Operator Overloading

The material in this handout is collected from the following reference:

- Chapter 14 of the text book C++ Primer.
- Various sections of Effective C++.
- Various sections of More Effective C++.

The `StopWatch` class

Class `StopWatch` is a simple class that keeps track of how many seconds have elapsed and is used to demonstrate operator overloading. Consider the definition of the class:

```
1  // in interface file stopwatch.h
2  class StopWatch {
3  public:
4    StopWatch(); // default constructor
5    StopWatch(int seconds); // conversion constructor
6    StopWatch(int hours, int minutes, int seconds); // non-default ctor
7
8    // tell other programmers and compiler that you want default semantics
9    StopWatch(StopWatch const&) = default;
10   StopWatch& operator=(StopWatch const&) = default;
11   ~StopWatch() = default;
12
13   void Increment(int seconds = 1);
14   void Reset();
15   int GetSeconds() const;
16   void Display() const;
17
18 private:
19   int seconds;
20 };
```

The member functions of class `StopWatch` are defined here.

Suppose we have two `StopWatch` objects, and we want to make a third `StopWatch` object that represents the sum of the first two `StopWatch`es:

```
1  StopWatch sw1(30); // 30 seconds
2  StopWatch sw2(10); // 10 seconds
3
4  // sw3 should be 40 seconds but following line will not compile
5  StopWatch sw3 = sw1 + sw2;
```

`g++` will produce this error message:

```
1  error: no match for 'operator+' in 'sw1 + sw2'
```

The code above will not compile because the compiler has no idea how to "add" two `StopWatch` objects. The message doesn't necessarily say it's illegal to add to `StopWatch` objects. It says that it can't find a function [`operator+`] that matches. The "C-style" way is to simply create a function to do the work:

```
StopWatch AddStopWatch(StopWatch const& sw1, StopWatch const& sw2) {
  // Construct unnamed StopWatch from parameters and
  // return result by value (a copy)
  return StopWatch(sw1.GetSeconds() + sw2.GetSeconds());
}
```

Here's how clients could use function `AddStopWatch`:

```
StopWatch sw1(30); // 30 seconds
StopWatch sw2(10); // 10 seconds
StopWatch sw3 = AddStopWatch(sw1, sw2);
sw3.Display(); // prints 00:00:40
```

This isn't all that bad for adding two objects, but it doesn't scale well:

```
StopWatch sw1(30); // 30 seconds
StopWatch sw2(10); // 10 seconds
StopWatch sw3(60); // 60 seconds
StopWatch sw4(20); // 20 seconds

// We have to do this (and it will get worse with more)
StopWatch tempsw1 = AddStopWatch(sw1, sw2);
StopWatch tempsw2 = AddStopWatch(tempsw1, sw3);
StopWatch sw5 = AddStopWatch(tempsw2, sw4);
sw5.Display(); // 00:02:00

// We could've "chained" the calls to save some typing:
StopWatch sw6 =
  AddStopWatch(AddStopWatch(AddStopWatch(sw1, sw2), sw3), sw4);
sw6.Display(); // 00:02:00
```

What we really just want to be able to do is this:

```
StopWatch sw5 = sw1 + sw2 + sw3 + sw4;
```

The solution is to *instruct* or *teach* the compiler how to add `StopWatch`es by overloading operator `+` for `StopWatch` objects.

Overloading binary `+` operator

To overload operator `+` [or most any operator for that matter], we use an *operator function*, which has the form:

```
return-type operator op(argument-list)
```

where:

- `return-type` is the type of the function's returned value, just as with other functions.

- `operator` is a reserved word
- `op` is the operator to overload (such as `+`, `*`, `-`, and so on).
- `argument-list` is the list of arguments to the operator function.

We simply need to create a non-member function called `operator+` that takes two `StopWatch` objects as parameters:

```
StopWatch operator+(StopWatch const& lhs, StopWatch const& rhs) {
    // lhs is left-hand side
    // rhs is right-hand side
  StopWatch sw(lhs.GetSeconds() + rhs.GetSeconds());

    // Return the result by value (a copy)
  return sw;
}
```

This makes life much simpler for clients of class `StopWatch`.

- Notice that the code does exactly same thing as function `AddStopWatch` that was defined earlier. Only the function name has changed, really.
- Now, when the compiler sees operator `+` between two `StopWatch` objects, it knows what to do.
- It calls the function and passes the two `StopWatch`es as operands: left operand first, right operand second).
- Notice that the function is just a "normal" global function.
- Where should the function go? Which source file?

Example usage:

```
StopWatch sw1(30); // 30 seconds
StopWatch sw2(10); // 10 seconds
StopWatch sw3(60); // 60 seconds
StopWatch sw4(20); // 20 seconds

StopWatch sw5 = sw1 + sw2;
StopWatch sw6 = sw1 + sw2 + sw3 + sw4;

sw5.Display(); // 00:00:40
sw6.Display(); // 00:02:00

// Functional notation (this is really what's happening at runtime)
StopWatch sw7 = operator+(sw1, sw2);
sw7.Display(); // 00:00:40
```

In practice, the functional notation is seldom used. Essentially, overloading an operator means writing a function that the compiler will call when it sees the operator being used when at least one argument is a user-defined data type.

