10.9 — Null pointers

ALEX JULY 4, 2021

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)
    std::cout << "ptr is pointing to a double value.";
else
    std::cout << "ptr is a null pointer.";
```

Best practice

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

Indirection through null pointers

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

Conceptually, this makes sense. Indirection through 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

In C++, there is a special preprocessor macro called NULL (defined in the <cstddef> header). This macro was inherited from C, where it is commonly used to indicate a null pointer.

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

  double* ptr { NULL }; // ptr is a null
  pointer
```

The value of NULL is implementation defined, but is usually defined as the integer constant 0. Note: as of C++11, NULL can be defined as nullptr instead (which we'll discuss in a bit).

Best practice

Because NULL is a preprocessor macro with an implementation defined value, avoid using NULL.

The perils of using 0 (or NULL) for null pointers

Note that the value of 0 isn't a pointer type, so assigning 0 (or NULL, pre-C++11) 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:

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

void print(int x)
{
 std::cout << "print(int): " << x << '\n';
}

void print(int* x)
{
 if (!x)
  std::cout << "print(int*): null\n";
 else
  std::cout << "print(int*): " << *x << '\n';
}

int main()
{
 int* x { NULL };
 print(x); // calls print(int*) because x has type int*
 print(0); // calls print(int) because 0 is an integer literal
 print(NULL); // likely calls print(int), although we probably wanted print(int*)

 return 0;
}
```

In the likely case where NULL is defined as value 0, print(NULL) will call print(int), not print(int*) like you might expect for a null pointer literal.

---

nullptr in C++11

To address the above issues, C++11 introduces a new keyword callednullptr. nullptr is a keyword, 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:

```cpp
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.

We can also call a function with a nullptr literal, which will match to any parameter that takes a pointer value:

```cpp
#include <iostream>

void print(int x)
{
 std::cout << "print(int): " << x << '\n';
}

void print(int* x)
{
 if (!x)
  std::cout << "print(int*): null\n";
 else
  std::cout << "print(int*): " << *x << '\n';
}

int main()
{
 int* x { nullptr };
 print(x); // calls print(int*)

 print(nullptr); // calls print(int*) as desired

 return 0;
}
```

Best practice

Use nullptr to initialize your pointers to a null value.

std::nullptr_t

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 only 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.

Next lesson
10.10 Pointers and arrays

Back to table of contents

Previous lesson
10.8 Introduction to pointers

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest