COMP0204: Introduction to Programming for Robotics and AI

Lecture 6: Pointers

Course lead: Dr Sophia Bano

MEng Robotics and AI

UCL Computer Science

Tips for improving programming skills

For those who think they are still behind in programming:

- **Reachout to the CS Programming Tutor – for one-to-one sessions!**
  (you should have information about this in your induction week notes)
- Make sure you revise the lectures
- Make sure to practice (exercises with solutions added in additional resources of each week)

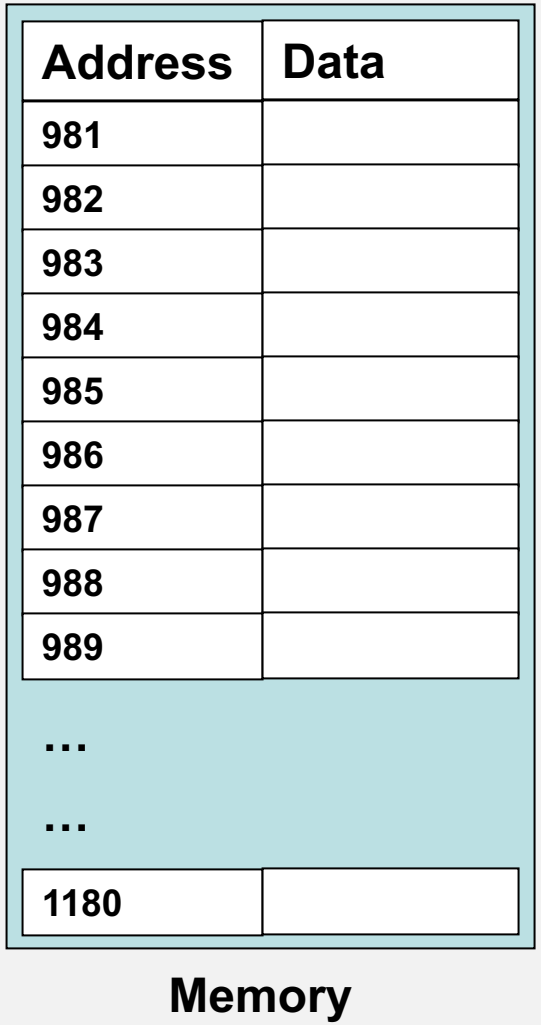
Today's lecture

- Pointers basics - declaration, referencing, dereferencing
- Pass by value vs pass by reference
- Arithmetic on Pointers
- Logical operators on Pointers
- Accessing arrays with Pointers
- 2D and multi-dimensional arrays with Pointers

Pointers basics

Memory addresses

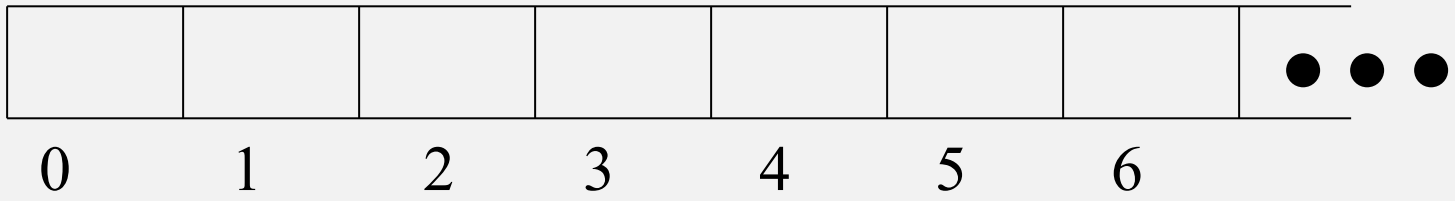
- All data (software instructions, variables, arrays) is in memory.
- Memory holds our program as it executes, and it also holds our program's variables.
- Memory addresses are all unique.
- When a variable is defined, C finds an unused place in memory and attaches the variable name to this location.
- Two variables defined back-to-back doesnot mean that C stores them back-to-back in memory.
- For any variable x, &x returns the address of x.



Memory

Bits and Bytes

- Memory is divided into bytes, each of which are further divided into bits
  - Each bit can have a value of 0 or 1
  - A byte is eight bits
    - Can hold any value from 0 to 255
  - Memory can be thought of as a sequence of bytes, numbered starting at 0



Storing Variables

- Each variables is stored in some sequence of bytes
  - Number of bytes depends on what?

  Number of bytes depends on the data type
  - Can two variables share a byte?

  Two variables will never share a byte – a variable uses all of a byte, or none of it

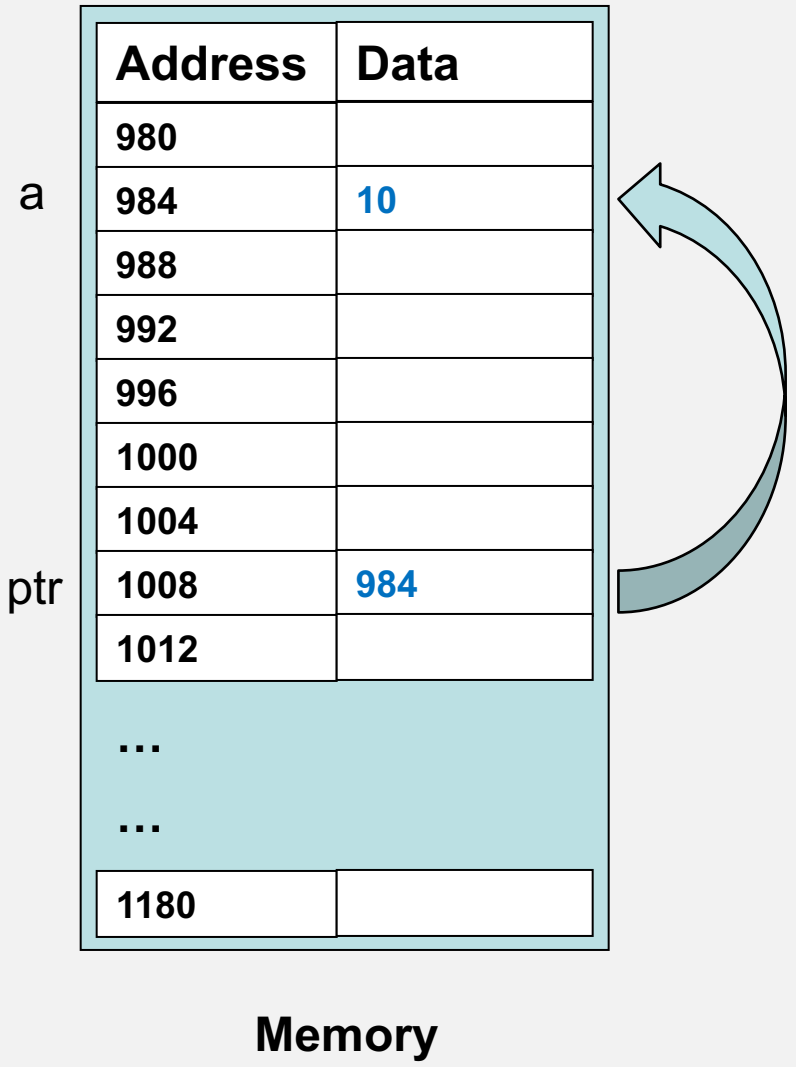
- Example:
  - An int is usually stored as a sequence of four bytes

Pointer basics

- A variable that stores the memory address of another variable
  - * - accessing or storing the value at the address in the pointer.
  - & - accessing or storing the memory **address** in the pointer.

```
int a = 10;
int *ptr = &a; // storing address of a
```

- But * has two different roles in pointers. What is the other role?

