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.

```
#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;
    // the next line prints 0012FF68
    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<<endl;
    return 0;
}
```
Your Computer’s Memory...

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

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

What you visualize:

```
b[0][0] b[0][1] b[0][2]
b[1][0] b[1][1] b[1][2]
b[2][0] b[2][1] b[2][2]
```

In memory:

```
b[0][0] b[0][1] b[0][2] b[1][0] b[1][1] b[1][2] b[2][0] b[2][1] b[2][2]
```

2D arrays are **1D arrays of 1D arrays**!
int a[4] = {1, 5, 0, 7};

You’re used to coding like this…

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

…here’s another way.

```
*(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
```
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 an operator specially for working with addresses.

- The * operator is called the "pointer" operator.

Declare an integer pointer as:

```
int *ip
```

Do not confuse pointer with other operators, such as:

```
int a = 4;
```

- `a = a * 5;` // This is multiplication

- `a = a * 5;` // This is also multiplication with comments */ 5;
• int a[5] = {0}, x = 5;

• int *ip = &x;

• Note: int *ip != int **ip
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.
Pointer Arithmetic

• 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 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+0 is 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);  //12(bytes)
cout << b; //0x22fe30
Here, b is seen as a **pointer to the first element** of the array

Here, b+1 points to the **next array element**, which is 12 bytes past array base (start) address
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
Hexadecimal

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

<table>
<thead>
<tr>
<th>Decimals - Base 10</th>
<th>Binary - Base 2</th>
<th>Hexadecimals - Base 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
<td>11</td>
</tr>
</tbody>
</table>