Pointers

Part 01

2022/2023

Hemza.loucif@univ-msila,dz

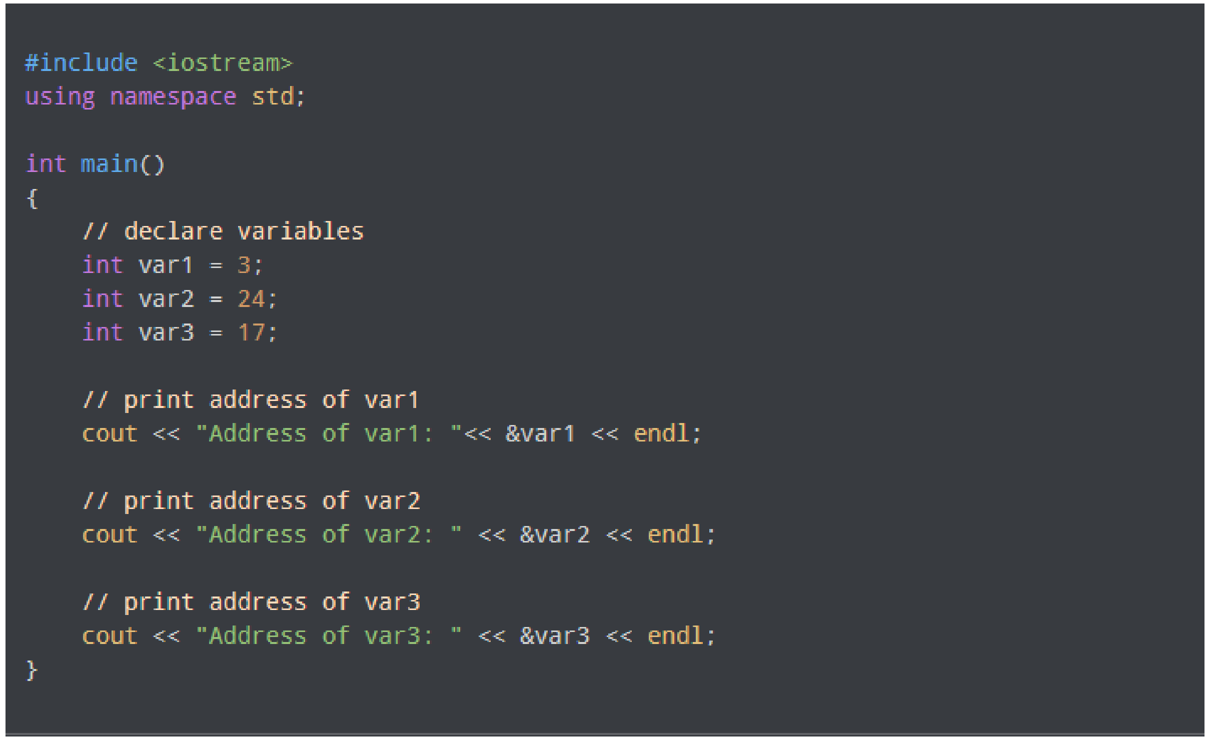
Pointers

- A pointer is a variable that stores the memory address of an object (another variables).

Address in C++

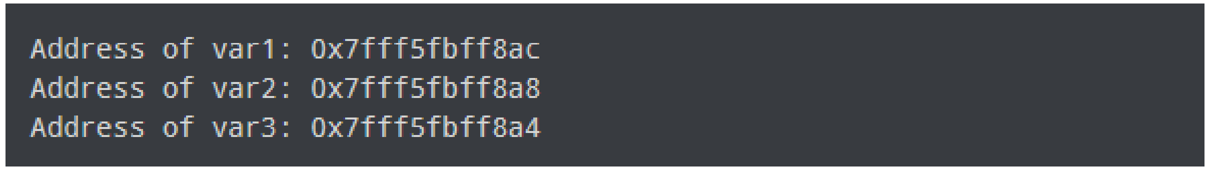
- If we have a variable var in our program, &var will give us its address in the memory. For example,

Example 1: Printing Variable Addresses in C++



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

int main()
{
    // declare variables
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;

    // print address of var1
    cout << "Address of var1: "<< &var1 << endl;

    // print address of var2
    cout << "Address of var2: " << &var2 << endl;

    // print address of var3
    cout << "Address of var3: " << &var3 << endl;
}
```

Output



```
Address of var1: 0x7fff5fbff8ac
Address of var2: 0x7fff5fbff8a8
Address of var3: 0x7fff5fbff8a4
```

Pointers

C++ Pointers

- As mentioned above, pointers are used to store addresses rather than values.
- Here is how we can declare pointers:

```
int *pointVar;
```

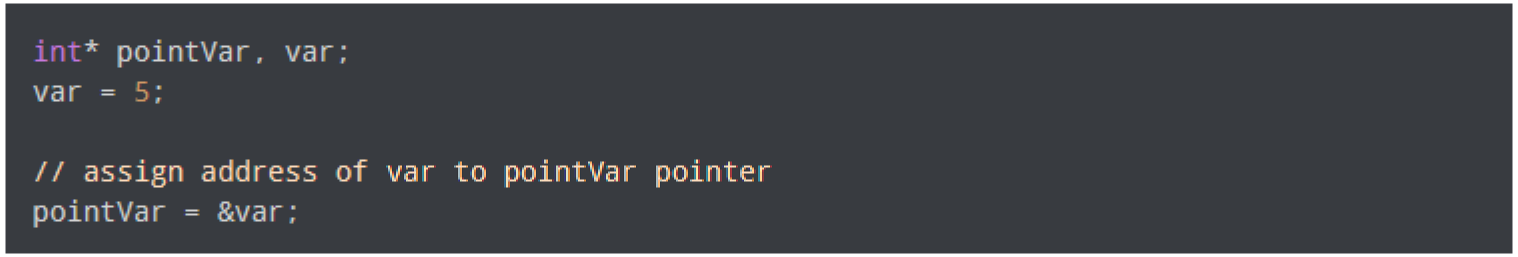
- Here, we have declared a pointer pointVar of the int type.
- We can also declare pointers in the following way.

```
int* pointVar; // preferred syntax
```

Pointers

Assigning Addresses to Pointers

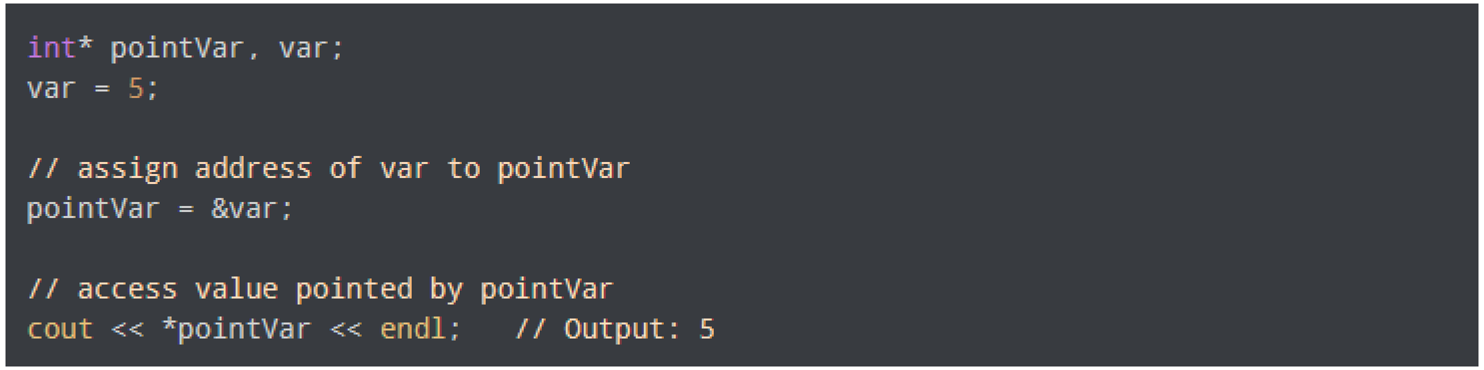
- Here is how we can assign addresses to pointers:



```
int* pointVar, var;
var = 5;

// assign address of var to pointVar pointer
pointVar = &var;
```

Get the Value from the Address Using Pointers

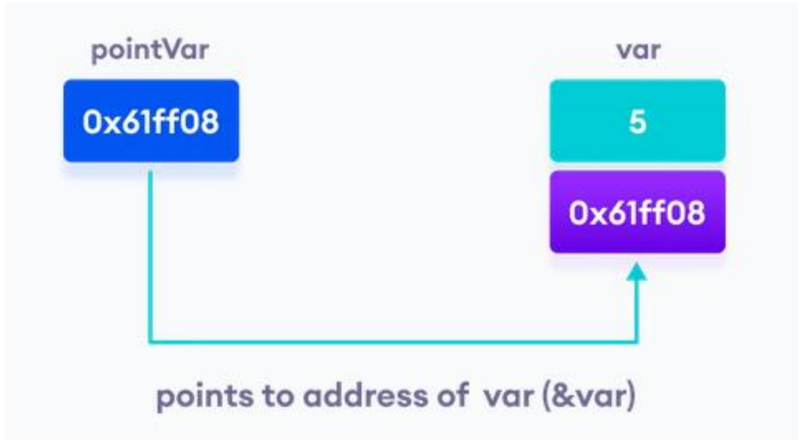
- To get the value pointed by a pointer, we use the * operator. For example:



```
int* pointVar, var;
var = 5;

// assign address of var to pointVar
pointVar = &var;

