```
            char  c0 =  cBuf[0];  // OK, special case, char member of S2
            char  c1 =  cBuf[1];  // Bug? -- platform dependent (might be UB)
            char  c2 =  cBuf[2];  //  "           "       "        "    "  "
            char  c3 =  cBuf[3];  //  "           "       "        "    "  "

         assert('A' == ucBuf[0]); // OK, corresponds to s.c
              int i1 = ucBuf[1];  // Bug (UB), convert from indeterminate value.
              int i3 = ucBuf[2];  // OK, convert from value-representing byte.
                     ++ucBuf[3];  // OK, increment value-representing byte.

         assert('A' ==  cBuf[0]); // OK, special case: corresponds to s.c
              int i2 =  cBuf[1];  // Bug (UB), convert from indeterminate value.
              int i4 =  cBuf[2];  // Bug? -- platform dependent (might be UB)
                      ++cBuf[3];  // Bug? -- platform dependent (might be UB)
}
```

While it is always permissible to read *any* value-representing byte from an array of **unsigned char**, in the special case (e.g., `cBuf[0]` above) where the byte being read corresponds to an initialized **char** or **signed char** from the original object `s`, the initialized byte can be read reliably from an array of **char**, even if **char** is signed on the platform.

Note that we were able to reliably access the copy of `s.c` in both `ucBuf` and `cBuf` because the original initialized object (1) was of standard-layout type and (2) had a **char** as its first non**static** data member, which always has offset 0. Had that member not been first, we could have instead employed the `offsetof` macro to learn, in a portable way, its precise location within the array — one of the few unambiguously well-defined uses of `offsetof`; see *Aggressive use of* `offsetof` on page 520. In no event, however, are we permitted to "read" or operate on a byte corresponding to padding bytes as those are always of indeterminate value. What's more, if the original object was not of standard-layout type, none of the fields could be portably accessed through the byte array, although `offsetof` would work for many types and platforms.

Finally, now that we understand the special privileges afforded only to unsigned ordinary character types, let's explore the pitfalls that await the programmer who abuses this information in a misguided attempt to optimize copying of objects. For example, the `myMemCpy` function (below) provides a valid, albeit suboptimal, alternative implementation satisfying the functional requirements of `std::memcpy` (but not `std::memmove`):

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

void* myMemCpy(void* dstPtr, const void* srcPtr, std::size_t numBytes)
{
    unsigned char*       dp = reinterpret_cast<unsigned char*>(dstPtr);
    const unsigned char* sp = reinterpret_cast<const unsigned char*>(srcPtr);
```