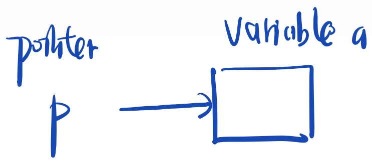
Learning Objectives: Pointer Basics

- **Define what a pointer is**
- **Declare a pointer** `int* p ;`
- **Apply the reference or address operator “&”**
- **Apply the dereference or content operator "*"**

`int *p = &a;`

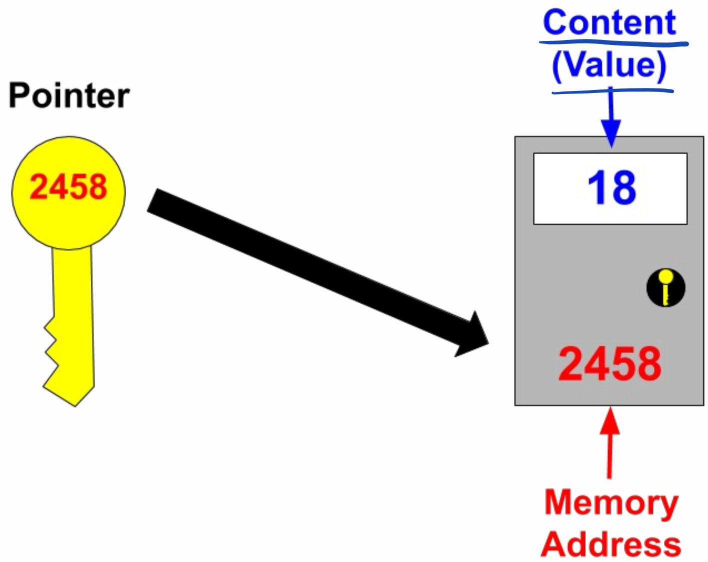


*p: value of a | &a: memory address of a

What Is a Pointer?

Pointer Introduction

A **pointer** is a data type that stores a memory address of another piece of data. Much like how an **array** *points* to all of its elements as a collection, pointers *point* to the memory address of the data that they are associated with.



.guides/img/PointerBasics

The picture above shows how pointers work. A pointer is like a key that stores the address of the locker that it is associated with. This association also enables the pointer to gain access to the content of what's inside the locker.

The advantage of using a pointer is that you do not need to worry about the value of the data that the pointer is pointing to. Thus, if the data ever changes its value, the pointer will still be able to access the new data as long as the pointer still points to the data's memory address.

Declaring a Pointer

Pointer Declaration

All pointers have a data type and a name that they are referred to. To declare a pointer, you need to have the following syntax in order:

- The data type of the pointer (e.g. int, string, etc.).
- An asterisk symbol *.
- A name for the pointer.

```
int* p;

cout << p << endl;
```
→ 0

challenge

What happens if you:

- change `int* p;` in the original code to `double* p;` (0), `bool* p;` (0), or `string* p;` (0)?
- change `int* p;` in the original code to `int *p;`? → 0
- change `int* p;` in the original code to `int *p = 2;`? → error

important

IMPORTANT

- The asterisk symbol can be placed anywhere between the end of the data type (i.e. `int`) and the variable name (i.e. `p`). `int* p;`, `int *p`, and `int * p` all work the same way.
- Pointers can only be assigned a memory address, which is why trying to assign `2` to a pointer will result in an error.
- Pointers that are not assigned a memory address will have a default output of `0`, also referred to as `null` pointers.

Reference Operator

Pointer Reference

A pointer can only be assigned a **memory address**. They cannot be assigned values that are `int`, `double`, `string`, etc. A memory address is denoted with the `&` symbol, called the **reference** operator, and they go in front of the variable that the address is associated with.

```
int a = 2;
int* p = &a;

cout << p << endl;
```

→ output memory address

Code Visualizer

challenge

What happens if you:

- run the same exact code again by clicking `TRY IT`?
- change `int* p = &a;` in the original code to `int* p = & a;`? → also works

Code Visualizer

important

IMPORTANT

- Memory is dynamic in C++ so whenever programs are compiled or executed again, they will often output memory addresses that are different from before.
- Though memory address is dynamic, once a pointer has been assigned a memory address, that association remains until the program is re-compiled or re-executed.

