Because `std::nullptr_t` is its own distinct type, overloading on it is possible:

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

void g(void*);           // (1)
void g(int);             // (2)
void g(std::nullptr_t);  // (3)

void f()
{
    char buf[] = "hello";
    g(buf);      // OK, (1) void g(void*)
    g(0);        // OK, (2) void g(int)
    g(nullptr);  // OK, (3) void g(std::nullptr_t)
    g(NULL);     // Error, ambiguous --- ~~(1), ~~(2)~~,~~ or (3)
}
```

Use Cases

Improvement of type safety

In pre-C++11 codebases, using the `NULL` macro was a common way of indicating, mostly to the human reader, that the literal value the macro conveys is intended specifically to represent a *null address* rather than the literal **`int`** value `0`. In the C Standard, the macro `NULL` is defined as an **implementation-defined** integral or **`void`**`*` constant. Unlike C, C++ forbids conversions from **`void`**`*` to arbitrary pointer types and instead, prior to C++11, defined `NULL` as an "integral constant expression rvalue of integer type that evaluates to zero"; any integer literal, e.g., `0`, `0L`, `0U`, or `0LLU`, satisfies this criterion. From a type-safety perspective, its implementation-defined definition, however, makes using `NULL` only marginally better suited than a raw literal `0` to represent a null pointer. It is worth noting that as of C++11, the definition of `NULL` has been expanded to, in theory, permit **`nullptr`** as a conforming definition; as of this writing, however, no major compiler vendors ~~do so~~.[1]

As just one specific illustration of the added type safety provided by **`nullptr`**, imagine that the coding standards of a large software company historically required that values returned via output parameters (as opposed to a **`return`** statement) are always returned via pointer to a modifiable object. Functions that return via argument typically do so to reserve the function's return value to communicate status.[2] A function in this codebase might "zero" the output parameter's local pointer variable to indicate and ensure that nothing more is to be written. The function below illustrates three different ways of doing this:

[1] Both GCC and Clang default to `0L` (**`long int`**), while MSVC defaults to `0` (**`int`**). Such definitions are unlikely to change since existing code could cease to compile or possibly silently present altered runtime behavior.

[2] See **lakos96**, section 9.1.11, "Pass Argument by Value, Reference, or Pointer," pp. 621–628, specifically the *Guideline* at the bottom of p. 623: "Be consistent about returning values through arguments (e.g., avoid declaring non**`const`** reference parameters)."