Introduction to Computing

Lecture 13

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Memory Address, References and Pointers

Memory Address

Getting Memory Address

When a variable is created in C++, a memory address is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.

To access it, use the **&** operator, and the result will represent where the variable is stored:

Example:

```
int a = 2;

cout << &a; // Outputs 0x6dfed4
```

Reference Variable

A reference variable is a "reference" to an existing variable, and it is created with the & operator:

```
int a = 2;
int &b= a; // reference to a
```

- The reference variable can only be initialized at the time of its creation
- The reference variable returns the address of the variable preceded by the reference sign ‘&’
- The reference variable can never be reinitialized again in the program
- The reference variable can never refer to NULL

Pointers

Pointer Variable

A pointer is a variable that stores the memory address as its value.

- A pointer variable points to a data type of the same type
- It is created with the * operator.
- The address of the variable you're working with is assigned to the pointer

Example:

```
int *a ;
int b = 2;
a= &b; // a stores the address of b
```

Accessing Memory Address and Value using Pointer Variable

- Pointer variable holds the address of a variable, so its not a problem
- We can also get the value of the variable through pointer, by using the * operator (**the dereference operator**).
- We can also change the value of the variable by using the * operator

Example:

```
int *a ;
int b = 2;
a= &b;          // a stores the address of b
cout << *a;     // using dereference operator we get value of 'b'
*a = 3;         // using dereference operator we set value of 'b'
cout << b;      // we get 3
```

Pointers

Accessing Memory Address and Value using Pointer Variable

- Poi
- W
  op
- W

Exa

```
int
int
a= &
cout << *a;     // using dereference operator we get value of 'b'
*a = 3;         // using dereference operator we set value of 'b'
cout << b;      // we get 3
```

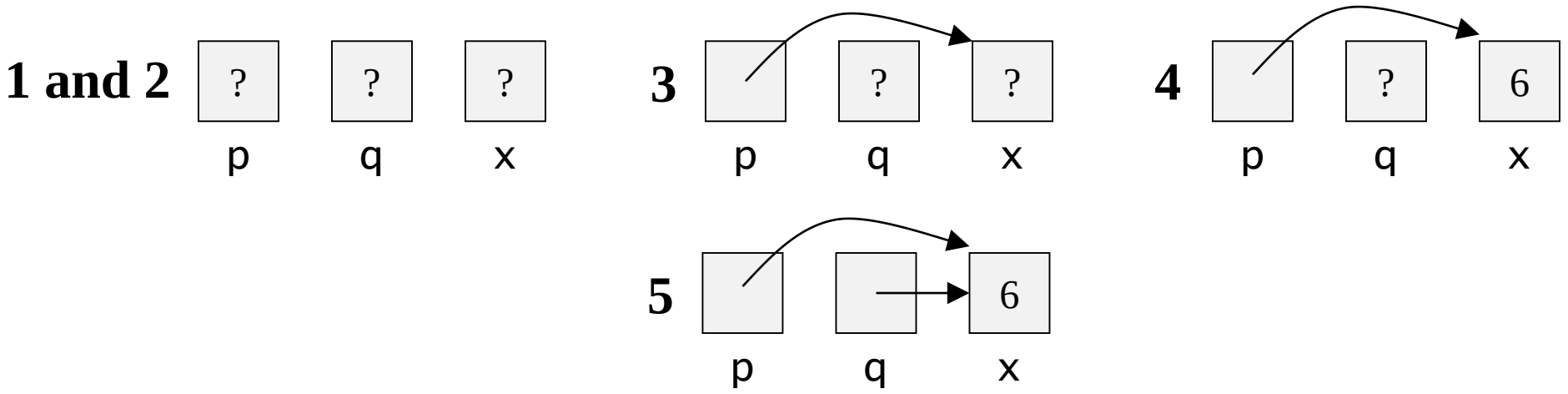
Note:

The * sign can be confusing here, as it does two different things in our code:

- When used in **declaration** (string* ptr), it creates a pointer variable.
- When **not used in declaration**, it act as a dereference operator.