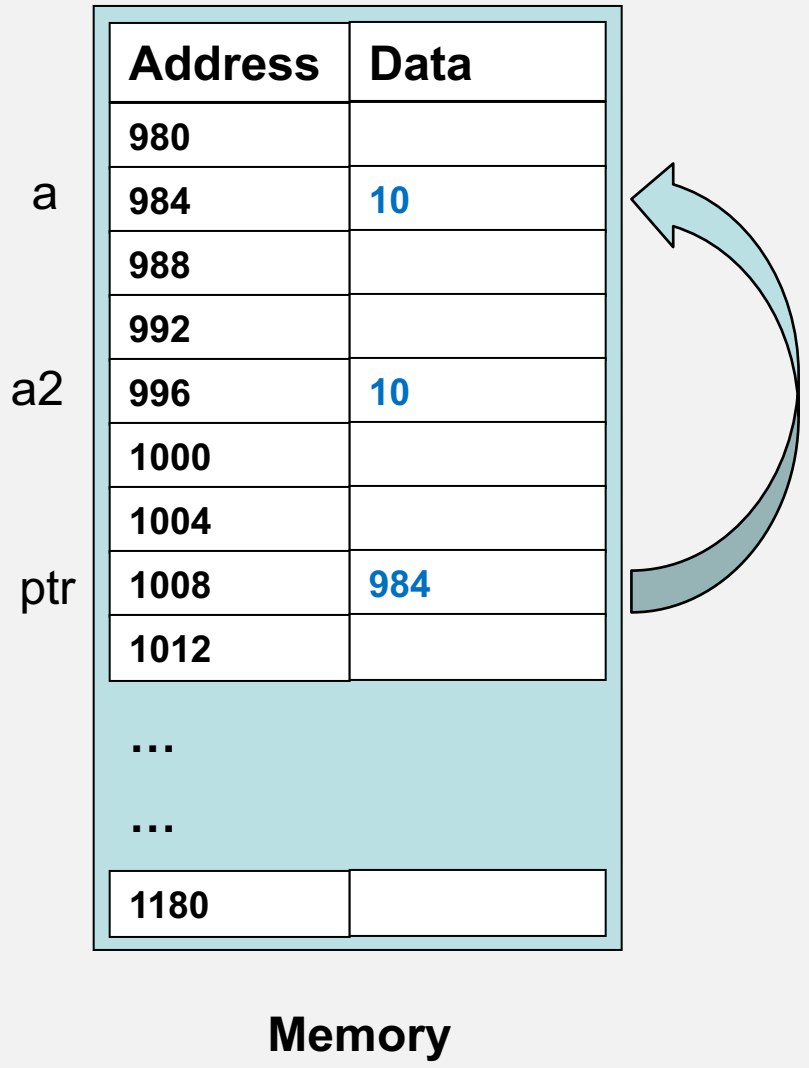


Pointer basics

- A variable that stores the memory address of another variable
  - * - accessing or storing the value at the address in the pointer.
  - & - accessing or storing the memory **address** in the pointer.

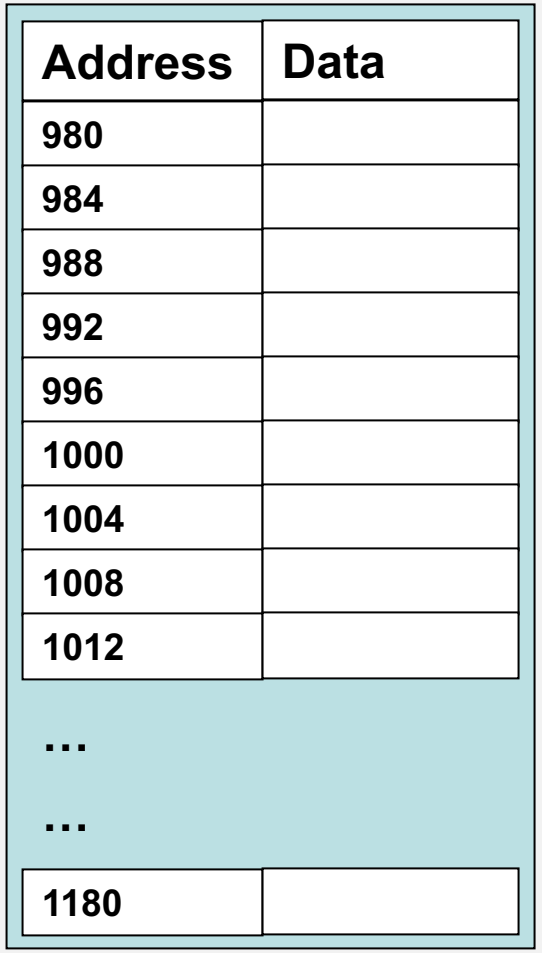
```
int a = 10;
int *ptr = &a; // storing address of a

int a2 = *ptr;
```

- Is * in pointer the same as the * in multiplication?



Pointer basics

- All data type have corresponding pointer data types.
- Only the address of same data type of variable can be assigned to pointer of the same data type.
- Pointer variables hold addresses of other variables.
- Until an address of a variable is assigned to pointer, the pointer remains uninitialized (indeterminate) and cannot be used.



Memory

Pointer basics

```
3   #include <stdio.h>
4
5   int main(){
6       int a = 10; // declare a variable
7       int *ptr = &a; // declare a pointer and assign the address of a to it
8
9       printf("The value of a is: %d\n", a);
10      printf("The address of a is: %p\n", &a);
11      printf("The address of ptr is: %p\n", &ptr);
12      printf("The value of ptr is: %p\n", ptr);
13      printf("The value of *ptr is: %d\n", *ptr);
14      return 0;
15  }
```

Referencing

De-referencing

Output

```
The value of a is: 10
The address of a is: 0061FF1C
The address of ptr is: 0061FF18
The value of ptr is: 0061FF1C
The value of *ptr is: 10
```

What will be the output of the following?

```
Printf("%d\n",*(&a));

Output: 10
```

Practice exercise


```
int *ptr;
int n;

ptr = &n; // address of n assigned to ptr
*ptr = 10; // value of n assigned to 10
printf("n = %d\n", n);
printf("*ptr = %d\n", *ptr);

*ptr += 1; // value of n incremented by 1
printf("n = %d\n", n);
printf("*ptr = %d\n", *ptr);

(*ptr)++;
printf("n = %d\n", n);
printf("*ptr = %d\n", *ptr);
```

Output:
n = 10
*ptr = 10
n = 11
*ptr = 11
n = 12
*ptr = 12

Pointers & Allocation

- After declaring a pointer:

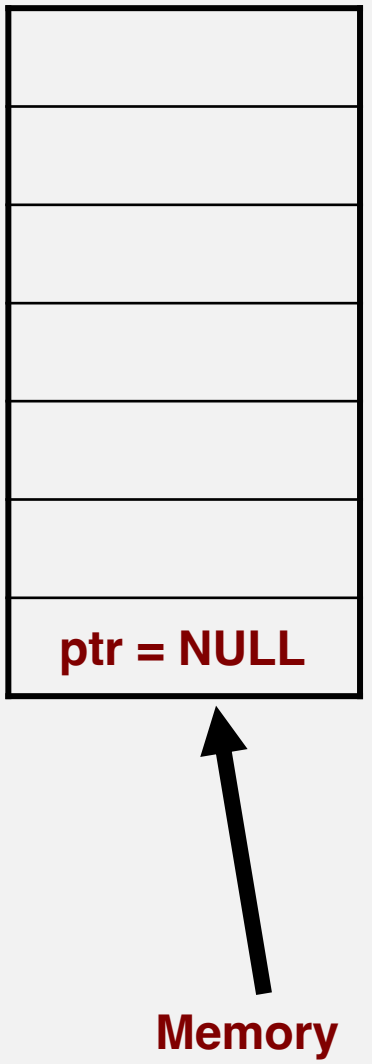
  `int *ptr1;`

  `ptr1` doesn't actually point to anything yet. It will contain some garbage value (indeterminate).
  (**Note:** some compiler by default assigns this to NULL)

  int *ptr1 = NULL; // address is NULL

- We can either:
  - make it point to something that already exists,

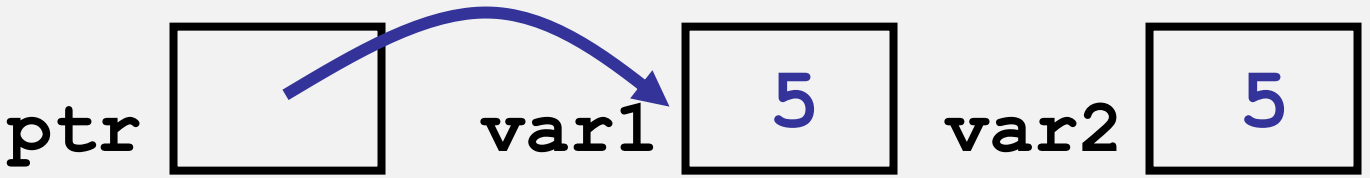
    `ptr1 = &c`
  - or
  - allocate room in memory for something new that it will point to… (dynamic memory will discuss later)



Pointers & Allocation

- Pointing to something that already exists:

```
int *ptr, var1, var2;
var1 = 5;
ptr  = &var1;
var2 = *ptr;
```



More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!