// access value pointed by pointVar
cout << *pointVar << endl;   // Output: 5
```

Pointers

- **Example 2:** Working of C++ Pointers



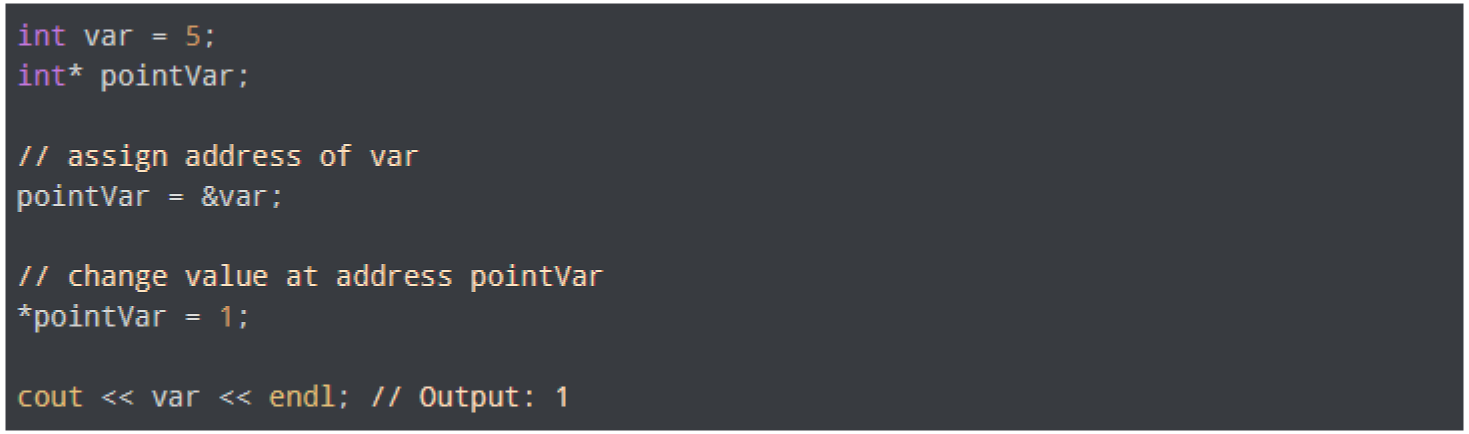
```cpp
#include <iostream>
using namespace std;
int main() {
    int var = 5;

    // declare pointer variable
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print value of var
    cout << "var = " << var << endl;

    // print address of var
    cout << "Address of var (&var) = " << &var << endl
         << endl;

    // print pointer pointVar
    cout << "pointVar = " << pointVar << endl;

    // print the content of the address pointVar points to
    cout << "Content of the address pointed to by pointVar (*pointVar) = " << *pointVar << endl;

    return 0;
}
```

```
var = 5
Address of var (&var) = 0x61ff08

pointVar = 0x61ff08
Content of the address pointed to by pointVar (*pointVar) = 5
```

Pointers

Changing Value Pointed by Pointers

- If pointVar points to the address of var, we can change the value of var by using *pointVar.



```
int var = 5;
int* pointVar;

// assign address of var
pointVar = &var;

// change value at address pointVar
*pointVar = 1;

cout << var << endl; // Output: 1
```

- Here, pointVar and &var have the same address, the value of var will also be changed when *pointVar is changed.

Pointers

- **Example 3:** Changing Value Pointed by Pointers

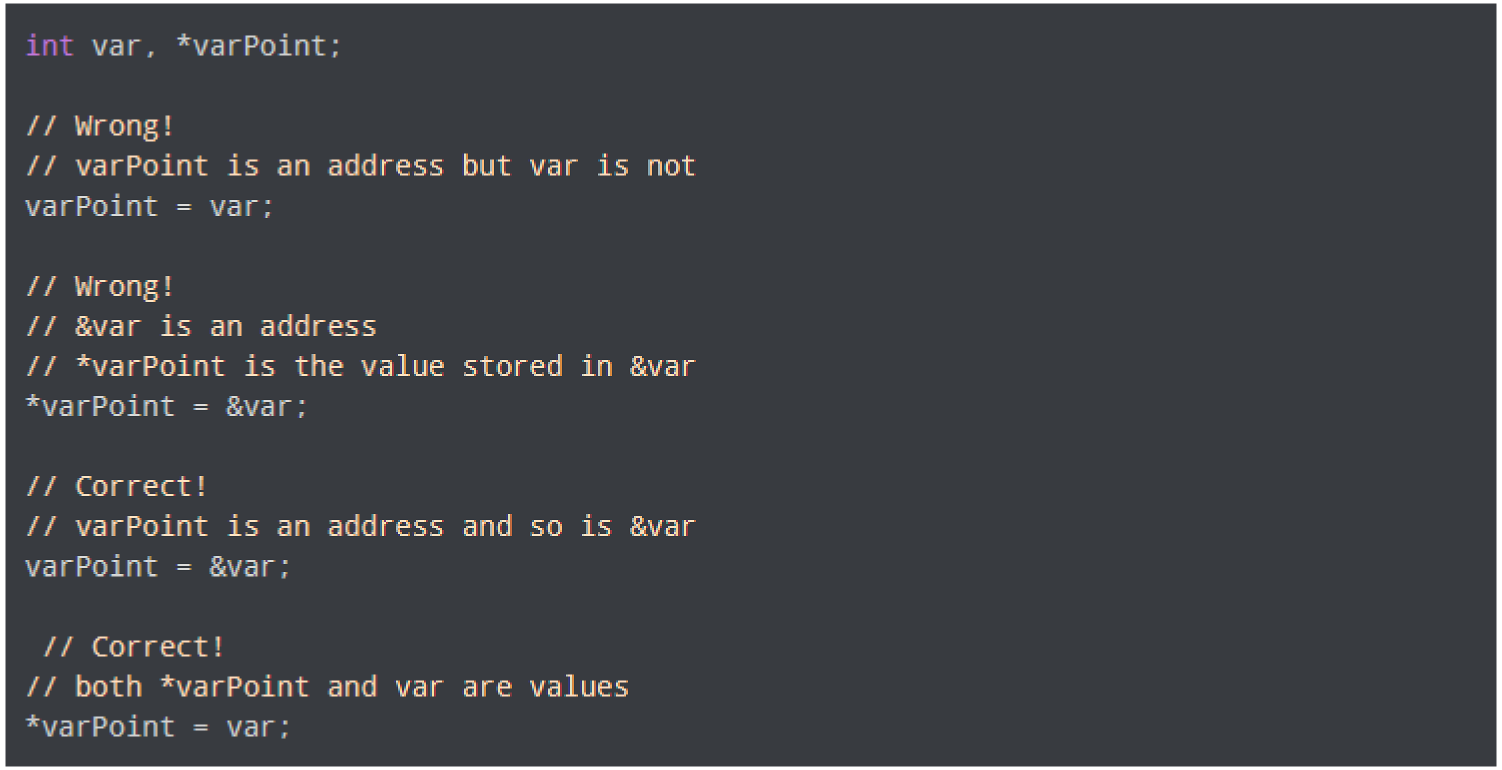
```
#include <iostream>
using namespace std;
int main() {
    int var = 5;
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of var to 7:" << endl;

    // change value of var to 7
    var = 7;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of *pointVar to 16:" << endl;

    // change value of var to 16
    *pointVar = 16;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl;
    return 0;
}
```

```
var = 5
*pointVar = 5

Changing value of var to 7:
var = 7
*pointVar = 7

Changing value of *pointVar to 16:
var = 16
*pointVar = 16
```

Pointers

Common mistakes when working with pointers

Suppose, we want a pointer varPoint to point to the address of var. Then,



```
int var, *varPoint;

// Wrong!
// varPoint is an address but var is not
varPoint = var;

// Wrong!
// &var is an address
// *varPoint is the value stored in &var
*varPoint = &var;

// Correct!
// varPoint is an address and so is &var
varPoint = &var;

 // Correct!
// both *varPoint and var are values
*varPoint = var;
```

C++ Pointers and Arrays

- In C++, Pointers are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an array.

```cpp
int *ptr;
int arr[5];

// store the address of the first
// element of arr in ptr
ptr = arr;
```

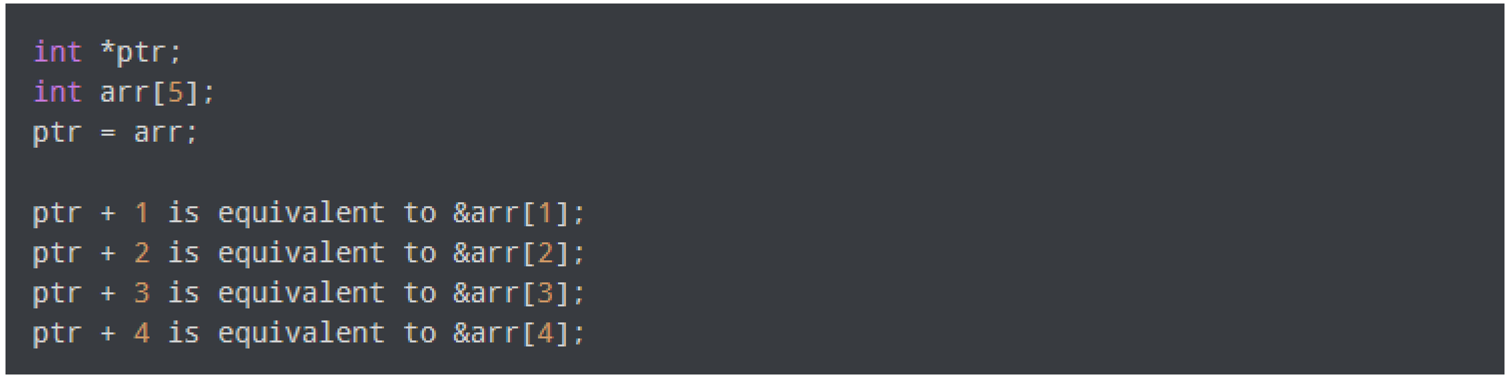
- Here, ptr is a pointer variable while arr is an int array. The code ptr = arr; stores the address of the first element of the array in variable ptr.
- Notice that we have used arr instead of &arr[0]. This is because both are the same. So, the code below is the same as the code above.