Again, notice how similar the functional notation is to our original function:

```
StopWatch sw7 = operator+(sw1, sw2);
StopWatch sw8 = AddStopWatch(sw1, sw2);
```

Overloading more operators for `StopWatch` class

While we're at it, we'll overload some more operators. First, let's overload operator `-` to determine the "difference" between two `StopWatch` objects:

```
StopWatch operator-(StopWatch const& lhs, StopWatch const& rhs) {
  int secs = lhs.GetSeconds() - rhs.GetSeconds(); // don't want -ve
  secs = (secs < 0) ? 0 : secs;
  return StopWatch(secs); // Create a new one and return it
}
```

Test it:

```
StopWatch sw1(1, 30, 0); // 01:30:00
StopWatch sw2(10, 0, 0); // 10:00:00
StopWatch sw3 = sw2 - sw1; // Subtract smaller from larger
sw3.Display(); // 00:08:30
sw3 = sw1 - sw2; // Subtract larger from smaller
sw3.Display(); // 00:00:00
```

Simple. In a nutshell, that's about it. [Like everything in C++, there are lots of little details involved.]

How about something like this:

```
StopWatch sw2 = sw1 * 2; // double the time
```

Of course, we'll get this:

```
error: no match for 'operator*' in 'sw1 * 2'
```

Fair enough. Another trivial function to implement:

```
// Multiply: StopWatch * int
StopWatch operator*(StopWatch const& lhs, int rhs) {
  return StopWatch(lhs.GetSeconds() * rhs);
}
```

And the problem is solved. Or is it?

```
StopWatch sw3 = 2 * sw1; // Double the time?
```

Error message:

```
error: no match for 'operator*' in '2 * sw1'
note: candidates are: StopWatch operator*(const StopWatch&, int)
```

So, we add an overload to `operator*`:

```
// Multiply: int * StopWatch
StopWatch operator*(int lhs, StopWatch const& rhs) {
  return StopWatch(lhs * rhs.GetSeconds());
}
```

Let us make one more observation about the two `operator*` overloads. Except for their return values, they do the same thing: they multiply a `StopWatch` and an `int` value. That is, they're *supposed* to do the same thing. How can you be sure the behavior of `sw*2` is consistent with that of `2*sw`? What guarantee do you have that their implementations won't diverge over time, possibly as a result of different programmers maintaining and enhancing them? At this moment, you've no such guarantee. A design principle to guarantee both functions will give the same result is for `2*sw` to be implemented *in terms* of `sw*2`. You or any other programmer need only maintain the `sw*2` version, because the `2*sw` version will then automatically behave in a consistent fashion:

```
StopWatch operator*(int lhs, StopWatch const& rhs) {
  return rhs * lhs; // implement 2*sw in terms of sw*2
}
```

Conversion constructor and `explicit` keyword

How about adding an integer to a `StopWatch`?

```
StopWatch sw1(60); // 00:01:00
StopWatch sw2 = sw1 + 60; // Add another minute?
```

Let's think about how the language has implemented the binary operator `+` for built-in types. The rule is very simple: both operands must have the same type. Otherwise, one of the operands is promoted or otherwise converted to the second operand's type. For example, we know that `3 + 4` will evaluate to an `int` value `7`; that `3.1 + 4.1` will evaluate to `double` value `7.2`; that `2.4f + 5.1f` evaluates to `float` value `7.5f`. What does the compiler do for expression `4.1 + 3`? The rule is that operand `3` of type `int` is promoted to type `double` with value `3.0`. That is, the compiler will evaluate the expression as `4.1 + (double)3` which is `4.1 + 3.0` with the evaluation resulting in value `7.1` of type `double`.

Using the knowledge of how operator `+` is implemented for built-in types, let's think about how the compiler evaluates expression `sw1 + 60`. The compiler will try to convert `sw1` to an `int` value but it sees no function that can convert a `StopWatch` object into an `int` object. Next, the compiler will try to convert an `int` object to a `StopWatch` object. It finds no function in the global scope that perform the conversion. However, in the definition of `StopWatch` class, it finds this function declared:

```
class StopWatch {
public:
  StopWatch(int seconds);
// other stuff ...
};
```

Constructor `StopWatch(int);` is a special constructor that can convert an `int` value into a `StopWatch`. Such constructors are called *conversion constructors* and are implicitly called by the compiler when it needs to evaluate an expression consisting of `StopWatch` and `int` values.

