Operator Overloading

PIC 10B, UCLA

Operator Overloading

Most of the operations we use in C++ invoke operators (functions). The symbols such as **++, - -, *=, %, ( ), [ ], <<, >>** and many others are operators. We're often used to the meanings of these operations for numeric types; sometimes we use them with different objects such as printing to the console with **<<** or we apply **+=** on **std::string**s in a way that "makes sense" (concatenation). Because these operations can add such richness to programming, it is worth studying how we can give our own meaning to various operators: this is called operator overloading.

Motivation

We will consider operator overloading in the context of a **Time** class, used to store the integer number of hours, minutes, and seconds for a time.

We would like to be able to add/subtract times, multiply/divide them by numbers, get individual components of the time (hours/minutes/seconds), reset the time, print the time, read in new times, compare times, and even interpret some expressions as times.

Motivation

```
// default construction and setting hr:min:sec
Time t1, t2(3,57,19);
std::cout <<t1 <<" "<<t2 <<'\n'; // can cout the times

t1 = Time(4,2,56); // set new time (move assignment)
std::cout <<t1 <<'\n';

// can be used as constexpr, can recognize as 124 seconds:
constexpr Time t3 = 124_sec;
std::cout <<t3 <<'\n';
```

```
0:0:0 3:57:19
4:2:56
0:2:4
```

Motivation

```
t1 += t2; // add the times
std::cout <<t1 <<'\n';

--t1; // go back a second
std::cout <<t1 <<'\n';

t1/=4; // divide the time by 4, rounding to nearest second
std::cout <<t1 <<'\n';
```

```
8:0:15
8:0:14
2:0:4
```

Motivation

```
// compare
std::cout << ( (t2 < t1) ? "t1 longer" : "t1 not longer" ) << '\n';

std::cout << t2['h'] << '\n'; // subscript

std::cout << static_cast<int>(t1) << '\n'; // convert Time to int
```

```
t1 not longer
3
7204
```

Operator Overloading: the Gist

When we wish to cout a **Time** object with **<<** so that

```
constexpr Time t(1,20,40);
std::cout <<t <<'\n';
```

we define a function called **operator<<** with signature

```
std::ostream& operator<<(std::ostream&, const Time&);
```

It takes as input arguments an **std::ostream** object (on the left) and a **Time** object (on the right), outputting the time in some format, and returning the stream so that it can combine again with **'\n'.**

Operator Overloading: the Gist

Effectively what happens is:

display the time and return std::cout

$$\overbrace{(\texttt{std::cout} \ll t)} \qquad \ll \texttt{'\textbackslash n'};$$

append a new line, clear the buffer, return std::cout

$$\overbrace{\texttt{std::cout} \ll \texttt{'\textbackslash n'};}$$

nothing more to do

$$\overbrace{\texttt{std::cout};}$$

The first operator evaluation is one we need to define: how **std::cout** should combine on the left of **<<** with a **Time** object on the right.

The next operator evaluation is already written into **<iostream>** for how **std::cout** combines on the left of **<<** with a **std::ios_base& (*)(std::ios_base&)** object, such as **'\n'**.

Historical Aside on << and >>

The meaning of **<<** and **>>** outside of our convenient overloads has to do with **bit shifting**. Recall that integer types are stored as a collection of bits (on/off, 1/0) and these bit patterns are interpreted as numbers in binary (base 2).

In base 10, we use symbols 0-9 and interpret

329 = 3 $\times 10^2$ **+ 2** $\times 10^1$ **+ 9** $\times 10^0$.

In base 2, we use symbols 0-1 and interpret

11011 = 1 $\times 2^4$ **+ 1** $\times 2^3$ **+ 0** $\times 2^2$ **+ 1** $\times 2^1$ **+ 1** $\times 2^0$ (which, in base 10 would be 27).

We decompose a number into sums of powers of the base.