Pointers (Recap)

1.	Pointer variables	int *p, *q;
2.	Static allocation	int x;
3.	Address-of operator	p = &x;
4.	Memory cell to which P points	*p = 6;
5.	Pointer operations	q = p;



“Pass by Value” and “Pass by Reference”

Pass by Value:

- Makes a copy in memory of the actual parameters

- Use pass by value when you are only **using** the parameter for some computation, not changing it

Pass by Reference:

- Forwards the actual parameters

- Use pass by reference when you are **changing** the parameter passed in the program

“Pass by Value”

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

“Pass by Reference”

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}


int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

“Pass by Value”

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Declaration

“Pass by Reference”

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}


int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Declaration

“Pass by Value”

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}

int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
return 0;
}
```

Function Definition

“Pass by Reference”

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}

int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Definition

“Pass by Value”

Pass by Pointer ~~“Pass by Reference”~~

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

int add(int a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Calling

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

int add(int* a)
{
  int b = 0;
  *a = *a + 1;
  b=*a;

  return b;
}


int main() {
  int x = 0;
  int result = add(&x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Function Calling

Another way for “Pass by Reference”

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

int add(int &a)
{
  int b = 0;
  a = a + 1;
  b=a;

  return b;
}


int main() {
  int x = 0;
  int result = add(x);
  cout << result << endl;
  cout << x << endl;
  return 0;
}
```

Reference Variable:
Reference variable is an alias for a variable which is assigned to it.

Different from pointer:

- The reference variable can only be initialized at the time of its creation
- The reference variable returns the address of the variable preceded by the reference sign ‘&’
- The reference variable can never be reinitialized again in the program
- The reference variable can never refer to NULL

Passing array as argument

Passing array as argument

Two ways:

```
void function(int a[5]);
int main()
{
    int x[5] = { 88, 76, 90, 61, 69 };

    function(x);

    return 0;
}
```

```
void function(int a[], int size);
int main()
{
    int x[5] = { 88, 76, 90, 61, 69 };

    function(x,5);

    return 0;
}
```

Sorting

Why Study Sorting?

- When an input is sorted, many problems become easy (e.g. searching, min, max)

Sorting algorithms

- **Bubble Sort**
- Selection Sort
- Insertion Sort
- Merge Sort

Bubble Sort: Idea

Given an array of n items

1. Compare pair of adjacent items
2. Swap if the items are out of order
3. Repeat until the end of array
    - The largest item will be at the last position
4. Reduce **n** by 1 and go to Step 1