If you or your clients find such implicit conversions off-putting, then you can tell the compiler to only perform such conversions when *explicitly* asked to by adding the keyword `explicit` in the declaration of `StopWatch`'s constructor:

```
class StopWatch {
public:
  // this prevents the conversion ctor from being used to perform
  // implicit type conversions, though they may still be used for
  // explicit conversions
  // note: use explicit keyword only for declarations
  explicit StopWatch(int seconds);
// other stuff ...
};
```

In this case, the compiler will flag expression `sw1 + 60` as an error. The programmer can explicitly request for the conversion like this:

```
StopWatch sw1(60); // 00:01:00
StopWatch sw2 = sw1 + 60; // error! there is no implicit conversion from
                          // int to StopWatch
StopWatch sw2 = sw1 + StopWatch(60); // fine!!! uses the StopWatch ctor
                                     // explicitly convert (that is, cast)
                                     // the int to a StopWatch just for this
                                     // call.
```

Constructors declared `explicit` are usually preferable to non-`explicit` ones, because they prevent compilers from performing unexpected [and often unintended] type conversions.

> ***Unless you've have a good reason for allowing a constructor to be used for implicit type conversions, declare it*** `explicit`.

Test your knowledge

Given the following overloads for class `StopWatch`:

```
// Multiply: StopWatch * int
StopWatch operator*(StopWatch const& lhs, int rhs) {
  return StopWatch(lhs.GetSeconds() * rhs);
}

// Multiply: int * StopWatch
StopWatch operator*(int lhs, StopWatch const& rhs) {
  return rhs * lhs; // Simply reverse the operands
}
```

explain whether the compiler will flag the following code as error?

```
StopWatch sw1(60);
StopWatch sw2(10);
StopWatch sw3 = sw1 * sw2; // Multiply two StopWatches
```

Pointers vs references

By now it should be very obvious why we are passing references instead of pointers with these overloaded operator functions. If we used pointers, we'd have to use the address-of operator every time we used them. For example, we'd have to do this:

```
StopWatch sw3 = &sw1 * &sw2;
```

which would be *very* undesirable [not to mention, illegal, as you can't multiply pointers].

Overloaded operators as member functions

Up to this point, all of our overloading was done with functions that are not member functions of class `StopWatch`. We could do this because class `StopWatch` has declared a public member function `GetSeconds`:

```
StopWatch operator+(StopWatch const& lhs, StopWatch const& rhs) {
  return StopWatch(lhs.GetSeconds() + rhs.GetSeconds());
}
```

If this method wasn't available, none of this would work. One solution would be to declare additional member functions in class `StopWatch`:

```
class StopWatch {
public:
  // other function from before ...

  // Overloaded operators (member functions). Notice the const at the end.
  StopWatch operator+(StopWatch const& rhs) const;
  StopWatch operator-(StopWatch const& rhs) const;
  StopWatch operator*(int rhs) const;

private:
  int seconds;
};
```

The functions will be defined in implementation file `stopwatch.cpp`:

```
StopWatch StopWatch::operator+(StopWatch const& rhs) const {
  return StopWatch(seconds + rhs.seconds);
}

StopWatch StopWatch::operator-(StopWatch const& rhs) const {
  return StopWatch(seconds - rhs.seconds);
}

StopWatch StopWatch::operator*(int rhs) const {
  return StopWatch(seconds * rhs);
}
```

Notice that the methods are accessing the private data from the `StopWatch` object that was passed in to them. This is allowed because the function is a member of class `StopWatch`.

All of this code works as before:

```
void f1() {
  StopWatch sw1(30);
  StopWatch sw2(10);
  StopWatch sw3 = sw1 + sw2; // StopWatch sw3 = sw1.operator+(sw2);
  StopWatch sw4 = sw1 + 20;  // StopWatch sw4 = sw1.operator+(20);
  StopWatch sw5 = sw1 - sw2; // StopWatch sw5 = sw1.operator-(sw2);
  StopWatch sw6 = sw1 - 20;  // StopWatch sw6 = sw1.operator-(20);
  StopWatch sw7 = sw1 * 10;  // StopWatch sw7 = sw1.operator*(10);

  sw3.Display(); // 00:00:40;
  sw4.Display(); // 00:00:50;
  sw5.Display(); // 00:00:20;
  sw6.Display(); // 00:00:10;
  sw7.Display(); // 00:05:00;
}
```

Notice which object the method is invoked on. This:

```
StopWatch sw3 = sw1 + sw2;
```

is the same as:

```
 // Method is invoked on left object
StopWatch sw3 = sw1.operator+(sw2);
```

and not this:

```
 // This is NOT how it works.
StopWatch sw3 = sw2.operator+(sw1);
```

It is ***not*** random, it is ***not*** arbitrary, it is ***not*** which ever object the compiler chooses. It is ***always*** the left object [with a binary operator]. *Always*.

What about this now?

```
StopWatch sw1(30);

  // What about this?
StopWatch sw2 = 2 + sw1;
```

With the global function, the functional notation looks like this:

```
StopWatch sw2 = operator+(2, sw1);
```

which converts to this [using the *conversion constructor*]:

```
StopWatch sw2 = operator+(StopWatch(2), sw1);
```

But with built-in types, there are no member functions, so it can't even attempt this:

```
StopWatch sw2 = 2.operator+(sw1);
```

This sometimes makes global functions more useful than member functions.