- What does the following code do?

```
void f()
{
    int *ptr;
    *ptr = 5;
}
```

TAKEAWAY:
- Always initialize pointers
  Must at least set to NULL

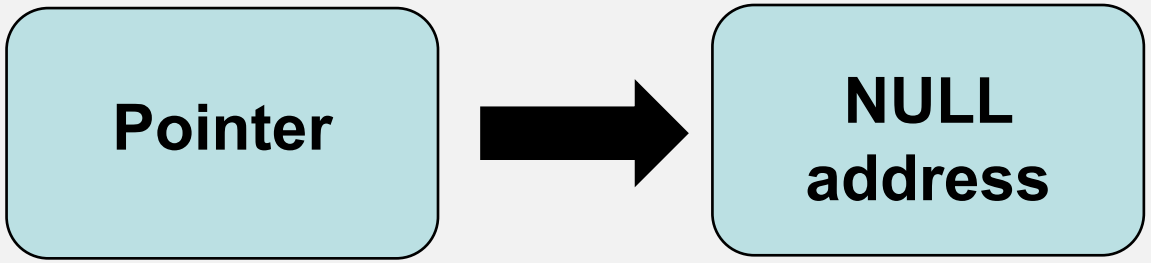
We can't store anything in the pointer (ptr) unless ptr contains some address.

Pointer Types - NULL Pointer

If you assign a NULL value to a pointer during its declaration, it is called Null Pointer.

Syntax:

Int *var = NULL;



Example:

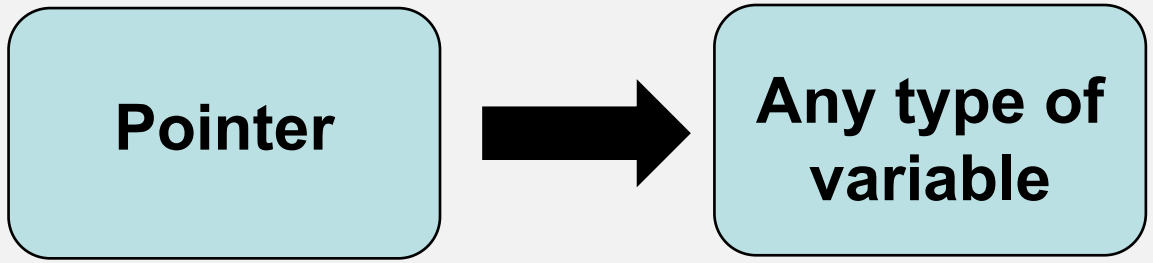
```
#include<stdio.h>
int main()
{
int *var = NULL;
printf("var=%d",*var);
}
```

Pointer Types - Void Pointer

When a pointer is declared with a void keyword, then it is called a void pointer. To print the value of this pointer, you need to typecast it.

Syntax:

```
void *var;
```



Example:

```
#include<stdio.h>
int main(){
  int a=2;
  void *ptr;
  ptr= &a;
  printf("After Typecasting, a = %d", *(int *)ptr);
  return 0;
}
```

Pointer Operators

Arithmetic on Pointers

- A pointer may be incremented or decremented.
- This means only address in the pointer is incremented or decremented.
- An integer (can be constant) may be added to or subtracted from a pointer.
- Pointer variables may be subtracted from one another.
- Pointer variables can be used in comparisons, but usually only in a comparison to pointer variables or NULL.

Arithmetic on Pointers

- When an integer (n) is added to or subtracted from a pointer (ptr)
  - The new pointer value (ptr) is changed by

    ptr current value + n * (bytes of ptr data type)

Address	data
1000	
1004	
1008	
1012	
1016	

Example 1:

```
int *ptr;
ptr = 1000;
ptr = ptr + 2;
// ptr address is now changed to 1000 + 2*4 (because integer
consumes 4 bytes) New address is now 1008
```

Remember: The operator (+ / -) on a pointer updates its location (address) based on the operator.

Arithmetic on Pointers

Example 2:

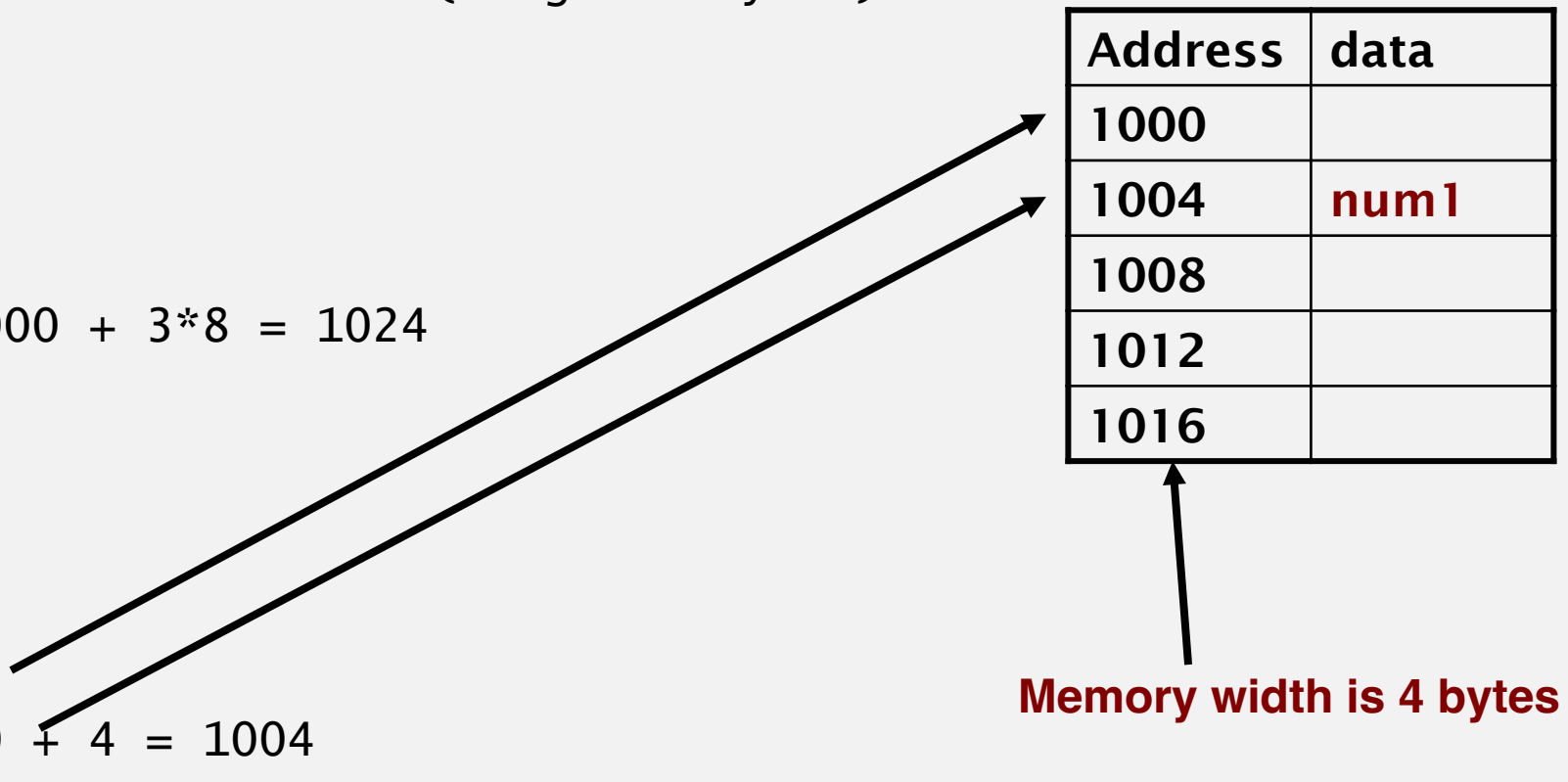
```
long *ptr;
ptr = 1000;
ptr++; // 1000 + 1*8 = 1008 (long = 8 bytes)
```

