**assignment**, the copy and move constructor, and the destructor are all guaranteed to be trivial but not necessarily usable, and only the destructor and one of the copy operations is declared and not deleted:

```
#include <cassert>  // standard C assert macro
#include <new>      // placement new
#include <cstring>  // std::memcpy

void copy1a()
{
    int a = 1, b = 2;
    a = b;                          // assignment
    assert(2 == a);
}

void copy1b()
{
    int a = 1, b = 2;
    new(&a) int(b);                 // copy construction using placement new
    assert(2 == a);
}

void copy1c()
{
    int a = 1, b = 2;
    std::memcpy(&a, &b, sizeof b);  // bitwise copy
    assert(2 == a);
}
```

All three of the functions above produce well-defined results that are indistinguishable from one another.

Let's now consider a **struct** (e.g., S) that is of trivial type and yet contains a *non*static ~~const~~ data member (e.g., **const int** i) and another **struct** (e.g., B) that is of trivial type and yet contains a *non*static data member of reference type (e.g., **int&** r). In both cases, the implicitly declared default constructor, copy-assignment operator, and move-assignment operator are deleted:

```
#include <type_traits>  // std::is_trivially_copyable
                        // std::is_trivially_copy_constructible (etc.)

struct S  // S is trivial yet neither default constructible nor assignable.
{
    const int i;  // const member i must be initialized at construction.
};

static_assert( std::is_trivial<S>::value, "");                          // OK
static_assert(!std::is_trivially_default_constructible<S>::value, "");  // OK
static_assert(!std::is_default_constructible<S>::value, "");            // OK
```