Overloading Support Operators with User-Defined Types

Overloading also extends to operators:
- **most operators can be redefined** in ways
- specific to the types of their operands.

In fact,
- **"natural" use of operators** with user-defined types
- **was the original goal of overloading**.
- Overloading is necessary to make operator redefinition useful.

Example of Code Simplification with Operator Overloading

The **canonical motivating example** for operator overloading is **complex numbers**.

Consider this task:
- given complex numbers **P** and **Q**,
- calculate **R = P^2 + Q^2**.

Looks nice in math notation, right? Here's **C**:

```
R = complex_add
      (complex_multiply (P, P),
       complex_multiply (Q, Q));
```

And here's **C++**:

```
R = P * P + Q * Q;
```


Overloading Integrates Naturally with Auto-Conversion

Redefining operators should also
- fit in with the "natural" conversions
- among `int`, `float`, `double`, and so forth.

For example, one **should not have to define all of these multiplication operators separately**:
- `complex * int`
- `complex * double`
- `int * complex`
- `double * complex`
- and so on…

C++ Combines Implicit Casts with Symmetric Definitions

To simplify the definitions, **C++**
- **allows creation of new implicit casts** (auto-conversions)
- and **uses `friend` functions** for symmetry.

A **`friend` function** is
- a function **outside of a class** (neither a member function nor a class function)
- with **full access rights to the class**
- (which, of course, **must appear** as a **`friend` in the class definition**).