```cpp
int *ptr;
int arr[5];
ptr = &arr[0];
```

C++ Pointers and Arrays

Point to Every Array Elements

- Suppose we need to point to the fourth element of the array using the same pointer ptr.
- Here, if ptr points to the first element in the above example then ptr + 3 will point to the fourth element. For example,



```
int *ptr;
int arr[5];
ptr = arr;

ptr + 1 is equivalent to &arr[1];
ptr + 2 is equivalent to &arr[2];
ptr + 3 is equivalent to &arr[3];
ptr + 4 is equivalent to &arr[4];
```

C++ Pointers and Arrays

Point to Every Array Elements

- Similarly, we can access the elements using the single pointer. For example,

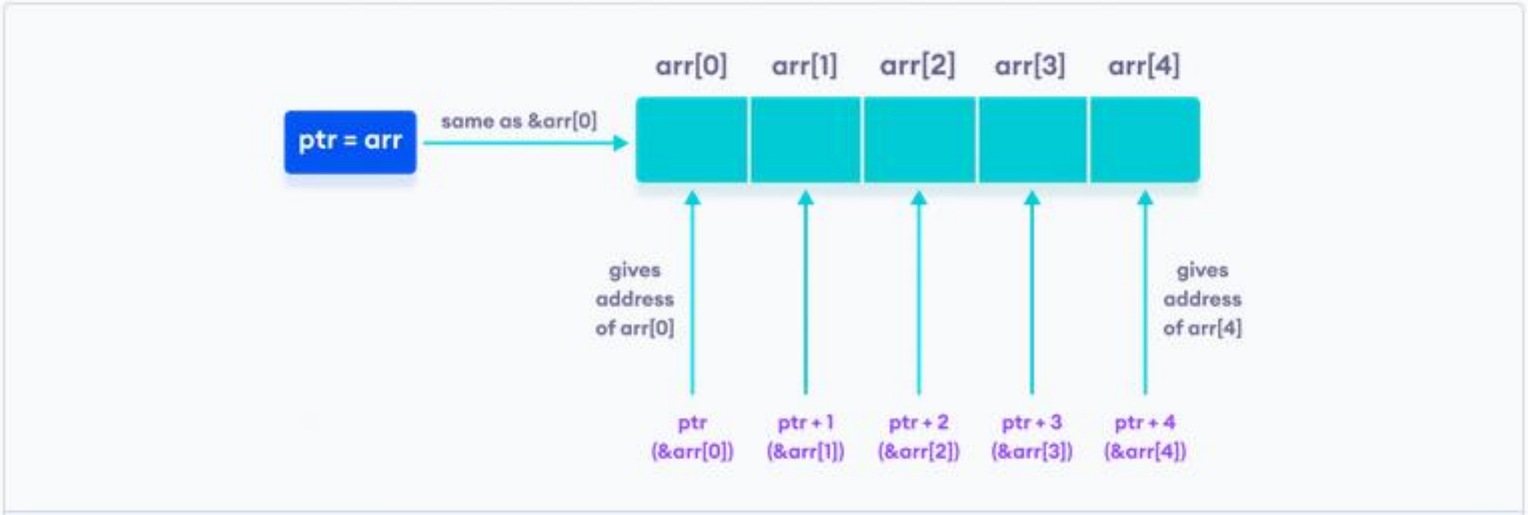
```
// use dereference operator
*ptr == arr[0];
*(ptr + 1) is equivalent to arr[1];
*(ptr + 2) is equivalent to arr[2];
*(ptr + 3) is equivalent to arr[3];
*(ptr + 4) is equivalent to arr[4];
```

- Suppose if we have initialized ptr = &arr[2]; then

```
ptr - 2 is equivalent to &arr[0];
ptr - 1 is equivalent to &arr[1];
ptr + 1 is equivalent to &arr[3];
ptr + 2 is equivalent to &arr[4];
```

C++ Pointers and Arrays

Point to Every Array Elements



Working of C++ Pointers with Arrays

C++ Pointers and Arrays

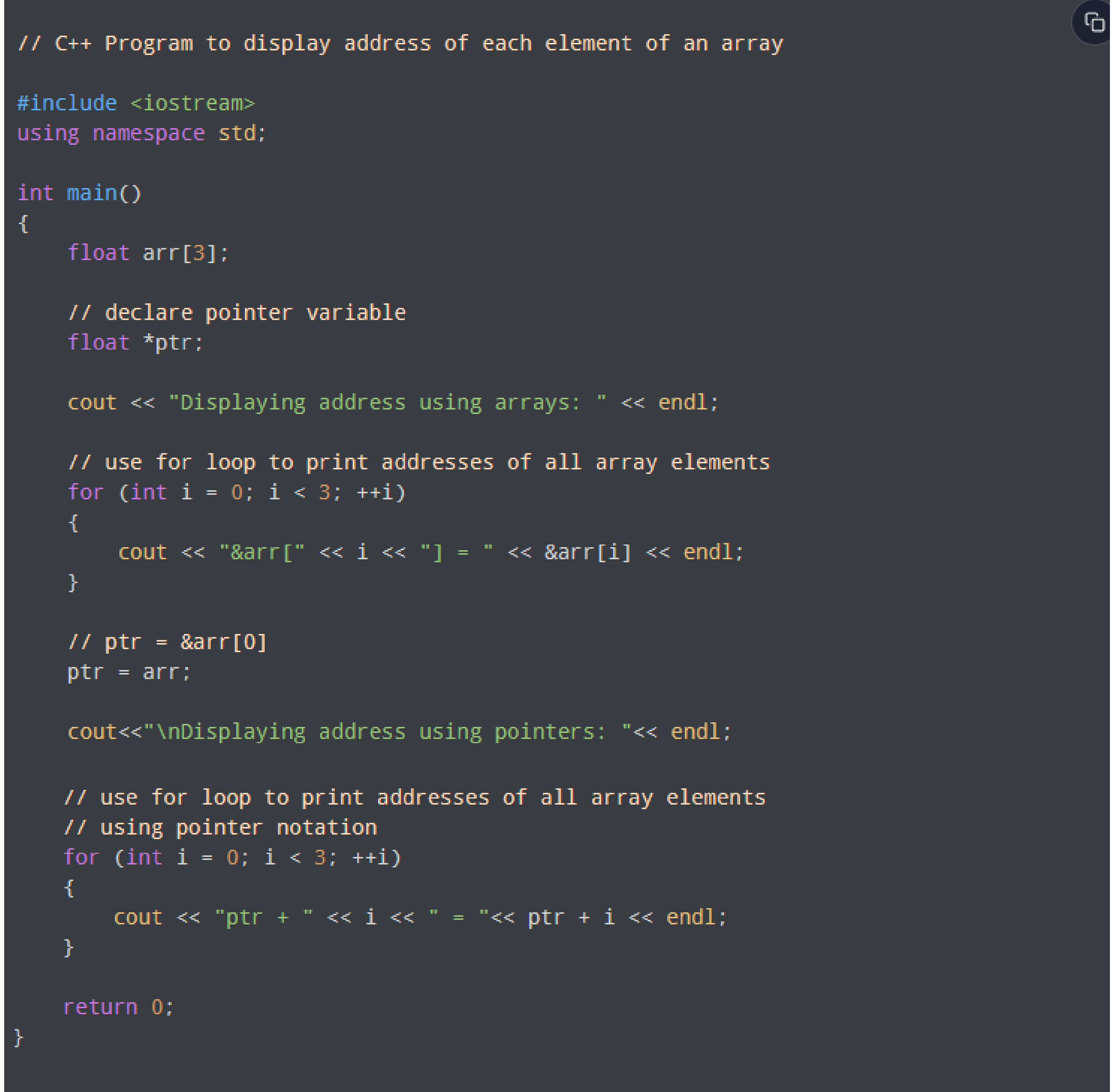
Point to Every Array Elements

Note: The address between ptr and ptr + 1 differs by 4 bytes. It is because ptr is a pointer to an int data. And, the size of int is 4 bytes in a 64-bit operating system.

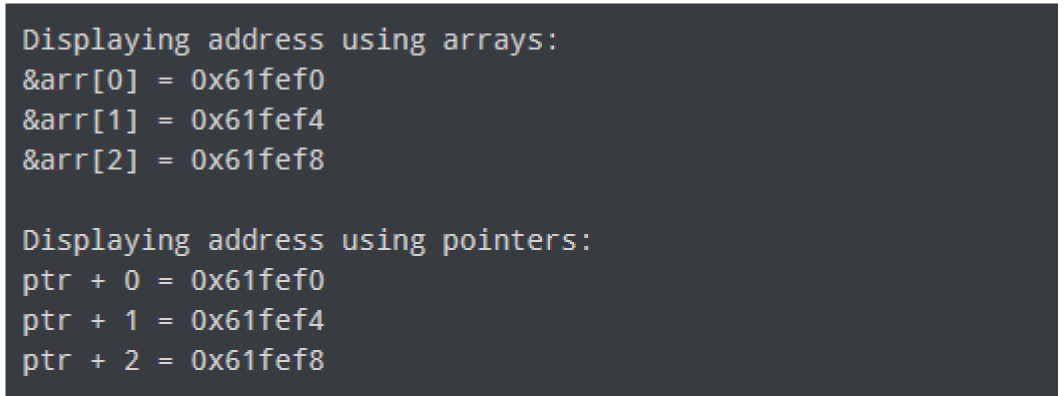
Similarly, if pointer ptr is pointing to char type data, then the address between ptr and ptr + 1 is 1 byte. It is because the size of a character is 1 byte.

Pointers

- **Example I:** C++ Pointers and Arrays



```cpp
// C++ Program to display address of each element of an array

#include <iostream>
using namespace std;

int main()
{
    float arr[3];

    // declare pointer variable
    float *ptr;

    cout << "Displaying address using arrays: " << endl;

    // use for loop to print addresses of all array elements
    for (int i = 0; i < 3; ++i)
    {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }

    // ptr = &arr[0]
    ptr = arr;

    cout<<"\nDisplaying address using pointers: "<< endl;

