# Pointers

## Introduction to Modern C++

Ryan Baker

## Contents

# 1 Pointers

One of the major appeals of C++ is the ability to work with memory directly. At the heart of this feature lies the concept of pointers. A **pointer** is a variable that holds a memory address–often the address of another variable. Pointers "point" to locations in memory and allow indirect access and manipulation of the data stored there.
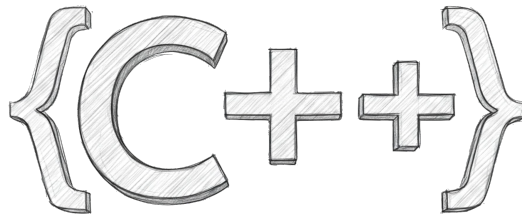
## 1.1 Declaring and Defining Pointers

Pointers are declared in relation to the type of data they point to. This is important: the compiler needs to know what type of value a pointer refers to so it knows how many bytes to read and how to interpret them. To declare a pointer, use an asterisk * along with the type:

```
1 int*    p1; // declares a pointer to an int
2 char*   p2; // declares a pointer to a char
3 double* p3; // declares a pointer to a double
```

A brief aside: there's frequent debate over whether to write `int* ptr` or `int *ptr` when declaring pointers. Both forms are valid and functionally identical–the choice comes down to coding style and personal preference.

I tend to prefer `int* ptr` because, conceptually, I think of `int*` as its own data type: "pointer to `int`". However, this style can introduce a subtle pitfall when declaring multiple variables on the same line:

```
1 int* p1, p2;  // p1 is a pointer, p2 is an int
2 int *p1, *p2; // p1 is a pointer, p2 is a pointer
3 int* p1, *p2; // p1 is a pointer, p2 is a pointer
```

Because of this ambiguity, many developers advocate for the second style–`int *ptr`–as a more honest representation of how the C++ parser actually reads the declaration. Personally, I find this case fringe enough to dismiss. I tend to believe we should prioritize how humans read the declaration over how the parser does.

### 1.1.1 Address-Of Operator &

To define a pointer that "points to" a variable, we use the **address-of operator** (&) to fetch the address of that variable:

```cpp
1 int x;
2 int* ptr { &x }; // ptr points to x
```

When we write &x, we're asking C++ to give us the memory address where x lives. This address is just a number–somewhat like a street address–that tells us where we can find the value of x in memory. Addresses are typically shown in hexadecimal:

```cpp
addressof.cpp

1 #include <iostream>
2
3 int main()
4 {
5     int x;
6     std::cout << &x << std::endl;
7 }
```

```
Terminal

$ clang++ -std=c++23 addressof.cpp && ./a.out
0x16db9ed7c
```

### 1.1.2 Dereference Operator *

To access the value pointed to by a pointer, we prefix the pointer with the **dereference operator** *:

```cpp
1 int* ptr;
2 int x { *ptr }; // assigns x to the value pointed to
```

Here's a more complete example of pointer dereferencing in action:

**dereference.cpp**

```cpp
1  #include <iostream>
2
3  int main()
4  {
5      int x { 42 };
6      int* ptr { &x };
7      std::cout << "x = " << *ptr << std::endl;
8  }
```

**Terminal**

```
$ clang++ -std=c++23 dereference.cpp
$ ./a.out
x = 42
```

When we write *ptr we're telling C++: "go to the address stored in ptr, and give me the value that resides there." Since pointers provide direct memory access, they can also be used to modify variables indirectly:

**increment.cpp**

```cpp
1  #include <iostream>
2
3  int main()
4  {
5      int x { 42 };
6      int* ptr { &x };
7      (*ptr)++;
8      std::cout << "x = " << x << std::endl;
9  }
```

**Terminal**

```
$ clang++ -std=c++23 increment.cpp && ./a.out
x = 43
```

## 1.2  `NULL` Pointers

A **null pointer** is a pointer that does not point to any valid memory address, typically represented by memory address 0. In C, `NULL` is a preprocessor macro that resolves to 0. In C++, the `nullptr` keyword was introduced to provide a type-safe representation of a null pointer, replacing `NULL`. Therefore, all of the following are valid ways to define null pointers:

```
1 int* p { 0 };
2 int* q { NULL };
3 int* r { nullptr };
```

In C++-land, `int* r { nullptr };` is typically preferred for better type safety and clarity.

We use `NULL` pointers because a pointer that points to nothing is safer than one that points to anything. As we'll soon see, stack memory is not default-initialized, meaning the following code:

```
1 int main() {
2     int* ptr;
3 }
```

creates a pointer that points to a (virtually) random address. Dereferencing this pointer before it's been initialized can cause a program-ending memory error– or worse, it might appear to work just fine, leading to a subtle runtime bug because you've accidentally manipulated memory you didn't intend to touch.

> For this reason, it's always a good idea to initialize pointers at the time of declaration. If there's no meaningful address to give them, use `nullptr`.
>
> ```
> 1 int* ptr { nullptr };
> ```
>
> This gives your program more deterministic behavior and allows it to fail fast and cheaply, reducing the chances of subtle bugs.

