Discussion Session
Week 5

Week 5:
Pointers and more on arrays

Pointers

- A **pointer** is a variable that holds the memory address of another variable
  - A memory address is always a hex value
  - When defining a pointer, we must always set it equal to a memory address by using the ‘&’ symbol
  - In the rare case that we have nothing to set out pointer equal to, default it to NULL

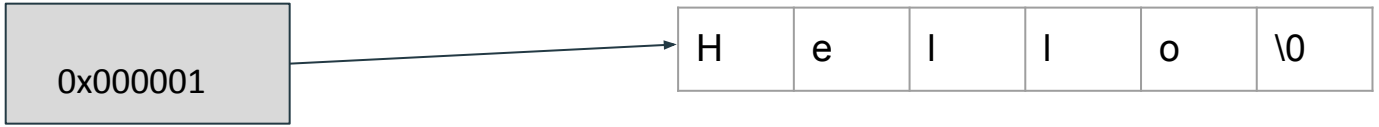
- Declaring a pointer
  - {type} *varName;
- Accessing values within a pointer
  - *varName
- Accessing the address of a pointer
  - varName

Pointers Visualized

int *a

*a = 5

char *temp = “Hello”



Example:

- int* ptr declares a pointer to an integer.
  - *ptr de-references the pointer and lets us access the data being stored there

```
4   int main()
5   {
6     int num = 5;
7     std::cout << num << " " << &num << std::endl;
8
9     int* ptr = &num;
10    std::cout << *ptr << " " << ptr << std::endl;
11  }
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
URI+david_perrone@DESKTOP-KSNEQOR MINGW64 ~/Documents/URI
$ g++ pointers.cpp && ./a.exe
5 0x61ff08
5 0x61ff08
```

Arrays and Pointers

- Ptr stores the memory address of array
  - This lets us access the elements in array
- There is only one array being stored in memory here

```
3   int main()
4   {
5       int array[5] = {1,2,3,4,5};
6       int* ptr = array;
7       for(int i = 0; i < 5; i++)(std::cout << array[i]);
8       std::cout << std::endl;
9       for(int i = 0; i < 5; i++)(std::cout << ptr[i]);
10      std::cout << std::endl;
11  }
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

URI+david_perrone@DESKTOP-KSNEQOR MINGW64 ~/Documents/URI
$ g++ pointers.cpp -o ptr && ./ptr
12345
12345
```

Arrays and Pointers

- Array and ptr are both pointers to the same memory address
- Changing an element in ptr will change the element in array (and vice versa)

```
8     std::cout<<array<<std::endl;
9     std::cout<<ptr<<std::endl;
10 }
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
URI+david_perrone@DESKTOP-KSNEQOR MINGW64 ~/Documents/URI
$ g++ pointers.cpp -o ptr && ./ptr
0x61fef8
0x61fef8
```

```
8     ptr[0] = 10;
9     array[1] = 12;
10
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
URI+david_perrone@DESKTOP-KSNEQOR MINGW64 ~/Documents/URI
$ g++ pointers.cpp -o ptr && ./ptr
10 12 3 4 5
10 12 3 4 5
```

Arrays ~are~ pointers

Creating an array without brackets

```
6  int main(void)
7  {
8      int *array = (int *) calloc(ARRAY_SIZE, sizeof(int));
9      printf("%p\n", array);
10
11     for(int i = 0; i < ARRAY_SIZE; ++i) std::cout << array[i] ;
12     std::cout << std::endl;
13
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
derek@DESKTOP-3L8T6AU:/mnt/c/Users/Derek Jacobs/Desktop/CSC/TA/211$ g++ test.cpp && ./a.out
0x7fffc41fde70
00000
```

Pointer arithmetic

- Because arrays are stored contiguously, we can use pointer arithmetic to iterate through an array

```
4   int main()
5   {
6     int array[5] = {1,2,3,4,5};
7     int* ptr = array;
8
9     for(int i =0; i < 5; i++) std::cout << *(array + i);
10    std::cout << std::endl;
11
12    for(int i =0; i < 5; i++) std::cout << *(ptr + i);
13    std::cout << std::endl;
14  }
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
URI+david_perrone@DESKTOP-KSNEQOR MINGW64 ~/Documents/URI
$ g++ pointers.cpp -o ptr && ./ptr
12345
12345
```

