Chapter 10: Operator Overloading and Type Safety

Why overload operators? The goal is to improve readability and create more expressive code. Operator overloading allows your custom types to integrate with familiar C++ syntax, making them feel like first-class citizens of the language. An expression like `total = price1 + price2;` is far more natural and intuitive than `total = price1.add(price2);`.

10.1 Overloading Deep Dive

You can overload most C++ operators. The few that **cannot** be overloaded are `.` (member access), `.*` (member pointer access), `::` (scope resolution), `sizeof`, and `?:` (the ternary operator).

Member vs. Non-Member Functions: A Key Design Choice

This is a critical design decision when overloading operators.

- **Overload as a Member Function when:**
  - The operator modifies the state of the object. This applies to all compound assignment operators (`+=`, `-=`, `*=`, etc.).
  - The operator requires access to `private` or `protected` members (though a `friend` function is also an option).
  - The operator is one of `()`, `[]`, `->`, or any assignment operator (`=`). The language requires these to be members.
- **Overload as a Non-Member (Free) Function when:**
  - The operator is a symmetric binary operator (like `+`, `-`, `*`, `==`, `<`).
  - **Why?** This allows for type conversions on **both** the left-hand and right-hand operands. If `operator+` for a `Money` class were a member function, `money_object + 5` would work, but `5 + money_object` would fail because the `int` type has no member function for adding a `Money` object. A non-member function can handle both cases.

Canonical Implementation: `+` in terms of `+=` The standard, most robust pattern is to implement the logic in the compound assignment operator (`+=`) as a member function, and then implement the binary operator (`+`) as a non-member function that calls `+=`.

```cpp
class Money {
public:
    // Member function: modifies the object's state
    Money& operator+=(const Money& rhs) {
        this->cents_ += rhs.cents_;
        return *this;
    }
private:
    long long cents_;
};

// Non-member function: provides symmetry
// Takes lhs by value to create a temporary copy to modify.
inline Money operator+(Money lhs, const Money& rhs) {
    lhs += rhs; // Re-use the logic from the member function
    return lhs; // Return the modified copy
}
```

10.2 User-Defined Conversions

Why have them? Conversions allow your type to be used where another type is expected. This can be convenient, but it comes at a high cost: the compiler can perform conversions you never intended, leading to ambiguous calls and subtle, hard-to-find bugs.

There are two ways to define an implicit conversion:

1. **Single-Argument Constructors (Converting Constructors):** A constructor that can be called with a single argument defines a conversion *from* the argument's type *to* your class's type.
2. **Conversion Operators:** A special member function `operator Type()` defines a conversion *from* your class's type *to* `Type`.

The Dangers of Implicit Conversions & The `explicit` Solution

Why is `explicit` so important? It is the tool C++ gives you to prevent unintended conversions. It allows a conversion to exist but requires the programmer to explicitly ask for it (e.g., with a `static_cast`), making the intent clear.

Case 1: `explicit` Constructors

```cpp
class Buffer {
public:
    // Without explicit, `Buffer b = 1024;` would compile, which is weird.
    // Does it mean a buffer containing the number 1024, or a buffer of size 1024?
    // It's ambiguous and potentially misleading.
    explicit Buffer(size_t size) : size_(size) { /* allocate memory */ }
private:
    size_t size_;
};

// Buffer b1 = 1024; // ERROR: constructor is explicit, implicit conversion forbidden.
Buffer b2(1024);   // OK: direct initialization is explicit and clear.
```

Case 2: `explicit` Conversion Operators

A classic example is `operator bool()`. If not `explicit`, it can lead to surprising behavior, like allowing your object to be added to an `int` (because the object converts to `bool`, which promotes to `int`).

```cpp
class SmartPointer {
public:
    // ...
    // With `explicit`, this can only be used in direct boolean contexts (if, while, for)
    // It prevents `int x = ptr + 5;` from compiling.
    explicit operator bool() const {
        return ptr_ != nullptr;
    }
private:
    Widget* ptr_;
};

SmartPointer ptr(new Widget());

if (ptr) { // OK: `explicit operator bool` is specially allowed in `if` conditions.
    // ...
}

// bool is_valid = ptr; // ERROR: implicit conversion is forbidden.
bool is_valid = static_cast<bool>(ptr); // OK: explicit cast.
```

Rule of Thumb: Always declare your single-argument constructors and conversion operators as `explicit` unless you have a very good, specific reason for wanting implicit conversions.

Projects for Chapter 10

Project 1: The `Money` Class

- **Problem Statement:** Create a `Money` class that stores a monetary value as a single `long long` representing the total number of cents. Implement the following overloaded operators:
  1. `operator+=` and `operator-=` as member functions.
  2. `operator+` and `operator-` as non-member functions that use the compound assignment operators.
  3. `operator==`, `operator!=`, `operator<`, and `operator>` as non-member functions.
  4. `operator<<` for `std::ostream` to print the `Money` object in a standard format (e.g., `$123.45`).
- **Core Concepts to Apply:** Operator overloading (arithmetic, comparison, stream insertion), member vs. non-member functions, canonical implementation patterns.

Project 2: The `explicit SafeFlag` Wrapper

- **Problem Statement:** Create a class `SafeFlag` that wraps a `bool`. Its constructor from a `bool` must be `explicit`. It should also have an `explicit operator bool() const`. In `main`, write code that demonstrates the following:
  1. You cannot implicitly convert a `bool` to a `SafeFlag` (`SafeFlag f = true;` should fail).
  2. You *can* explicitly construct it (`SafeFlag f(true);`).
  3. You *can* use it directly in an `if` statement (`if (f)`).
  4. You *cannot* implicitly convert it to a `bool` for assignment (`bool b = f;` should fail).
  5. You *can* explicitly convert it to a `bool` (`bool b = static_cast<bool>(f);`).
- **Core Concepts to Apply:** `explicit` constructors, `explicit` conversion operators, context-specific conversions.

Project 3: The `GradeMap` Subscript Operator

- **Problem Statement:** Create a `GradeMap` class that stores student grades. It should use a `std::map<std::string, int>` to map student names to their grades. Overload the subscript operator (`operator[]`) to provide access to the grades. You should provide two versions:
  1. A non-`const` version (`int& operator[](const std::string& name)`) that allows both reading and writing grades. This version should allow adding new students to the map.
  2. A `const` version (`int operator[](const std::string& name) const`) for read-only access. This version should return `0` or throw an exception if the student is not found, but it must not modify the map.
- **Core Concepts to Apply:** Subscript operator overloading, `const` and non-`const` overloads for read/write vs. read-only access, `std::map`.