C++ Pointers Tutorial

Contents

Time required: 120 minutes

Online Tutorial

Go through the following tutorials before going through the tutorial assignments.

C++ Pointers

Pointers are an important concept in C++ that allow you to manipulate memory directly and work with dynamic memory allocation. A pointer is a variable that stores the memory address of another variable.

Imagine a pointer as a map leading to a hidden treasure. Just like the map doesn't hold the treasure itself, a pointer doesn't store the actual data it points to; it simply stores the memory address of that data. This allows you to access and manipulate data indirectly, offering greater flexibility and control.

Here's an analogy:

- Think of a variable as a box containing a value (e.g., an integer).
- A pointer is like a label attached to the box, telling you where the box is locate

Declare a Pointer

Pointers are declared using the `*` symbol followed by the variable name.

```
// Declares a pointer to an integer
int* ptr;
// Declares a pointer to a float
float* fptr;
// Declares a pointer to a character
char* cptr;
```

Assign a Value to a Pointer

You assign a memory address to a pointer by using the address-of operator (&) followed by the variable name.

```
// Declare and initialize an integer variable
int x = 10;
// Declare a pointer to an integer
int* ptr;
// Initialize the pointer with the address of x
ptr = &x;
```

The following code shows the memory address of a pointer. Try it a couple of times. Notice each time the memory address is different.

```
#include <iostream>

int main()
{
    // Declare variable
    int num = 42;

    // Creat a pointer called p
    // Asign it the memory address of the variable num
    int *p = &num;

    // Display the memory address
    std::cout << p;
```

```
    return 0;
}
```

Example run:

```
0x6ae51ffdb4
```

Dereferencing Pointers

You can access the value stored at the memory location pointed to by a pointer using the `*` operator. For example:

```
int x = 10;
int* ptr = &x;
cout << "Value of x: " << x << endl; // Output: Value of x: 10
cout << "Value of *ptr: " << *ptr << endl; // Output: Value of *ptr: 10
```

- The **&** operator is used to access the memory location of a variable.
- The ***** operator is used to access the value of a memory address that is stored in a pointer.
- The same ***** sign is also used to declare a pointer, and it is different from the dereference operator.

Modify Value of Pointer

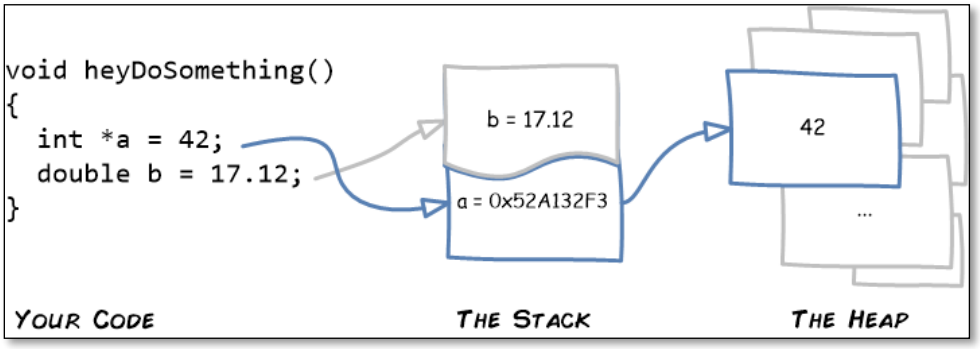
You can modify the value of the memory location pointed to by a pointer by dereferencing the pointer and assigning a new value.

```
int num = 10;
int* ptr = &num;
// Update the value of num through the pointer ptr
*ptr = 20;
// Output: Value of num after update: 20
cout << "Value of num after update: " << num << endl;
```

We have changed the value of a variable using the pointer.

***ptr** is basically an alias for `num`, meaning that they represent the same thing. When you change the value of ***ptr**, **num** is also changed, and vice-e-versa, changing **num** will also change the value of ***ptr**.

Stack vs Heap



In computer science, the terms "stack" and "heap" refer to different regions of memory that are used for storage during a program's execution. They have distinct characteristics, allocation methods, and purposes.

In programming, **stack and heap** are two crucial memory areas used for storing data in different ways. Understanding their differences is essential for efficient and optimized code.

Stack

- **Imagine a stack of plates:** As you add plates, they sit on top, and you remove them from the top first. This LIFO (Last In, First Out) order is how the stack works.
- **Purpose:** The stack is used for **temporary data**, like function arguments, local variables, and return values.
- **Allocation:** Memory on the stack is **automatically allocated** by the program when a function is called and **deallocated** when the function exits.
- **Benefits:** Faster access due to its LIFO structure and proximity to CPU.
- **Drawbacks:** Limited size and not suitable for long-lasting data.

Heap

- **Think of a messy room:** Data is scattered throughout, and you need to keep track of where each piece is. This is how the heap operates.