Example 3:

```
float *ptr;
ptr = 1000;
ptr+=3; // 1000 + 3*8 = 1024
```

Example 4:

```
int *ptr;
int num1 = 0;
ptr = &num1;
ptr++; //1000 + 4 = 1004
```



Arithmetic on Pointers

Example 4:

```
int *ptr;
int num1 = 0;
ptr = &num1;
ptr++; //1000 + 4 = 1004
```

```c
#include <stdio.h>

int main()
{
    int *ptr;
    int num1 = 0;
    ptr = &num1;

    printf("%u\n",ptr);
    printf("%d\n",*ptr);
    ptr++; // 1000 + 1*4 = 1004

    printf("%u\n",ptr);
    printf("%d",*ptr);

    return 0;
}
```

Output:

```
2465927468
0
2465927472
-1829039824
```

Arithmetic on Pointers

Example 5:

```
int main (){
    int *pointer1, *pointer2;
    int num1 = 93;
    pointer1 = &num1; //address of num1
    pointer2 = pointer1; //pointer1 address is assign to pointer2

    printf("The value of num1 is: %d\n", num1);
    printf("pointer1 value is: %u\n", pointer1);
    printf("pointer2 value is: %u\n", pointer2);

    printf("pointer1 address %u\n", &pointer1);
    printf("pointer2 address %u\n", &pointer2);

    return 0;
}
```

Address	data
1000	
1004	num1 = 93
1008	
1012	
1016	pointer1 = 1004
1020	
1024	
1028	pointer2 = 1004

Arithmetic on Pointers

Example 5:

```
int main (){
    int *pointer1, *pointer2;
    int num1 = 93;
    pointer1 = &num1; //address of num1
    pointer2 = pointer1; //pointer1 address is assign to pointer2

    printf("The value of num1 is: %d\n", num1);
    printf("pointer1 value is: %u\n", pointer1);
    printf("pointer2 value is: %u\n", pointer2);

    printf("pointer1 address %u\n", &pointer1);
    printf("pointer2 address %u\n", &pointer2);

    return 0;
}
```

Address	data
1000	
1004	num1 = 93
1008	
1012	
1016	pointer1 = 1004
1020	
1024	
1028	pointer2 = 1004

Logical operators on Pointers

- We can apply logical operators (<, >, <=, >=, ==, !=) on pointers.
  - But remember pointers can be compared to pointers or NULL

- Example 6:

```c
#include <stdio.h>

int main(){
    int *pointer1 = NULL, *pointer2 = NULL; // both pointer2 contains NULL addresses
    int num1 = 93;

    printf("The value of pointer1 is: %p\n", pointer1);
    if( pointer1 == NULL ){ // pointer compared to NULL
        pointer1 = &num1;
        printf("The value of pointer1 is: %p\n", pointer1);
        }
    pointer2 = &num1;
    if( pointer1 == pointer2 ){ // pointer compared to pointer
        printf("Both pointers are equal");
        printf("The value of pointer2 is: %p\n", pointer2);
    }

    return 0;
}
```

Output:

```
The value of pointer1 is: 0000000000000000
The value of pointer1 is: 000000ed107ffccc
Both pointers are equal
The value of pointer2 is: 000000ed107ffccc
```

Pointer Operators (using & and * operators)

- * and & are inverses of each other
  - Will “cancel one another out” when applied consecutively to either order

Pointer Operators (using & and * operators)

```c
#include <stdio.h>

int main(){
    int a; // a is an integer
    int *aPtr; // aPtr is a pointer to an integer

    a = 7; // assign 7 to a
    aPtr = &a; // assign address of a to aPtr

    printf("The address of a is %p", &a);
    printf("\nThe value of aPtr is %p", aPtr);
    printf("\n\nThe value of a is %d", a);
    printf("\nThe value of *aPtr is %d", *aPtr);
    printf("\n\nShowing that * and & are complements of ");
    printf("each other\n&*aPtr = %p", &*aPtr);
    printf("\n*&aPtr = %p\n",*&aPtr);

    return 0;
}
```

Output:

```
The address of a is 000000a5d9fff81c
The value of aPtr is 000000a5d9fff81c

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 000000a5d9fff81c
*&aPtr = 000000a5d9fff81c
```

Address of **a** and the value of **aPtr** are

Value of **a** and the dereferenced **aPtr** are identical

***** and **&** are inverses; same result when both are applied to **aPtr**

Passing Pointers to Function

Passing Pointers to Function

- We know pointers contains two things (address and value in the address).
- We can pass the pointers to functions as input parameter or output parameter as
  - By value and
  - By address
- When we pass the pointers to functions by value. It is called passing by value.
- When we pass the pointers to functions by address. It is called passing by reference.

Passing Pointers to Function

- A function can return only one value
- Arguments passed to a function using reference arguments or pointer arguments
  - Function can modify original values of arguments
  - More than one value “returned”

Passing Pointers to Function

- Pass-by-reference with pointer arguments
  - Pass address of argument using & operator
  - Arrays not passed with & because array name is already a pointer

Passing Pointers to Function

Passing by value (example)

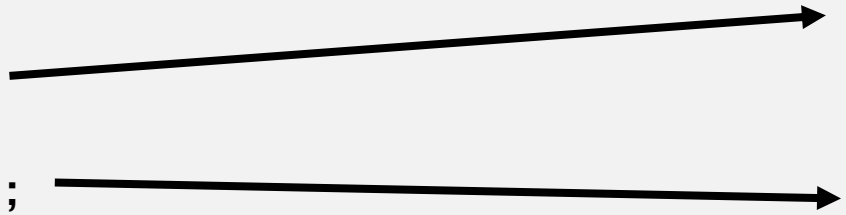
```
void fun1(int pass_by_value){
    printf("value is %d\n", pass_by_value);
}

int main(){
    int *pointer;
    int num1 = 10;

    pointer = &num1;
    fun1(*pointer); // Pass by value
    return 0;
}
```

Output:
The value is 10

Passing Pointers to Function

Passing by reference (Example)

```
void fun1 (int *pass_by_reference){
    printf("The value is %d\n", *pass_by_reference);
    *pass_by_reference = *pass_by_reference + 5;
}

int main (){
    int *pointer;
    int num1 = 10;

    pointer = &num1;
    fun1 (pointer); // address of num1 is passed
    fun1 (&num1); // address of num1 is passed
    printf("The value is %d\n", num1);

    return 0;
}
```



Address	data
1000	
1004	num1 = 10
1008	
1012	pointer = 1004
1016	

Output:

?

Practice – pass by value and pass by reference

- Performing sum, product and average using pointers

```c
#include <stdio.h>

void sum_prod_avg(int a, int b, int *sum, int *product, int *avg);

int main(){
    int a = 3, b = 7; // local variables
    int sum, product, avg;

    // address of sum, product, avg are passed
    sum_prod_avg(a, b, &sum, &product, &avg);

    printf("The sum is %d\n", sum);
    printf("The product is %d\n", product);
    printf("The avg is %d\n", avg);

    return 0;
}

// a and b - pass by value, sum, product, avg - pass by reference
void sum_prod_avg(int a, int b, int *sum, int *product, int *avg){
    *sum = a + b;
    *product = a * b;
    *avg = (a + b) / 2;
}
```

Output:

```
The sum is 10
The product is 21
The avg is 5
```

sizeof Operator

- Returns size of operand in bytes
- For arrays, sizeof returns
  - (size of 1 element) * (number of elements)
- If `sizeof(int)` returns 4 then

```
int myArray[ 10 ];
printf("%d", sizeof(myArray));
```

  will print 40
- Can be used with
  - Variable names
  - Type names
  - Constant values

Relationship Between Pointers and Arrays

Relationship Between Pointers and Arrays

- Arrays and pointers are closely related
  - Array name is like constant pointer
  - Pointers can do array subscripting operations

Relationship Between Pointers and Arrays

- Accessing array elements with pointers
  - Assume declarations:

```
int b[ 5 ];
int *bPtr;
bPtr = b;
```

  - Element `b[n]` can be accessed by `*(bPtr + n)`
    - Called pointer/offset notation
  - Addresses
    - &b[3] is same as bPtr + 3
  - Array name can be treated as pointer
    - b[3] is same as *(b + 3)
  - Pointers can be subscripted (pointer/subscript notation)
    - bPtr[3] is same as b[3]

