Potential Pitfalls

Sometimes implicit conversion *is* indicated

Implicit conversions to and from common arithmetic types, especially **`int`**, are generally ill advised given the likelihood of accidental misuse. However, for proxy types that are intended to be drop-in replacements for the types they represent, implicit conversions are precisely what we want. Consider, for example, a `NibbleConstReference` proxy type that represents the 4-bit integer elements of a `PackedNibbleVector`:

```
class NibbleConstReference
{
    // ...
public:
    operator int() const; // implicit

    // ...
};

class PackedNibbleVector
{
    // ...
public:
    bool empty() const;
    NibbleConstReference operator[](int index) const;

    // ...
};
```

The `NibbleConstReference` proxy is intended to interoperate well with other integral types in various expressions, and making its conversion operator **`explicit`** hinders its intended use as a drop-in replacement by requiring an explicit conversion, a.k.a. cast:

```
int firstOrZero(const PackedNibbleVector& values)
{
    return values.empty()
        ? 0
        : values[0]; // compiles only if conversion operator is implicit
}
```

Sometimes a named function is better

Other kinds of overuse of even *explicit* conversion operators exist. As happens with any user-defined operator, when the operation being implemented is not somehow either canonical or ubiquitously idiomatic for that operator, expressing that operation by a named, i.e., nonoperator, function is often better. Recall from *Description* on page 61 that we used a conversion operator of class `Point` to represent the distance from the origin. This example