Historical Aside on << and >>

Consider an **unsigned char**, which takes up 1 byte; 1 byte is often 8 bits of information.

unsigned char c = 43;
Internally, it could be stored as **00101011**.

c = c >> 1; // right shift all bits by 1, possibly losing a bit
Now we have **00010101** (the value is 21). 0's come in from the left and bits "fall off" the right

c <<= 3; // same as c = (c <<3): bit shift left by 3
Now we have **10101000** (the value is 168). 0's come in from the right and bits "fall off" the left.

Managing integers does have its subtleties in terms of storing negative values (often via the 2's complement), but these operators are used in the process of various integer arithmetic processes.

Operator Overloading: the Gist

In defining the class interface, there are a few things we need to be aware of. Operators can be member functions or non-member functions.

Some operators can only be member functions such as the
subscript **[]**,
call **()**,
assignment **=**,
conversion/cast, and
arrow **->** operators.

Operator Overloading: the Gist

Operators that we overload can be **unary** (taking one argument) or **binary** (taking two arguments).

When an operator is overloaded as a member function, its *first argument* is implicitly the object itself and a unary operator would take no explicit parameters and binary operator would take one explicit parameter.

If the operator is implemented as a non-member function, we must explicitly state the argument(s) being passed to the operator.

Operator Overloading: the Gist

Useful digression: there are binary and unary versions of **+** and **-**. The binary versions add/subtract two arguments. The unary **+** and **-** are used below for **int**s:

```
int x = 9;
std::cout <<+x; // prints 9: unary +
std::cout <<-x; // prints -9: unary -
```

Operator Overloading: the Gist

Suppose that **t1** and **t2** are two time variables.

With **operator+** as a **binary non-member** function we have that

```
Time t3 = t1 + t2; // this is equivalent to ....
Time t3 = operator+(t1,t2); // this.
```

With **operator+=** as a **binary member** function, we have that

```
t1+=t2; // this is equivalent to...
t1.operator+=(t2); // this
```

Operator Overloading: the Gist

With **operator-** as a **unary member** function (returning the negated time - *not used for subtraction when unary*), we have that

```
t3 = -t2; // this is equivalent to ...
t3 = t2.operator-(); // this.
```

With **operator+** as a **unary non-member** function (*note this violates the style conventions and is for illustration only*), we have

```
+t1; // this is equivalent to ...
operator+(t1); // this
```

Operator Overloading: the Gist

Style conventions:

Asymmetric and unary operations such as
+=, **-=**, ***=**, **/=**, **%=**,

the *increment/decrement* operators **++**, **- -**, and

unary operators like unary **+**, unary **-**, unary ***,

are implemented as *member functions* by convention.

Operators like **+=** are "asymmetric" in the sense that usually just one of the two parameters is modified (the one on the left).

Operator Overloading: the Gist

Binary operators that do not change their input arguments such as

binary **+**, binary **-**, binary *, /, **%**,

and the comparison operators **<**, **>**, **==**, **!=**, **<=**, **>=**, etc.

are implemented as *non-member functions*.

The benefit of the operators being free functions (i.e. non-member) is that implicit conversions are also allowed.

Operator Overloading: the Gist

There are many operators that can be overloaded:

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

Inherent within the language, some of these operators have greater precedence than others: binary * before binary **+**, for example.

Also, some of these operators operate from left to right, like **+**, and others from right to left, like **=**.

Operator Overloading: the Gist

In operator overloading, we can give a meaning to certain operators, but we cannot change their direction (i.e. whether the compiler reads them left-to-right or right-to-left) nor their precedence, i.e. the C++ order of operations.

We can also *only overload operators for classes* and only those operators that do not already have a meaning. We can't change the meaning of * for **int**s, for example.

We will write selected operators for the **Time** class.

Time Class Setup I

We begin by considering the declarations of the various operators for our **Time** class. We only consider a small subset of the possible operators.

For brevity, the operators/constructors are not documented with /** */ but for practical code, the operators should all be documented.