    // use for loop to print addresses of all array elements
    // using pointer notation
    for (int i = 0; i < 3; ++i)
    {
        cout << "ptr + " << i << " = "<< ptr + i << endl;
    }

    return 0;
}
```

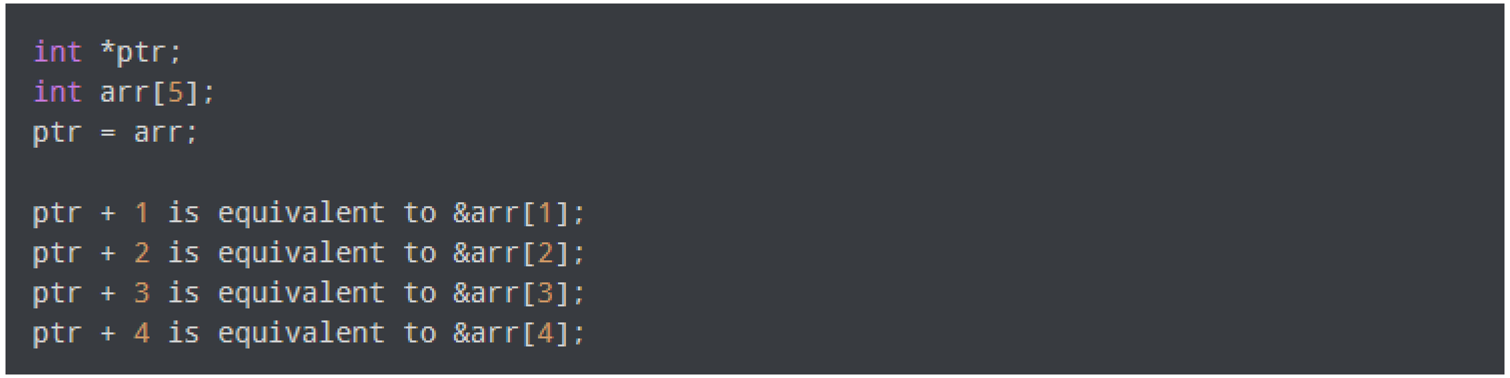


```
Displaying address using arrays:
&arr[0] = 0x61fef0
&arr[1] = 0x61fef4
&arr[2] = 0x61fef8

Displaying address using pointers:
ptr + 0 = 0x61fef0
ptr + 1 = 0x61fef4
ptr + 2 = 0x61fef8
```

C++ Pointers and Arrays

Point to Every Array Elements

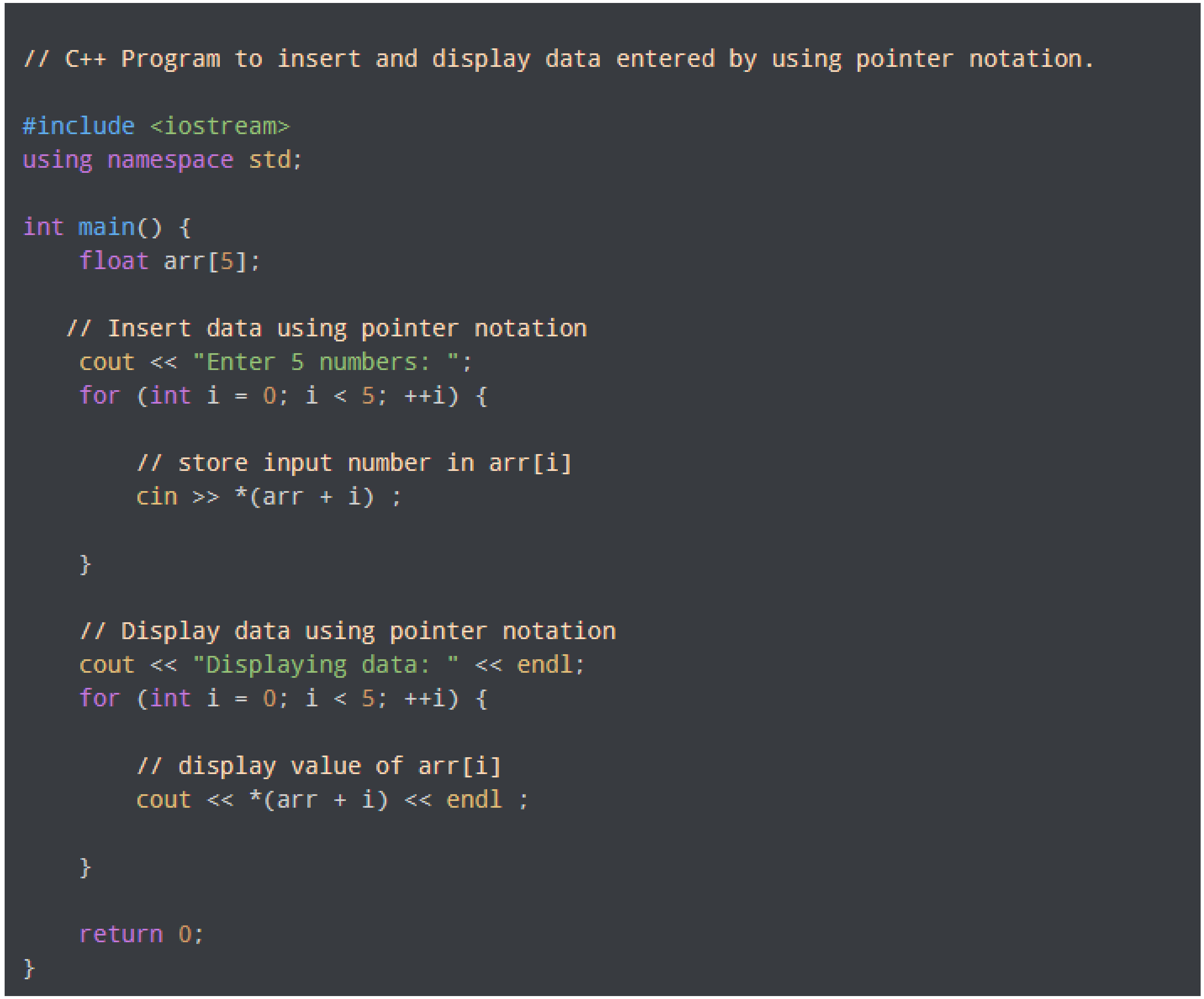
- In most contexts, array names are converted to pointers.
- That's the reason why we can use pointers to access elements of arrays.



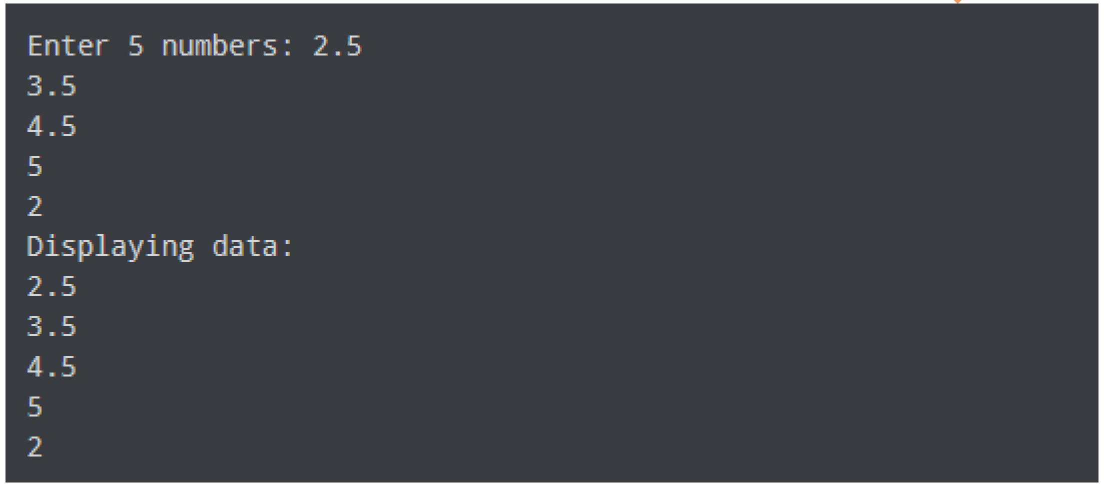
```
int *ptr;
int arr[5];
ptr = arr;

ptr + 1 is equivalent to &arr[1];
ptr + 2 is equivalent to &arr[2];
ptr + 3 is equivalent to &arr[3];
ptr + 4 is equivalent to &arr[4];
```

Pointers

- **Example 2:** Array name used as pointe



```cpp
// C++ Program to insert and display data entered by using pointer notation.

#include <iostream>
using namespace std;

int main() {
    float arr[5];

   // Insert data using pointer notation
    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; ++i) {

        // store input number in arr[i]
        cin >> *(arr + i) ;

    }

    // Display data using pointer notation
    cout << "Displaying data: " << endl;
    for (int i = 0; i < 5; ++i) {

        // display value of arr[i]
        cout << *(arr + i) << endl ;

