Null Pointer

This lesson highlights the key features of the null pointer.

we'll cover the following ^ • Properties • Generic code

Before C++11, was often used to represent an empty or null value when the NULL macro was not applicable. The issue with the literal is that it can be the null pointer (void*) or the number o. This is defined by the context.

Therefore, a small program with the number o should be confusing.

```
#include <iostream>
#include <typeinfo>

int main(){

  std::cout << std::endl;

  int a= 0;
  int* b= 0;
  auto c= 0;
  std::cout << typeid(c).name() << std::endl;

auto res= a+b+c;
  std::cout << "res: " << res << std::endl;
  std::cout << typeid(res).name() << std::endl;
}</pre>
```

The variable c is of type int, and the variable res is of type pointer to int:
int*. Pretty simple, right? The expression a+b+c in line 13 is pointer
arithmetic.

An alternative is the **NULL** macro, but the issue is that it implicitly converts to int.

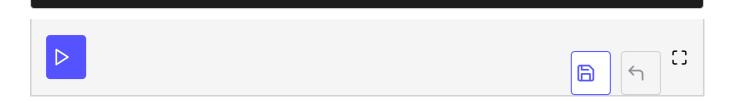
The new null pointer, nullptr, cleans up the ambiguity of the number o and the macro NULL. nullptr is of type std::nullptr t.

Properties

- We can assign nullptr to arbitrary pointers.
- The pointer becomes a null pointer and points to no data.
- We cannot dereference a nullptr.
- A **nullptr** pointer can be compared with all other pointers.
- A nullptr can be converted to all pointers. This also holds true for pointers to class members.

We cannot compare or convert a nullptr to an integral type. There is one exception to this rule. We can implicitly compare and convert a bool value with a nullptr. Therefore, we can use a nullptr in a logical expression.

```
#include <iostream>
                                                                                             G
#include <string>
std::string overloadTest(char*){
 return "char*";
std::string overloadTest(long int){
  return "long int";
int main(){
  std::cout << std::endl;</pre>
 long int* pi = nullptr;
  // long int i= nullptr;
                                 // ERROR
  auto nullp= nullptr;
                                 // type std::nullptr_t
  bool b(nullptr);
  std::cout << std::boolalpha << "b: " << b << std::endl;</pre>
  if ( nullptr < &val ){ std::cout << "nullptr < &val" << std::endl; }</pre>
  // calls char*
  std::cout << "overloadTest(nullptr)= " << overloadTest(nullptr)<< std::endl;</pre>
  std::cout << std::endl;</pre>
```



The simple rule is: Use nullptr instead of o or NULL. Still not convinced? Here is my final and strongest point.

Generic code

The literal o and NULL show their true nature in generic code. Thanks to template argument deduction, both literals are integral types in the function template. There is no hint that both literals were null pointer constants.

The code below will give an error:

```
#include <cstddef>
#include <iostream>

template<class P >
void functionTemplate(P p){
   int* a= p;
}

int main(){
   int* b= NULL;
   int* c= nullptr;

functionTemplate(0);
   functionTemplate(NULL);
   functionTemplate(nullptr);
}
```

We can use o and NULL to initialize the int pointer in line 10 and 11, but if we use the values o and NULL as arguments of the function template, the compiler will loudly complain.

The compiler deduces of in the function template to type int; it deduces NULL to the type long int. But these observations will not hold true for nullptr.

The nullptr in lines 12 and 16 is of the type std::nullptr_t.

In the next lesson, we'll look at a few more examples of pointers.