Time Class Setup II

```
class Time {
private:
    int hours, minutes, seconds; // data members

    // constants used for converting
    constexpr static int sec_per_hr = 3600, sec_per_min = 60;

    // gives the corresponding number of seconds
    constexpr int totalSeconds() const {
        return sec_per_hr * hours + sec_per_min*minutes + seconds;
    }

    constexpr void reduce(); // puts time into proper hh:mm:ss format
```

Time Class Setup III

```
public:
    // accepts number of seconds, uses 0 for a default construction
    constexpr Time(int _seconds = 0);

    // sets the time from input hours, minutes, and seconds given
    constexpr Time(int _hours, int _minutes, int _seconds) :
        Time(sec_per_hour * _hours + sec_per_min * _minutes + _seconds)
        { }

    // constructor to read from input stream
    Time(std::istream& in);
```

Time Class Setup IV

```
// unary -
constexpr Time operator-() const
    { return Time(-hours,-minutes,-seconds); }

constexpr Time& operator+=(const Time& other);

constexpr Time& operator--(); // prefix
constexpr Time operator--(int unused); // postfix

constexpr Time& operator*=(double factor);

constexpr Time& operator()(); // call operator: reset

constexpr int operator[](char i) const; // subscript
```

Time Class Setup V

```
    constexpr operator int() const { // conversion to int operator
        return total_seconds();
    }

    // to read from an input stream
    friend std::istream& operator>>(std::istream& in, Time& t);

};
```

Time Class Setup VI

```
// some non-member operator overloads

// to compare
constexpr bool operator<(const Time& left, const Time& right) {
    return left.operator int() < right.operator int();
}

constexpr Time operator+(Time left, const Time& right); // binary +

constexpr Time operator*(Time left, double factor); // binary *

constexpr Time operator*(double factor, Time right); // another binary *

// user-defined literal
constexpr Time operator ""_sec(unsigned long long int seconds);
```

Time Class Implementations: private

```
private:
   int hours, minutes, seconds; // data members

   constexpr static int sec_per_hr = 3600, sec_per_min = 60;
```

The **minutes** and **seconds** will range from -59 to 59. The **hours** can be any **int**.

We choose to make **sec_per_hr** and **sec_per_min** as **static constexpr** member variables: we can avoid “magic numbers” of 60/3600 appearing in our code and preserve high efficiency.

Time Class Implementations: private

The private member functions are for use by the class itself.

totalSeconds computes the total number of seconds represented by a time with its hours, minutes, and seconds.

Time Class Implementations: private

reduce properly formats the time to
(hours):(minutes from -59 to 59):(seconds from -59 to 59).

The **reduce** member function should find the total number of signed (could be negative) seconds found in the number of hours, minutes, and seconds stored in the object; then, it should compute the corresponding number of hours, minutes, and seconds that correspond to that number of seconds.

If the total number of seconds should be negative, we will have that all of the hours, minutes, and seconds are nonpositive: -63 seconds should be -1 minute and -3 seconds.

Time Class Implementations: constexpr and inline

Our **Time** class can be a **constexpr** class, having some **constexpr** constructors, because it only contains literal types (**int**s in this case). Were it to store **std::string**s, etc., this would not be possible.

Intuitively: if a class has a "complicated" destruction process, which requires freeing heap memory, or being part of an inheritance structure, it is not suitable to be a **constexpr**-type class.

And many member functions can also be **constexpr** when they only involve fundamental types and reading/setting values that can be done at compile time.

Time Class Implementations: constexpr and inline

A **constexpr** function, including member function or constructor, is implicitly **inline** (to be discussed).

This means the definition of such a function should appear in the header file in which it is declared! We can still separate the declaration and definition but they must both be in the header file.

A **constexpr** function's definition must be provided before it can be used in a **constexpr** context.

Time Class Implementations: constexpr and inline

We mark as many things as possible as **constexpr** so the compiler can do as much as possible for us when it compiles things. Whenever a method could be run during compilation because it only uses literal types and other **constexpr**-classes/variables, we mark it as **constexpr**.

We are using C++14 (and beyond) features by allowing a **constexpr** function to have multiple statements (not just a single statement).

