Lecture 2:

Composite Data Structures

LEGO blocks of data structures

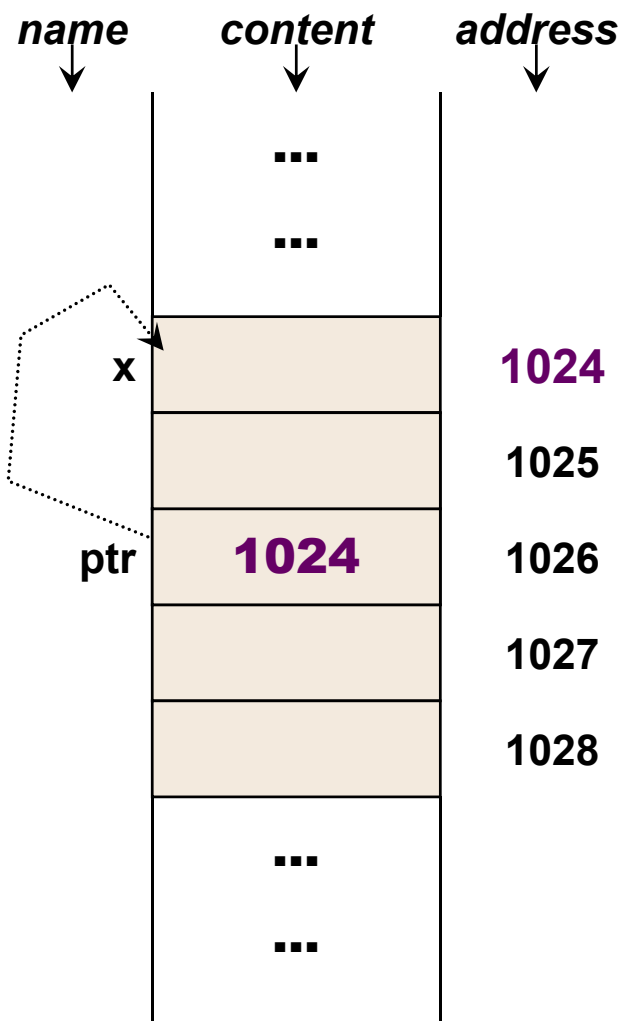
Lecture Outline

- Quick Recap of **Pointers**
- Quick Recap of **Reference**
- Composite Data Structures
  - Emphasis on **multidimensional array**
- Multidimensional Array Applications

Pointer: **Declaration**

- A ***pointer variable*** stores the address of a memory location

```
int x;

int *ptr;      ptr is an int pointer

ptr = &x;      ptr points to x
```



- The "**&**" operator gives the address of a variable
  - known as **address-of** operator
- The **ptr** variable **points to** the variable **x**
  - Hence the name **pointer**