    }

    return 0;
}
```



```
Enter 5 numbers: 2.5
3.5
4.5
5
2
Displaying data:
2.5
3.5
4.5
5
2
```

C++ Pointers and Arrays

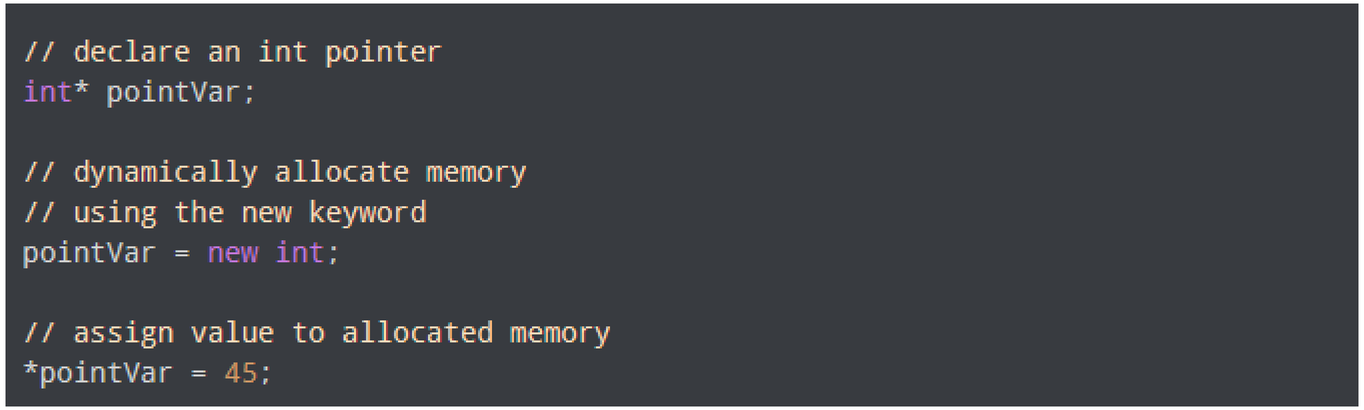
C++ Memory Management: new and delete

- C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.
- In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.
- We can allocate and then deallocate memory dynamically using the new and delete operators respectively.

C++ Pointers and Arrays

C++ new Operator

- The new operator allocates memory to a variable. For example,



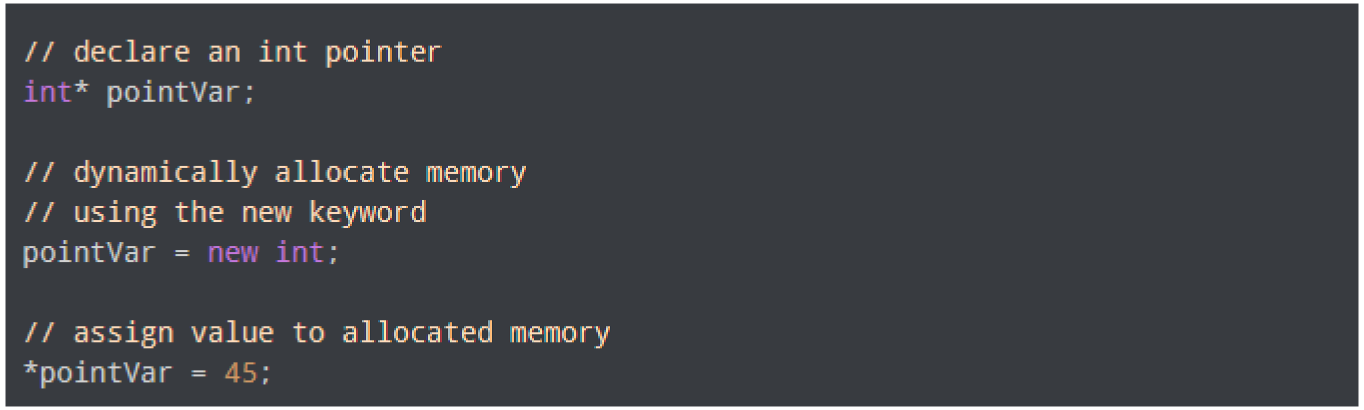
```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// using the new keyword
pointVar = new int;

// assign value to allocated memory
*pointVar = 45;
```

- Here, we have dynamically allocated memory for an int variable using the new operator.
- Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.

C++ Pointers and Arrays

C++ new Operator

- The new operator allocates memory to a variable. For example,



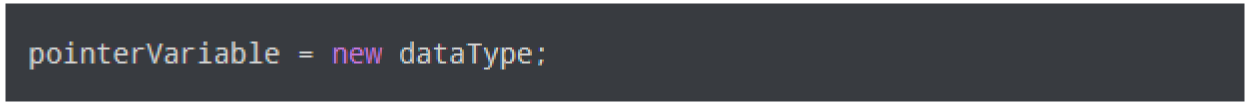
```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// using the new keyword
pointVar = new int;

// assign value to allocated memory
*pointVar = 45;
```

- Here, we have dynamically allocated memory for an int variable using the new operator.
- Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.

C++ Pointers and Arrays

C++ new Operator

- In the case of an array, the new operator returns the address of the first element of the array.
- From the example above, we can see that the syntax for using the new operator is



```
pointerVariable = new dataType;
```

C++ Pointers and Arrays

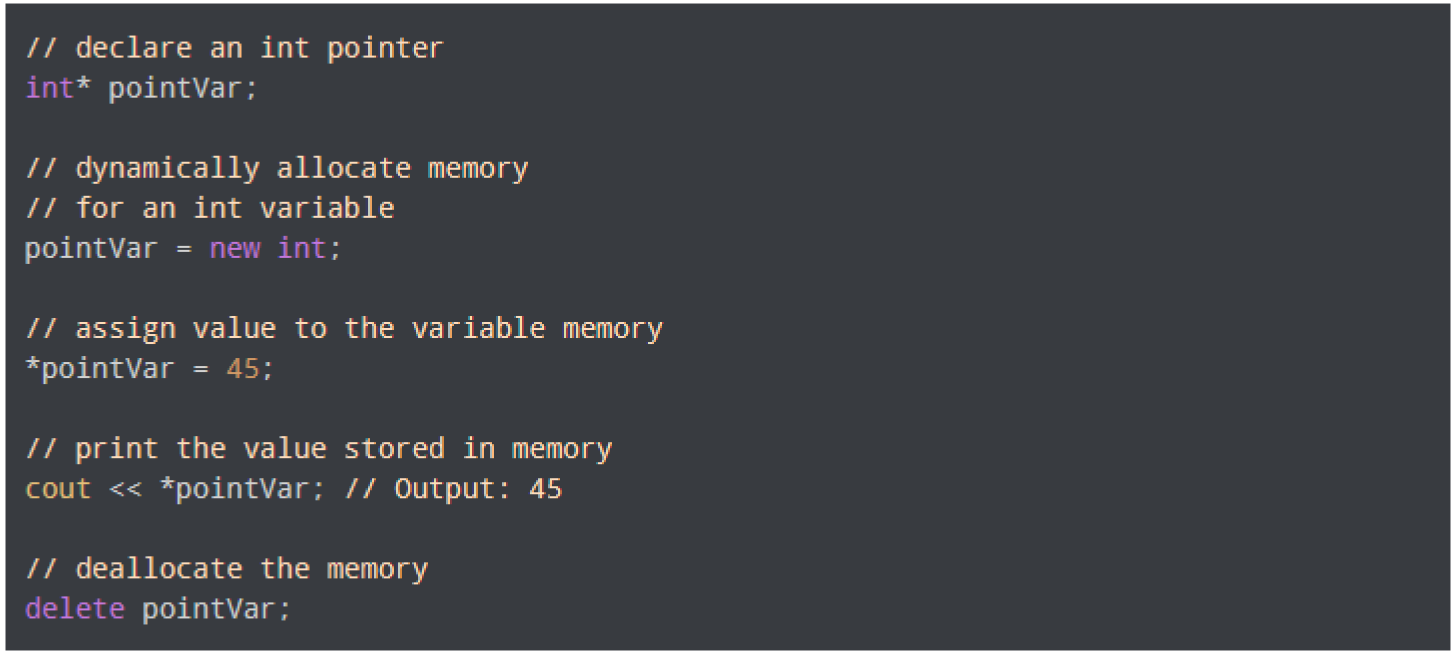
delete Operator

- Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.
- For this, the delete operator is used.
- It returns the memory to the operating system. This is known as memory deallocation.
- The syntax for this operator is

```
delete pointerVariable;
```

C++ Pointers and Arrays

C++ Memory Management: new and delete

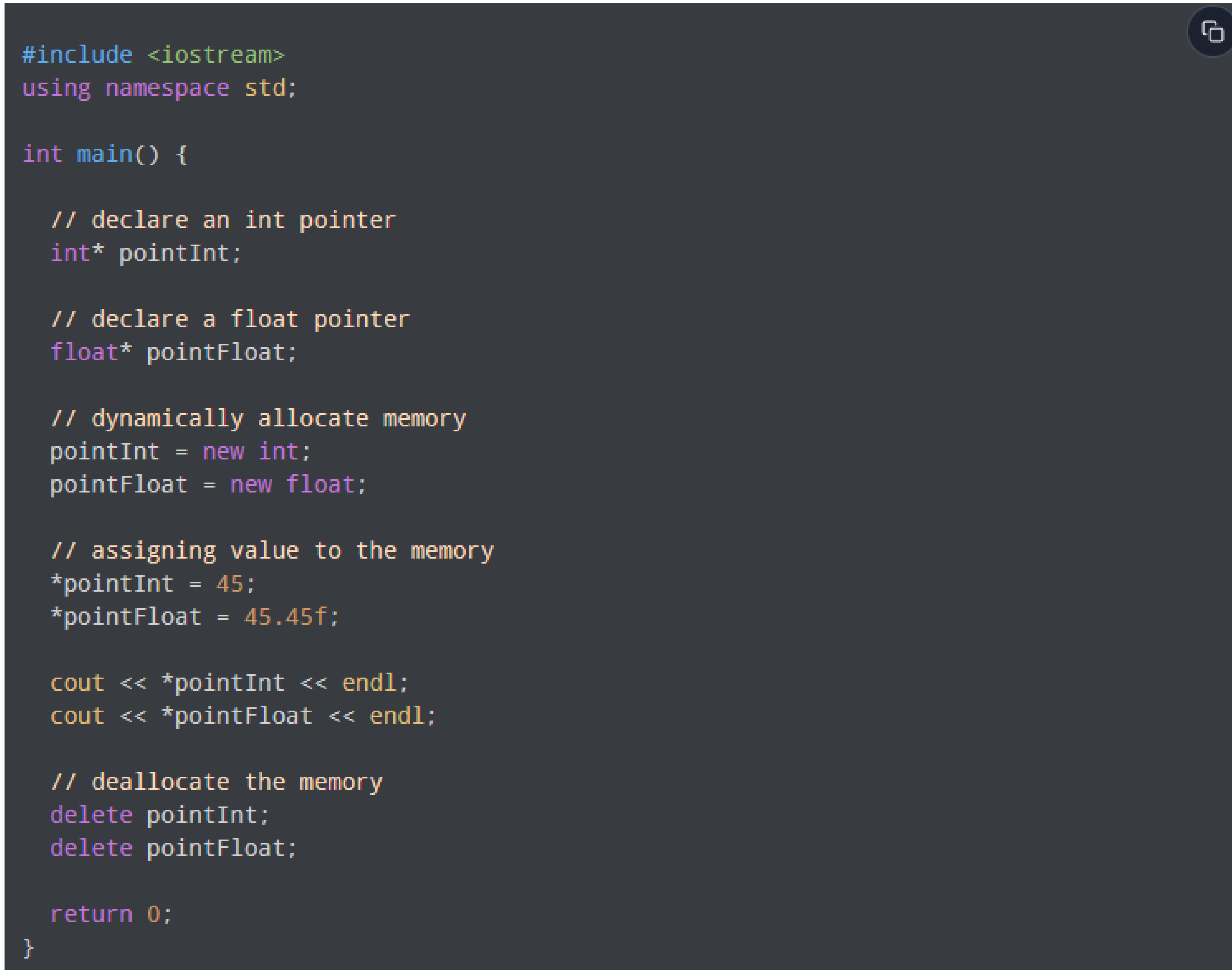


