TutorTube: Pointers in C++

Fall 2020

Introduction

Hello and welcome to TutorTube, where The Learning Center's Lead Tutors help you understand challenging course concepts with easy to understand videos. My name is Kalvin Garcia, Lead Tutor for Computer Science. In today's video, we will explore **pointers**.

Sometimes programs can get very large, especially when we declare a bunch of variables in our **functions**, our **structs**, our **classes**. **Pointers** can help minimize the memory being used by our programs, while also allowing for dynamic **allocation** of memory.

Value Types and Addresses

In C and C++ variables come in a variety of **value types**. They all store a particular data type: **int** stores integers, **char** stores characters, **string** stores strings, etc.

That data is stored in a particular location in the memory. The location is marked by an **address**, much like your family's home has an address. Whenever the variable is r**eferenced** by the program, the computer goes to that location in memory to find our **value**.

We as programmers can also **reference** our variables. You may have heard "**pass by reference.**" Just like our program gives the **address** to the computer to find a value, we can ask the program for the addresses of our variables. This is done with the **& (ampersand)** symbol: **&var**. This passes the **address** of the variable, rather than the **value** of the variable.

For example, if we declared an **int** variable of the name **number**, we could output the value: **cout << number << endl**. The console should then output the **value** and a **new line**. If we were to instead output the **reference** of our variable, by putting the **& (ampersand)** symbol before the name of our variable, **cout << &number << endl**, we would instead get the **address** of our variable.

Pointer Types and Addresses

A variable's **address** is the location a variable's **value** is stored. What does that have to do with **pointer**? Well, a **pointer** is a variable, but **addresses** are the **values** it holds. Wikipedia's hyperlinks that lead to other wiki articles are **pointers** to those articles. They use the address of the pages to point to their location.

A **pointer** is declared using the same data types as variables. To show the data type is a pointer we add an *** (asterisk)** symbol: **type* pointer_name** or **type *pointer_name**. A particular **pointer** type can only store **address** of the same **value** type. This means, for example, **int*** can only store **int addresses.**

Let's say we have an integer variable, number. We can declare an int pointer called "ptr," by putting an * (asterisk) before the name of our pointer. We can then assign the reference to our integer variable to the pointer by using the & (ampersand) symbol before out variable name: ptr = &number. If we output our pointer **(cout << ptr << endl)**, we will see the address to our variable, which we can confirm by also outputting the reference to out variable **(cout << &number << endl)**. What if we wanted to access the value inside the memory location? We would then use the *** (asterisk)** symbol before our **pointer**'s name, **cout << *ptr << endl**. This would instead output the **value** inside of the **address** our **pointer** holds.

Allocating Memory

Pointers don't need to be assigned variable **addresses**. We can also **allocate** the memory they use ourselves, as programmers. Our program allocates the memory a pointer uses during **run-time**.

If we have an **int pointer**, **ptr**, we can **allocate** memory to the **pointer** using the C++ operator **new**: **ptr = new int**. We can then assign **value** to the **pointer** by **dereferencing** it using the *** (asterisk)**. If we were to output the **pointer**, what would be displayed? What if we output the **dereferenced pointer**?

Using the concept of **dereferencing**, we can perform arithmetic with our **pointer**. In this case, we will add 10 to out **pointer**'s stored **value**. If we again output the **pointer**, what will we see? What if we **dereference** it?

In both cases, we find that outputting the **pointer** displays an **address**. This **address** remains the same even after performing arithmetic to our **pointer**'s **value**, which does change.

What if we instead wanted to create an **array**? We would use the same **new** operator, but instead add **[] (square brackets)**: **ptr = new int[SIZE]**. Inside of the brackets, we would need to declare the size of the **array**.

There are many ways to **dereference array pointers**. One method is using **array** notation, which uses the **[] (square brackets)** to denote the **index** of the **array** we are using: **ptr[i]**. This makes sense since we **allocated** an **array** of memory. Another method is using the *** (asterisk)** as before and adding the **index** to the **initial address** stored in our **pointer**: ***(ptr + i)**. This is because the memory is **allocated** in series.

Therefore, we can add to the **address** and find the next **address** of our **array**. To show this is the case, we can output **ptr + i** and **&ptr[i]**, side-by-side. We know that **referencing** a variable gives its **address**, so the **address &ptr[i]** should match **ptr + i**, which is our **initial address** plus the **index**.

Deallocating Memory

When we **allocate** the memory ourselves within our program, the compiler and computer don't know when to stop reserving the **allocated** memory slots, so we must tell the computer. We do this by **freeing** the **pointer** within our program whenever we are done using it.

In our previous example program, I did not use the C++ **delete** operator to **free** our **pointer**, but it still compiled. Again, the compiler does not know when we are **allocating** memory because it is done during **run-time**. It is good practice to always **free pointers** to avoid memory leaks.

In C++, this is done using the **delete**, as in: **delete ptr**, for single variables, and **delete [] ptr**, for array pointers.

Outro

Though we did not talk about C allocation methods, it is still important to note: we cannot **allocate pointers** by combining C and C++ methods. That is, if we **allocate** using **new**, we cannot use **realloc()** to resize the memory allocated. This is also true for **delete** and **free()**. If a **pointer** was **allocated** using **malloc()**, we cannot use **delete**. If a **pointer** was **allocated** using **new**, we cannot use **free()**. Remember, **new** goes with **delete** and **malloc()**/**calloc()** go with **free()**.

Thank you for watching TutorTube! I hope you enjoyed this video. Please subscribe to our channel for more exciting videos. Check out the links in the description below for more information about The Learning Center and follow us on social media. See you next time!

Code

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

int main() {
	//Value Types and Addresses
	//Decalre a variable
	int number;
	//Assign a value to the variable
	number = 10;
	//Output the variable
	cout << number << endl;
	//Output the reference
	cout << &number << endl;
	return 0;
}
#include <iostream>
using namespace std;

int main() {
	//Pointer Types and Addresses
	int number = 1;

	//Declare a pointer
	int *ptr;
	//Assign the address to the pointer
	ptr = &number;
```

```
	//Output the pointer
	cout << ptr << endl;
	cout << &number << endl;
	//Output the dereferenced pointer
	cout << *ptr << endl;

	return 0;
}
#include <iostream>
using namespace std;

#define SIZE 3

int main() {
	//Allocating Memory in C++
	int* ptr;
/*
	//Use the new operator to allocate memory
	ptr = new int;
	//Assign a value to the pointer
	*ptr = 10;
	//Output the pointer
	cout << ptr << endl;
	//Output the dereferenced pointer
	cout << *ptr << endl;
	//Do arithmatic with the pointer
	*ptr = *ptr + 10;
```

```
        //Output the pointer
        cout << ptr << endl;
        //Output the dereferenced pointer
        cout << *ptr << endl;
        //delete the allocated memory
        delete ptr;
*/
        //Use the new[] operator to allocate memory
        ptr = new int[SIZE];
        //Asign values to the pointer
        for(int i = 0; i < SIZE; ++i)
                ptr[i] = i;
        //Output the pointer and dereferenced pointer
      for(int i = 0; i < SIZE; ++i) {
                cout << *(ptr + i) << endl;
                cout << ptr + i << endl;
                cout << &ptr[i] << endl;
        }

        delete [] ptr;

        return 0;
}
```