Using subscripting and pointer notations with arrays

```c
#include <stdio.h>

int main(){
    int b[] = {10, 20, 30, 40}; // create 4-element array b
    int *bPtr = b; // set bPtr to point to array b

    //output array b using array subscript notation
    printf("Array b printed with:\nArray subscript notation\n");
    for (size_t i = 0; i < 4; ++i){
        printf("b[%u] = %d\n", i, b[i]);
    }

    //output array b using array name and pointer/offset notation
    printf("\nPointer/offset notation where\n"
           "the pointer is the array name\n");
    for (size_t offset = 0; offset < 4; ++offset)
        printf("*(b + %u) = %d\n", offset, *(b + offset));

    // output array b using bPtr and array subscript notation
    printf("\nPointer subscript notation\n");
    for (size_t i = 0; i < 4; ++i)
        printf("bPtr[%u] = %d\n", i, bPtr[i]);

    // output array b using bPtr and pointer/offset notation
    printf("\nPointer/offset notation\n");
    for (size_t offset = 0; offset < 4; ++offset)
        printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));

    return 0;
}
```

Using array subscript notation

Using array name and pointer/offset notation

Using pointer subscript notation

Using pointer name and pointer/offset notation

Using subscripting and pointer notations with arrays

```c
#include <stdio.h>

int main(){
    int b[] = {10, 20, 30, 40}; // create 4-element array b
    int *bPtr = b; // set bPtr to point to array b

    //output array b using array subscript notation
    printf("Array b printed with:\nArray subscript notation\n");
    for (size_t i = 0; i < 4; ++i){
        printf("b[%u] = %d\n", i, b[i]);
    }

    //output array b using array name and pointer/offset notation
    printf("\nPointer/offset notation where\n"
           "the pointer is the array name\n");
    for (size_t offset = 0; offset < 4; ++offset)
        printf("*(b + %u) = %d\n", offset, *(b + offset));

    // output array b using bPtr and array subscript notation
    printf("\nPointer subscript notation\n");
    for (size_t i = 0; i < 4; ++i)
        printf("bPtr[%u] = %d\n", i, bPtr[i]);

    // output array b using bPtr and pointer/offset notation
    printf("\nPointer/offset notation\n");
    for (size_t offset = 0; offset < 4; ++offset)
        printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));

    return 0;
}
```

Output:

```
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Accessing 1-Demensional Array

- We know, Array name denotes the memory address of its first slot.
  - Example:

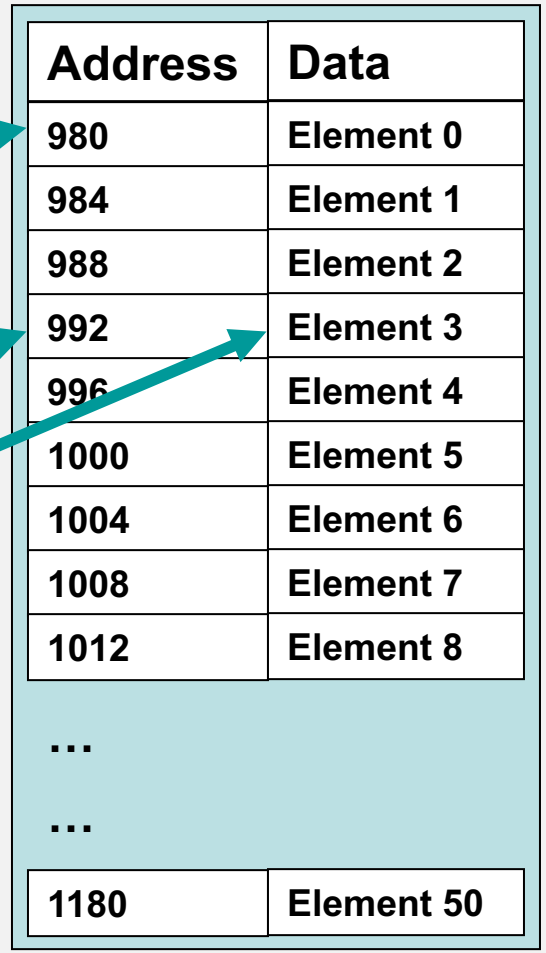
```
int List [ 50 ];
int *Pointer;
Pointer = List;
```

- Other slots of the Array (List [50]) can be accessed by performing Arithmetic operations on Pointer.
- For example, the address of (element 4th) can be accessed using:

```
int *Value = Pointer + 3;
```

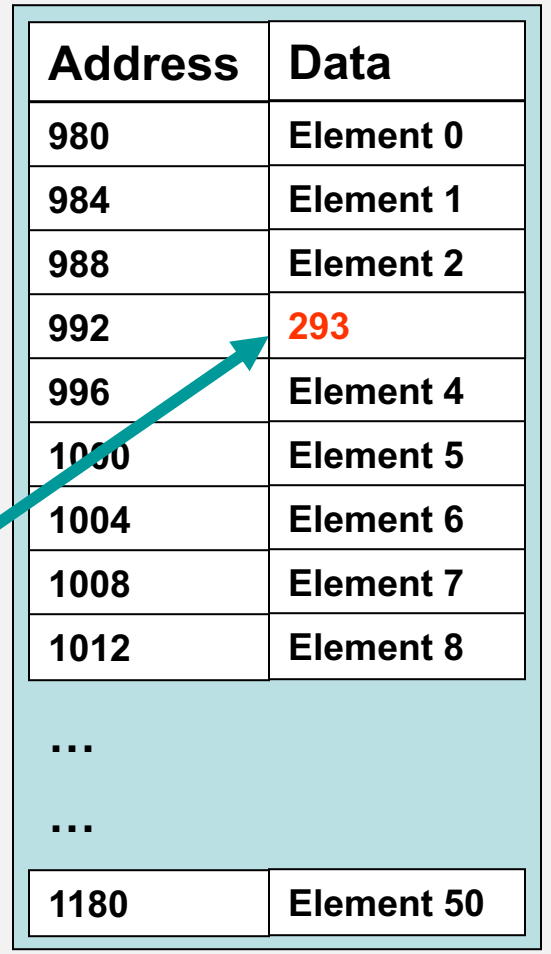
- The value of (element 4th) can be accessed using:-

```
int Value = *(Pointer + 3);
```



Accessing 1-Demensional Array

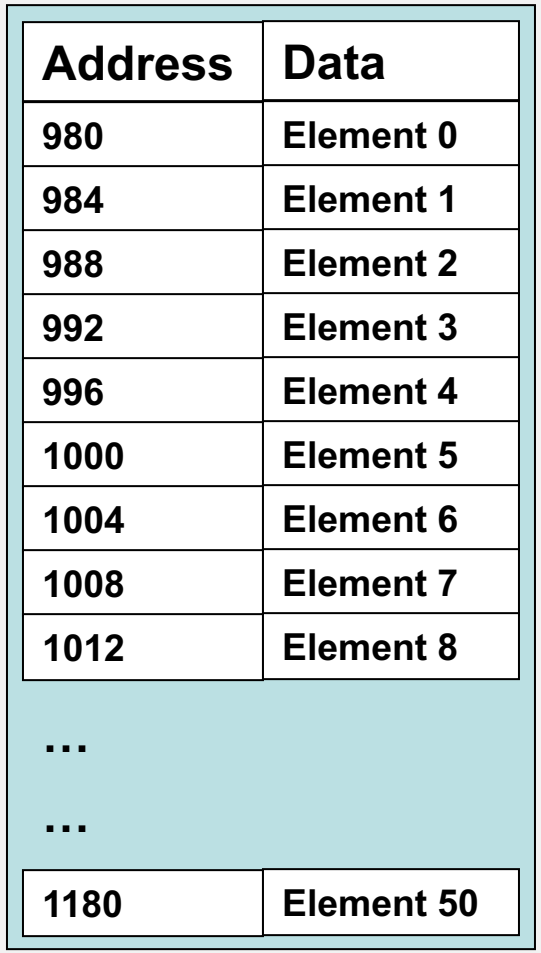
```
#include <stdio.h>
int main (){

    int List [ 50 ];
    int *Pointer;
    Pointer = List; // Address of first Element

    int *ptr;
    ptr = Pointer + 3; // Address of 4th Element
    *ptr = 293; // 293 value store at 4th element
    address

    printf("Value is %d", *ptr);
    return 0;
}
```



Accessing 1-Demensional Array

We can access all element of List [50] using Pointers and for loop combinations.

