As a concrete example, consider a `ConnectionHandle` class that can be in either a *valid* or *invalid* state. For the user's convenience and consistency with other proxy types, e.g., raw pointers, that have a similar *invalid* state, representing the invalid or null state via an explicit conversion to **`bool`** might be desirable:

```
#include <cstddef>  // std::size_t
#include <iostream> // std::cerr
struct ConnectionHandle
{
    std::size_t maxThroughput() const;
        // Return the maximum throughput (in bytes) of the connection.

    explicit operator bool() const;
        // Return true if the handle is valid and false otherwise.
};
```

Instances of `ConnectionHandle` will convert to **`bool`** only where one might reasonably want them to do so, say, as the predicate of an **`if`** statement:

```
int ping(const ConnectionHandle& handle)
{
    if (handle)  // OK, contextual conversion to bool
    {
        // ...
        return 0;  // success
    }

    std::cerr << "Invalid connection handle.\n";
    return -1;  // failure
}
```

Having an **`explicit`** conversion operator prevents unwanted conversions to **`bool`** that might otherwise happen inadvertently:

```
bool hasEnoughThroughput(const ConnectionHandle& ingress,
                         const ConnectionHandle& egress)
{
    return ingress.~~t~~hroughput() <= egress;  // Error, thankfully
//                                     ^~~~~~
}
```

In the example above, the programmer mistakenly wrote `egress` instead of `egress.maxThroughput()` after `<=`, the relational operator. Fortunately, the conversion operator of `ConnectionHandle` was declared to be **`explicit`**, and a compile-time error ensued; if the conversion had been *implicit*, ~~the example code above would have compiled,~~ and, if executed, the above faulty implementation of the `hasEnoughThroughput` function would have silently exhibited well-defined but incorrect behavior.