Overloading operator `<<`

Although `StopWatch::Display` member function is nice, we'd like to write characters to the standard output stream the C++ way:

```
StopWatch sw1(60);
sw1.Display();    // ok: works just fine
std::cout << sw1; // error: but we'd like to do this
```

Defining output operator `<<` for a given type is typically trivial. Here is a simple output operator for `StopWatch` that simply prints data similar to `StopWatch::Display`:

```
std::ostream& operator<<(std::ostream& os, StopWatch const& sw) {
  int total_seconds = sw.GetSeconds(); // get seconds
  int hours = total_seconds / 3600; // hrs
  int minutes = (total_seconds - (hours * 3600)) / 60; // mins
  int seconds = total_seconds % 60; // secs

  os.fill('0'); // print them
  os << std::setw(2) << hours << ':';
  os << std::setw(2) << minutes << ':';
  os << std::setw(2) << seconds << "\n";

  return os; // must return the ostream reference
}
```

Given the above definition of operator `<<` for class `StopWatch`, the meaning of

```
std::cout << sw1; // infix notation
```

where `sw1` is a `StopWatch` object is just the call

```
operator<<(std::cout, sw1); // functional notation
```

Note how `operator<<()` takes an `ostream&` as its first argument and returns it again as its return value. That's the way the output stream is passed along so that you can "chain" output operations. For example, we could output two `StopWatch`es like this:

```
StopWatch sw1(60);
StopWatch sw2(100);
std::cout << sw1 << sw2;
```

This will be handled by first resolving the first `<<` and after that the second `<<` [because left-shift operator `<<` is left associative]

```
std::cout << sw1 << sw2; // means operator<<(std::cout, sw1) << sw2;
                         // means operator<<(operator<<(std::cout, sw1),
sw2);
```

That is, first output `sw1` to `std::cout` and then output `sw2` to the output stream that is the result of the first output operation. In fact, we can use any of those three variants to write out `sw1` and `sw2`. We know which one is easier to read, though.

What if member function `StopWatch::GetSeconds` was removed [or made private]? Should method `operator<<` be made a member function of class `StopWatch`? This is what the functional notation looks like for a member function:

```
std::cout.operator<<(sw1); // Functional notation (member function)
```

See the problem? We can't make it a member function of class `StopWatch`. It would need to be a member of class `ostream`, which we cannot modify. Therefore, overload function `operator<<` will have to be implemented as a non-member function of class `StopWatch`.

The `friend` keyword

Since we can't make output operator `operator<<` a member of class `ostream`, we need a way to allow the overloaded output operator function to access private members of class `StopWatch`. We do this by making the overloaded output operator a *friend* of class `StopWatch`. A function that is not a member of a class can be granted access to all members through a `friend` declaration. For example:

```
class StopWatch {
public:
  // other public stuff ...

  // make this function a friend so it can access private members of this class
  friend std::ostream& operator<<(std::ostream& os, const StopWatch& rhs);

private:
  int seconds;
};
```

The definition of `operator<<` would look like this:

```
std::ostream& operator<<(std::ostream& os, StopWatch const& sw){
  // A friend can access private members of sw
  int hours = sw.seconds / 3600;
  int minutes = (sw.seconds - (hours * 3600)) / 60;
  int seconds = sw.seconds % 60;

  os.fill('0');  // Print them
  os << std::setw(2) << hours << ':';
  os << std::setw(2) << minutes << ':';
  os << std::setw(2) << seconds << "\n";

  return os; // Must return a reference to the ostream
}
```

> ***Just like in the Real World, friendship is a one-way street: "I may be your friend, but that doesn't make you my friend." Friendship is granted from one entity to another. With `operator<<`, class `StopWatch` is granting friendship to the operator [the operator is NOT granting friendship to class `StopWatch`]. If you need it to work both ways, then both entities must grant the friendship.***
>
> ***Friendship is an advanced technique and should be limited to `operator<<` until you have a very good understanding of C++ class design. In other words, it should be used as a last resort.***

Automatic conversions and user-defined types

Recall that a single-argument constructor is called a *conversion constructor* because it "converts" its argument into an object of the class type.

```
void f1() {
  StopWatch sw1; // Default constructor
  sw1 = 60; // same as sw1 = (StopWatch)60;
            // same as sw1 = StopWatch(60);
}
```

Another example:

```
void fooSW(const StopWatch& sw) {
  sw.Display(); // Do something with sw
}

StopWatch sw1(60);
fooSW(sw1); // Pass StopWatch
int time = 90;
fooSW(time); // Convert int to StopWatch
```

Both examples *implicitly* convert the integer to a `StopWatch`; implicitly meaning automatically and silently. However, the opposite isn't the same thing:

```
void f1() {
  StopWatch sw1;
  int seconds1 = sw1;       // Error, how do you convert StopWatch to int?
  int seconds2 = (int) sw1; // Error, how do you convert StopWatch to int?
  int seconds3 = int(sw1);  // Error, how do you convert StopWatch to int?
}
```