```
#include <stdio.h>
int main (){

    int List [ 50 ];
    int *Pointer;
    Pointer = List;
    for ( int i = 0; i < 50; i++ ){
        printf("%d\n",*Pointer);
        Pointer++; // Address of next element
    }
    return 0;
}
```



Address	Data
980	Element 0
984	Element 1
988	Element 2
992	Element 3
996	Element 4
1000	Element 5
1004	Element 6
1008	Element 7
1012	Element 8
...	
...	
1180	Element 50

This is Equivalent to

```
for ( int loop = 0; loop < 50; loop++ )
    printf("%d\n"), List[ loop ]);
```

Accessing 2-Demensional Array

- Note that the statements
  - int *Pointer;
  - Pointer = &List [3];
- represents that we are accessing the address of 4th slot.

- In 2-Demensional array the statements
  - int List [5] [6];
  - int *Pointer;
  - Pointer = List[3];
- Represents that we are accessing the address of 4th row **or** the address the 4th row and 1st column.

Address	Data
980	Element 0
984	Element 1
988	Element 2
992	Element 3
996	Element 4
1000	Element 5
1004	Element 6
1008	Element 7
1012	Element 8
…	
…	
1180	Element 50

Exercise: Pre/post increment/decrement


```
int main(){
    int a[] = {2, 4, 8, 10, 12, 14, 16, 18};
    int *p = &a[0];

    printf("%d ", *(p++));
    printf("%d ", *p);

    return 0;
}
```

Output:
?

Address	Data
980	2
984	4
988	8
992	10
996	12
1000	14
1004	16
1008	18
1012	
…	
…	
1180	

Exercise: Pre/post increment/decrement


```
int main(){
    int a[] = {2, 4, 8, 10, 12, 14, 16, 18};
    int *p = &a[0];

    printf("%d ", *(++p));
    printf("%d ", *p);

    return 0;
}
```

Output:
?

Address	Data
980	2
984	4
988	8
992	10
996	12
1000	14
1004	16
1008	18
1012	
…	
…	
1180	

Exercise: Pre/post increment/decrement


```
int main(){
    int a[] = {2, 4, 8, 10, 12, 14, 16, 18};
    int *p = &a[2];

    printf("%d ", *(--p));
    printf("%d ", *(p--));

    return 0;
}
```

Output:
?

Address	Data
980	2
984	4
988	8
992	10
996	12
1000	14
1004	16
1008	18
1012	
…	
…	
1180	

Comparing pointers – explained with arrays

- Use relational operators (<,>,<=,>=) and equality operators (==,!=) to compare pointers
- Only possible when both pointers point to same array
- Output depends upon the relative positions of both the pointers

Exercise: Comparing pointers


```
int main(){
    int a[] = {2, 4, 8, 10, 12, 14, 16, 18};
    int *p = &a[3];
    int *q = &a[5];

    printf("%d\n", p<=q);
    printf("%d\n", p>=q);

    q = &a[3];
    printf("%d ", p==q);

    return 0;
}
```

Output:
?

Address	Data
980	2
984	4
988	8
992	10
996	12
1000	14
1004	16
1008	18
1012	
…	
…	
1180	

Exercise: Pointers operations


```
int main(){
    int a[] = {2, 4, 8, 10, 12, 14, 16, 18};
    int *ptr1 = &a[1], *ptr2 = &a[5];

    printf("%d\n", *(ptr1+3));
    printf("%d\n", *(ptr2-4));
    printf("%d\n", ptr2-ptr1);
    printf("%d\n", ptr1<ptr2);
    printf("%d\n", *ptr1<*ptr2);

    return 0;
}
```

Output:
?

Address	Data
980	2
984	4
988	8
992	10
996	12
1000	14
1004	16
1008	18
1012	
…	
…	
1180	

Exercise: Sum of elements of array using pointer

Given an array containing five positive integers, calculate the sum of the elements of array using pointer.

Exercise: Minimum and maximum element of an array

Given an array containing five positive integers, find the minimum and maximum element in this array using pointer.

Accessing 2-Demensional Array

int b[2][3]

b[0]
b[1]

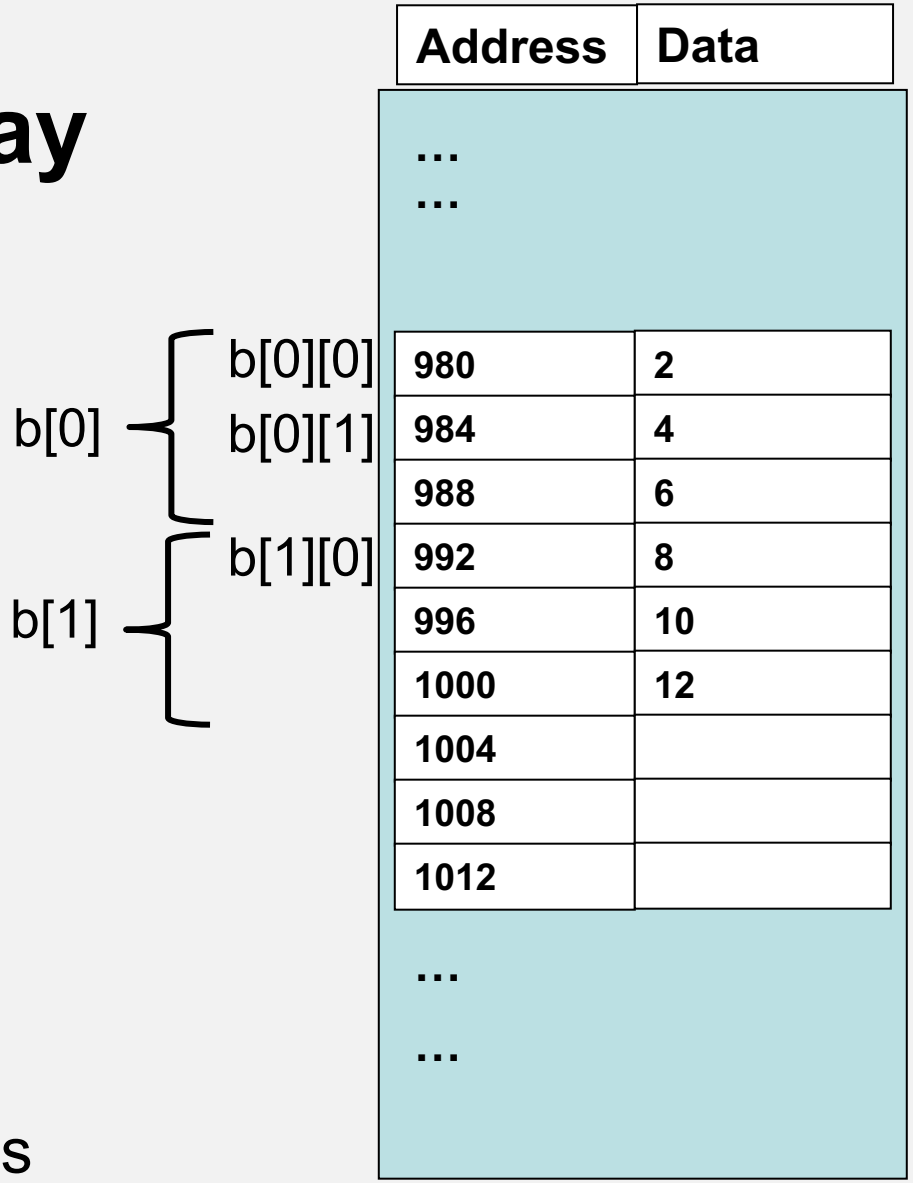
1D arrays of 3 elements each

int *p = b; **ERROR**

(return a pointer to 1D array of 3 integers)

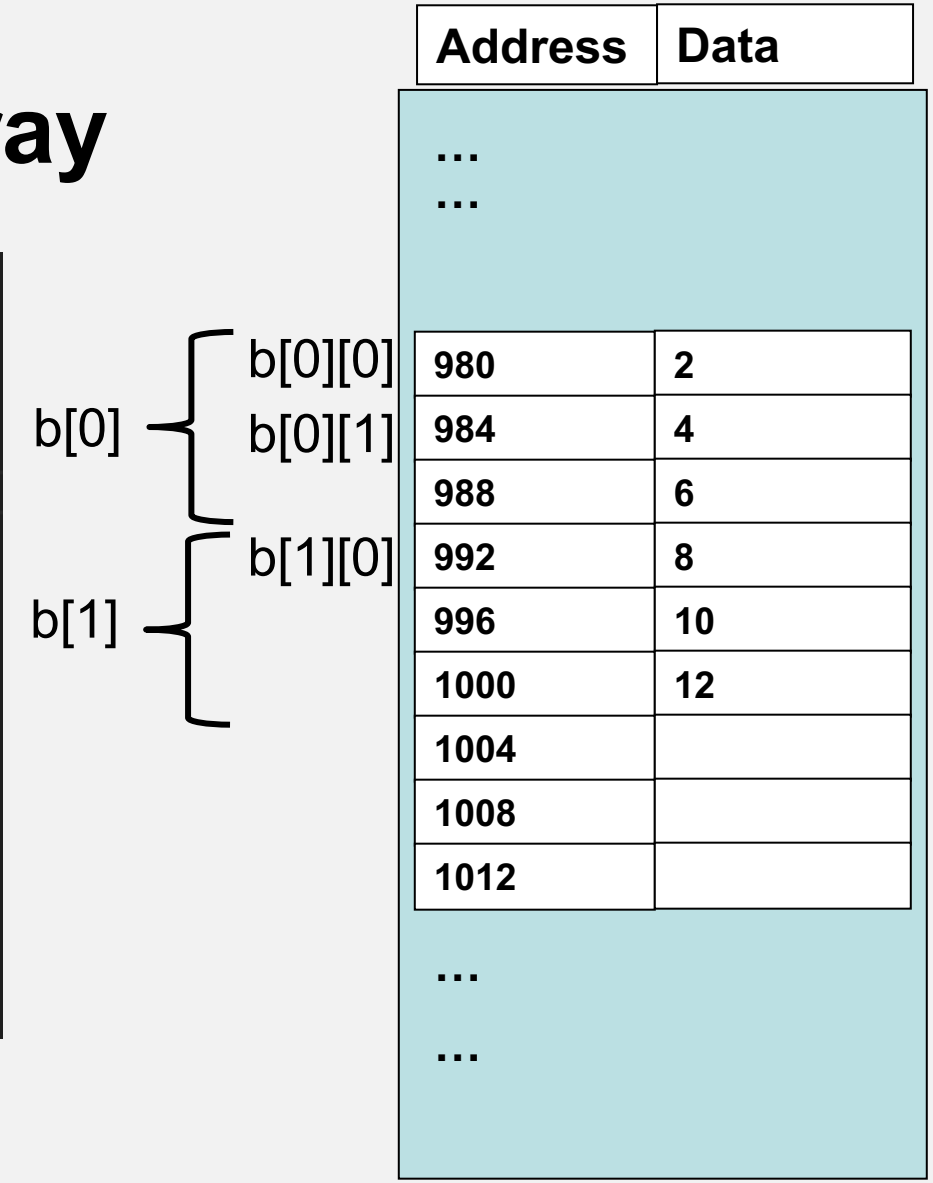
int *p = b[0]; **CORRECT**

b[i][j], *(b[i]+j), *(*(b+i)+j) are all same expressions



Accessing 2-Demensional Array