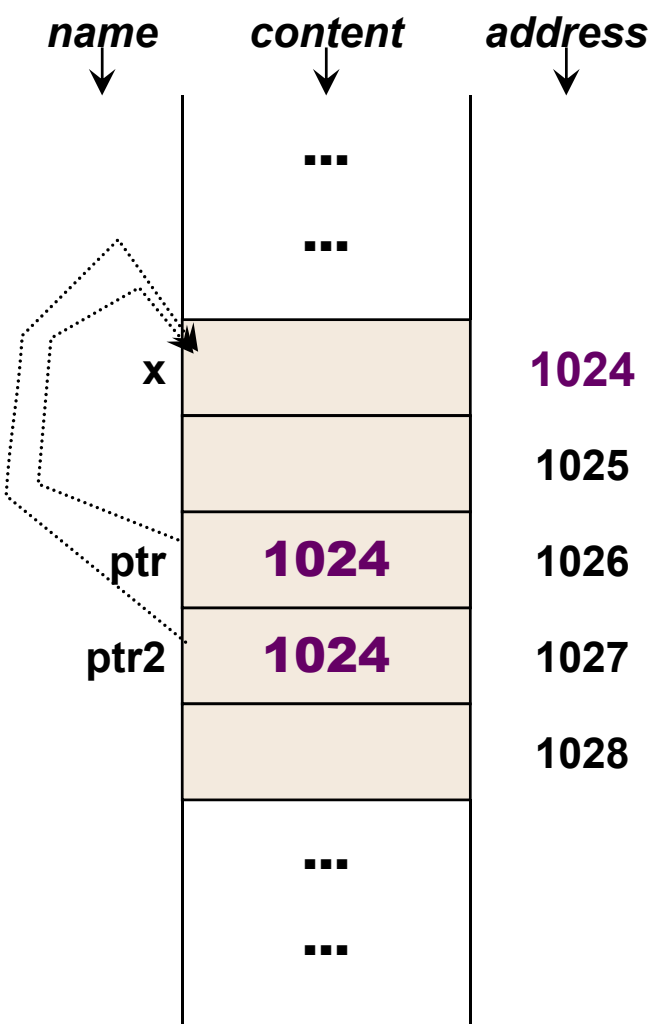
Pointer: **Usage**

- Pointer variable behaves in the same way as normal variable
  - Manipulation works in the same way

```
int x;

int *ptr, *ptr2;

ptr = &x;

ptr2 = ptr;
```

Note the "*"

Content of ptr copied over to ptr2

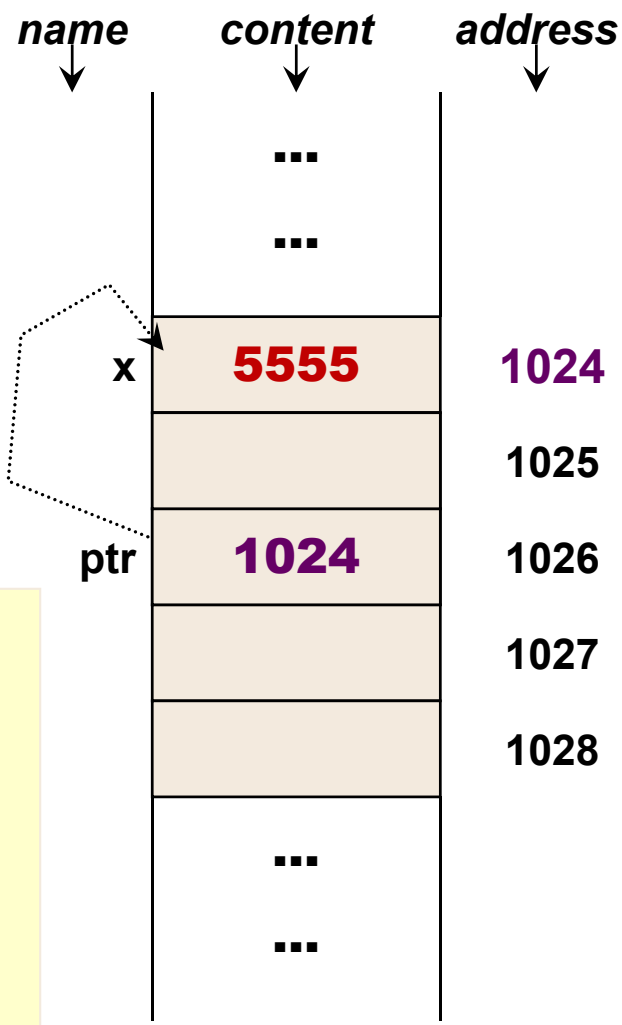


Pointer: **Dereferencing**

- We can **follow the address stored in a pointer variable and manipulate the destination**
  - Known as **dereferencing**

- A dereferenced pointer works like a normal variable of that type

```
int x;
int *ptr;

ptr = &x;

*ptr = 1234;          Same as x = 1234

*ptr = *ptr + 4321;   Same as x = x+4321
```



Pointer: **Common Confusion**

- Note the different meanings of the "*"
  - During declaration: **declare a pointer variable**
  - During usage: **dereference a pointer variable**

```
int x;
int *ptr;

*ptr = &x;
```
incorrect!

