12.8 — Null pointers

learncpp.com/cpp-tutorial/null-pointers/

In the previous lesson (12.7 -- Introduction to pointers), we covered the basics of pointers, which are objects that hold the address of another object. This address can be dereferenced using the dereference operator (*) to get the object at that address:

```
#include <iostream>

int main()
{
    int x{ 5 };
    std::cout << x << '\n'; // print the value of variable x

    int* ptr{ &x }; // ptr holds the address of x
    std::cout << *ptr << '\n'; // use dereference operator to print the value of the
object at the address that ptr is holding (which is x's address)

    return 0;
}
```

The above example prints:

```
5
5
```

In the prior lesson, we also noted that pointers do not need to point to anything. In this lesson, we’ll explore such pointers (and the various implications of pointing to nothing) further.

Null pointers

Besides a memory address, there is one additional value that a pointer can hold: a null value. A **null value** (often shortened to) is a special value that means something has no value. When a pointer is holding a null value, it means the pointer is not pointing at anything. Such a pointer is called a **null pointer**.

The easiest way to create a null pointer is to use value initialization:

```
int main()
{
    int* ptr {}; // ptr is now a null pointer, and is not holding an address

    return 0;
}
```

Best practice

Value initialize your pointers (to be null pointers) if you are not initializing them with the address of a valid object.

Because we can use assignment to change what a pointer is pointing at, a pointer that is initially set to null can later be changed to point at a valid object:

```
#include <iostream>

int main()
{
    int* ptr {}; // ptr is a null pointer, and is not holding an address

    int x { 5 };
    ptr = &x; // ptr now pointing at object x (no longer a null pointer)

    std::cout << *ptr << '\n'; // print value of x through dereferenced ptr

    return 0;
}
```

The nullptr keyword

Much like the keywords `true` and `false` represent Boolean literal values, the **nullptr** keyword represents a null pointer literal. We can use `nullptr` to explicitly initialize or assign a pointer a null value.

```
int main()
{
    int* ptr { nullptr }; // can use nullptr to initialize a pointer to be a null
pointer

    int value { 5 };
    int* ptr2 { &value }; // ptr2 is a valid pointer
    ptr2 = nullptr; // Can assign nullptr to make the pointer a null pointer

    someFunction(nullptr); // we can also pass nullptr to a function that has a
pointer parameter

    return 0;
}
```

In the above example, we use assignment to set the value of `ptr2` to `nullptr`, making `ptr2` a null pointer.

Best practice

Use `nullptr` when you need a null pointer literal for initialization, assignment, or passing a null pointer to a function.

Dereferencing a null pointer results in undefined behavior

Much like dereferencing a dangling (or wild) pointer leads to undefined behavior, dereferencing a null pointer also leads to undefined behavior. In most cases, it will crash your application.

The following program illustrates this, and will probably crash or terminate your application abnormally when you run it (go ahead, try it, you won’t harm your machine):

```
#include <iostream>

int main()
{
    int* ptr {}; // Create a null pointer
    std::cout << *ptr << '\n'; // Dereference the null pointer

    return 0;
}
```

Conceptually, this makes sense. Dereferencing a pointer means “go to the address the pointer is pointing at and access the value there”. A null pointer holds a null value, which semantically means the pointer is not pointing at anything. So what value would it access?

Accidentally dereferencing null and dangling pointers is one of the most common mistakes C++ programmers make, and is probably the most common reason that C++ programs crash in practice.

Warning

Whenever you are using pointers, you’ll need to be extra careful that your code isn’t dereferencing null or dangling pointers, as this will cause undefined behavior (probably an application crash).

Checking for null pointers

Much like we can use a conditional to test Boolean values for `true` or `false`, we can use a conditional to test whether a pointer has value `nullptr` or not:

```
#include <iostream>

int main()
{
    int x { 5 };
    int* ptr { &x };

    if (ptr == nullptr) // explicit test for equivalence
        std::cout << "ptr is null\n";
    else
        std::cout << "ptr is non-null\n";

    int* nullPtr {};
    std::cout << "nullPtr is " << (nullPtr==nullptr ? "null\n" : "non-null\n"); //
explicit test for equivalence

    return 0;
}
```

The above program prints:

```
ptr is non-null
nullPtr is null
```

In lesson 4.9 -- Boolean values, we noted that integral values will implicitly convert into Boolean values: an integral value of `0` converts to Boolean value `false`, and any other integral value converts to Boolean value `true`.

Similarly, pointers will also implicitly convert to Boolean values: a null pointer converts to Boolean value `false`, and a non-null pointer converts to Boolean value `true`. This allows us to skip explicitly testing for `nullptr` and just use the implicit conversion to Boolean to test whether a pointer is a null pointer. The following program is equivalent to the prior one:

```
#include <iostream>

int main()
{
    int x { 5 };
    int* ptr { &x };

    // pointers convert to Boolean false if they are null, and Boolean true if they
are non-null
    if (ptr) // implicit conversion to Boolean
        std::cout << "ptr is non-null\n";
    else
        std::cout << "ptr is null\n";

    int* nullPtr {};
    std::cout << "nullPtr is " << (nullPtr ? "non-null\n" : "null\n"); // implicit
conversion to Boolean

    return 0;
}
```

Warning

Conditionals can only be used to differentiate null pointers from non-null pointers. There is no convenient way to determine whether a non-null pointer is pointing to a valid object or dangling (pointing to an invalid object).