Dereference Operator

Pointer Dereference

Every memory address holds a value and that value can be accessed by using the **dereference operator**. The dereference operator is denoted by the asterisk symbol `*`.

```
int a = 5; //regular int variable set to 5
int* p = &a; //int pointer points to a's memory address

cout << *p << endl; //dereference p to print its content
```

→ 5

Code Visualizer

challenge

What happens if you:

- change `int a = 5;` in the original code to `int a = 50;`? → 50
- change `int* p = &a;` in the original code to `string* p = &a;`? → error

Code Visualizer

important

IMPORTANT

- A pointer can only be assigned a memory address of a variable that holds a value of the same type as the pointer. For example, if `&a` is the memory address of an `int` variable, then you cannot assign it to a `string` pointer (`string* p = &a`).

- Though memory address is dynamic, once a pointer has been assigned a memory address, that association remains until the program is re-compiled or re-executed.

```
int a = 123;
int b = 456;
int* p1 = &a;
int* p2 = &b;
int** p3 = &p1;
```

Determine what the output will be for each of the following.

1. `cout << b;` will return 456.
2. `cout << *p1;` will return 123.
3. `cout << *p2;` will return 456.
4. `cout << *p3;` will return a's memory address.
5. `cout << **p3;` will return ~~a's memory address~~ 123.

Solution:

1. `b` is not a pointer and will simply return `456`.
2. `*p1` will return `123` because `p1` —> `a` whose value is `123`.
3. `*p2` will return `456` because `p2` —> `b` whose value is `456`.
4. `*p3` will return `a's memory address` because `p3` —> `p1` whose value is a's memory address.
5. `**p3` will return `123` because `p3` —> `p1` —> `a` whose value is `123`.

Pointer to another Pointer

It is possible to have a pointer point to another pointer. To assign the memory address of a pointer to a new pointer, that new pointer must be denoted with two asterisk symbols `**`.

```
int a = 5;
int* p = &a;
int** p2 = &p;

cout << *p << endl;
cout << **p2 << endl;
```

p2 → p → a

→ 5

→ 5

Code Visualizer

challenge

What happens if you:

- change `int a = 5;` in the original code to `int a = 100;`? → 100 100
- change `cout << **p2 << endl;` in the original code to `cout << *p2 << endl;`? → 100 the memory address of variable a

Code Visualizer

important

IMPORTANT

*p2

Dereferencing a new pointer to an old pointer will return the memory address of the old pointer. If that pointer is dereferenced again, then the value of the variable that the old pointer pointed to will be returned. For example, `**p2` and `*p` both returned `5` because `p2` points to `p` which points to `a` which equals `5`.

Variable	Memory Address (&)	Content/Value (*)
(int) a	0x00	5
(int*) p	0x01	0x00
(int**) p2	0x02	0x01

.guides/img/PointerTable

challenge

Fun Fact:

If you dereference an array, it will return only the first element in the array.

```
int array[] = {24, 52, 97};

cout << *array << endl;
```

→ 24

Why Use Pointers?

Array Memory Usage

Before we can see how useful pointers can be, let's take a look at how memory is used within an array:

```
char names[3][6] = { "Alan",
                     "Bob",
                     "Carol" };

for (int i = 0; i < sizeof(names) / sizeof(names[0]); i++) {
  cout << names[i] << endl;
}
```