- All pointer variables **store memory address**
  - The different "type" of pointer variables refer to the data type **of the destination**

```
int *intPtr;
double *doublePtr;
```

Both store memory address. The destination of `intPtr` is an integer, while **doublePtr** points to a double value

Pointer: **Passing into function**

- By passing the **address of a variable** into function:
  - Allows the function to manipulate the **actual argument!**

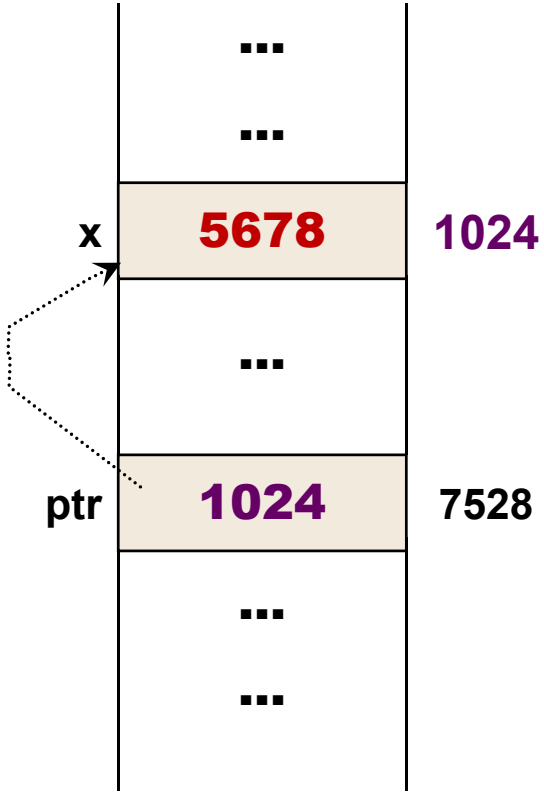
```
int main( )
{
   int x = 0;

   function( &x );

   cout << "X = " << x  << endl;

}
```

Note the "&"

```
void function( int* ptr )
{
    *ptr = 5678;
}
```

Note the "*"

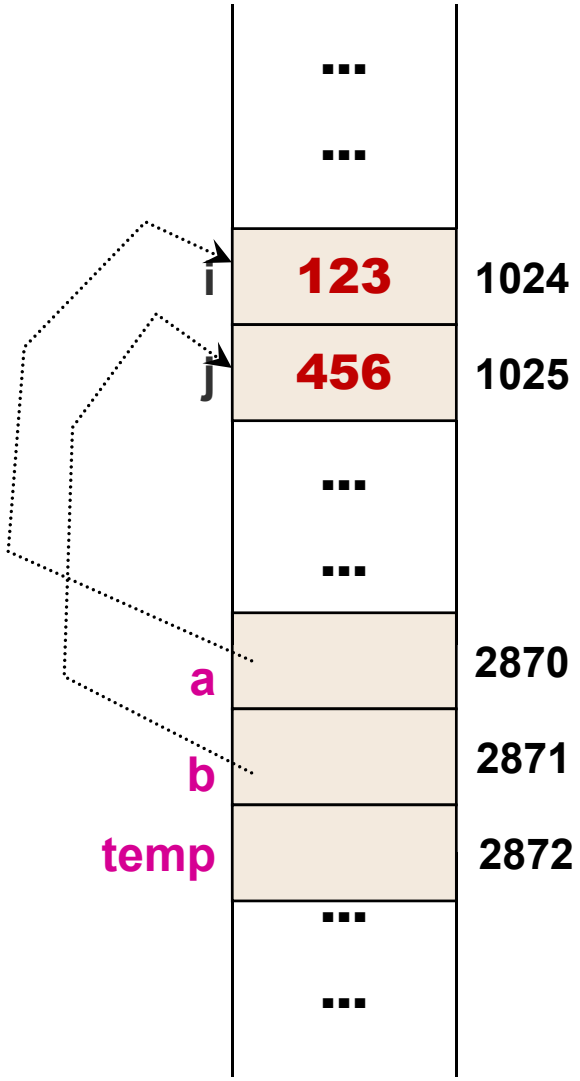
Modify actual parameter through dereferencing