Since C++14, a **constexpr** member function can be called in a non-const manner on a class as in **Time::reduce**.

Time Class Implementations: reduce

```
constexpr void Time::reduce() {
    int total = totalSeconds(); // compute total number of seconds

    hours = total / sec_per_hr; // integer division gives hours

    // remainder with int division gets minutes
    minutes = (total % sec_per_hr) / sec_per_min;
    seconds = (total % sec_per_min); // seconds are remainder

}
```

Time Class Implementations: Default Constructor

```
constexpr Time(int _seconds = 0);
```

This function accepts a default argument: if none are provided, it constructs a **Time** object with 0 seconds; otherwise if a single **int** is provided, it constructs a **Time** object from that many seconds.

Time Class Implementations: Default Constructor

```
constexpr Time(int _seconds = 0);
```

The **Time** class is a **literal type**. As such, it is often possible that we can construct a **Time** object as a **constant expression** and we are allowed to mark some of its constructors as **consexpr**. Such constructors can be run at compile time.

The precise definition of a **literal type** from the C++ Standard Follows but in a nutshell for our purposes, an entity is a literal type if it only stores fundamental types, other literal types, and its destructor is *not virtual* (so no base classes with virtual destructors either!) and the *destructor does not need to free any memory/resources*.

Time Class Implementation: Default Constructor

A type is a literal type if it is:
— possibly cv-qualified void; or
— a scalar type; or
— a reference type; or
— an array of literal type; or
— a possibly cv-qualified class type [...] that has all of the following properties:
    — it has a trivial destructor,
    — it [...] *has at least one constexpr constructor* [...]
— if it is a union, at least one of its non-static data members is of non-volatile literal type, and
— if it is not a union, all of its non-static data members and base classes are of non-volatile literal types.
[*Note: A literal type is one for which it might be possible to create an object within a constant expression. It is not a guarantee that it is possible to create such an object, nor is it a guarantee that any object of that type will be usable in a constant expression. — end note*]

Time Class Implementations: Default Constructor

We define the constructor below (note how the default argument is not repeated in the definition):

```
constexpr Time::Time(int _seconds) :
    hours(_seconds / sec_per_hour),
    minutes( (_seconds % sec_per_hr) / sec_per_min )
    seconds( _seconds % sec_per_min) { }
```

It makes use of integer division and modular arithmetic. Suppose **_seconds** is 3922 = 1 hour, 5 minutes, 22 seconds:

3922/3600 = 1 and
(3922 % 3600) = 322 and **322/60 = 5** and
3922 % 60 = 22.

Time Class Implementations: Default Constructor

A **constexpr** constructor must initialize all member variables within the constructor initializer list.

Further assignments are permitted in the body, but all members must be initialized through a constructor initializer list. In general, a **constexpr** constructor must satisfy the conditions of an ordinary **constexpr** function.

Time Class Implementations: Time(int, int, int)

```
constexpr Time(int _hours, int _minutes, int _seconds) :
    Time(sec_per_hour * _hours + sec_per_min * _minutes + _seconds)
    { }
```

This constructor delegates work to the constructor accepting a total number of seconds. Being marked as **constexpr**, it can only invoke another **constexpr** constructor if it delegates work.

Time Class Implementations: Time(std::istream&)

We define this constructor by:

```
Time::Time(std::istream& in) : hours(std::numeric_limits<int>::max()),
    minutes(std::numeric_limits<int>::max()),
    seconds(std::numeric_limits<int>::max()) {

    in >>hours >>minutes >>seconds; // read in data

    if(!in) { // if the stream failed
        throw std::runtime_error("bad format");
    }

    reduce();
}
```

When working with streams, there is always the chance of failure. We check that the stream has not failed; if it has, we throw an exception so the object is never used.

Time Class Implementations: Time(std::istream&)

Imagine reading from a file **times.txt** line by line to get **Time**s but the file has a format like:

```
3 14 0
9 22 53
0 something very evil
2 4 43
```

The **exception**, if managed, would prevent that evil line from corrupting the **Time**s.