```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// for an int variable
pointVar = new int;

// assign value to the variable memory
*pointVar = 45;

// print the value stored in memory
cout << *pointVar; // Output: 45

// deallocate the memory
delete pointVar;
```

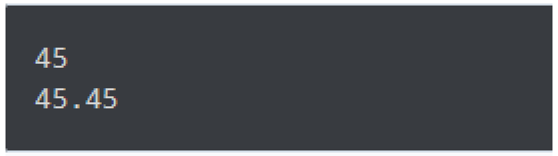
- Here, we have dynamically allocated memory for an int variable using the pointer pointVar.
- After printing the contents of pointVar, we deallocated the memory using delete.

Pointers

- **Example 1:** C++ Dynamic Memory Allocation



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

int main() {

  // declare an int pointer
  int* pointInt;

  // declare a float pointer
  float* pointFloat;

  // dynamically allocate memory
  pointInt = new int;
  pointFloat = new float;

  // assigning value to the memory
  *pointInt = 45;
  *pointFloat = 45.45f;

  cout << *pointInt << endl;
  cout << *pointFloat << endl;

  // deallocate the memory
  delete pointInt;
  delete pointFloat;

  return 0;
}
```



```
45
45.45
```

Pointers

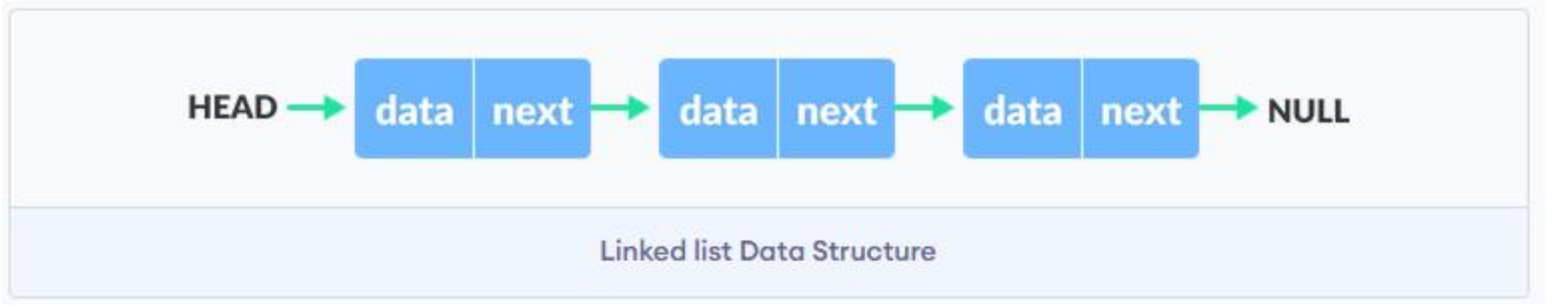
- **Example 2:** C++ new and delete Operator for Arrays

```cpp
// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user

#include <iostream>
using namespace std;

int main() {

  int num;
  cout << "Enter total number of students: ";
  cin >> num;
  float* ptr;

  // memory allocation of num number of floats
  ptr = new float[num];

  cout << "Enter GPA of students." << endl;
  for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ": ";
    cin >> *(ptr + i);
  }

  cout << "\nDisplaying GPA of students." << endl;
  for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ": " << *(ptr + i) << endl;
  }


   // ptr memory is released
   delete[] ptr;

   return 0;
 }
```

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9

Displaying GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9
```

Linked list Data Structure

- A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node. For example,



- You have to start somewhere, so we give the address of the first node a special name called HEAD. Also, the last node in the linked list can be identified because its next portion points to NULL.

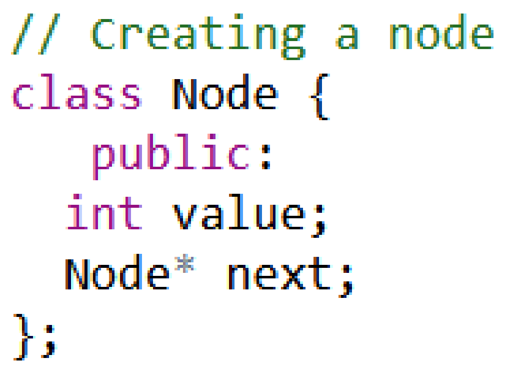
Linked list Data Structure

Representation of Linked List

Let's see how each node of the linked list is represented. Each node consists:

- A data item
- An address of another node

- We wrap both the data item and the next node reference in a struct as:



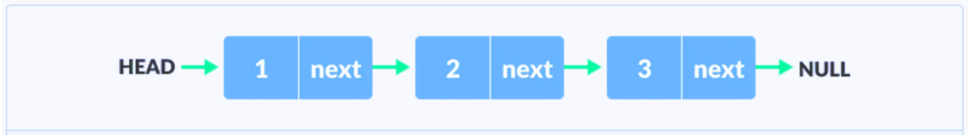
```
// Creating a node
class Node {
   public:
  int value;
  Node* next;
};
```

- Each struct node has a data item and a pointer to another struct node.

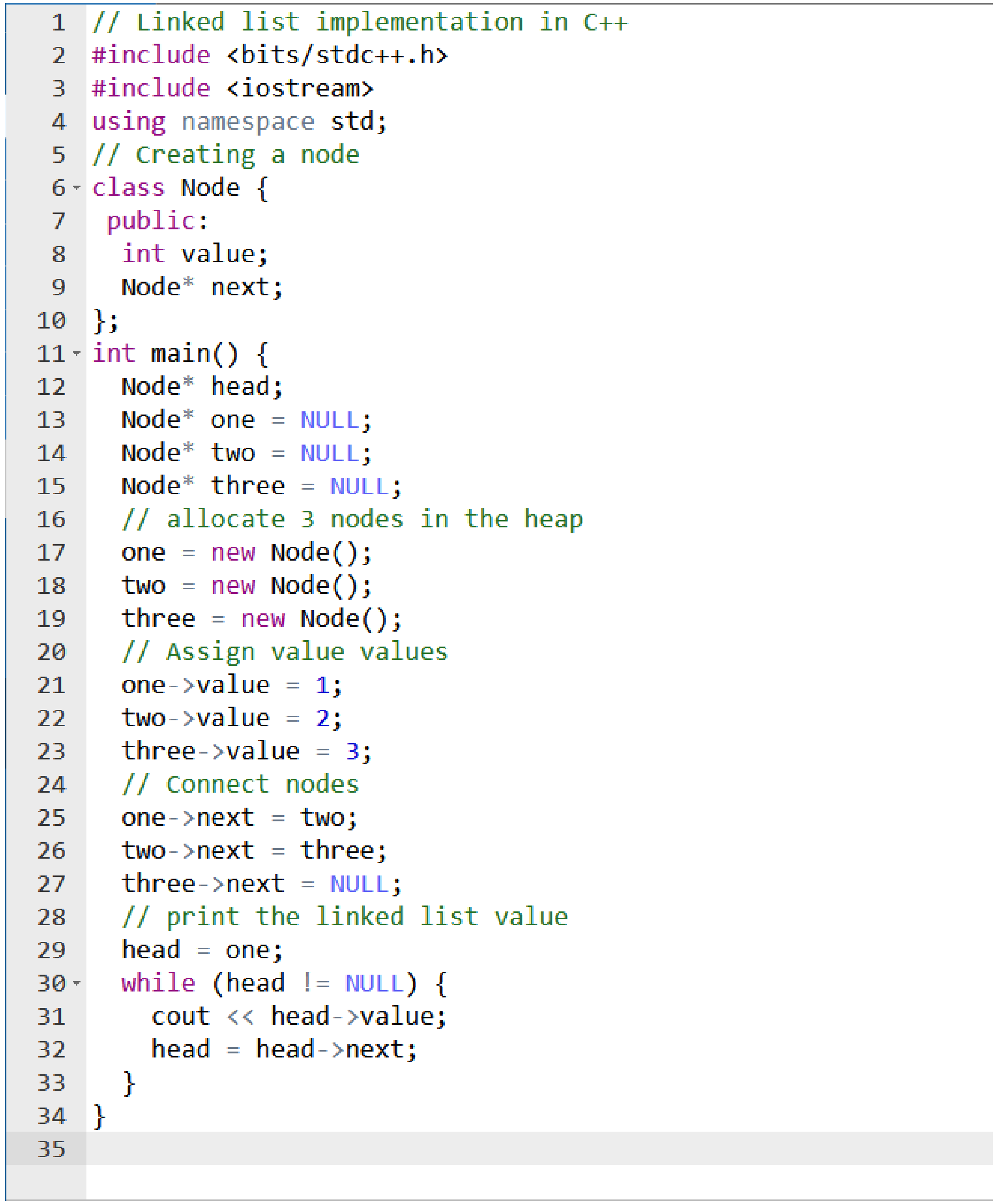
Linked list Data Structure

Representation of Linked List

Let us create a simple Linked List with three items to understand how this works.



Linked list Representation