Pointer: **Pass by address**

```
int main()
{
    int i = 123, j = 456;

    swap_ByAdr( &i, &j );

}
```

```
void swap_ByAdr( int* a, int* b )
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```



Pointer: **Helpful "pointers"**

- Real World Analogy:
  - Variable is your home, which has an address
  - Pointer variable stores the address
    - If you follow the address, you find the actual house!
    - Your friends can hold multiple copies of the address, but there is only one actual house

- We show the addresses to illustrate the actual working of a pointer
  - To understand program execution, you most likely only need to know which variable a pointer is referring to

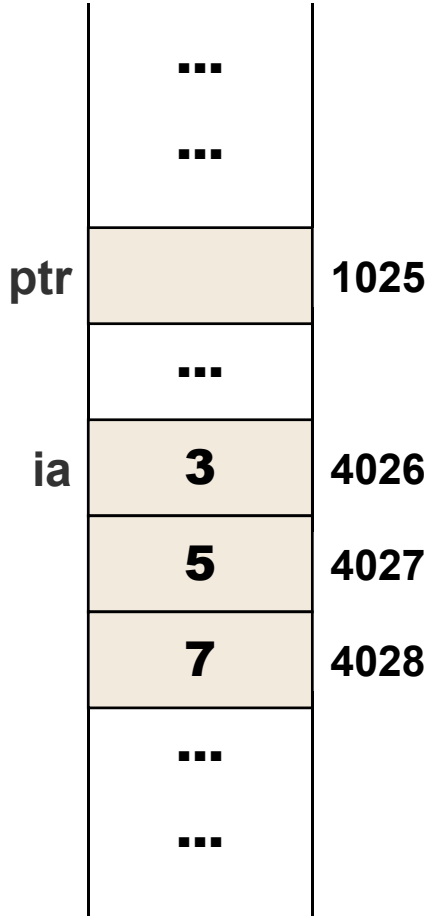
Pointers and Arrays

- Normal pointer can work just like an array if you set it up properly

```
int ia[3] = {3, 5, 7};
int *ptr;

ptr = ia;
ptr[1] = 333;


ptr = &(ia[1]);
ptr[1] = 4444;
```

array name evaluates to address of 0th element

Don't Panic! Just apply what you have learnt

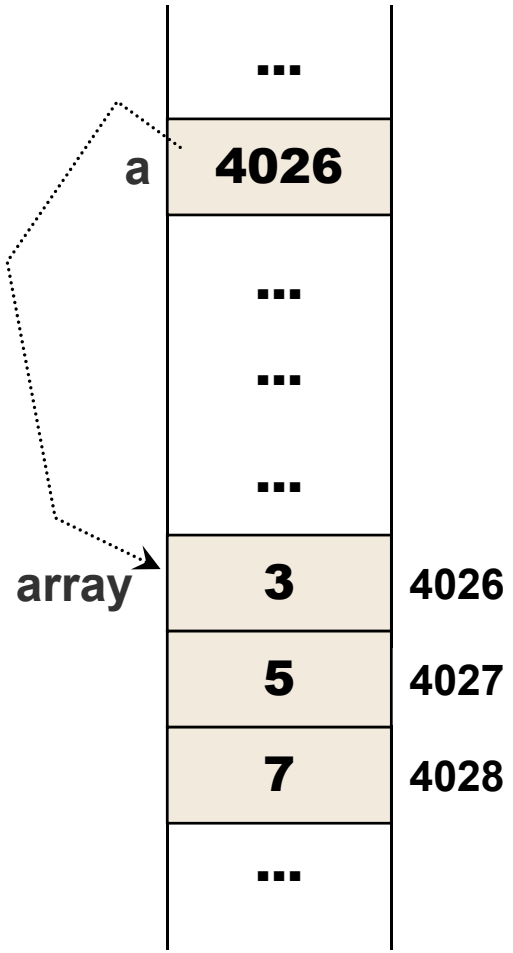


Array: **Passing into function**

- When an array is passed into function
  - **pointer to the 0th element is passed**

```
void swapItem( int a[], int x, int y )
{
    int temp;

    temp = a[x];
    a[x] = a[y];
    a[y] = temp;
}
```

```
int main( )
{
     int array[2] = { 3, 5 };

     swapItem( array, 0, 1 );
     ……
}
```



Array: **Passing into function**

- There is no difference between declaring the ***function parameter*** **as pointer** or **as an array**!