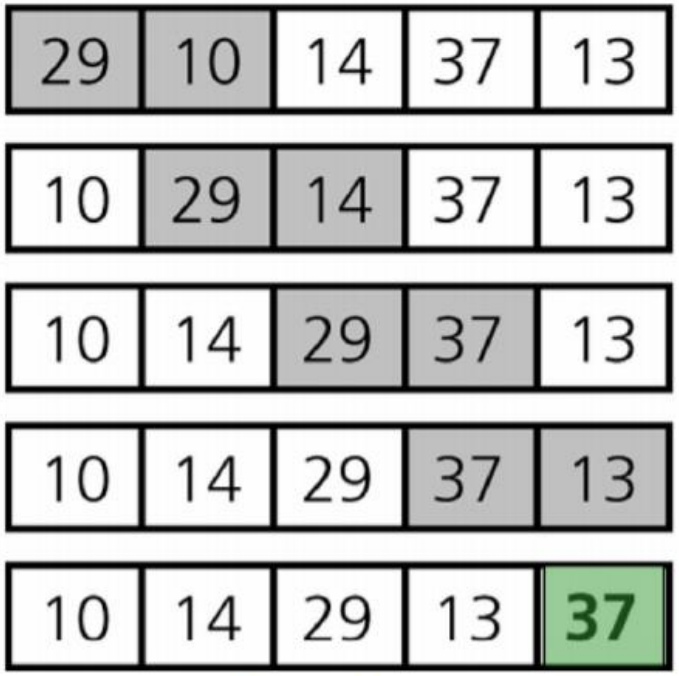
Analogy

- Large item is like "**bubble**" that floats to the end of the array

Bubble Sort: Idea

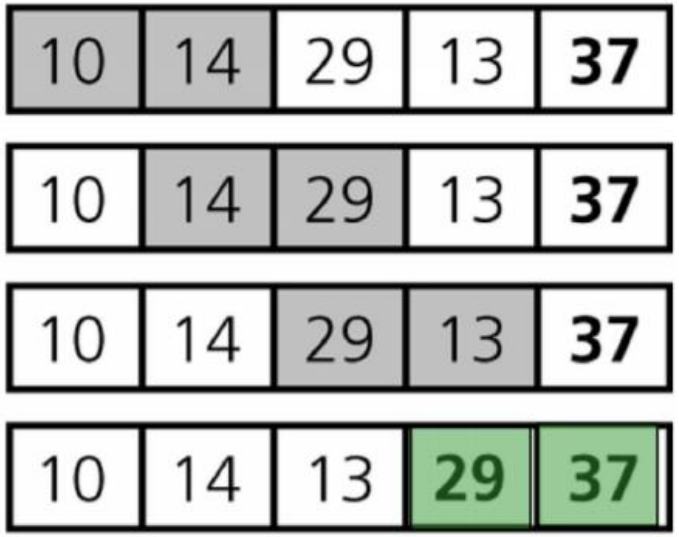
Illustration:

(a) Pass 1



At the end of **Pass 1**, the largest item **37** is at the last position.

(b) Pass 2



At the end of **Pass 2**, the second largest item **29** is at the second last position.

Bubble Sort: Code

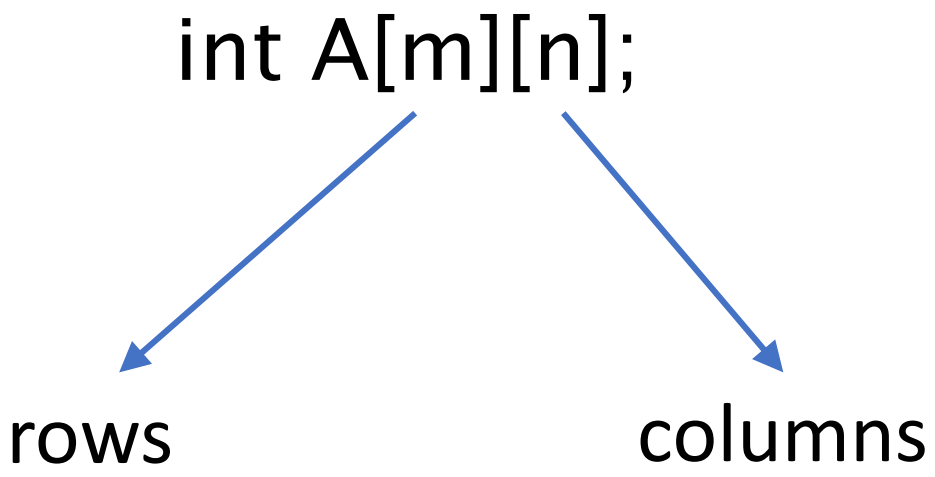
```
void bubbleSort(int a[], int n)
{
    for (int i = n - 1; i >= 1; i--)
    {
        for (int j = 1; j <= i; j++)
        {
            if (a[j - 1] > a[j])
            {
                int temp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = temp;
            }
        }
    }
}
```

```
int main()
{

    int a[5] = { 7,3,6,4,1 };
    bubbleSort(a,5);

    for (int i = 0; i < 5; i++)
    {
        cout << a[i];
    }

    return 0;
}
```

2D arrays and Beyond (again)

Multi-dimensional Arrays

- An array is a contiguous block of memory.
- A 2D array of size m by n is defined as:



What is the number of bytes necessary to hold **int A[2][2]** ?