Time Class Implementations: Time(std::istream&)

It is always good practice to initialize a variable before it is used. We initialized **hours**, **minutes**, and **seconds** to the largest possible **int** values.

Not entirely necessary here given we check the state of the stream, if we saw a bunch of gigantic **int**-values floating around in the times, this would suggest a reading error.

Time Class Implementations: Time(std::istream&)

Note the syntax: **std::numeric_limits<int>::max()**

The **std::numeric_limits** is a templated class found in the **<limits>** header.

max is a static member function of that class.

Time Class Implementations: unary-

```
// unary -
constexpr Time operator-() const {
    return Time(-hours,-minutes,-seconds);
}
```

This is the **unary -** operator: it does not mean subtraction.

We use it here to return a negated version of a **Time** variable. Assuming **operator<<** has been overloaded then:

```
Time t(10,11,12);

std::cout << -t << '\n';
```

would generate

```
-10:-11:-12
```

Time Class Implementations: operator+=

We define **operator+=** outside the class:

```
constexpr Time& Time::operator+=(const Time& other){
    hours += other.hours; // update members
    minutes += other.minutes;
    seconds += other.seconds;

    reduce(); // format

    return *this; // return updated object
}
```

This is implemented as a member function. As it is a member function of the **Time** class, it is given access to the private member variables of **other**, which is also a **Time** object.

Right to Left

The **+=** operator is a right-to-left operator and in general it returns a reference to the updated variable. For example we could have:

```
Time t1(0,0,1), t2(0,0,7);
t1 += t2 += t1 += t1;
std::cout << t1 <<" "<< t2;
```

outputs

```
0:0:11 0:0:9
```

It is resolved as:

t1 now (0,0,2), returned by reference

t1 += t2 += $\overbrace{\texttt{t1+=t1}}$;

t2 now (0,0,9), returned by reference

t1 += $\overbrace{\texttt{t2 +=t1}}$;

t1 now (0,0,11), returned by reference

$\overbrace{\texttt{t1 += t2}}$;

t1;

Right to Left

While a similar output could be achieved in returning by value instead of reference, there would be an additional overhead of creating temporary objects (and destroying them), whereas the references ensure there are no copies being made.

The convention for these sorts of operators is to return by reference. There would also be differences in behaviour for code such as

```
Time t1(0,0,1), t2(0,0,4), t3(0,0,9);
(t1 += t2) += t3;
std::cout << t1;
```

When the signature is **Time& operator+=(const Time&)**, the output is

```
0:0:14
```

When the signature is **Time operator+=(const Time&)**, the output is

```
0:0:5
```

because: t2 is added to t1 making it 0:0:5, but only the copy has t3 added to it.

Right to Left

All of
+= -= *= /= %= =
are right-to-left operators.

Consider:
```
Time t1(0,0,1), t2(0,0,7), t3;
t3 = t2 = t1; // now all are 0:0:1
```

It is resolved as:

t2 now (0,0,1), returned by reference

t3 = $\overbrace{\texttt{t2=t1}}$;

t3 now (0,0,1), returned by reference

$\overbrace{\texttt{t3 =t2}}$;

t3;

Time Class Implementations: operator- -

```
constexpr Time& operator- -(); // prefix
constexpr Time operator- -(int unused); // postfix
```

There are prefix and postfix versions of **++** and **- -**; the way the compiler tells them apart is by requring the postfix to accept an unused **int** parameter. Otherwise they would have the same signature.

The *prefix versions are unary*; the *postfix versions are binary*.

Time Class Implementations: operator--

Defined outside the interface:

```
constexpr Time& Time::operator--(){
    --seconds; // decrement the seconds
    reduce(); // format
    return *this; // return
}

constexpr Time Time::operator--(int unused){
    Time copy(*this); // copy the current time
    --seconds; // decrement the copy
    reduce();
    return copy; // return the copy
}
```

Time Class Implementations: operator- -

With the **postfix**, we follow the convention that the object is *copied* first; then, it gets *decremented*; finally, the *copy of the original*, pre-decremented object is *returned*.