```
void swapItem( int a[], int x, int y )
{
   //code not shown
}
```

```
void swapItem( int *a, int x, int y )
{
   //code not shown
}
```

Equivalent Version. The coding is **exactly the same.**

- Take a look at the online C reference on the prototype for string functions

Problem: **Cumulative Sum**

- Given the following function

```
void sum( int a[], int size )
{
   int i, result = 0;

   for ( i = 0; i < size; i++)
        result += a[i];
   return result;
}
```

- Write a function to return the cumulative sum:
  - **`Result[i] = sum of A[i] to A[size-1]`**

```
void cumulative( int a[], int result[], int size );
```

a	1	2	3	4	5
result	15	14	12	9	5

Example:

`a[]` is given, `result[]` is filled by the `cumulative()` function

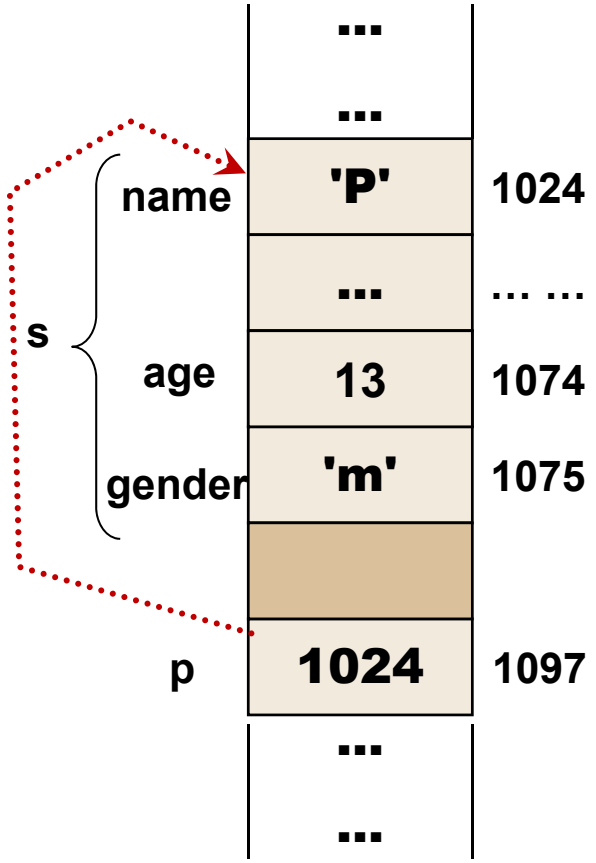
Pointer and Structure (Recap)

- Pointer can points to a structure as well

```
struct Person {
    string name;
    int age;
    char gender;
};
```

```
int main()
{
    Person s =
            { "Potter", 13, 'm' };

    Person *p; //Person Pointer


    p = &s;

    p->age = 14;