```cpp
// Linked list implementation in C++
#include <bits/stdc++.h>
#include <iostream>
using namespace std;
// Creating a node
class Node {
 public:
  int value;
  Node* next;
};
int main() {
  Node* head;
  Node* one = NULL;
  Node* two = NULL;
  Node* three = NULL;
  // allocate 3 nodes in the heap
  one = new Node();
  two = new Node();
  three = new Node();
  // Assign value values
  one->value = 1;
  two->value = 2;
  three->value = 3;
  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = NULL;
  // print the linked list value
  head = one;
  while (head != NULL) {
    cout << head->value;
    head = head->next;
  }
}
```

Linked list Data Structure

- The power of a linked list comes from the ability to break the chain and rejoin it.
- E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

  - Create a new struct node and allocate memory to it.
  - Add its data value as 4
  - Point its next pointer to the struct node containing 2 as the data value
  - Change the next pointer of "1" to the node we just created.

Linked list Data Structure

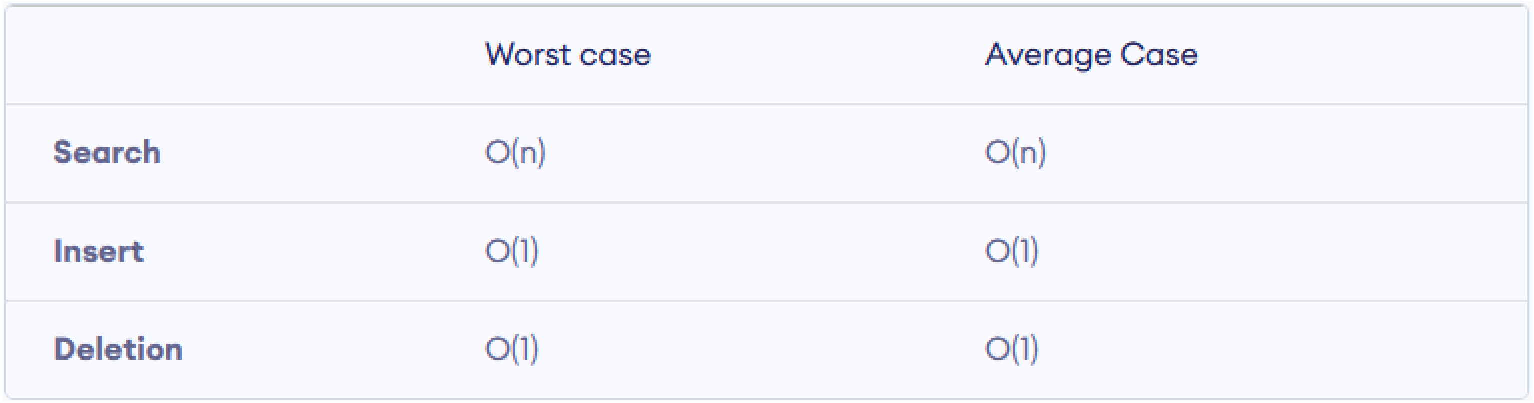
Linked List Utility

- Lists are one of the most popular and efficient data structures, with implementation in every programming language like C, C++, Python, Java, and C#.
- Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

Linked list Data Structure

Linked List Complexity

Time Complexity



	Worst case	Average Case
Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

Linked list Data Structure

Linked List Applications

- Dynamic memory allocation
- Implemented in stack and queue
- Hash tables, Graphs
- ,,,,

Linked list Data Structure

Linked List Operations: Traverse, Insert and Delete

There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.

Here's a list of basic linked list operations that we will cover in this article.

- Traversal - access each element of the linked list
- Insertion - adds a new element to the linked list
- Deletion - removes the existing elements
- Search - find a node in the linked list
- Sort - sort the nodes of the linked list

Linked list Data Structure

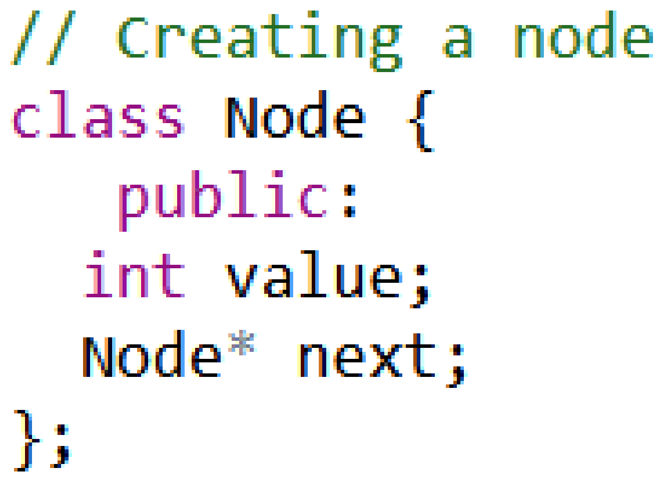
Linked List Operations: Traverse, Insert and Delete

Things to Remember about Linked List

- **head** points to the first node of the linked list
- **next** pointer of the last node is **NULL**, so if the next current node is NULL, we have reached the end of the linked list.

Linked list Data Structure

In all of the examples, we will assume that the linked list has three nodes 1 --->2 --->3 with node structure as below:



```
// Creating a node
class Node {
   public:
  int value;
  Node* next;
};
```

Linked list Data Structure

Traverse a Linked List

- Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.
- When temp is NULL, we know that we have reached the end of the linked list so we get out of the while loop.

Things to Remember about Linked List

- **head** points to the first node of the linked list
- **next** pointer of the last node is NULL, so if the next current node is NULL, we have reached the end of the linked list.

Linked list Data Structure

Insert Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

1. **Insert at the beginning**
   - Allocate memory for new node
   - Store data
   - Change next of new node to point to head
   - Change head to point to recently created node

Linked list Data Structure

Insert Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

2. Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

Linked list Data Structure

Insert Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

3. Insert at the Middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

Linked list Data Structure

Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

1. **Delete from beginning**
   - Point head to the second node
2. **Delete from end**
   - Traverse to second last element
   - Change its next pointer to null

3. Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

Linked list Data Structure

Search an Element on a Linked List

You can search an element on a linked list using a loop using the following steps. We are finding **item** on a linked list.

- Make **head** as the **current** node.
- Run a loop until the **current** node is NULL because the last element points to NULL.
- In each iteration, check if the key of the node is equal to **item**. If it the key matches the item, return **true** otherwise return **false**.

Linked list Data Structure

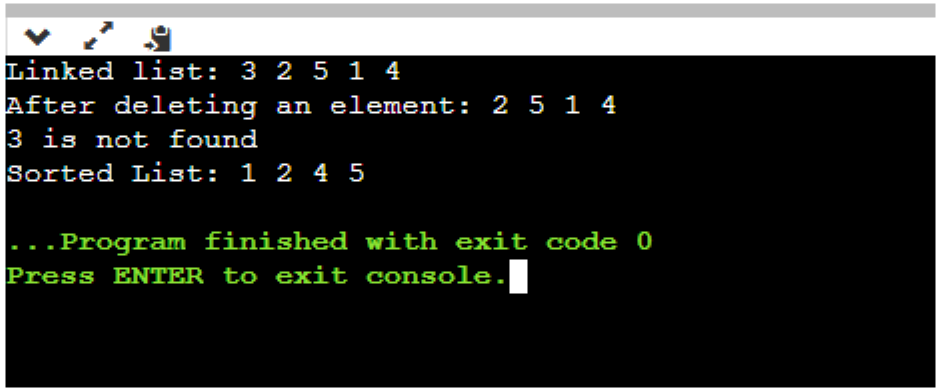
Sort Elements of a Linked List

We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

1. Make the head as the current node and create another node index for later use.
2. If head is null, return.
3. Else, run a loop till the last node (i.e. NULL).
4. In each iteration, follow the following step 5-6.
5. Store the next node of current in index.
6. Check if the data of the current node is greater than the next node. If it is greater, swap current and index.

Linked list Data Structure

Linked list Implementations in C++



```
Linked list: 3 2 5 1 4
After deleting an element: 2 5 1 4
3 is not found
Sorted List: 1 2 4 5

