serves to illustrate how conversion operators might be both ambiguous and insufficient at the same time. Consider that (1) many mathematical operations on a 2-D integral point might return a **`double`** (e.g., `magnitude`, `angle`) and (2) we might want to represent the same information but in different units (e.g., `angleInDegrees`, `angleInRadians`). Another valid design decision would be to return an object of user-defined type, say, `Angle`, that captures the amplitude and provides named accessors to the different units (e.g., `asDegrees`, `asRadians`).

Rather than employing any conversion *operator* (**`explicit`** or otherwise), consider instead providing a named function, which (1) ~~is automatically~~ ~~**`explicit`**~~ and (2) affords both flexibility in writing and clarity in reading for a variety of domain-specific functions — now and in the future — that might well have had overlapping return types:

```
class Point  // only explicitly convertible and from only an int
{
    int d_x, d_y;

public:
    explicit Point(int x = 0, int y = 0);  // explicit converting constructor
    // ...
    double magnitude() const;  // Return distance from origin as a double.
};
```

Note that defining **nonprimitive functionality**, like `magnitude`, in a separate *utility* at a higher level in the physical hierarchy, e.g., `PointUtil::magnitude(const Point& p)`, might be better still.[3]

[3]For more on separating out nonprimitive functionality, see **lakos20**, section 3.2.7, "Not Just Minimal, Primitive: The Utility `struct`," through section 3.2.8, "Concluding Example: An Encapsulating Polygon Interface," pp. 529–552.