- **Purpose:** The heap stores **dynamically allocated data** whose lifespan is not confined to a specific function. This includes objects, dynamically sized arrays, and allocated memory using **new** and **delete** operators.
- **Allocation:** You, the programmer, are responsible for **explicitly allocating and deallocating** memory on the heap.
- **Benefits:** Flexible size and suitable for data needed across functions or the entire program.
- **Drawbacks:** Slower access compared to the stack due to its scattered nature and potential for memory leaks if deallocation isn't handled properly.

Key Differences

	Stack	Heap
Purpose	Temporary data	Dynamic data
Allocation	Automatic	Manual
Deallocation	Automatic	Manual
Structure	LIFO (Last In, First Out)	No specific order
Access speed	Faster	Slower
Size	Limited	Flexible
Lifetime	Function scope	Across functions or program

Choosing the right memory area

Choose the stack for temporary data requiring fast access and automatic memory management. Use the heap for dynamically allocated data needed beyond a function's lifetime, but be mindful of manual allocation and deallocation to avoid memory leaks.

- **Stack**: is used for static memory allocation, manages local variables and function calls, operates in a LIFO manner, and has limited size but faster access.
- **Heap**: is used for dynamic memory allocation, requires explicit management, offers more flexibility in memory usage, has a larger but more variable size, and might be slower due to dynamic allocation.

Dynamic Memory Allocation

You can use pointers to dynamically allocate memory on the heap using the **new** operator. The `new` operator returns a pointer to the newly allocated memory.

It's important to note that when you allocate memory dynamically using `new`, you should also free the memory using the **delete** operator to avoid memory leaks.

```
// Allocate memory for an integer and assign the pointer to it
int* ptr = new int;
// Store a value in the dynamically allocated memory
*ptr = 20;
// Output: Value stored in dynamically allocated memory: 20
cout << "Value stored in dynamically allocated memory: " << *ptr << endl;
// Always free the dynamically allocated memory to avoid memory leaks
delete ptr;
```

Pointers and Functions

Pointers can also be passed as arguments to functions, allowing functions to modify the value of a variable outside of its scope. For example:

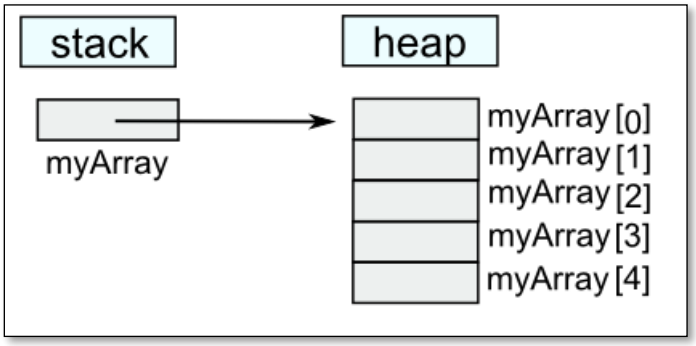
```
void increment(int* ptr) {
    // Increment the value pointed to by ptr
    (*ptr)++;
}

int main() {
    int num = 10;
    int* ptr = &num;

    increment(ptr); // Pass a pointer to x to the function
    // Output: Value of num after increment: 11
    cout << "Value of num after increment: " << num << endl;

    return 0;
}
```

Arrays & Pointers



The name of an array is actually a pointer to its first element. Each element can be accessed by incrementing the pointer.

Here is an example:

```
int arr[] = {2, 4, 6, 8};

int *p = arr;

// first element
cout << *p << endl;

// second element
cout << *(p+1) << endl;

// third element
cout << *(p+2) << endl;
```

This code used ***p = arr** and not **&arr**. This is because the array name is already a pointer and is the same as **&arr[0]**. The code used the dereference operator like this: ***(p+1)**, so that it accessed the incremented address. The parentheses are important!

You can use pointers to loop over an array.

```
int arr[] = {2, 4, 6, 8};

int *p = arr;

for(int i=0;i<4;i++) {
  cout << *p << endl;
  p++;
}
```

During each iteration of the loop the code increments the pointer by 1, making it point to the next element of the array.

Takeways

Pointers might seem confusing at first, but don't worry, you'll get the hang of it! Here are the key takeaways:

- A pointer is a variable that stores the memory address of another variable.
- It is declared using a ***** and the type of the value it points to, for example: **int *ptr;**
- The memory address of a variable can be accessed using the **&** operator, and assigned to a pointer, for example: **int *ptr = #**
- The value of a memory address can be accessed using the ***** operator, for example ***ptr** is the value stored at the address that **ptr** points to.
- A pointer can also be assigned to an array and be used to access the elements of the array, by simply incrementing the pointer.

Assignment Submission

1. Attach the program files.
2. Attach screenshots showing the successful operation of the program.
3. Submit in Blackboard.