Explicit Conversion Operators

Ensure that a user-defined type is convertible to another type only in contexts where the conversion is made obvious in the code.

Description

Though sometimes desirable, implicit conversions achieved via user-defined *conversion ~~functions~~* — either **converting constructors** accepting a single argument or **conversion operators** — can also be problematic, especially when the conversion involves a commonly used type (e.g., **int** or **double**):

```
class Point  // implicitly convertible from an int or to a double
{
    int d_x, d_y;

public:
    Point(int x = 0, int y = 0);  // default, conversion, and value constructor
    // ...
    operator double() const;  // Return distance from origin as a double.
};
```

Using a conversion operator to calculate distance from the origin in this unrealistically simple `Point` example is for didactic purposes only. In practice, we would typically use a named function for this purpose; see *Potential Pitfalls — Sometimes a named function is better* on page 66.

As ever, calling a function that takes a `Point` but accidentally passing an **int** can lead to surprises:

```
void g0(Point p);         // arbitrary function taking a Point object by value
void g1(const Point& p);  // arbitrary function taking a Point by const reference

void f1(int i)
{
    g0(i);  // oops, called g0 with Point(i, 0) by mistake
    g1(i);  // oops, called g1 with Point(i, 0) by mistake
}
```

This problem could have been solved even in C++03 by declaring the constructor to be **explicit**:

```
explicit Point(int x = 0, int y = 0);  // explicit converting constructor
```