What to do? Same as always, create **Y**et **A**nother **F**unction. We'll call the function `ToInt` because it will convert a `StopWatch` into an `int`:

```
class StopWatch {
public:
  // other public methods from before ...
  int ToInt() const; // convert to int (explicit)

  private:
    int seconds;
};
```

The function is defined outside the class definition like this [notice that the function definition must specify the `const` keyword]:

```
int StopWatch::ToInt() const {
  return seconds;
}
```

You use member function `StopWatch::ToInt` like this:

```
void f2() {
  StopWatch sw1(60);
  int seconds = sw1.ToInt(); // convert to an int explicitly
  std::cout << seconds << "\n";
}
```

If you want the compiler to perform the conversions *automatically*, then you have to use an *implicit type conversion operator*. An implicit type conversion operator is simply a member function with a strange-looking name: the word `operator` followed by a type specification. You aren't allowed to specify a type for the function's return value, because the type of the return value is basically just the name of the function. For example, to allow `StopWatch` objects to be implicitly converted to `int`s, you might define class `StopWatch` like this:

```
class StopWatch {
public:
  // other public methods ...
  operator int() const; // implicit type conversion operator

private:
  int seconds;
};
```

The type conversion operator is trivially defined:

```
StopWatch::operator int() const {
  return seconds;
}
```

This function would be automatically invoked in contexts like this:

```
void f3() {
  StopWatch sw1(60);
  int seconds = sw1; // Convert to an int implicitly
  std::cout << seconds << std::endl;
}
```

Let's review the general form of the type conversion operator:

```
operator type() const;
```

- `type` can be a built-in type such as `int` or a user-defined type such as `std::string` or `StopWatch`.

- *Must* be a member function. Most other operators can be overloaded as either a member function or global function.
- *Zero* arguments. You may specify `void` for the arguments, but it's not required.
- *No* return type because the type of the return value is basically just the function's name.
- Can be marked `explicit` to prevent the compiler from making the conversion implicitly:

```
explicit operator int() const;
```

- Conversion operators are generally marked `const`.

Be wary of type conversion operator

In general, you usually don't want to provide type conversion functions of *any* ilk. The fundamental problem is that such functions often end up being called when you neither want nor expect them to be. The result can be incorrect and unintuitive program behavior that is maddeningly difficult to diagnose. Here's an example:

```
int array[10];
StopWatch sw(60);
// other code...
int value = array[sw]; // disaster: sw converted to 60!!!
std::cout << value << "\n";
int swval = sw
std::cout << swval << "\n";
```

Further suppose you forgot to write an `operator<<` for class `StopWatch`. You would probably expect that the attempt to print `sw` would fail, because there is no appropriate `operator<<` to call:

```
StopWatch sw(60);
std::cout << sw;
```

You would be mistaken. Your compilers, faced with a call to a function called `operator<<` that takes a `StopWatch`, would find that no such function existed, but they would then try to find an acceptable sequence of implicit type conversions they could apply to make the call succeed. The rules defining which sequences of conversions are acceptable are complicated, but in this case your compilers would discover they could make the call succeed by implicitly converting `sw` to an `int` by calling `StopWatch::operator int()`. The result of the code above would be to print `sw` as an `int`, not as a stopwatch. This is hardly a disaster, but it demonstrates the disadvantage of implicit type conversion operators: their presence can lead to the *wrong function* being called [that is, one other than the one intended].

The solution is to replace the operators with equivalent functions that don't have the syntactically magic names. For example, to allow conversion of a `StopWatch` object to an `int`, replace `operator int` with the previously defined member function `StopWatch::ToInt`:

```
int array[10];
StopWatch sw(60);

// other code...
int temp2 = 0;
int value = array[temp2];  // correct
std::cout << value << "\n";
int swval = sw.ToInt(); // explicit
std::cout << swval << "\n";
```

The other alternative is to instruct the compiler not to implicitly convert a `StopWatch` to an `int` by specifying the implicit type conversion operator to be `explicit`:

```
class StopWatch {
public:
  StopWatch();
  explicit StopWatch(int seconds); // suppress implicit conversions
  StopWatch(int hrs, int mins, int secs);
  // other public methods

private:
  int seconds;
};
```

In this case, the compiler must be instructed to perform the conversion from a `StopWatch` object to an `int` value:

```
StopWatch sw(60);
int seconds1 = sw;       // error: can't convert to an int implicitly
int seconds2 = int(sw1); // ok: explicit conversion
```

In most cases, the inconvenience of having to call conversion functions explicitly is more than compensated for by the fact that unintended functions can no longer be silently invoked. In general, the more experience C++ programmers have, the more likely they are to eschew type conversion operators. The members of the committee working on the C++ standard library, for example, are among the most experienced in the business, and perhaps that's why the `string` type they added to the library contains no implicit conversion from a `string` object to a C-style `char*`. Instead, there's an explicit member function, `c_str`, that performs that conversion. Coincidence? I think not.

