```
    for (; numBytes; --numBytes)
    {
        *dp++ = *sp++;
    }
}
```

We have deliberately chosen to use **`unsigned char`** as it is the only ordinary character type that is guaranteed to represent a unique valid value for every possible bit pattern on every conforming platform.

Misuse of `unions`

It is a common misconception that it is *ever* well-defined behavior to write to a scalar member of a **`union`** and then read from another member of that union, sometimes referred to as a *union cast* or *type punning*, even when the two members are of identical type:

```
union U0  // Writing to a and then reading from b has undefined behavior.
{
    int a;  // scalar element of type int
    int b;  //    "      "     "   "    "
};
```

A motivation for this form of misuse would be to write a function that determines endianness:

```
union U1
{
    int           a;
    unsigned char b[sizeof a];
} const u1 = { 1 };

bool isBigEndian1() { return 0 == u1.b[0]; }  // Bug, type punning has UB.
```

A proper portable implementation can be achieved using, e.g., the Posix `htonl` function[39]:

```
#include <arpa/inet.h>  // htonl

bool isBigEndian2()
{
    return htonl(1) == 1;  // OK
}
```

No *reinterpretation* of the bit representation of a scalar value via access to parallel members of a **`union`** is *ever* well-defined behavior in C++. There are, however, other ways to accomplish this specific task natively; see *Abuse of* ***`reinterpret_cast`*** on page 506.

[39] As of C++20, we can query the `big` member of the standard **`enum`** `std::endian`, defined in the standard header `<bit>`, to resolve this question portably at compile time.