CS 121 Worksheet - Week 11 - Pointers Introduction & Practice

Preface to pointers:

We've covered how to store values into variables, e.g.

double x = 5.0;

Now we're going to go over how variables are stored in memory and how to manipulate the locations in which they are stored at.

Two points that cannot be emphasized nearly enough (when dealing with pointers):

- **Memory Address:** Location in memory in which a variable is stored at.
- The memory address of a variable does not necessarily relate to the value a variable holds

Before continuing you **must** understand the above. It's very easy to mistake the second point.

Onto what a pointer is:

- **Pointer:** A variable that holds a memory address.

The definition of a pointer is pretty straightforward, but can be tricky to fully comprehend.

The general syntax of a pointer:

*<datatype_to_point_to> * <identifier>;*

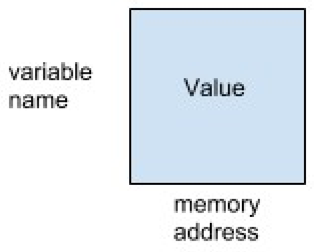
Examples:

*int * int_ptr;*
*double * another_ptr;*

*bool * myptr;*

In the following page, refer to the following notes:

- Use the following image as a reference (for the visual diagrams):



- Arrows are used to emphasize that a pointer is pointing to the memory location of a variable.
- The memory addresses are all made up and are small numbers for simplicity
    - Realistically, they would be in hexadecimal format, looking something like 0xbf83775A
- "*&var*" means to get the memory address of a variable.
- "**pointer_var*" means to dereference a pointer
    - This means to change the value of the memory location the pointer

Example application of pointers (creation of a pointer and dereferencing pointers):

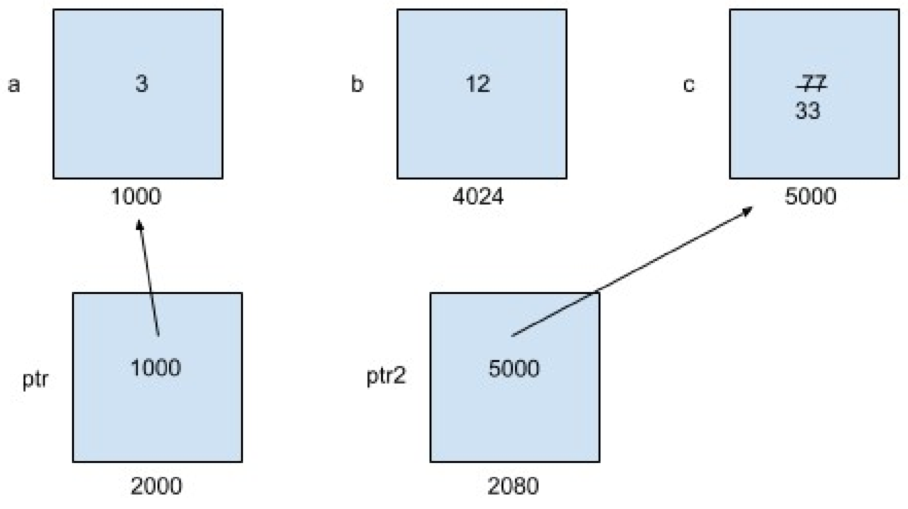
Syntax (C++):	Visualization:

```
int main()
{
  int a = 3; b = 12, c = 77;

  int* ptr = &a;

  int* ptr2 = &c;

  *ptr2 = 33;

  return 0;
}
```
	 a 3 1000
	b 12 4024
	c ~~77~~ 33 5000
	ptr 1000 2000
	ptr2 5000 2080

Another example of pointers (pointing to arrays and pointer arithmetic):

Syntax (C++):	Visualization:

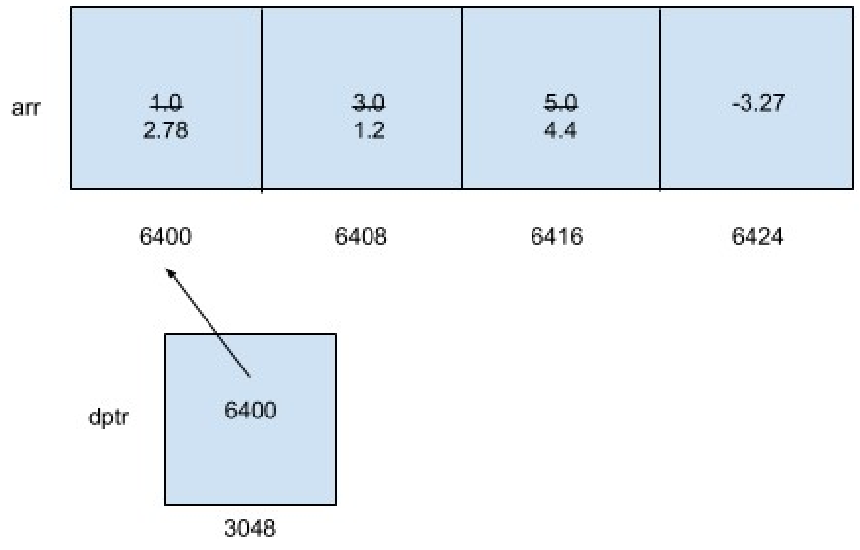
```
int main()
{
  double arr[] = {1.0, 3.0, 5.0,
                  -3.27};
  double* dptr;

  dptr = arr;

  *dptr = 2.78;

  *(dptr+1) = 1.2;

  dptr[2] = 4.4;

  return 0;
}
```
	 arr
	~~1.0~~
	2.78
	~~3.0~~
	1.2
	~~5.0~~
	4.4
	-3.27
	6400
	6408
	6416
	6424
	dptr
	6400
	3048

Before continuing on, recall the following:

The memory address of a variable does not necessarily relate to the value a variable has!

At this point you might be wondering “What is the point of pointers? They only seem to complicate things.” And you would be right: They mostly complicate things. They are however extremely powerful for (at least) the three following reasons:

1. **They allow us to create dynamic memory**
   - **a.** Pointers are how strings and vectors are able to dynamically change their sizes.

2. **They allow polymorphism to exist in C++**
   - **a.** This is a bit tricky to explain right now and might be covered later on. Here is a tutorial on it if you are interested.

3. **They allow an alternate way to pass by reference (specifically this is called passing by pointer)**
   - **a.** Using ‘&’ in the parameter list didn’t exist in C (it’s a C++ thing)
   - **b.** Also, a side-note: Pointers allow us to pass a NULL value (whereas by-reference does not)

1) Dynamic memory:

- Ever think it was annoying or wasteful to use partially-filled arrays to cover user-defined array sizes? Using pointers, we can create arrays with a size defined by **any** integer-based variable.

- Noted below, we dynamically allocate memory by using the *new* keyword, and deallocate it by using the *delete* keyword. This memory is created in the heap, which is a block of memory designed for dynamic memory allocation (you can think of it like a sandbox for messing with memory allocation).
  - **IMPORTANT:** Remember to use the *delete* keyword or else we run into memory leak errors.
- Syntax:
  - Example:

```
char* my_str = new char[7];
//…
delete my_str;
```

  - Example (with user input): (this example is used in problem two)

```
int size;
int* usr_list;

cout << “Enter the size of the list you want: “;
cin >> size;
// do error-checking with size (i.e. verify it’s greater than zero)
```

```
usr_list = new int[size];
//…
delete usr_list;
```

Questions:

1. If you haven't yet, observe and make comments about the diagrams on pages 2 & 3. Are there confusing parts or anything interesting you notice?

2. Draw out visually what you believe is occurring with the second dynamic memory allocation example. Use the reference diagram on page one to help with your illustration.

3. Translate the following to a pass by-pointer function. The function header for the new function is given to you for convenience.

```
void swap(int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
}

void swap(int* a, int* b)
{




}
```

4. Translate the following to use pointer arithmetic instead of the bracket operator. Assume all variables below are already defined.

```
for(int i = 0; i < SIZE; ++i)
{
  arr[i] = 2 * i;  // this line needs to be changed
}
```

5. Before we used partially-filled arrays to read data from a file. How could you improve memory efficiency by implementing dynamically-allocated arrays instead? HINT: Think of ways on how we may get the array size.