Overloading operators with side-effects

Up to this point:

- All of our operators created a new object.
- This new object was returned by value.
- The operands were not modified in anyway [which makes sense].

Consider the following two examples:

```
// built-in types that implicitly support operator+
int a = 1, b = 2, c;
c = a + b; // new value stored in c; a and b are unchanged

// StopWatch type overloads operator+
StopWatch sw1(30), sw2(60);
StopWatch sw3 = sw1 + sw2; // new StopWatch object is created
                           // sw1 and sw2 unchanged
```

We would like to support *side-effect* operators like this:

```
// built-in operators that are implicitly support operator+=
int a = 5, b = 7;
a += b; // a is changed; b is unchanged

// StopWatch type must overload operator+=
StopWatch sw1(30), sw2(60);
sw1 += sw2; // sw1 is changed; sw2 is unchanged
```

Member functions `operator+` and `operator+=` are declared like this:

```
class StopWatch {
public:
  // overloads for op+ and op+=
  StopWatch operator+(StopWatch const& rhs) const;
  StopWatch& operator+=(StopWatch const& rhs);

  // other public methods ...

private:
  int seconds;
};
```

These overloads are defined like this:

```
StopWatch StopWatch::operator+(StopWatch const& rhs) const {
  return StopWatch(seconds + rhs.seconds);
}

StopWatch& StopWatch::operator+=(StopWatch const& rhs) {
  seconds += rhs.seconds; // Modify this object directly
  return *this;
}
```

Member function `operator+=` is not marked `const` because you're changing the object. The return type is a reference because a new object is not being created. In addition, returning a reference must support this syntax that is valid with `int`s and must therefore be valid with `StopWatch`es:

```
int a = 1, b = 2, c = 3;
a += b += c; // associates right to left; changes a and b
std::cout << a << "\n"; // 6
std::cout << b << "\n"; // 5
std::cout << c << "\n"; // 3

StopWatch sw1(30), sw2(60), sw3(90);
sw1 += sw2 += sw3; // associates right to left; changes sw1 and sw2
std::cout << "\n"; // 00:03:00
std::cout << "\n"; // 00:02:30
std::cout << "\n"; // 00:01:30
```

If the function returned `void`, this syntax would be illegal. Since `this` is a *pointer* to the object, `*this` is the object, and that's what you want to return a reference to. You'll see this technique used more later.

Not a good idea to overload `&&`, `||`, and `,` operators

C++ allows you to customize the behavior of operators `&&` and `||` for user-defined types. You do it by overloading functions `operator&&` and `operator||`, and you can do this at the global scope or on a per-class basis. If you decide to take advantage of this opportunity, however, you must be aware that you are changing the rules of the game quite radically, because you are replacing short-circuit semantics with *function call* semantics. That is, if you overload `operator&&`, what looks to you like this:

```
if (expression1 && expression2) ...
```

looks to compilers like one of these:

```
// when operator&& is a member function
if (expression1.operator&&(expression2)) ...

// when operator&& is a global function
if (operator&&(expression1, expression2)) ...
```

This may not seem like that big a deal, but function call semantics differ from short-circuit semantics in two crucial ways. First, when a function call is made, *all* parameters must be evaluated, so when calling functions `operator&&` and `operator||`, *both* parameters are evaluated. There is, in other words, no short circuit evaluation. Second, the language specification leaves undefined the order of evaluation of function call arguments, so there is no way of knowing whether `expression1` or `expression2` will be evaluated first. This stands in stark contrast to short-circuit evaluation, which *always* evaluates its arguments in left-to-right order.

As a result, if you overload `&&` or `||`, there is no way to offer programmers the behavior they both expect and have come to depend on. So don't overload `&&` or `||`.

The comma operator is used to form *expressions*, and you're most likely to run across it in the update part of a `for` loop. Consider the following example:

```
// reverse string s in place
void reverse(char s[]) {
  for (int i = 0, j = strlen(s)-1; i < j; ++i, --j) {
    int c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}
```

Here, `i` is incremented and `j` is decremented in the final part of the `for` loop. It is convenient to use the comma operator here, because only an expression is valid in the final part of a `for` loop; separate statements to change the values of `i` and `j` would be illegal.

Just as there are rules in C++ defining how `&&` and `||` behave for built-in types, there are rules defining how the comma operator behaves for such types. An expression containing a comma is evaluated by first evaluating the part of the expression to the left of the comma, then evaluating the expression to the right of the comma; the result of the overall comma expression is the value of the expression on the right. So in the final part of the loop above, compilers first evaluate `++i`, then `--j`, and the result of the comma expression is the value returned from `--j`.