...Program finished with exit code 0
Press ENTER to exit console.
```

Stack Data Structure

A stack is a linear data structure that follows the principle of Last In First Out (LIFO).

This means the last element inserted inside the stack is removed first.

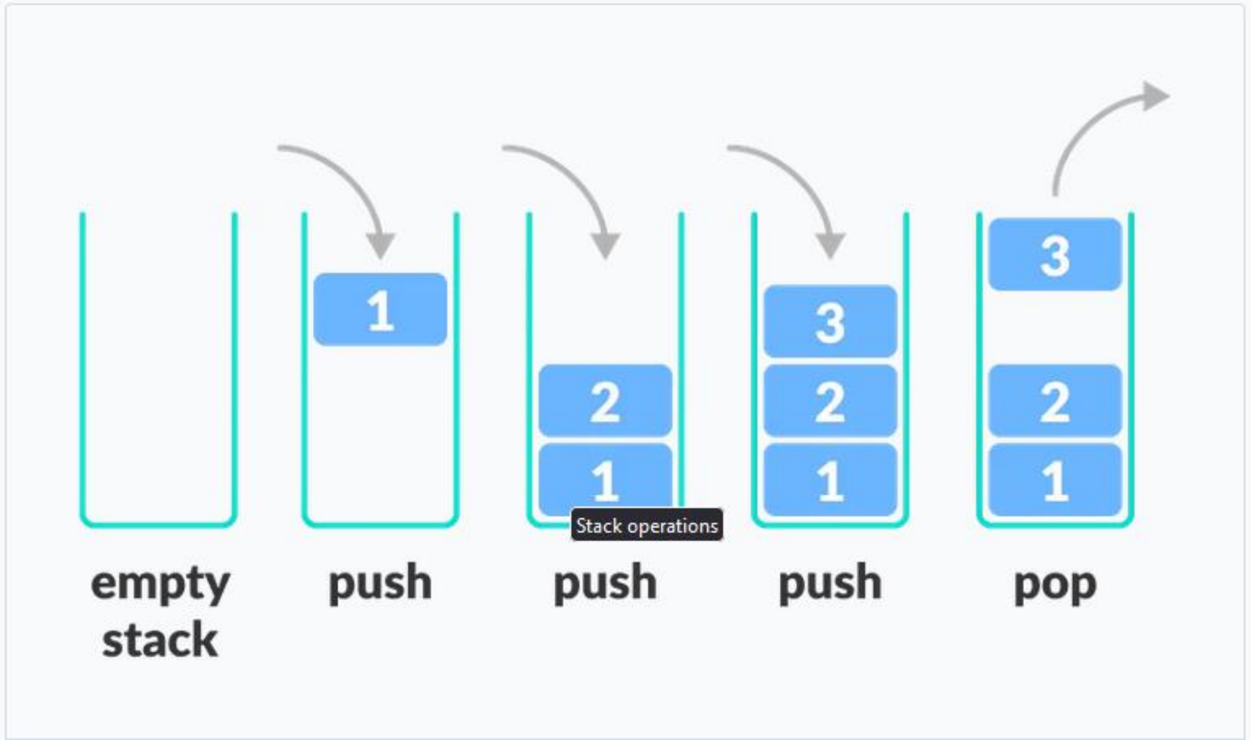
You can think of the stack data structure as the pile of plates on top of another.

Stack representation similar to a pile of plate

Stack Data Structure

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called push and removing an item is called pop.



Stack Push and Pop Operations

Stack Data Structure

Basic Operations of Stack

There are some basic operations that allow us to perform different actions on a stack.

- **Push**: Add an element to the top of a stack
- **Pop**: Remove an element from the top of a stack
- **IsEmpty**: Check if the stack is empty
- **IsFull**: Check if the stack is full
- **Peek**: Get the value of the top element without removing it

Stack Data Structure

Working of Stack Data Structure

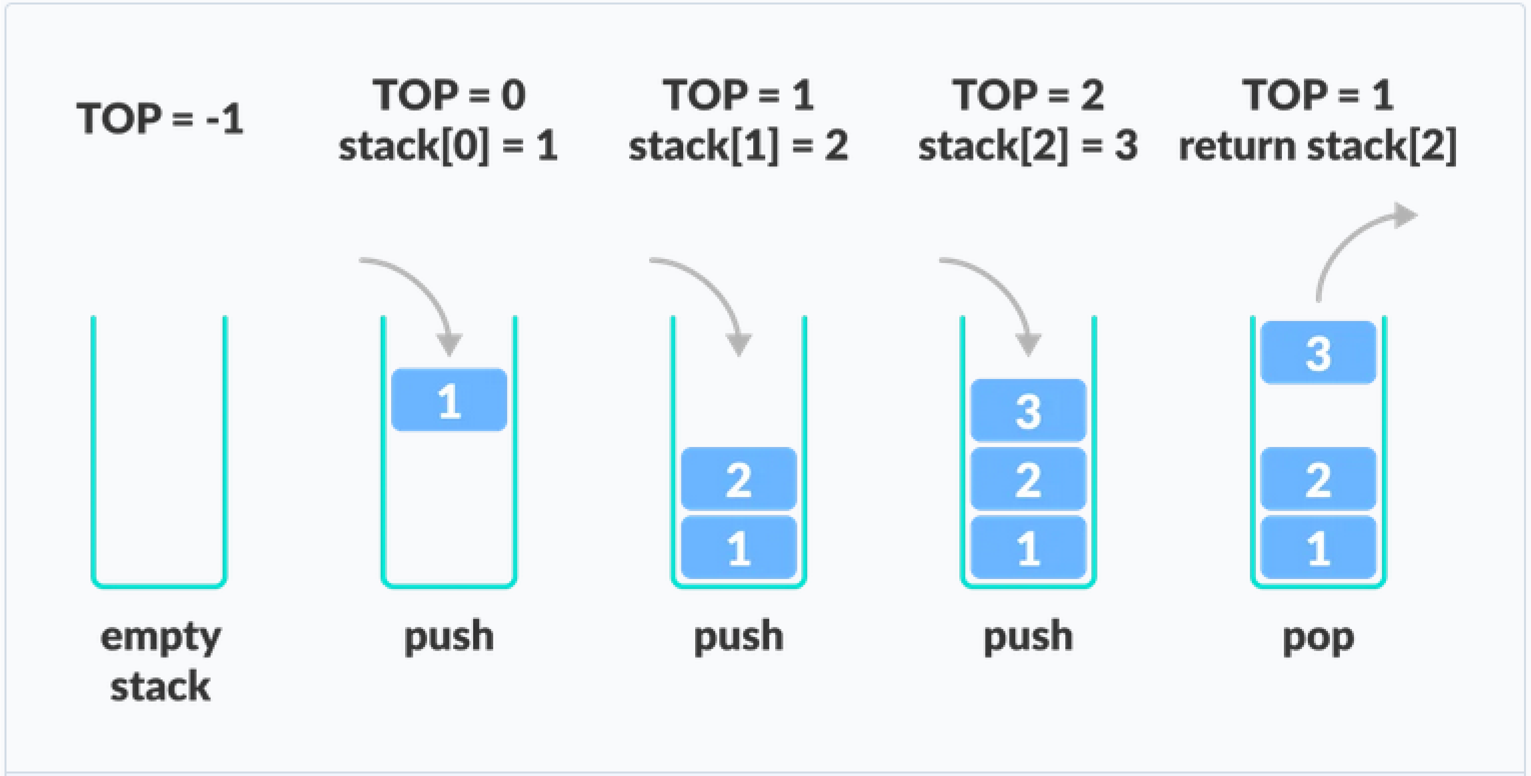
The operations work as follows:

1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty

Stack Data Structure

Working of Stack Data Structure

The operations work as follows:



Working of Stack Data Structure

Stack Data Structure

Stack Implementations in C++

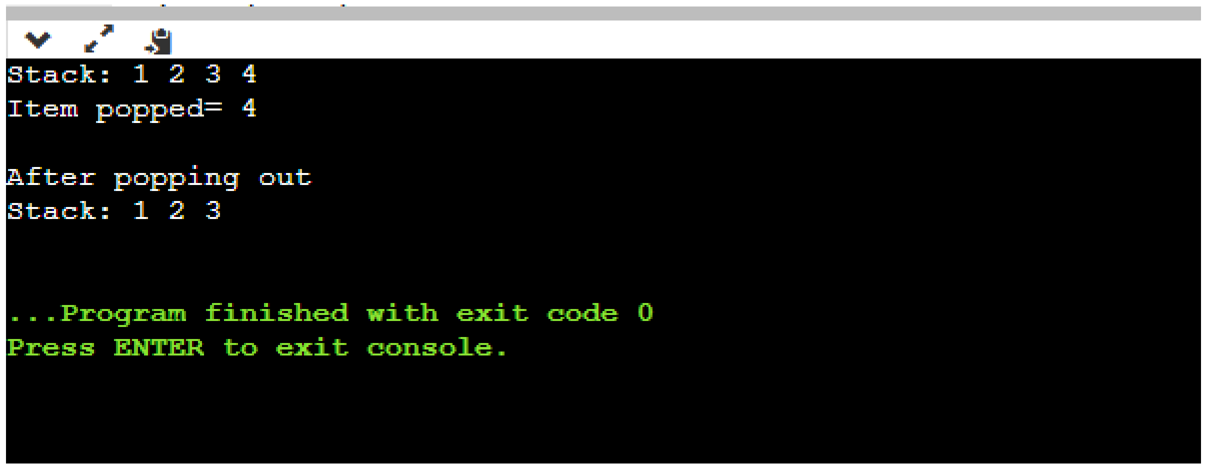
The most common stack implementation is using arrays, but it can also be implemented using lists.

A pointer called TOP is used to keep track of the top element in the stack.

1. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
2. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
3. On popping an element, we return the element pointed to by TOP and reduce its value.
4. Before pushing, we check if the stack is already full
5. Before popping, we check if the stack is already empty

Stack Data Structure

Stack Implementations in C++



```
Stack: 1 2 3 4
Item popped= 4

After popping out
Stack: 1 2 3


...Program finished with exit code 0
Press ENTER to exit console.
```

Stack Data Structure

Stack Time Complexity

For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1).

Applications of Stack Data Structure

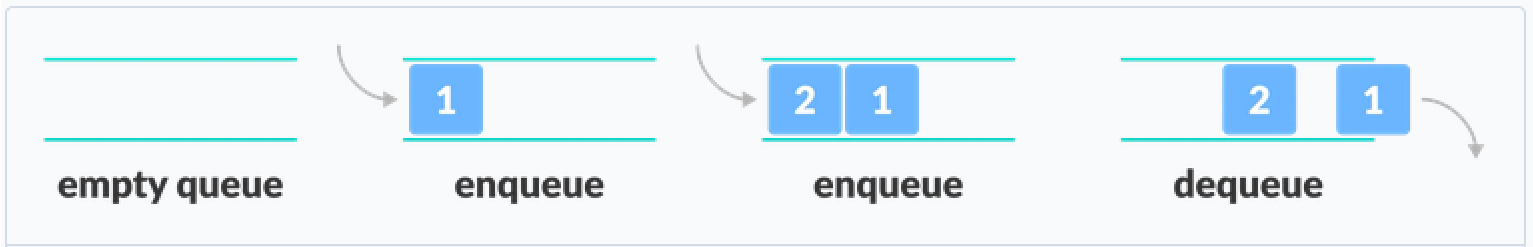
Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

- To reverse a word
- In compilers
- In browsers

Queue Data Structure

A queue is a useful data structure in programming. It is similar to the ticket queue outside a post office hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the First In First Out (FIFO) rule - the item that goes in first is the item that comes out first.



FIFO Representation of Queue

In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**.

Queue Data Structure

Basic Operations of Queue

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- **Enqueue:** Add an element to the end of the queue
- **Dequeue:** Remove an element from the front of the queue
- **IsEmpty:** Check if the queue is empty
- **IsFull:** Check if the queue is full
- **Peek:** Get the value of the front of the queue without removing it

Queue Data Structure

Working of Queue

Queue operations work as follows:

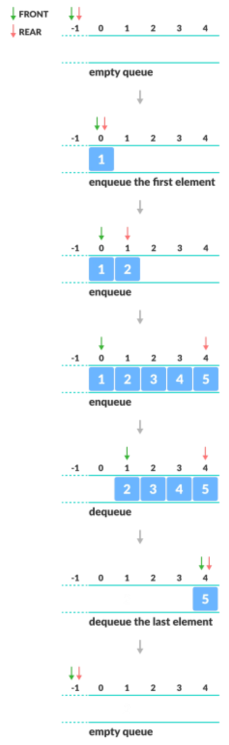
- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

Queue Data Structure

Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

Queue Data Structure



Queue Data Structure

Complexity Analysis

The complexity of enqueue and dequeue operations in a queue using an array is O(1). If you use pop(N) in python code, then the complexity might be O(n) depending on the position of the item to be popped.

Applications of Queue

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes.The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc
- Handling of interrupts in real-time systems.

References:

- https://www.programiz.com/cpp-programming/pointers
- https://www.programiz.com/dsa/linked-list
- https://www.onlinegdb.com/online_c++_compiler