}
```

```
(*p).age = 14;
```



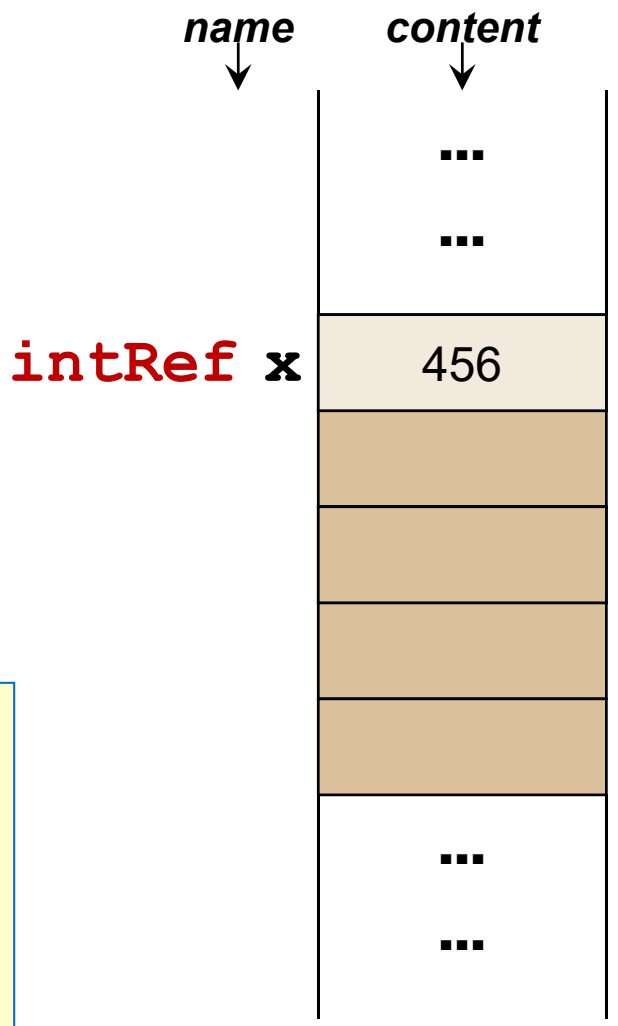
Can you be my referee?

REFERENCE

Reference

- **Reference** == ***alias*** (alternative name) for a variable

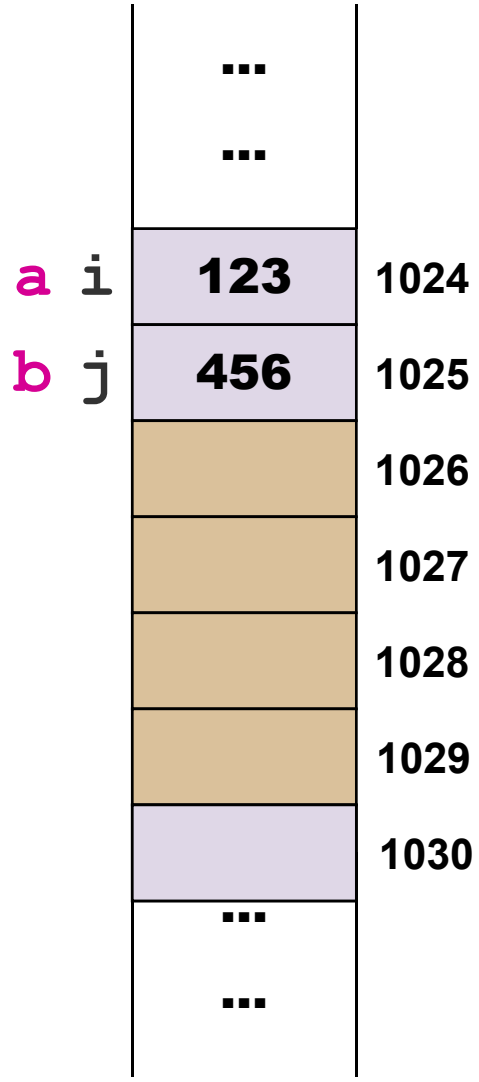
```
int x = 456;

int& intRef = x;

intRef++;
cout << x << endl;     //result?
```




```
int& intRef;

int I;
int& ref = &I;
```

Function : **Pass by reference**

```
void swap_ByRef( int& a, int& b )
{    int temp;

     temp = a;
     a = b;
     b = temp;
}

int main()
{
     int i = 123, j = 456;

     swap_ByRef( i, j );

     cout << i << endl;
     cout << j << endl;
}
```



Function : **Passing Parameters**

- **By Value:**
  - Simple data types (`int`, `float`, `char` etc) and structures are passed by value
  - Function **cannot** change the actual parameter
- **By Address:**
  - Requires the caller to pass in the address of variables using "**&**"
  - Requires dereferencing of parameters in the function
  - Arrays are pass by address
- **By Reference:**
  - No additional syntax except to declare the parameters as references
  - No additional memory storage
    - Faster execution and less memory usage

Combine this with that and you get…..

COMPOSITE DATA STRUCTURES

Composite Data Structures

- Basic data structures:
  a. **Array**
  b. **Structure**
  - Can be ***combined*** to meet new needs!

- We have seen (lecture 1):
  - Structure with arrays
  - Structure with structures
  - Array of structures

- Let's look at **multidimensional arrays**

Multidimensional Array: **Motivation**

- Array we used so far is known as ***one dimensional array***:
  - Only a **single index** is required to locate an item

- Many problems require array of higher dimensions, e.g.:
  - To represent **matrix** in mathematics
  - To store a **collection of strings** as dictionary
  - To represent a two dimensional area
  - To represent a three dimensional space
  - etc

Two dimensional (2D) array : **Syntax**

SYNTAX

Declaration:

```
datatype identifier[ #rows ][ #columns ];
```

Declaration with initialization:

```
datatype identifier[#rows][#cols] = init_list;
```

- Example:

```
int myMatrix[3][4];
```

- Visualization (not actual memory layout):

myMatrix

?????	?????	?????	?????
?????	?????	?????	?????
?????	?????	?????	?????

2D Array: **Initialization list**

- Initialization list for 2D array is a **collection of initialization lists for each row**
  - Rows are specified from top to bottom
  - Columns in each rows are specified from left to right
  - Missing values take zero automatically

```
int myMatrix[3][4] = {
                        { 1, 2, 3, 4 },
                        { 5, 6, 7, 8 },
                        { 9, 10, 11, 12 }
                      };
```

myMatrix

1	2	3	4
5	6	7	8
9	10	11	12

2D Array: **Accessing individual item**

- Need **two indices** to **uniquely** identify an item in the 2D array

myMatrix	[,0]	[,1]	[,2]	[,3]
[0,]	1	2	3	4
[1,]	5	6	7	8
[2,]	9	10	11	12

```
cout << "M[0][0]: " << myMatrix[0][0] << endl;
cout << "M[1][2]: " << myMatrix[1][2] << endl;
cout << "M[2][1]: " << myMatrix[2][1] << endl;

myMatrix[2][3]++;
cout << "M[2][3]: " << myMatrix[2][3] << endl;

myMatrix[2][2] = myMatrix[0][0] + myMatrix[2][0];
cout << "M[2][2]: " << myMatrix[2][2] << endl;
```

2D Array: **As function parameter**

- **Unlike** 1D array, 2D array function parameter **requires the** last dimension to be specified:
  - It is **not optional**, and the size **must match the actual argument**

```
void printMatrix( int M[][4], int rows, int cols );
```

- In general, only the size of the first dimension of a multidimensional array can be omitted:
  - **Size of all other dimensions are compulsory**

2D Array: **As function parameter**

```
int main()
{
    int myMatrix[3][4] = { { 1, 2, 3, 4 },
                           { 5, 6, 7, 8 },
                           { 9, 10, 11, 12 } };

    printMatrix( myMatrix, 3, 4 );

    return 0;
}
```

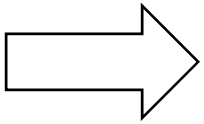
```
void printMatrix( int M[][4], int rows, int cols )
{
    int i, j;

    for ( i = 0; i < rows; i++ ){
        for ( j = 0; j < cols; j++ ){
            cout << M[i][j];
        }
        cout << endl;
    }
}
```

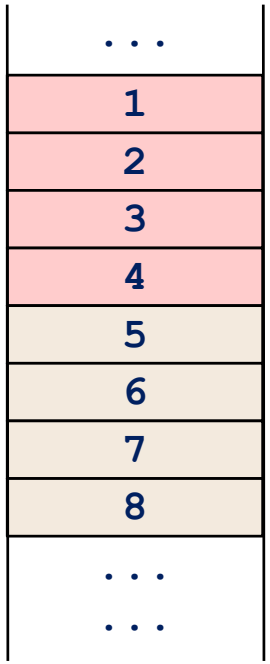
2D Array: **Actual Memory Layout**

- Memory is one dimensional
- C uses the last dimension as the ***basic unit***:
  - Commonly known as the **row-major layout**