```
3   int main(){
4       int b[2][3] = {{2, 4, 6}, {8, 10, 12}};
5       int *p = b[0];
6       int *q = b[1];
7
8       printf("*b = %p, b[0] = %p, &b[0][0]=%p\n", *b, b[0], &b[0][0]);
9
10      printf("**b = %d\n", *(*b));
11      printf("*p = %d\n\n", *p);
12
13      printf("*(*b+1) = %d\n", *(*b+1));
14      printf("*(p+1) = %d\n\n", *(p+1));
15
16      printf("*(*(b+1)+1) = %d\n", *(*(b+1)+1));
17      printf("*(q+1) = %d\n\n", *(q+1));
18
19      return 0;
20  }
```



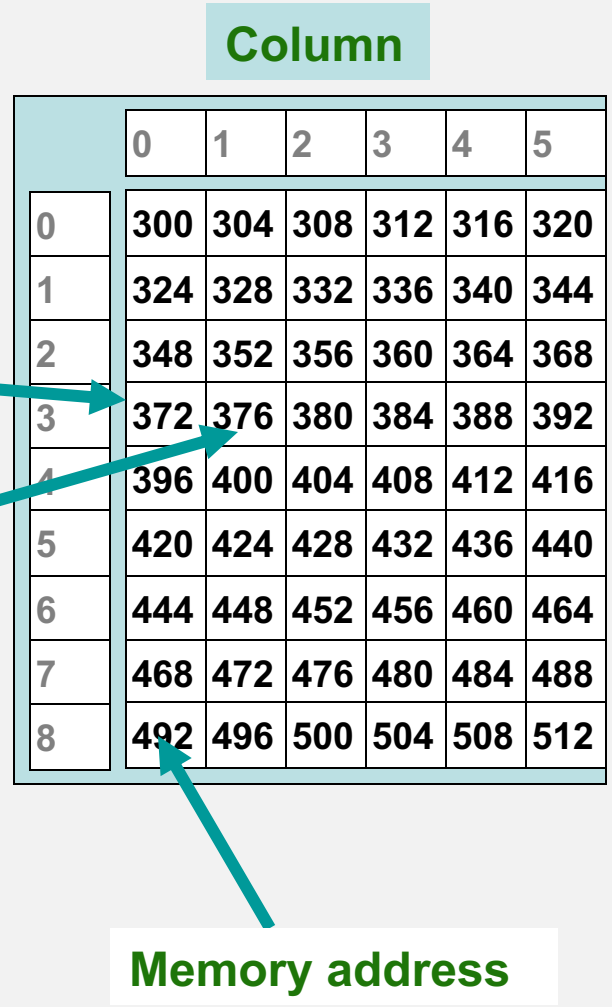
Accessing 2-Demensional Array