Because we never use **unused**, we can even remove its name altogether.

The only way the compiler can find the definition of the postfix **++** or **- -** is by finding an **operator++** or **opeator- -** that explicitly accepts an **int**!

Time Class Implementations: operator- -

Assuming prefix**++**:

```
Time t(0,0,58);
++++++t; // now t == Time(0,1,1)
```

The **reduce** ensures the format is correct.

Time Class Implementations: operator- -

From an efficiency standpoint, loops with iterators as below should use the prefix, not postfix ++ and - - to avoid extra copying:

```
for ( auto itr = start; itr != end; ++ itr ) { // good
    // stuff
}

for ( auto itr = start; itr != end; itr++ ) { // bad
    // stuff
}
```

++ and - - for Fundamental Types

Fundamental types also have prefix and postfix **++** and **- -** operators; the prefix always increments/decerements and returns the updated value by reference and the postfix always copies, increments/decrements, and returns the copy value.

Although the prefix operators can be chained together, the postfix operators can only be applied to l-values for fundamental types (there is no pr-value to x-value promotion as for classes).

```
int x = 7;
++++x; // now x == 9
double y=4.2;
std::cout <<y- -; now y==3.2 and the output was 4.2

y- - - -; // ERROR: cannot do this because (y- -) returns pr-value
```

Time Class Implementations: operator*=

Defining **operator*=** to multiply a **Time** by a **double**.

```
// inside the class
constexpr Time& Time::operator*=(double factor) {
    double product = totalSeconds() * factor;
    bool pos = (product > 0); // check sign
    int absTotal = static_cast<int>( (pos ?
        (product + 0.5) : (-product + 0.5) ) ); // rounded absolute sec value
    hours = absTotal / sec_per_hr; // integer division gives hours
    minutes = (absTotal % sec_per_hr) / sec_per_min; /* remainder and int
        division gets minutes */
    seconds = (absTotal % sec_per_min); // seconds are remainder
    if ( ! pos) { // if total was negative, switch all the signs
        hours = -hours;        minutes = -minutes;        seconds = -seconds;
    }
    return *this;
}
```

Time Class Implementations: operator()

```
constexpr Time& operator()(); // call operator
```

The **operator()** is known as the **call** operator. It must always be a member function of the class, and it can take 0 or more arguments. There can be more than one version of the call operator, overloaded by argument types, implemented in a class.

The name of the operator above is **operator()**, and because this version takes no arguments, there is an extra set of parentheses indicating no arguments are taken. Had there been arguments to the operator, they would appear in a comma separated list in the second pair of parentheses.

Time Class Implementations: operator()

We wish to implement the operator taking no arguments, acting as a reset function, setting the time to 0:0:0, and returning the new object.

```
constexpr Time& Time::operator()() {
    hours = 0;
    minutes = 0;
    seconds = 0;

    return *this;
}
```

Time Class Implementations: operator()

We can use the operator we just defined as below:

```
Time t(1,2,3);
t(); // now t == Time(0,0,0)
```

Time Class Implementations: operator[]

constexpr int operator[](char i) const;

The **operator[]** is known as the **subscript** operator. It is always a member function. It can only take a single input parameter. But it can be overloaded with multiple possible single input types.

We consider the subscript operator with valid subscripts ‘h’, ‘m’, and ‘s’ for the hours, minutes, and seconds. Anything else is invalid.

Remark 1: if we chose to implement this function to accept an **std::string**, it could not be **constexpr** because **std::string**s cannot be **constexpr**.

Remark 2: a non-void function must return a value - or throw an exception.

Time Class Implementations I

```
constexpr int Time::operator[](char i) const {
    switch(i) {
        case 'h': return hours; break;
        case 'm': return minutes; break;
        case 's': return seconds; break;
        default: throw std::logic_error("bad index");
    }
}
```

Time Class Implementations: operator[]

Sometimes a subscript returns a reference type (not reference to const or a copy) in which case it would not be a const member function. Or the return type could depend upon the constness of the object (more later).