If you decide to write your own comma operator, you will need to mimic this behavior. Unfortunately, you can't perform the requisite mimicry. If you write `operator,` as a non-member function, you'll never be able to guarantee that the left-hand expression is evaluated before the right-hand expression, because both expressions will be passed as arguments in a function call (to `operator,`). But you have no control over the order in which a function's arguments are evaluated. So the non-member approach is definitely out. That leaves only the possibility of writing `operator,` as a member function. Even here you can't rely on the left-hand operand to the comma operator being evaluated first, because compilers are not constrained to do things that way. Hence, you can't overload the comma operator and also guarantee it will behave the way it's supposed to. It therefore seems imprudent to overload it at all.

> ***Don't overload operators `,`, `&&`, and `||`!!!***

Overloading prefix and postfix increment and decrement operators

Both prefix and postfix versions of the increment operator use the same token `++` and must be overloaded. Likewise the prefix and postfix versions of the decrement operator use the same token `--`. Overloaded functions are differentiated on the basis of the parameter types they take, but neither prefix nor postfix increment or decrement takes an argument. To surmount this linguistic conundrum, it was decreed that postfix forms take an `int` argument, and compilers silently pass `0` as that `int` when those functions are called:

```
class StopWatch {
public:
  // other functions ...
  StopWatch& operator++();          // prefix ++
  const StopWatch operator++(int);  // postfix ++

  StopWatch& operator--();          // prefix --
  const StopWatch operator--(int);  // postfix --

private:
  int seconds;
};
```

You may recall that the prefix form of the increment operator is sometimes called *increment and fetch*, while the postfix form is often known as *fetch and increment*. These two phrases are important to remember, because they all but act as formal specifications for how prefix and postfix increment should be implemented:

```
// prefix form: increment and fetch
StopWatch& StopWatch::operator++() {
  ++seconds;                            // increment
  return *this;                          // fetch
}
// postfix form: fetch and increment
const StopWatch StopWatch::operator++(int) {
  StopWatch oldValue = *this;            // fetch
  ++(*this);                             // increment
  return oldValue;                       // return what was
}                                        // fetched
```

Note how the postfix operator makes no use of its parameter. This is typical. The only purpose of the parameter is to distinguish prefix from postfix function invocation. Many compilers issue warnings if you fail to use named parameters in the body of the function to which they apply, and this can be annoying. To avoid such warnings, a common strategy is to omit names for parameters you don't plan to use; that's what's been done above.

It's clear why postfix increment must return an object [it's returning an old value], but why a `const` object? Imagine that it did not. Then the following would be legal:

```
StopWatch sw(30);
sw++++;             // apply postfix increment twice
```

This is the same as

```
sw.operator++(0).operator++(0);
```

and it should be clear that the second invocation of `operator++` is being applied to the object returned from the first invocation.

There are two problems with this. First, it's inconsistent with the behavior of the built-in types. A good rule to follow when designing classes is *when in doubt, do as the* `int`*s do*, and the `int`s most certainly do not allow double application of postfix increment:

```
int i;
i++++;  // error!
```

The second reason is that double application of postfix increment almost never does what clients expect it to. As noted above, the second application of `operator++` in a double increment changes the value of the object returned from the first invocation, *not* the value of the original object. Hence, if

```
i++++;
```

were legal, `i` would be incremented only once. This is counterintuitive and confusing [for both `int`s and `StopWatch`s], so it's best prohibited.

C++ prohibits it for `int`s, but you must prohibit it yourself for classes you write. The easiest way to do this is to make the return type of postfix increment a `const` object. Then when compilers see

```
i++++;   // same as i.operator++(0).operator++(0);
```

they recognize that the `const` object returned from the first call to `operator++` is being used to call `operator++` again. `operator++`, however, is a non-`const` member function, so `const` objects - such as those returned from postfix `operator++` - can't call it. If you've ever wondered if it makes sense to have functions return `const` objects, now you know: sometimes it does, and postfix increment and decrement are examples.

Let's evaluate the postfix increment function from the perspective of efficiency. That function has to create a temporary object for its return value, and the implementation above also creates an explicit temporary object (`oldValue`) that has to be constructed and destructed. The prefix increment function has no such temporaries. This leads to the conclusion that, for efficiency reasons alone, clients of `StopWatch` should prefer prefix increment to postfix increment unless they really need the behavior of postfix increment. Let us be explicit about this:

> ***When dealing with user-defined types, prefix increment should be used whenever possible, because it's inherently more efficient.***

Let us make one more observation about the prefix and postfix increment operators. Except for their return values, they do the same thing: they increment a value. How can you be sure the behavior of postfix increment is consistent with that of prefix increment? What guarantee do you have that their implementations won't diverge over time, possibly as a result of different programmers maintaining and enhancing them? Unless you've followed the design principle embodied by the code above, you have no such guarantee. That principle is that postfix increment and decrement should be implemented *in terms of* their prefix counterparts. You then need only maintain the prefix versions, because the postfix versions will automatically behave in a consistent fashion.