```
int A[2][4] = { { 1, 2, 3, 4 },
                { 5, 6, 7, 8 } };
```

A	[,0]	[,1]	[,2]	[,3]
[0,]	1	2	3	4
[1,]	5	6	7	8



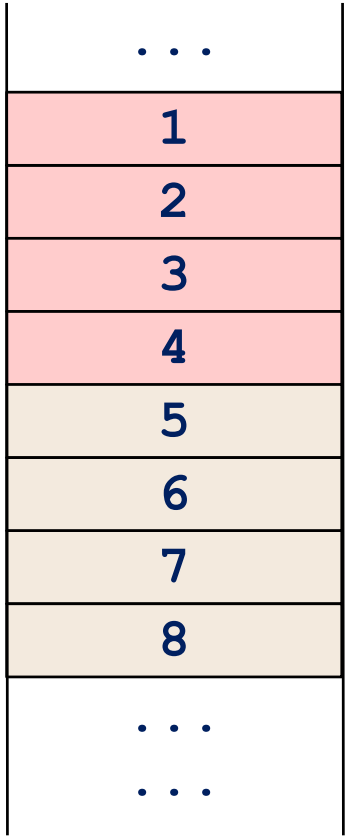


2D Array: Taking a single row

- Each **row can be treated as if it is a normal 1D-array**

```
void printArray( int a[], int size )
{
    int i;

    for (i = 0; i < size; i++) {
         cout << a[i];
    }
    cout << endl;
}
```

```
void printMatrix( int M[][4], int rows, int cols )
{
     int i;

     for ( i = 0; i < rows; i++ ){
          printArray( M[i], cols );
     }
}
```



Let's apply the knowledge

PROBLEMS AHEAD!!

Problem: **Peak of Excellence**

- Given:
  - 2D Area Map, where each point is the **elevation** (height of the ground from sea level)
- Your Task:
  - Find out all the "peaks" in the area
  - Peak is defined as a point higher than the 4 orthogonal neighbors
- Example:

3.75	4.02	4.89	6.01
4.75	8.78	6.55	4.53
5.50	6.02	9.78	5.33
4.91	3.33	8.98	7.64

Peak

Peak

Problem: **Not in my Dictionary**

- Given:
  - A dictionary of `N` words, each word at most `C` characters
- Your Task:
  - Check whether a user given word `W` is in the dictionary or not
- Key Problems:
  - How to represent the dictionary?
  - How to "search" the dictionary?

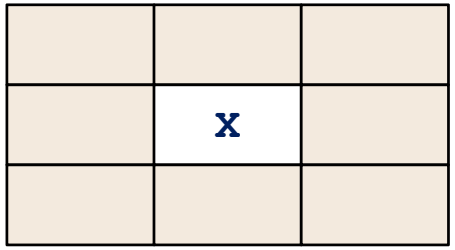
Problem: **Minesweeper!**

- Minesweeper is a popular mini-games:

  - Let's just do a minor but interesting part of this game:
    - Generation of the **randomly placed** mines
    - Calculate the numbers in non-mine cell

Problem: **Minesweeper!**

- **Given:**
  - A minefield of 20 x 20 cells
  - **`nMines`**: Number of mines to be planted
- ***Your task:***
  - Generate the mines
  - Update the adjacent cells



The 8 neighbors for a cell **X**

Mine	2	1
2	3	Mine
Mine	2	1
1	1	0

Challenge: **Matrix Multiplication**

- Given:
  - An **m** × **p** matrix ***A*** (**m** rows, **p** columns)
  - An **p** × **n** matrix ***B*** (**p** rows, **n** columns)
  - ***A*** **x** ***B*** is an **m×n** matrix **C** whose entries are

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} \times B_{k,j}$$

A	[,0]	[,1]	[,2]
[0,]	1	2	3
[1,]	4	5	6

X

B	[,0]	[,1]
[0,]	1	2
[1,]	3	4
[2,]	5	6

=

C	[,0]	[,1]
[0,]		
[1,]	49	

$$C_{1,0} = 4 \times 1 + 5 \times 3 + 6 \times 5 = 49$$

Challenge : **Matrix Multiplication**

- Suggestion:
  - Given `i` and `j`, figure out how to calculate for entry $C_{i,j}$
  - Use the above and loop through possible `i` and `j`

- You can assume:
  - A is 2 x 3
  - B is 3 x 2

- Try it out!

Summary

C++ Elements

Pointer:
- Declaration
- Dereference
- Passing into function
- Pointer and Array
- Pointer and Structure

Reference:
- Passing into function
- Reference and Structure

Composite Data Type:
- Multidimensional Arrays
  - Declaration and Usage
  - Passed into function
  - Memory layout
  - Taking one row out of a matrix