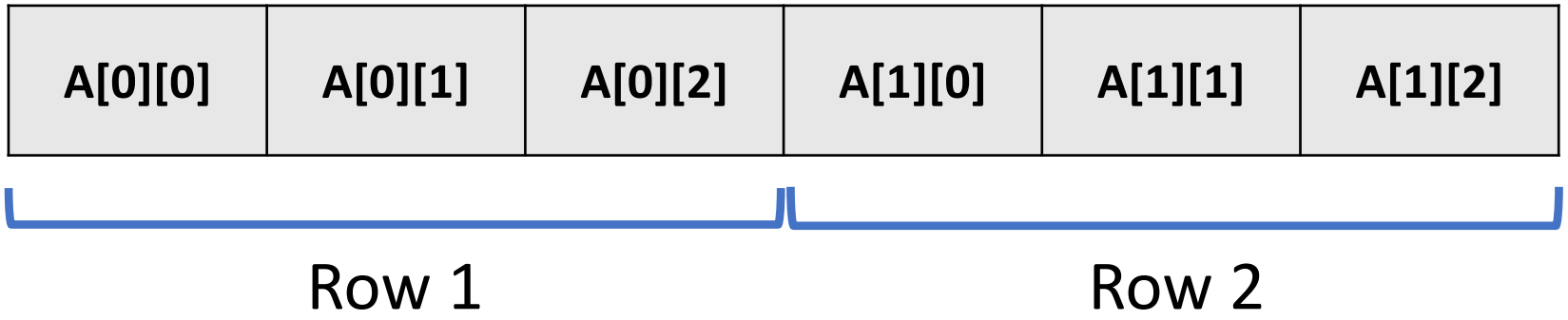
Multi-dimensional Arrays

- It can be initialized as:

$$A[2][3]=\{\{1,2,3\},\{4,5,6\}\};$$

	[0]	[1]	[2]
[0]	**1**	**2**	**3**
[1]	4	5	6

- How a 2D array is stored?



Multi-dimensional Arrays

- Accessing 1D arrays using Pointers:


```
cout<<A[1];
cout<<*(A+1);
```

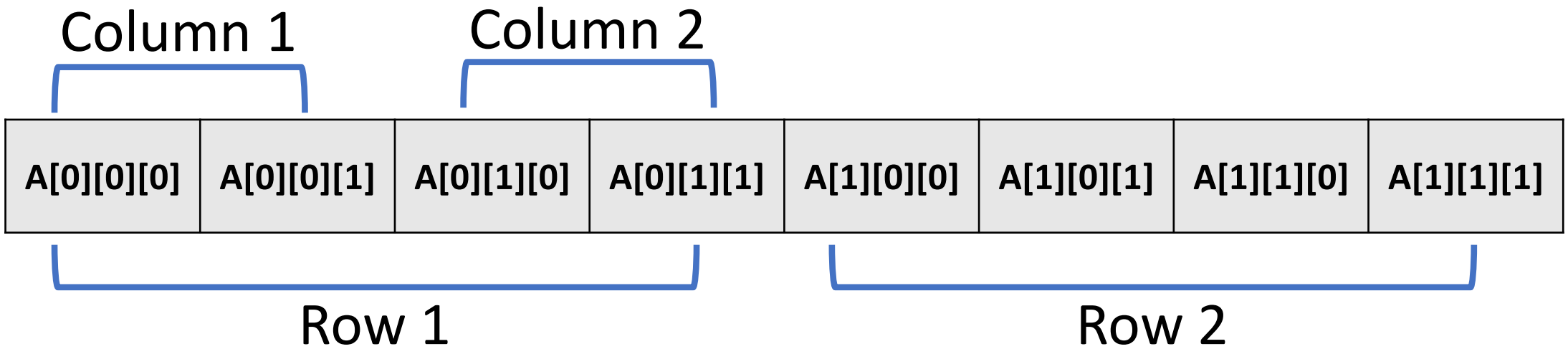
- How does 2D arrays and pointers relate:

Multi-dimensional Arrays

- Now… If I want access A[1][2] via pointer, what will I write?

