Reference: Big C++.

Exercises 14:

Exercise R14.1.

At least one argument be a user-defined data type.

Exercise R14.2.

When using an operator will make the code more readable and understandable. And I will use a function if the operator will make the program not understandable or ambiguous.

Exercise R14.3.

Employee operator+(const Employee& e, int percent)

{

return Employee(e.get\_name(), e.get\_salary() \* (percent / 100));

}

No this is not a good practice, since it’s mysterious and not clear that this int refers to percent, also it’s not easy to remember, a function raise\_salary(int percent) will be good.

Exercise R14.4.

Time operator-(const Time& t)

{

Time result = Time(t.get\_hours() - 1,t.get\_minutes(), t.get\_seconds());

return result;

}

This is not a good practice at all.

Exercise R14.5.

When the operator function requires access to private fields and they are not easily accessible, when the left arguments is to be modified, and in case of we don’t want the type conversion to be applied to the left argument.

Exercise R14.6.

accepted

double pi = (2,14159);

double pi\_two = 3,14159;

First expression to the left of comma gets evaluated, then the one to its right. The result of the right most expression is stored in the variable to the left of = sign.

Exercise R14.7.

When overloaded as a member function, a << b is interpreted as a.operator<<(b), so it only takes one explicit parameter (with this as a hidden parameter).

Since this requires that the overload be part of the class used as the left-hand operand, it's not useful with normal ostreams and such. It would require that your overload be part of the ostreamclass, not part of your class. Since you're not allowed to modify ostream, you can't do that. That leaves only the global overload as an alternative.

There is, however, a fairly widely-used pattern where you overload the operator globally, but have that call a member function:

class whatever {

// make this public, or the global overload a friend.

std::ostream &write(std::ostream &dest) const {

// write self to dest

}

};

std::ostream &operator<<(std::ostream &os, whatever const &w) {

return w.write(os);

}

This is particularly useful when/if you want polymorphic behavior. You can't make the overloaded operator polymorphic itself, but you make the member function it calls virtual, so it acts polymorphic anyway.

Edit: to (I hope) clarify the situation, you can do this a few different ways. The first and probably most obvious is to just make our write member public, and have the global operator call it. Since it is public, we don't have to do anything special to let the operator use it:

class myClass {

public:

std::ostream &write(std::ostream &os) const {

// write stuff to stream

return os;

}

};

std::ostream &operator<<(std::ostream &os, myClas const &m) {

// since `write` is public, we can call it without any problem.

return m.write(os);

}

A second alternative is to make write private, and declare operator<< a friend to give it access:

class myClass {

// Note this is private:

std::ostream &write(std::ostream &os) const {

// write stuff to stream

return os;

}

// since `write` is private, we declare `operator<<` a friend to give it access:

friend std::ostream &operator<<(std::ostream &, myClass const &);

};

std::ostream &operator<<(std::ostream &os, myClas const &m) {

return m.write(os);

}

There's a third possibility that's almost like the second:

class myClass {

// Note this is private:

std::ostream &write(std::ostream &os) const {

// write stuff to stream

return os;

}

// since `write` is private, we declare `operator<<` a friend to give it access.

// We also implement it right here inside the class definition though:

friend std::ostream &operator<<(std::ostream &os, myClas const &m) {

return m.write(os);

}

};

This third case uses a rather strange (and little known) rule in C++ called "name injection". The compiler knows that a friend function can't be part of the class, so instead of defining a member function, this "injects" the name of that function into the surrounding scope (the global scope, in this case). Even though operator<< is defined inside the class definition, it's not a member function at all -- it's a global function.

Exercise R14.9.

Don't worry about performance, say 97% of the time. Premature Optimization is the Root of all Evil.

-- Donald Knuth

Now that this is out of our way, let's make our choice *sanely*:

* ++i: *prefix increment*, increment the current value and yields the result
* i++: *postfix increment*, copy the value, increment the current value, yields the copy

Unless a copy of the old value is required, using *postfix increment* is a round-about way of getting things done.

Exercise R14.10.

i = 7;

j = ++i + i++; // ++i yields 8 and i = 8, i++ yields 8 and i = 9. Then k = 8 + 8 = 16

std::cout << j << "\n";

k = j++ + ++j; // j++ yields 16, j = 17 , ++j yields 18 and j = 18. Then k = 17 + 18 = 16

Exercise R14.11.

++ & -- operators.

Exercise R14.12.

It will not be used by the objects outside the class.

Exercise R14.13.

When a class contains pointers, assigning pointers in the default way will lead to a disaster.

Exercise R14.14.

By providing a constructor for the fraction that constructs a fraction from a floating-point value.