Pointers and functions

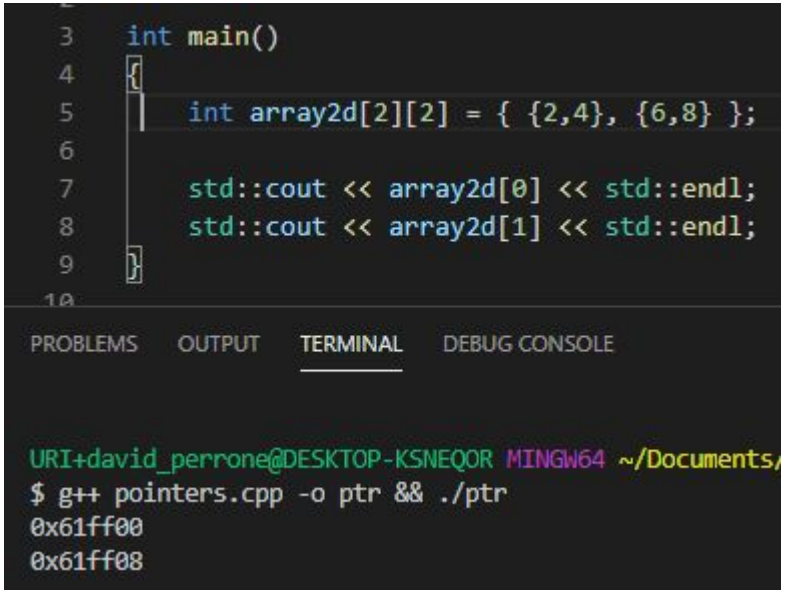
- Functions in c++ can only return one variable
- Pointers allow us to modify the value of variables without returning anything

```cpp
#include <iostream>

void add2(int num1, int& num2, int* num3);

int main()
{
    int a = 0;
    int b = 10;
    int c = 20;

    add2(a, b, &c);

    std::cout << a << ' ' << b << ' '  << c<<<std::endl;
}

void add2(int num1, int& num2, int* num3)
{
  num1 += 2;
  num2 += 2;
  *num3 += 2;
}
```

```
URI+david_perrone@DESKTOP-KSNEQOR MINGW64 ~/Documents/URI
$ g++ pointers.cpp -o ptr && ./ptr
0 12 22
```

Multi-Dimensional Arrays

- An array of arrays
- Array2d[0] is a pointer to an array of integers {2,4} stored at memory address 0x61ff00
- Array2d[1] is a pointer to an array of ints {6,8} stored at 0x61ff08.
- We can predict this memory address because each array stores 2 integers (8 bytes)



Dynamic Arrays

- Static arrays - Size needs to be known at compile time
- What if we don’t know how large the array will be?
- The keywords “new” and “delete” are used to allocate and deallocate dynamic memory.

```
int n;
std::cin >> n;
int* arr = new int[n];
delete[] arr;
```

```
int main()
{
    int n,m;
    std::cin >> n >> m;
    int** arr = new int*[n];
    for(int i =0; i < n; i++)
    {
        arr[i] = new int[m];
    }

    for(int i =0; i < n; i++)
    {
        delete[] arr[i];
    }

    delete[] arr;
}
```

Valgrind

A very useful tool used to check for memory leaks (memory that has not been freed when allocated on the heap)

```
3   int main(void)
4   {
5       std::cout << "Hello World" << std::endl;
6       int *temp = new int;
7
8       // delete temp;
9
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
djacobs@homework:~$ valgrind ./a.out
==1213788== Memcheck, a memory error detector
==1213788== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1213788== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1213788== Command: ./a.out
==1213788==
Hello World
==1213788==
==1213788== HEAP SUMMARY:
==1213788==     in use at exit: 4 bytes in 1 blocks
==1213788==   total heap usage: 3 allocs, 2 frees, 73,732 bytes allocated
==1213788==
==1213788== LEAK SUMMARY:
==1213788==    definitely lost: 4 bytes in 1 blocks
==1213788==    indirectly lost: 0 bytes in 0 blocks
==1213788==      possibly lost: 0 bytes in 0 blocks
==1213788==    still reachable: 0 bytes in 0 blocks
==1213788==         suppressed: 0 bytes in 0 blocks
==1213788== Rerun with --leak-check=full to see details of leaked memory
==1213788==
==1213788== For lists of detected and suppressed errors, rerun with: -s
==1213788== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```cpp
int main(void)
{
    std::cout << "Hello World" << std::endl;
    int *temp = new int;

    delete temp;

```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
djacobs@homework:~$ g++ hello.cpp
djacobs@homework:~$ valgrind ./a.out
==1214235== Memcheck, a memory error detector
==1214235== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1214235== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1214235== Command: ./a.out
==1214235==
Hello World
==1214235==
==1214235== HEAP SUMMARY:
==1214235==     in use at exit: 0 bytes in 0 blocks
==1214235==   total heap usage: 3 allocs, 3 frees, 73,732 bytes allocated
==1214235==
==1214235== All heap blocks were freed -- no leaks are possible
==1214235==
==1214235== For lists of detected and suppressed errors, rerun with: -s
==1214235== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exercise 1 (5 minutes)

- Create a void function that takes in a pointer to an array and fills it with numbers 0 through n, where n is the length of the array

Exercise 2 (5 min)

- Create a program that takes in 2 integers, swaps their values using pointers, printing the memory address of each variable and the value at each address along the way
    - **Preserve the memory address of each variable**

```
5 13
BEFORE SWAP
 A 0x7ffff5816200       5
 B 0x7ffff5816204       13
AFTER SWAP
 A 0x7ffff5816200       13
 B 0x7ffff5816204       5
```

Pointers vs references

- **Use references when you can and pointers when you have to**
  - References are easier to implement and can do a lot of the same things as pointers but have less flexibility and aren't able to be reassigned

- Pointers are best for
  - If we need a NULL memory address for any reason
  - We need pointer arithmetic (such as traversing an array)
  - Implementing data structures

- References are best for
  - Function parameters and return types

Final Remarks/Key Points

- Pointers point
  - They point to memory addresses that hold values, but they themselves do not hold values
- Important operators:
  - & “Address of” → Used when you are trying to access the memory address of a non-pointer
  - * “Value at” → Used to either de-reference a pointer to access the value at its memory address, or declare a pointer
  - [] Used when creating arrays

Questions?

- On anything...beyond just pointers and what we've covered today