First Solution	*(A[1] + 2)
Second Solution	*(*(A+1) + 2)

- You can view **3D** or **nD** array the same way, i.e., int A[2][2][2]



Typecasting (Type conversion)

- The process of converting one predefined type into another is called as type conversion

- C++ facilitates the type conversion into the following two forms :

  - ❑ Implicit Type Conversion
  - ❑ Explicit Type Conversion

Implicit Type Conversion

- Conversion performed by the compiler without programmer's intervention whenever differing data types are intermixed in an expression

- The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable)

- Example:

```
int main()
{
    short int x = 1417;
    char ch;
    ch = x;       // where ch is char (1 byte) and x is int (2 bytes)
    return 0;
}
```

Implicit Type Conversion

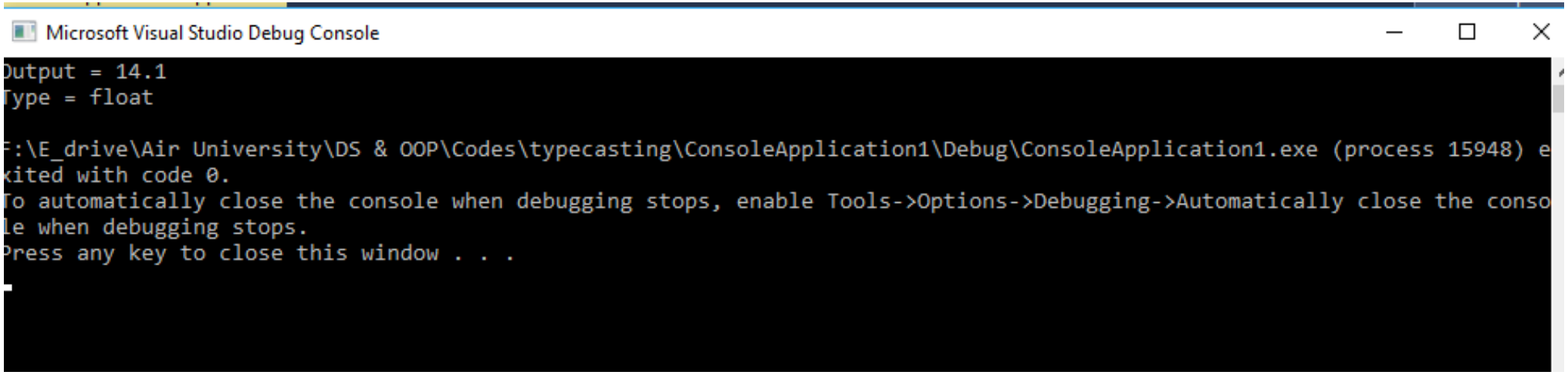
- **x** was having value 1417 (whose binary equivalent is 0000010110001001)

- **ch** will have lower 8-bits i.e., 10001001 resulting in loss of information.

Implicit Type Conversion

- Another example

```
int main()
{
    int x = 10;

    float y = 4.1;

    cout << "Output = " << x + y << endl;

    cout << "Type = " << typeid(x + y).name() << endl;
}
```



Explicit Type Conversion

- User-defined conversion that forces an expression to be of specific type

```
int main()
{
    int y = 3;
    cout << (float)(y) / 2;

}
```

Output= 1.5

```
int main()
{
    int y = 3;
    cout << (y) / 2;

}
```

Output= 1

Memory Allocation

- Dynamic memory allocation is necessary because, during compile time, we may not know the exact memory needs to run the program.

new **malloc()**

- C++ also does not have automatic garbage collection. Therefore a programmer must manage all dynamic memory used during the program execution

delete[] **free()**

Memory Allocation

new / delete[]

```
int main()
{
    int *x;
    x = new int[11];

    for (int i = 0; i <= 10; i++)
        x[i] = 0.1*i;

    delete[] x;
}
```

Return same pointer type

Is a operator

Allocate memory and
calls constructor
for initialization

malloc() / free() / realloc()

```
int main()
{
    int *x;
    x = (int*) malloc(11 * sizeof(int));

    for (int i = 0; i <= 10; i++)
        x[i] = 0.1*i;

    free(x);
}
```

return void *

stdlib function

Allocate memory and
Does not calls constructor

Memory Allocation (2D array)

new / delete[]

```
int main()
{
    int rowCount = 10;
    int colCount = 10;

    int** a = new int*[rowCount];
    for (int i = 0; i < rowCount; ++i)
        a[i] = new int[colCount];

    for (int i = 0; i < rowCount; ++i)
        delete[] a[i];
    delete[] a;
}
```

malloc() / free()

YOUR TURN

Thanks a lot

If you are taking a Nap, **wake up**.....Lecture Over