```
// file1.cpp:
#include <common.h>

const int *f1() { return &c; }

// file2.cpp:
#include <common.h>

const int *f2() { return &c; }

// main.cpp:
#include <common.h>
#include <cassert>  // standard C assert macro

int main()
{
    assert( f1() != f2() );    // different addresses in memory per TU
    assert( *f1() == *f2() );  // same value
    return 0;
}
```

For **`static`** data members, however, things become more difficult. While the declaration in the class definition needs to have an initializer, that is not itself a definition and will not result in static storage being allocated at run time for the object, ending in a link-time error when we try to build an application that tries to reference the **`static`** data member[7]:

```
struct S {
    static constexpr int d_i = 17;
};
void useByReference(const int& i) { /*...*/ }

int main()
{
    const int local = S::d_i;  // OK, value is only used at compile time.
    useByReference(S::d_i);    // Link-Time Error, S::d_i not defined
    return 0;
}
```

This link-time error would be averted by adding a definition of `S::d_i`. Note that the initializer needs to be omitted, as it has already been specified in the definition of `S`:

```
constexpr int S::d_i;  // Define constexpr data member.
```

[7]The C++17 change to make **`constexpr`** variables **`inline`** also applies to **`static`** data members, removing the need to provide external definitions when they are used.