```
int List [ 9 ] [ 6 ];
int *ptr;
ptr = List [3];
```

- To access the address of 4th row 2nd column, we can increment the value of (ptr).

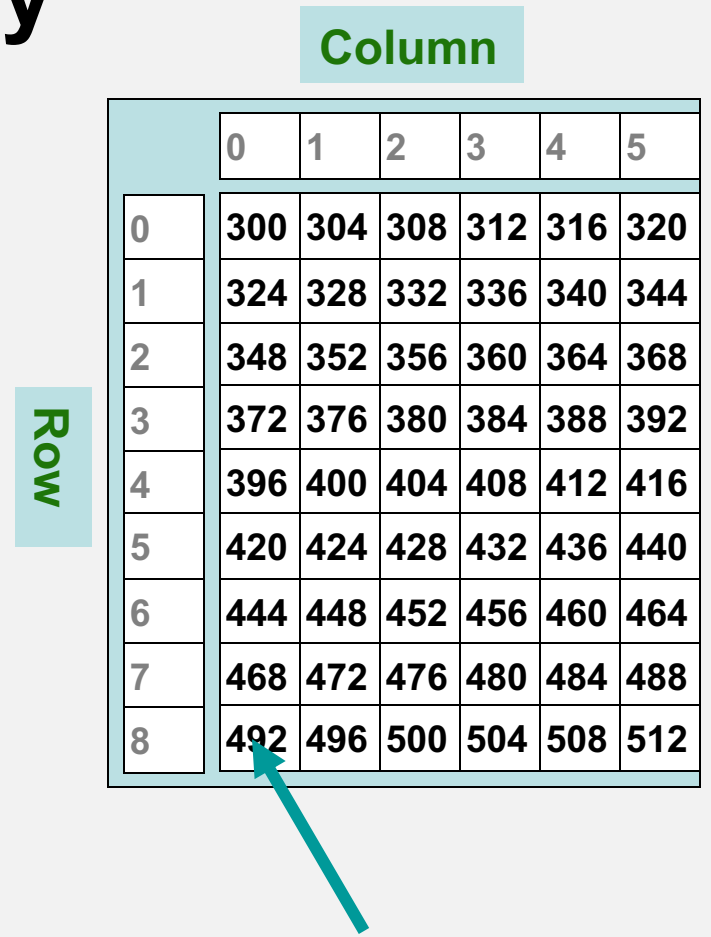
  ptr++; // address of 4th row 2nd column

  Equivalent to List [3][1] ;



Accessing 2-Demensional Array

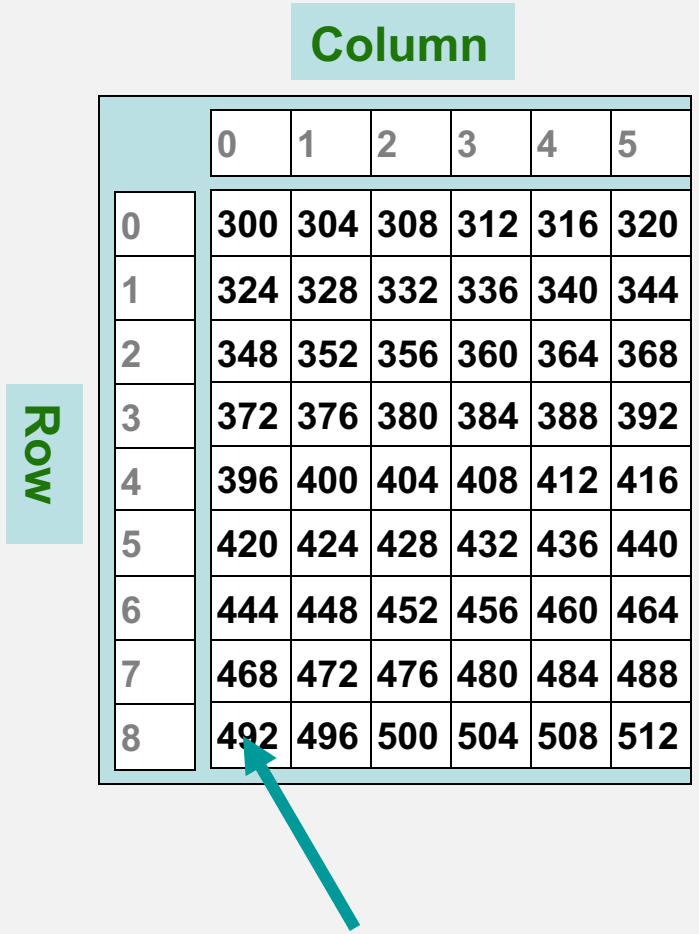
- Suppose you have to write a C code which calculates the average of the values of 6*9 table:-
- There are two methods to calculate the average of table
  - **Without pointer**
  - **With pointer**



Accessing 2-Demensional Array

Without pointer

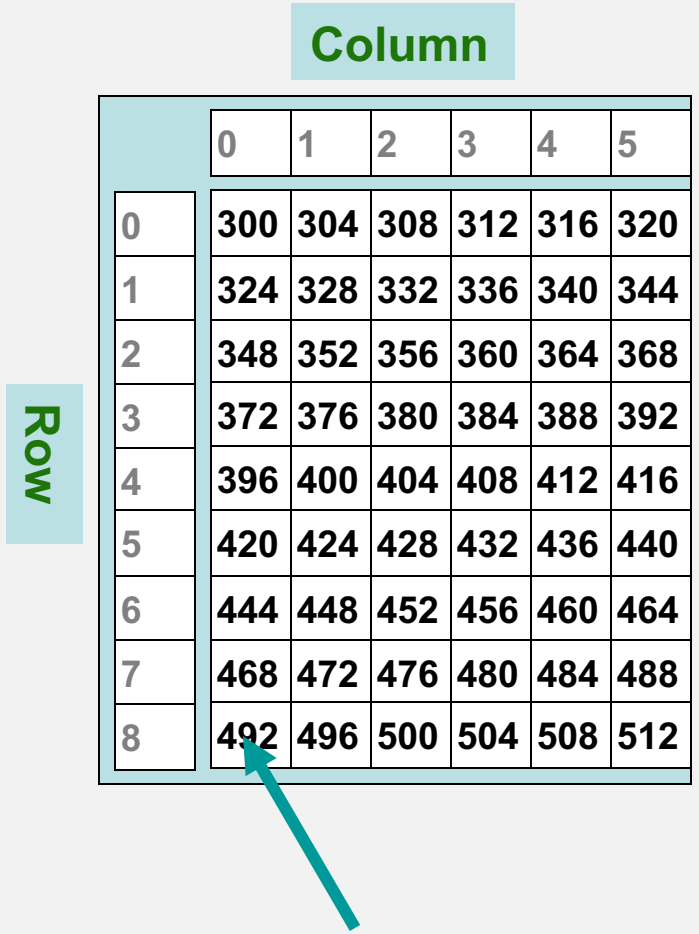
```
#include <iostream.h>
void main (void)
{
    int List [ 9 ][ 6 ];
    int Sum = 0;
    for ( int row = 0; row < 9; row++ ){
        for ( int col = 0; col < 6; col++ ){
            sum += List [row][col];
        }
    }
    cout << “Average is “ << Sum/(9*6);
    getch ();
}
```



Accessing 2-Demensional Array

With pointer

```
#include <iostream.h>
void main (void)
{
      int List [ 9 ][ 6 ];
      int Sum = 0, *Pointer;
      for ( int row = 0; row < 9; row++ )
      {
            Pointer = List [row];
            for ( int col = 0; col < 6; col++ )
            {
                  sum += *Pointer;
                  Pointer++;
            }
      }
      cout << “Average is “ << Sum/(9*6);
      getch ();
}
```



Accessing 3D array with pointers

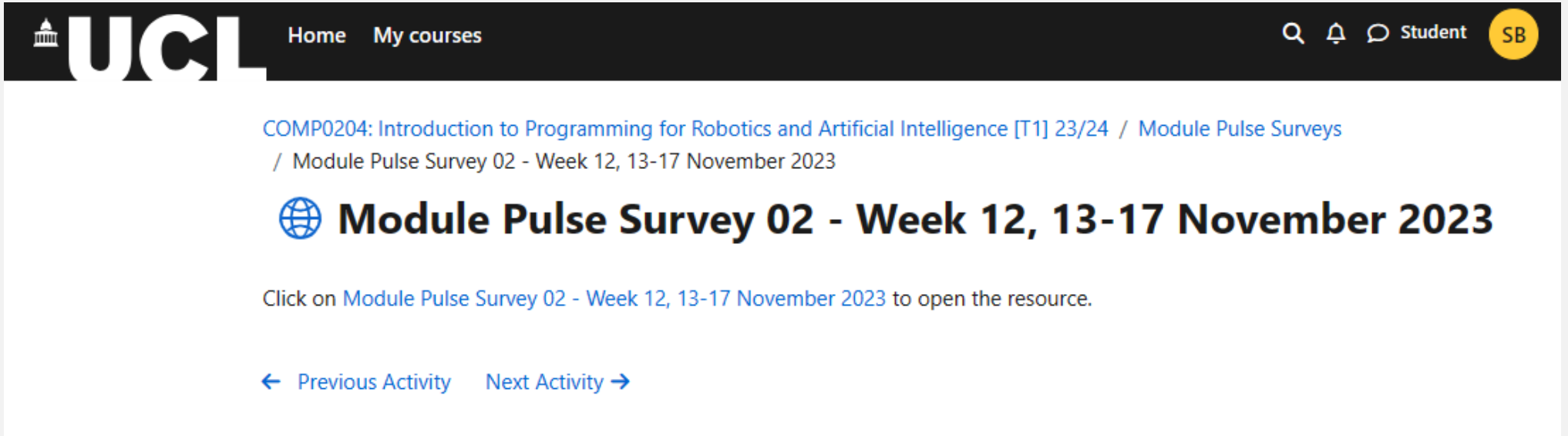
Declare a 3D array C of size 2 x 2 x 2.

Make the memory map showing how this will appear in the memory.

Print C, C[0], C[0][0]. What is the result? Discuss

Access C[0][0][1] and C[1][1][0] using pointers arithmetic

Module Pulse Survey



Assessment 5: Inperson lab assessment

- 27th Nov 2023

- Timed 40 minutes exercise

If you miss the inperson assessment, apply for EC.
No grade will be given to remotely submitted assessment

https://www.ucl.ac.uk/students/student-support-framework/short-term-illness-and-other-extenuating-circumstances