Lab 9
Understanding Arrays as Pointers

Run the following program and make sure that you can explain the output,

Note: the sizeof() function returns the number of bytes of the argument passed to it.

```cpp
#include <iostream>
using namespace std;

int main( ) {
    int b[3][2];
    cout<<sizeof(b)<<endl;
    cout<<sizeof(b+0)<<endl;
    cout<<sizeof(*(b+0))<<endl;
    cout<<"The address of b is: "<<b<<endl;
    cout<<"The address of b+1 is: "<<b+1<<endl;
    cout<<"The address of &b is: "<<&b<<endl;
    cout<<"The address of &b+1 is: "<<&b+1<<endl;
    return 0;
}
```
In Real Life

**Business Listing**

*Cheese Shop*
Address: 12-34 Whatever St.

**The Place Itself**
Pointers, like business listings, store addresses. To visit the location which the (memory) address refers to, we dereference the pointer.
Pointers are data types which store memory addresses. We visit the location the memory address refers to by dereferencing the pointer.

```c
int i = 10;
int *ip; // declaring a pointer to an integer
ip = &i; // storing address of i (an int variable) into pointer ip
*ip = 5; // dereference ip (get/visit variable i)
```

**Notes**

- When the * operator is part of a variable declaration, it means the variable is a pointer; elsewhere, when it is on the left-side of a pointer variable, it serves as the dereference operator.
- The & operator is the “address of” operator.
**Pointer Examples**

```cpp
int x = 5;
int *xptr = &x; // xptr is a pointer to an int variable
int *yptr, y; // yptr is a int pointer, y is a “regular” int
yptr = &y; // yptr stores the address of y
*yptr = 3; // yptr dereferenced gives you variable y itself

cout << x << endl; // 5
cout << xptr << endl; // prints address stored in xptr
cout << &x << endl; // prints address of x (same output as above)
cout << *xptr << endl; // 5 (dereferencing xptr gives you variable x)

cout << y << endl; // 3
cout << yptr << endl; // prints address stored in yptr
cout << &y << endl; // prints address of y (same output as above)
cout << *yptr << endl; // 3 (dereferencing yptr gives you variable y)

// xptr and yptr get dereferenced first, then multiplied
int product = *xptr * *yptr; // product = 15
```
Swap Function...with Pointers!

Variable scope is irrelevant when you have the memory address of a variable.

```c
int main() {
    int j = 1, k = 2;
    swap(&j, &k);
}

void swap(int* a, int* b) {
    int temp = *a; //get variable which a points to and store value
    *a = *b; //get variables which a and b point to, reassign value
    *b = temp; //reassign value of variable which b points to
}
```
De-reference

• Why do we use the integer pointer for both single integer and array of integers? Because we want to get the integer value.

• De-reference of an int: Take the 4 bytes that it points to, represent the 32 bits as an integer. Use * operator for de-referencing.
• What does it mean to add 1 to an pointer?

• int a[3]; int *ip = a; ip = ip + 1 ← the address of next integer
int a[4] = {1, 5, 0, 7};

You’re used to coding like this… …here’s another way.

```
a[0] = 4;
cout << a[2] << endl;
```

```
*(a+0) = 4;
cout << *(a+2) << endl;
```
int a[4] = {1, 5, 0, 7};

The array name itself refers to the contiguous block of memory where the array sits.

Array names are not pointers, though they can be treated like pointers in many cases. Recall that we dereference a pointer to access whatever is being pointed to.

cout<< *a << endl; //1
cout<< *(a+0) << endl; //1
cout<< *(a+2) << endl; //0
cout<< *(a+1) << endl; //5
Your Computer’s Memory…

…is just a giant 1D array!
2D Arrays in Memory

```c
int b[3][3];
```

What you visualize:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b[0][0]</td>
<td>b[0][1]</td>
<td>b[0][2]</td>
</tr>
<tr>
<td>b[1][0]</td>
<td>b[1][1]</td>
<td>b[1][2]</td>
</tr>
<tr>
<td>b[2][0]</td>
<td>b[2][1]</td>
<td>b[2][2]</td>
</tr>
</tbody>
</table>

In memory:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b[0][0]</td>
<td>b[0][1]</td>
<td>b[0][2]</td>
<td>b[1][0]</td>
<td>b[1][1]</td>
<td>b[1][2]</td>
</tr>
<tr>
<td>b[2][0]</td>
<td>b[2][1]</td>
<td>b[2][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2D arrays are **1D arrays of 1D arrays**!
int b[3][3];

Assume integers and memory addresses are 4 bytes.
//sizeof prints the “actual” size of the array if called within
//same scope where array was made
cout<< sizeof(b);  //36(bytes)
//b+0is seen as a pointer type by compiler
cout<< sizeof(b+0);  //4(bytes)
//*(b+0)gives you the 1D array at b[0]
cout<< sizeof(*(b+0));  //12(bytes)
cout << b; //0x22fe30

Here, \textbf{b} is seen as a \textit{pointer to the first element} of the array

Here, \textbf{b}+1 points to the \textbf{next array element}, which is 12 bytes past array base (start) address

cout << b+1; //0x22fe3c
cout << &b;  //0x22fe30
Here, &b holds the *address of the array*, and is therefore a *pointer to the entire array*

cout << &b+1;  //0x22fe54
&b+i brings us *(i * sizeof(b)) bytes past the array’s base (starting) address*
How we would usually print array a:

```cpp
for(int i= 0; i< 4; ++i)
    cout<< a[i] << ' ';  
```

Note the following: 
\[ a[i] = *(a+i) = *(i+a) = i[a] \]

So the following works too…

```cpp
for(int i= 0; i< 4; ++i)
    cout<< i[a] << ' ';  
```

…never do this though!
static addressing

• Static array: You can not assign an array to another.

• Ex:
  int a[5];
  int b[5];
  a = b; //compile error

  int x = 5, y = 0;
  x = y;
C++ provides a operator specially for working with address.

* operator → “pointer” operator

Declare an integer pointer as: int *ip Do not confuse pointer with other operator, such as,

- int b[3][3]={0}, a = 4;
- a = a * 5; //this is multiplication
- a = 5 * /*this is also multiplication with comments */ a;
- a = 5 ***b; //what this line mean
- a = 5 **(*b+1)+1); //what this line mean
• int x = 5;

• int *ip = &x;

• Note: int *ip != int **ip
Hexadecimal

- Internally, all addresses (memory locations) are calculated as hex.
- Memory unit is byte.
- The size of int is 4 bytes
Binary

- Decimals - Base 10
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - 9
  - 10
  - 11
  - 12
  - 13
  - 14
  - 15
  - 16
  - 17

- Binary - Base 2
  - 0
  - 1
  - 10
  - 11
  - 100
  - 101
  - 110
  - 111
  - 1000
  - 1001
  - 1010
  - 1011
  - 1100
  - 1101
  - 1110
  - 1111
  - 10000
  - 10001

- Hexadecimal - Base 16
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - 9
  - A
  - B
  - C
  - D
  - E
  - F
  - 10
  - 11