6.7a — Null pointers

BY ALEX ON AUGUST 12TH, 2015 | LAST MODIFIED BY ALEX ON MAY 19TH, 2018

Null values and null pointers

Just like normal variables, pointers are not initialized when they are instantiated. Unless a value is assigned, a pointer will point to some garbage address by default.

Besides memory addresses, there is one additional value that a pointer can hold: a null value. A **null value** is a special value that means the pointer is not pointing at anything. A pointer holding a null value is called a **null pointer**.

In C++, we can assign a pointer a null value by initializing or assigning it the literal 0:

```
float *ptr { 0 };  // ptr is now a null pointer

float *ptr2; // ptr2 is uninitialized
ptr2 = 0; // ptr2 is now a null pointer
```

Pointers convert to boolean false if they are null, and boolean true if they are non-null. Therefore, we can use a conditional to test whether a pointer is null or not:

```
double *ptr { 0 };

// pointers convert to boolean false if they are null, and boolean true if they are non-null
if (ptr)
    cout << "ptr is pointing to a double value.";
else
    cout << "ptr is a null pointer.";
```

Best practice: Initialize your pointers to a null value if you're not giving them another value.

Dereferencing null pointers

In the previous lesson, we noted that dereferencing a garbage pointer would lead to undefined results. Dereferencing a null pointer also results in undefined behavior. In most cases, it will crash your application.

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 doesn't have an address. So when you try to access the value at that address, what should it do?

The NULL macro

C (but not C++) defines a special preprocessor macro called NULL that is #defined as the value 0. Even though this is not technically part of C++, its usage is common enough that it should work in every C++ compiler:

```
double *ptr { NULL }; // assign address 0 to ptr
```

However, because NULL is a preprocessor macro and because it's technically not a part of C++, best practice in C++ is to avoid using it.

nullptr in C++11

Note that the value of 0 isn't a pointer type, so assigning 0 to a pointer to denote that the pointer is a null pointer is a little inconsistent. In rare cases, when used as a literal argument, it can even cause problems because the compiler can't tell whether we mean a null pointer or the integer 0:

```
doSomething(0); // is 0 an integer argument or a null pointer argument?  (It will assume integer)
```

To address these issues, C++11 introduces a new keyword called **nullptr**. nullptr is both a keyword and an rvalue constant, much like the boolean keywords true and false are.

Starting with C++11, this should be favored instead of 0 when we want a null pointer:

```
int *ptr { nullptr }; // note: ptr is still an integer pointer, just set to a null value
```

C++ will implicitly convert nullptr to any pointer type. So in the above example, nullptr is implicitly converted to an integer pointer, and then the value of nullptr assigned to ptr. This has the effect of making integer pointer ptr a null pointer.

This can also be used to call a function with a nullptr literal:

```
#include <iostream>

void doSomething(double *ptr)
{
    // pointers convert to boolean false if they are null, and boolean true if they are non-null
    if (ptr)
        std::cout << "You passed in " << *ptr << '\n';
    else
        std::cout << "You passed in a null pointer\n";
}

int main()
{
    doSomething(nullptr); // the argument is definitely a null pointer (not an integer)

    return 0;
}
```

Best practice: With C++11, use nullptr to initialize your pointers to a null value.

std::nullptr_t in C++11

C++11 also introduces a new type called std::nullptr_t (in header <cstddef>). std::nullptr_t can only hold one value: nullptr! While this may seem kind of silly, it's useful in one situation. If we want to write a function that accepts a nullptr argument, what type do we make the parameter? The answer is std::nullptr_t.

```
#include <iostream>
#include <cstddef> // for std::nullptr_t

void doSomething(std::nullptr_t ptr)
{
    std::cout << "in doSomething()\n";
}

int main()
{
    doSomething(nullptr); // call doSomething with an argument of type std::nullptr_t

    return 0;
}
```

You probably won't ever need to use this, but it's good to know, just in case.


6.8 -- Pointers and arrays


Index


6.7 -- Introduction to pointers