```
int illustrativeFunction(int* x)   // pointer to modifiable integer
{
    // ...
    if (/*...*/)
    {
        x = 0;       // OK, set pointer x to null address.
        x = NULL;    // OK, set pointer x to null address.
        x = nullptr; // Bug, set pointer x to null address.
    }
    // ...
    return 0;    // success
}
```

Now suppose that the function signature is changed (e.g., due to a change in coding standards in the organization) to accept a reference instead of a pointer:

```
int illustrativeFunction(int& x)  // reference to modifiable integer
{
    // ...
    if (/*...*/)
    {
        x = 0;       // OK, always compiles; makes what x refers to 0
        x = NULL;    // OK, implementation-defined; might warn
        x = nullptr; // Error, always a compile-time error
    }
    // ...
    return 0;    // SUCCESS
}
```

As the example above demonstrates, how we represent the notion of a null address matters.

1. `0` — Portable across all implementations but minimal type safety
2. `NULL` — Implemented as a macro; added type safety, if any, is platform specific
3. **`nullptr`** — Portable across all implementations and fully type-safe

Using **`nullptr`** instead of `0` or `NULL` to denote a null address maximizes type safety and readability, while avoiding both macros and implementation-defined behavior.

Disambiguation of `(int)0` vs. `(T*)0` during overload resolution

The platform-dependent nature of `NULL` presents additional challenges when used to call a function whose overloads differ only in accepting a pointer or an integral type as the same positional argument, which might be the case, e.g., in a poorly designed third-party library:

```
void uglyLibraryFunction(int* p);  // (1)
void uglyLibraryFunction(int  i);  // (2)
```