Here is some code that makes calls to the overloaded prefix and postfix increment functions:

```
StopWatch sw1(60);
StopWatch sw2 = ++sw1; // prefix increment: afterwards sw2 == sw1
std::cout << sw1; // 00:01:01
std::cout << sw2; // 00:01:01

sw2 = sw1++; // postfix increment: afterwards sw2 != sw1
std::cout << sw1; // 00:01:02
std::cout << sw2; // 00:01:01

sw2 = ++sw1++; // error: same as this: ++(sw1++)
sw2 = (++sw1)++; // prefix followed by postfix increment
std::cout << sw1; // 00:01:04
std::cout << sw2; // 00:01:03
```

Member, friend, or non-member?

In most cases, you have three choices when overloading operators: member or friend or non-member/non-friend function.

1. Member function: Binary operators will take one operand with the left operand implicit via the `this` pointer.

   ```
   StopWatch StopWatch::operator*(int factor) const {
     StopWatch sum(seconds_ * factor);
     return sum;
   }
   ```

   Unary operators will not take any parameters.

2. Friend function: Declare a friend function to overload the operator and implement it outside of the class. The friend function has access to the private members of the parameters. Binary operators will have two operands while unary operators will have one operand.

   ```
   StopWatch operator*(int factor, StopWatch const& sw) {
     return StopWatch(sw.seconds * factor);
   }
   ```

3. Non-member/non-friend function: Implement a non-member, non-friend function to overload the operator. The function can only access the public methods of the class which means that the class must have a public accessor. Binary operators will have two operands while unary operators will have one operand.

   ```
   StopWatch operator*(int factor, StopWatch const& sw) {
     return StopWatch(sw.GetSeconds() * factor);
   }
   ```

4. In certain cases, a non-member, non-friend function can be implemented in terms of a regular overloaded operator function. For example, assuming a member `operator*(int)` overload, the non-member/non-friend function can be defined like this:

   ```
   StopWatch operator*(int factor, StopWatch const& sw) {
     // the compiler sees it like this: sw.operator*(factor)
      return sw * factor;
   }
   ```

This assumes that the operator is commutative. That is `a*b` is the same as `b*a`.

> ***Study these four functions above and be sure you understand exactly how each one is different and why. Also make sure you know why some functions have 1 parameter and some have 2 parameters. Unary operators will have 0 or 1 parameter(s), depending on whether they are members or not.***

Restrictions on operator overloading

1. At least one operand must be a user-defined type. This means you can't overload built-in types.

```
// compiler error: cannot overload built-in operators for int
int operator+(int left, int right);
```

2. You can't violate the C++ syntax rules for the operator you wish to overload. This means that you can't change the precedence or associativity of an operator. This also means that if you overload binary modulus operator `%`, you can't use it with just a single operand:

```
int i;
StopWatch sw;

% i;  // error: % requires 2 integer operands
% sw; // error: % would require 2 operands
```

3. You can't create new operator symbols. As mentioned above, the `@` symbol is not a C++ operator for the built-in types, so you can't define it for a user-defined type.
4. You cannot overload the following operators: `::`, `.*`, `.`, `?:`, `sizeof`, `typeid`, `const_cast`, `dynamic_cast`, `reinterpret_cast`, and `static_cast`.
5. The following operators can only be overloaded with member functions: `=`, `()`, `[]`, `->`.
6. These operators can be overloaded using either member or non-member functions:

```
+       -       *       /       %       ^       &       |
~=      !       =       <       >       +=      -=      *=
/=      %=      ^=      &=      |=      <<      >>      >>=
<<=     ==      !=      <=      >=      &&      ||      ++
--      ,       ->*     ->      ()      []      new     delete
new []  delete []
```

7. A binary operator must be implemented either by a member function taking one parameter or by a non-member function taking two parameters. The exceptions are these operators [`=`, `()`, `[]`, `->`] that must be implemented as member functions.
8. A unary operator must be implemented either by a member function taking no parameters or by a non-member function taking one parameter.
9. Remember that friend functions are non-member functions so they can't be marked `const`.
10. Operators `&&`, `||`, and `,` have special properties with built-in types. Namely, they are *sequence guarantee operators*. However, when overloaded with user-defined types, this property does not hold.

`StopWatch` class implementation

```
#include <iostream>
#include <iomanip>
#include "StopWatch.h"

StopWatch::StopWatch() : seconds(0) {}
StopWatch::StopWatch(int secs) : seconds(secs) {}
StopWatch::StopWatch(int hour, int minutes, int secs)
: seconds(hour*3600 + minutes*60 + secs) {}
void StopWatch::Increment(int secs) {
  seconds += secs;
}
void StopWatch::Reset() {
  seconds = 0;
}
void StopWatch::Display() const {
  int hours   = seconds / 3600;
  int minutes = (seconds - (hours * 3600)) / 60;
  int secs    = seconds % 60;

  std::cout.fill('0');
  std::cout << std::setw(2) << hours << ':';
  std::cout << std::setw(2) << minutes << ':';
  std::cout << std::setw(2) << secs << "\n";
}

int StopWatch::GetSeconds() const {
  return seconds;
}
```