### 1.2.1 Dereferencing Null Pointers

Since a null pointer doesn't point to a valid memory address by definition, attempting to read or write through it results in undefined behavior. In most cases, this will manifest as a segmentation fault:

*nullptr.cpp*

```
1  #include <iostream>
2
3  int main()
4  {
5      int* ptr { nullptr };
6      std::cout << *ptr << std::endl;
7  }
```

Terminal

```
$ clang++ -std=c++23 nullptr.cpp
$ ./a.out
[1]    23783 segmentation fault  ./a.out
```

For this reason, if safety is critical and there's a chance a pointer might be null, you should always provide appropriate handling by checking for null before deferencing:

```
1  if (ptr != nullptr && *ptr /* ... */)
```

Some functions and algorithms return `nullptr` to indicate failure or the absence of a result. So, checking for null before dereferencing is essential to avoid undefined behavior.

One interesting note: null pointer dereferencing is allowed within the `sizeof` operator because the operand isn't actually evaluated:

```
1  int* ptr { nullptr };
2  std::cout << sizeof(*ptr) << std::endl; // legal!
```

## 1.3   Pointer Arithmetic

Pointers support various arithmetic operations that allow for navigation through memory. These operations fall into two broad categories: operations between pointers and integers, and operations between pointers themselves.

- **Pointer + Integer**: Addition and subtraction (which includes increment and decrement).

- **Pointer - Pointer**: Pointer subtraction and comparison.

Pointers can be incremented or decremented. Importantly, arithmetic on a `T*` respects the size of `T`. So, moving a pointer by an integer value adjusts its address by `sizeof(T)` bytes, not just one byte. For example, if `sizeof(int)` = 4, then incrementing an `int*` will move it by four bytes:



```cpp
#include <iostream>

int main()
{
    int* p { nullptr };
    std::cout << p++ << ',' << p++ << ',' << p << '\n';
}
```
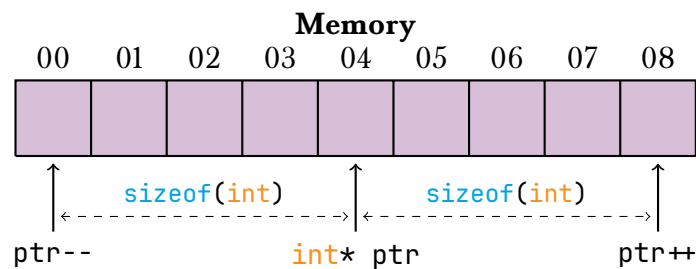
Terminal

```
$ clang++ -std=c++23 ppp.cpp
$ ./a.out
0x0,0x4,0x8
```

Similarly, adding an integer $n$ to a pointer will adjust the pointer by $n \times$ `sizeof`(T) bytes. You can think of this as moving the pointer by $n$ elements, where each element is of type T, assuming elements of that type are stored contiguously in memory.

```cpp
padd.cpp
1  #include <iostream>
2
3  int main()
4  {
5      short* p { nullptr }; // sizeof(short) == 2
6
7      std::cout << p << ',';
8      p += 4;
9      std::cout << p << ',';
10     p -= 3;
11     std::cout << p << std::endl;
12 }
```

```
Terminal
$ clang++ -std=c++23 padd.cpp
$ ./a.out
0x0,0x8,0x2
```

> When we add 4 to p, p is advanced by $4 \times$ `sizeof`(`short`) $= 8$ bytes, or 4 elements. Similarly, when we subtract 3 from p, it is advanced by $-3 \times$ `sizeof`(`short`) $= -6$ bytes, or $-3$ elements.

Subtracting two pointers of the same type will yield the number of elements between them–not the number of bytes. Given two pointers T* p1, T* p2, and letting $B$ representing the difference in bytes between them, we get:

$$\texttt{p1} - \texttt{p2} = \frac{B}{\texttt{sizeof(T)}}$$

In cases where the pointers are not properly aligned–that is, their addresses are not even multiples of `sizeof`(T)–the division truncates toward 0:

```
                         misaligned.cpp
1  #include <iostream>
2
3  int main()
4  {
5      int* p1 { (int*)1002 };
6      int* p2 { (int*)1000 };
7      std::cout << (p1 - p2) << std::endl; // 0
8  }
```

Here's a more realistic example demonstrating how pointer subtraction can be used to determine the number of elements between two variables:

```
                            psub.cpp
1   #include <iostream>
2
3   int main()
4   {
5       int a, b, c, d, e, f;
6       int* p1 { &a };
7       int* p2 { &f };
8
9       std::cout << (p1 - p2) << std::endl;
10  }
```

```
                            Terminal
$ clang++ -std=c++23 psub.cpp
$ ./a.out
5
```

> The exact value may vary depending on how variables are laid out on the stack, but this example assumes contiguous allocation in declaration order (which is common, though not guaranteed).

9

Pointers of the same type can be compared using standard relational operators like $=$, $\neq$, $<$, $>$, $\leqslant$, and $\geqslant$. The result of such comparisons depends solely on the memory addresses the pointers hold:

```cpp
// pcomp.cpp
1 #include <iostream>
2
3 int main()
4 {
5     int* p1 { nullptr };
6     int* p2 { p1 + 1 };
7
8     if (p1 < p2)
9         std::cout << "p1 points to a lower address\n";
10     else if (p1 == p2)
11         std::cout << "p1 and p2 point to the same
                address\n";
12     else
13         std::cout << "p1 points to a higher address\n";
14 }
```

```
Terminal
$ clang++ -std=c++23 pcomp.cpp
$ ./a.out
p1 points to a lower address
```

## 1.4   Pointers to Pointers