nullptr Instead of 0 or NULL

In this lesson, we will learn how new null pointer nullptr cleans up in C++ with the ambiguity of the number 0 and the macro NULL.

WE'LL COVER THE FOLLOWING

The Number 0

The issue with the literal 0 is that it can be either the null pointer `(void*)0` or the number 0 depending on the context of the problem in question.

Therefore, programs using the number 0 should be confusing.

```
// null.cpp

#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;

  std::cout << std::endl;

}
```

The question remains, what is the data type of variable `c` in line 12 and of variable `res` in line 15?

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 15 is pointer arithmetic.

The macro NULL

The issue with the null pointer `NULL` is that it implicitly converts to `int`.

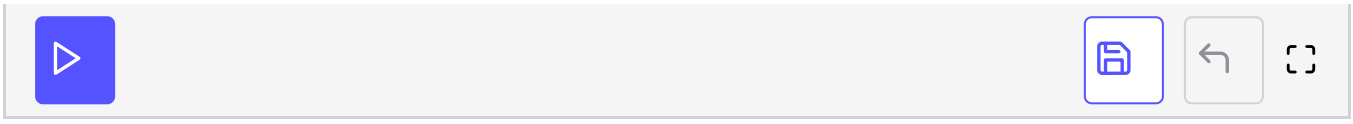
According to en.cppreference.com the macro NULL is an implemenentation-defined null pointer constant. A possible implementation is as follows:

```
#define NULL 0
//since C++11
#define NULL nullptr
```

But that will not apply to our platform. `NULL` seems to be of the type `long int`. We will return to this point later. The usage of the macro `NULL` raises some questions.

```
// nullMacro.cpp

#include <iostream>
#include <typeinfo>

std::string overloadTest(int){
  return "int";
}

std::string overloadTest(long int){
  return "long int";
}


int main(){

  std::cout << std::endl;

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

```
  std::cout << "overloadTest(NULL) = " << overloadTest(NULL) << std::endl;

  std::cout << std::endl;

}
```



The compiler complains about the implicit conversion to `int` in line 19. The warning in line 21 is confusing. The compiler automatically deduces the type of the variable `c` to `long int`. At the same time, it complains that the expression `NULL` must be converted. Our observation is in accordance with the call `overloadTest(NULL)` in line 26. The compiler uses the version for the type `long int` (line 10). If the implementation uses `NULL` of type `int`, the compiler will choose `overloadTest` for the parameter type `int` (line 6), which is according to the C++ standard.

Now, we want to know the current type of the null pointer constant `NULL`. Therefore, we will comment out lines 22 and 23 of the program.

```
// nullMacro.cpp

#include <iostream>
#include <typeinfo>

std::string overloadTest(int){
  return "int";
}

std::string overloadTest(long int){
  return "long int";
}


int main(){

  std::cout << std::endl;

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


  std::cout << "overloadTest(NULL) = " << overloadTest(NULL) << std::endl;

  std::cout << std::endl;

}
```

One one hand, `NULL` seems to be of type `long int` but on the other hand, it seems to be a constant pointer. This behavior shows the compilation of the above program.

So, we can conclude the following: **Don't use the macro NULL.**

The new null pointer constant `nullptr` can solve this problem.

`nullptr` - The Null Pointer Constant

The new null pointer `nullptr` cleans up in C++, with the ambiguity of the number 0 and the macro NULL. `nullptr` is and remains of type `std::nullptr_t`.

We can assign arbitrary pointers to a `nullptr`. The pointer becomes a null pointer and points to no data. We cannot dereference a `nullptr`. However, pointers of this type can be compared with all pointers and be converted to all pointers. This also holds true for pointers to class members, but we cannot compare and 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`, meaning that we can use a `nullptr` in a logical expression.

Generic Code

In the generic code, the literal `0` and `NULL` reveal their true nature. Due to template argument deduction, both literals are integral types in the function template. There is no indication that both literals were null pointer constants.

```
// generic.cpp

#include <cstddef>
#include <iostream>

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

int main(){
  int* a = 0;
```

```
  int* b = NULL;
  int* c = nullptr;

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

We can use `0` and `NULL` to initialize the `int` pointer in line 12 and 13. However, if we use the values `0` and `NULL` as arguments of the function template, the compiler will loudly complain. The compiler deduces `0` in the function template to type `int`, and it deduces `NULL` to the type `long int`. These observations will not hold true for the `nullptr`,however.

The example in the next lesson will further your understanding of this concept.