to include `stdio.h` and `stdlib.h`.

```
int *p = (int *)malloc(sizeof(int));
fprintf(stdout, "Value stored at address %p: %d\n", p, *p);

*p = 1;
fprintf(stdout, "Value stored at address %p: %d\n", p, *p);
free(p);

fprintf(stdout, "Value stored at address %p: %d\n", p, *p);
```

Explain what is happening in this code. Is there a memory leak?

a dynamic array is made w/ 1 element assigned w/ the value 1, no memory leak is happening.

Run the following command, replacing `<executable>` with the name of your executable.

```
valgrind --leak-check=full
         --show-leak-kinds=all
         --track-origins=yes
         ./<executable>
```

What parts of Valgrind's output helps you identify the problem in the code?

conditional jump or move depends on uninitialised values

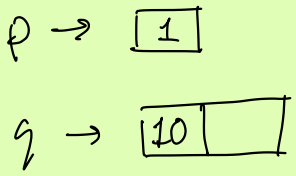
Try compiling your executable with the `-g` flag, and rerun the command above. What is the difference between compiling with vs. without the `-g` flag?

It didnt do anything, however -g flag is a debug flag

Task 5 Create a `lab2_task5.c` file with the contents of the `main` function shown below. Compile and run the code. Remember to include `stdlib.h`.

```
int *p = (int *)malloc(sizeof(int));                    1
int *q = (int *)malloc(sizeof(int) * 2);                2
*p = 1;                                                 3
*q = 10;                                                4
                                                        5
q = p;                                                  6
free(q);                                                7
free(p);                                                8
```

Draw a memory model for the above code.



Describe the memory malpractice in the above code. You may wish to use Valgrind as described previously.

double free detected

Since the pointers were equated, the memory allocated to q in the beginning is not freed.

Task 6 Create a `lab2_task6.c` file with the contents of the `main` function shown below. Compile and run the code. Remember to include `stdlib.h`.

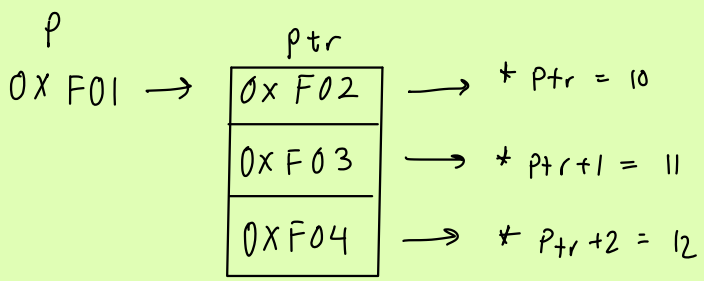
```c
int** p;

int x = 10;
int y = 11;
int z = 12;

int* ptr = (int *)malloc(sizeof(int) * 3);
*ptr = x;
*(ptr + 1) = y;
*(ptr + 2) = z;

p = &ptr;
```

Draw a memory model for the above code. You may use arbitrary memory addresses to simplify the task.



Modify the code to correctly free all allocated memory. Check your work with Valgrind.

Task 7 Create a `lab2_task7.c` file that implements the following function `append(...)`. Remember to include `stdlib.h`. Do **NOT** include a main function or any other `includes`. Do **NOT** allocate any new memory blocks with `malloc` or free any blocks with `free` in this function.

```
void append(int** x, int* n, int val){
  /**
  <*x> refers to a dynamically allocated integer array
  of size <*n> * sizeof(int).
  Modifies <*x> such that it has val added to the end.
  The modified array should be of length <*n> + 1.

  Modifies <*n> such that it is updated to be the value
  of the new length of the array <*x>.

  HINT: you may find the function realloc(...) helpful
  */
}
```

Would you be able to implement this function if the first parameter was changed to `int* x`? Explain why or why not.

maybe, realloc sometimes creates a copy, so it wouldn't change the actual array.

Would you be able to implement this function if the second parameter was changed to `int n`? Explain why or why not.

no, int n just passes a copy, int* n passes the location of n.