Alan
Bob
Carol

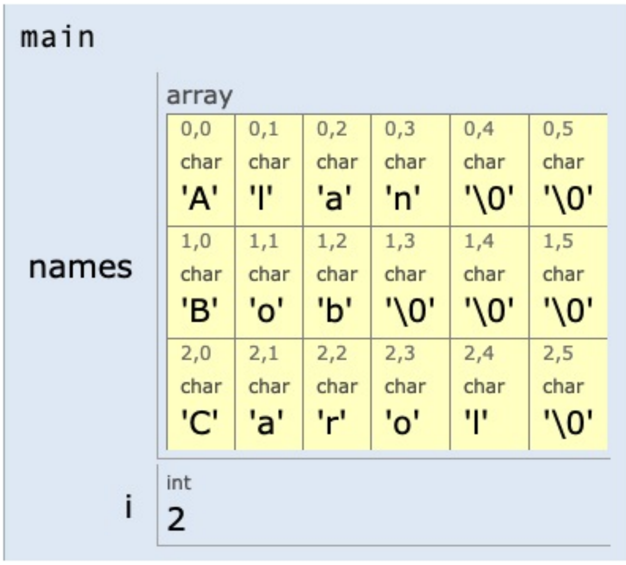
Code Visualizer

Remember: The row index `[3]` is *optional* but the *column* index `[6]` is *mandatory*.

▼ **Why is the column index 6 instead of 5?**

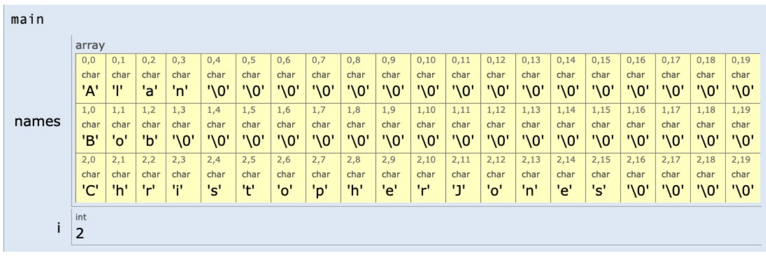
When working with a string of characters, the last character is always a special character known as a null character. This character is often referred as `NUL` or `\0`. Therefore, the maximum character length within this array is `'C'`, `'a'`, `'r'`, `'o'`, `'l'`, `'\0'`, which has 6 characters. This is why to be able to store all of the characters, the column index must be set to `6`.

The code above creates an array of characters where the row index `[3]` refers to the three starting characters `A` for `Alan`, `B` for `Bob`, and `C` for `Carol`, and the column index `6` refers to how many character each of the rows can hold which also includes the null characters (`NUL` or `\0`). Notice how the null characters also take up memory space.



.guides/img/ArrayNullCharacter

Here, we know how long the names would be, so we were able to budget the right amount memory for them. However, what if we didn't? In such a case, we would have to assign additional space for our characters, something larger, like `20` for example. That way, if the name `Carol` was a mistake and it was actually supposed to be `ChristopherJones`, we can feel more confident that the array will still be able to hold all of the characters. Unfortunately, this causes more memory to be wasted as depicted in the image below.



.guides/img/ArrayWasteMemory

Pointer Usage

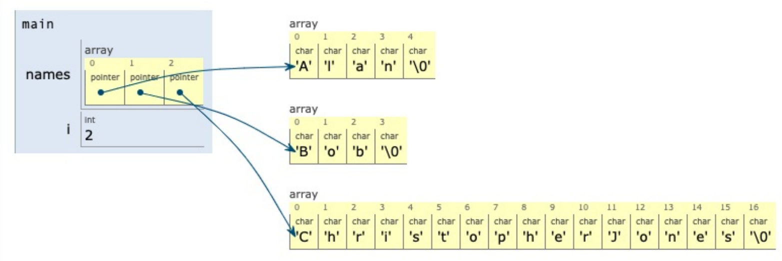
This is where pointers come in handy because they can help the system save memory. When using pointers for character arrays, the pointers will only point to the 3 leading characters `A`, `B`, and `C`. You do not need to specify the column index. **Note** that C++ requires the keyword `const` for pointers that point to characters within an array. This forces the characters to remain intact and prevents the pointer from potentially pointing elsewhere.

```
const char* names[] = { "Alan",
                        "Bob",
                        "ChristopherJones" };

for (int i = 0; i < sizeof(names) / sizeof(names[0]); i++) {
  cout << names[i] << endl;
}
```

→ Alan
Bob
ChristopherJones

Code Visualizer



.guides/img/PointerArray

Notice how we did not have to include any index values, which means the potential for wasting memory can be avoided. All we needed was to reserve enough memory for the creation of 3 pointers.