Return Types of Commonly Overload Operators in C++

Overloading operators in C++ gives you the ability to modify the behavior of certain operations (like `+`, `-`, `*`, etc.) on user-defined types (like classes and structs). When you overload these operators, you can choose to return values by value or by reference. Here's a simplified explanation of when to use each.

Returning by Value

In most cases, you'll want to return by value, especially when the result of the operation is a new object that doesn't exist before the operation. This is often the case with arithmetic operations, where the result is a new value.

For example, let's say you're overloading the `+` operator for a `Vector` class:

```
class Vector {
  //...
  Vector operator+(const Vector& rhs) const {
    Vector result;
    //... calculate result ...
    return result;
  }
};
```

Here, the result of `v1 + v2` is a new `Vector` that doesn't have a corresponding object before the `+` operation. Hence, it makes sense to return by value.

Another example is overloading the + operator in a `Complex` number class.Complex numbers are numbers that consist of a real and an imaginary part.

Here's how you might define the class and overload the `+` operator:

```
class Complex {
public:
  Complex(double real, double imag) : real(real), imag(imag) {}

  double getReal() const { return real; }
  double getImag() const { return imag; }

  // Overload the '+' operator
  Complex operator+(const Complex& other) const {
    return Complex(real + other.real, imag + other.imag);
  }

private:
  double real, imag;
};
```

```
int main() {
  Complex c1(1.0, 2.0);  // represents the complex number 1 + 2i
  Complex c2(2.0, 3.0);  // represents the complex number 2 + 3i

  Complex c3 = c1 + c2;  // uses the overloaded '+' operator
  // c3 now represents the complex number 3 + 5i
}
```

Note that it makes sense to return by value since we are returning a new Complex object.

Returning by Reference

Returning by reference is usually done when the object being returned already exists, and you want to avoid the overhead of a copy. This is common with assignment operators and stream insertion/extraction operators.

For instance, consider overloading the = operator for the same `Vector` class:

```
class Vector {
  //...
  Vector& operator=(const Vector& rhs) {
    if (this != &rhs) {
      //... assign values ...
    }
    return *this;
  }
};
```

Here, = is an assignment operator, and it modifies the existing object and returns it. It's common to return by reference in this case to avoid unnecessary copying. Note that it returns a reference to `*this`, i.e., the object it was called on.

In addition, for operators that are usually chained like = and <<, returning by reference allows the chaining to work correctly.

```
Vector v1, v2, v3;
v1 = v2 = v3;  // This works because operator= returns Vector&.

std::cout << v1 << v2 << v3; // This works because operator<< returns
std::ostream&.
```

Remember that returning by reference when you shouldn't can lead to dangling references if the object being referred to is destroyed or falls out of scope, which can lead to undefined behavior. Always ensure the lifetime of the object you're returning by reference extends beyond the scope of the function call.

Here's another example. Let's say we have a `Person` class and we want to be able to print out instances of this class using `std::cout` with the << operator. In this case, we would return as reference to the output stream.

```
#include <iostream>
#include <string>

class Person {
public:
  Person(const std::string& name, int age) : name(name), age(age) {}

  const std::string& getName() const { return name; }
  int getAge() const { return age; }

private:
  std::string name;
  int age;
};

std::ostream& operator<<(std::ostream& os, const Person& person) {
  os << "Name: " << person.getName() << ", Age: " << person.getAge();
  return os;
}

int main() {
  Person john("John Doe", 30);
  std::cout << john << std::endl;
}
```

Here, the << operator is overloaded to handle `Person` objects. It prints the person's name and age to the `std::ostream` object (which might be `std::cout`, or a `std::ofstream` for outputting to a file, etc.).

This function returns `os` by reference, so why does it make sense to return by reference here?

1. **Chaining**: Returning by reference allows for chaining of the << operator. This is why you can write statements like `std::cout << "Hello, " << name << "!" << std::endl;` in C++. Each << operation returns the stream, allowing the next << to operate on it.
2. **Efficiency**: Returning by reference avoids unnecessary copying. `std::ostream` objects can potentially be quite large, so copying them could be expensive.
3. **It doesn't make sense to return a copy**: The << operator modifies the state of the stream (by writing to it), and it doesn't make much sense to return a new stream object, because we generally want further operations to affect the same stream.

Returning by reference is a common practice when overloading the << operator and >> operator for custom types in C++. The returned reference must be to an object that continues to exist after the function call, so it's typical to return the stream object passed into the function, as we're doing here.

Here's a table showing common C++ operators and their canonical return types. Note that the `Type` refers to the type of object for which the operator is being overloaded. You should try to preserve the semantics of the operator as defined by C++ when you overload it yourself.

Operator	**Return Type**	**Note**
`+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `<<`, `>>`, `==`, `!=`, `<`, `>`, `<=`, `>=`	`Type` by value	Most arithmetic and comparison operators create new values. Note that << and >> in this case are the binary shift operators, not the stream operators.
`+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<=`, `>>=`	`Type&` (reference to `Type`)	These modify the existing object and should return a reference to allow for chaining.
`=`, `[]`, `()`	`Type&` (reference to `Type`)	The assignment, subscript, and function call operators modify existing objects.
`++` (prefix), `--` (prefix)	`Type&` (reference to `Type`)	Prefix increment/decrement modify the object and return it.
`++` (postfix), `--` (postfix)	`Type` by value	Postfix increment/decrement return a value representing the object before modification.
`<<`, `>>` (stream operators)	`std::ostream&` or `std::istream&`	Stream operators should return a reference to the stream to allow for chaining.
`new`, `delete`, `new[]`, `delete[]`	`void*` and `void`, respectively	These operators are used for object allocation and deallocation.

Remember that these are common practices and not hard rules. There might be valid reasons to return by value when normally you'd return by reference and vice versa. However, it's best to stick with these conventions unless you have a good reason to deviate, as they follow the principle of least surprise and make your code easier to understand and use correctly.