Use nullptr to avoid dangling pointers

Above, we mentioned that dereferencing a pointer that is either null or dangling will result in undefined behavior. Therefore, we need to ensure our code does not do either of these things.

We can easily avoid dereferencing a null pointer by using a conditional to ensure a pointer is non-null before trying to dereference it:

```
// Assume ptr is some pointer that may or may not be a null pointer
if (ptr) // if ptr is not a null pointer
    std::cout << *ptr << '\n'; // okay to dereference
else
    // do something else that doesn't involve dereferencing ptr (print an error
message, do nothing at all, etc...)
```

But what about dangling pointers? Because there is no way to detect whether a pointer is dangling, we need to avoid having any dangling pointers in our program in the first place. We do that by ensuring that any pointer that is not pointing at a valid object is set to `nullptr`.

That way, before dereferencing a pointer, we only need to test whether it is null -- if it is non-null, we assume the pointer is not dangling.

Best practice

A pointer should either hold the address of a valid object, or be set to nullptr. That way we only need to test pointers for null, and can assume any non-null pointer is valid.

Unfortunately, avoiding dangling pointers isn't always easy: when an object is destroyed, any pointers to that object will be left dangling. Such pointers are *not* nulled automatically! It is the programmer's responsibility to ensure that all pointers to an object that has just been destroyed are properly set to `nullptr`.

Warning

When an object is destroyed, any pointers to the destroyed object will be left dangling (they will not be automatically set to `nullptr`). It is your responsibility to detect these cases and ensure those pointers are subsequently set to `nullptr`.

Legacy null pointer literals: 0 and NULL

In older code, you may see two other literal values used instead of `nullptr`.

The first is the literal `0`. In the context of a pointer, the literal `0` is specially defined to mean a null value, and is the only time you can assign an integral literal to a pointer.

```
int main()
{
    float* ptr { 0 };  // ptr is now a null pointer (for example only, don't do this)

    float* ptr2; // ptr2 is uninitialized
    ptr2 = 0; // ptr2 is now a null pointer (for example only, don't do this)

    return 0;
}
```

As an aside…

On modern architectures, the address `0` is typically used to represent a null pointer. However, this value is not guaranteed by the C++ standard, and some architectures use other values. The literal `0`, when used in the context of a null pointer, will be translated into whatever address the architecture uses to represent a null pointer.

Additionally, there is a preprocessor macro named `NULL` (defined in the <cstddef> header). This macro is inherited from C, where it is commonly used to indicate a null pointer.

```
#include <cstddef> // for NULL

int main()
{
    double* ptr { NULL }; // ptr is a null pointer

    double* ptr2; // ptr2 is uninitialized
    ptr2 = NULL; // ptr2 is now a null pointer

    return 0;
}
```

Both `0` and `NULL` should be avoided in modern C++ (use `nullptr` instead). We discuss why in lesson 12.11 -- Pass by address (part 2).

Favor references over pointers whenever possible

Pointers and references both give us the ability to access some other object indirectly.

Pointers have the additional abilities of being able to change what they are pointing at, and to be pointed at null. However, these pointer abilities are also inherently dangerous: A null pointer runs the risk of being dereferenced, and the ability to change what a pointer is pointing at can make creating dangling pointers easier:

```
int main()
{
    int* ptr { };

    {
        int x{ 5 };
        ptr = &x; // assign the pointer to an object that will be destroyed (not
possible with a reference)
    } // ptr is now dangling and pointing to invalid object

    if (ptr) // condition evaluates to true because ptr is not nullptr
        std::cout << *ptr; // undefined behavior

    return 0;
}
```

Since references can’t be bound to null, we don’t have to worry about null references. And because references must be bound to a valid object upon creation and then can not be reseated, dangling references are harder to create.

Because they are safer, references should be favored over pointers, unless the additional capabilities provided by pointers are required.

Best practice

Favor references over pointers unless the additional capabilities provided by pointers are needed.

Quiz time

Question #1

1a) Can we determine whether a pointer is a null pointer or not? If so, how?

Show Solution

1b) Can we determine whether a non-null pointer is valid or dangling? If so, how?

Show Solution

Question #2

For each subitem, answer whether the action described will result in behavior that is: predictable, undefined, or possibly undefined. If the answer is “possibly undefined”, clarify when.

2a) Assigning a new address to a non-const pointer

Show Solution

2b) Assigning nullptr to a pointer

Show Solution

2c) Dereferencing a pointer to a valid object

Show Solution

2d) Dereferencing a dangling pointer

Show Solution

2e) Dereferencing a null pointer

Show Solution

2f) Dereferencing a non-null pointer

Show Solution

Question #3

Why should we set pointers that aren’t pointing to a valid object to ‘nullptr’?

Show Solution