The reason we can change an element of a **std::vector<int>**, **v**, for example by writing

```
v[0] = 7;
```

is precisely because **v[0]** is a reference to the **int** stored at index 0.

Time Class Implementations: operator[]

For the **Time** class, giving outside access to the private variables of **hours**, **minutes**, and **seconds** could be disastrous because if a user directly modified those without the class object being **reduce**d, the outputs could become garbled. Thus we chose to just return the member variables by value.

Time Class Implementations: operator<<

Having written the subscript operator, we can overload **operator<<** as a non-member function:

```
std::ostream& operator<<(std::ostream& out, const Time& _time) {

    // use subscript to access hours, minutes, seconds
    out << _time['h'] << ":" << _time['m'] << ":" << _time['s'];

    return out; // and return the stream
}
```

Note that we use and call upon the local reference variable **out** here (could be **std::cout** but could be an output file stream or output string stream): we aren't just using **std::cout** - gotta thank Polymorphism for that!

Time Class Implementations: Conversion Operator

```
constexpr operator int() const { // conversion to int operator
    return total_seconds();
}
```

This operator allows a **Time** to be converted into an **int.**

If this operator had been defined outside the class, it would be defined as:

```
constexpr Time::operator int() const { return total_seconds(); }
```

Remark: the return type does not appear before the function name because the function name **operator int() const** already says what that is, **int** in this case.

Time Class Implementations: Conversion Operator

```
constexpr operator int() const { // conversion to int operator
   return total_seconds();
}
```

Note: the function can be marked **constexpr** because it only invokes another **constexpr** member function.

Time Class Implementations: Conversion Operator

```
constexpr operator int() const { // conversion to int operator
    return total_seconds();
}
```

Right now, the operator allows for implicit conversions. For example:

```
constexpr Time t(1,0,0);
constexpr int i = t; // i == 3600, Time converted implicitly
```

Time Class Implementations: Conversion Operator

Assuming we never chose to define
std::ostream& operator<<(std::ostream&,const Time&) then by writing

```
std::cout << t;
```

we would get the output

```
3600
```

since there is already a **operator<<** for **std::ostream**s to print **int**s.

Time Class Implementations: Conversion Operator

```
explicit constexpr operator int() const { // conversion to int operator
    return total_seconds();
}
```

To prevent the implicit conversions and only allow this for **static_cast**s, we use the **explicit** keyword as above.

The **explicit** keyword only appears within the class interface and not without.

With explicit:

```
constexpr Time t;
int j = t; // ERROR
int j = static_cast<int>(t); // okay, j could be constexpr, too
```

Time Class Implementations: operator>>

```
friend std::istream& operator>>(std::istream& in, Time& t);
```

The **friend** keyword gives a function (or class) access to the **private**/**protected** members of the class in which the **friend** keyword appears.

In this case, **std::istream& operator>>(std::istream& in, Time& t)** can access the private members of **Time** objects.

The **friend** keyword also makes it explicit that this function *is not a member function* of the class.

Time Class Implementations: operator>>

```
friend std::istream& operator>>(std::istream& in, Time& t);
```

The **friend** keyword should be used sparingly: with it, a class willingly violates encapsulation granting another function/class access to its private/protected members.

Time Class Implementations: operator>>

```
friend std::istream& operator>>(std::istream& in, Time& t);
```

It cannot be a member function: **operator>>** is binary. If it were a member function then the first argument would have to be a **Time** object, not a **std::istream&**.

Since we define it inside the class, to prevent the compiler from trying to make it into a member function (which would then require 3 arguments: the implicit object plus the two listed - nonsense for a binary operator), it knows the function is not a member because of the **friend** keyword.

Time Class Implementations: operator>>

A **friend** function can be defined inside or outside the class. When declared inside, it must have the **friend** keyword but it cannot have the **friend** keyword outside of the class.

Defined outside the class:

```
std::istream& operator>>(std::istream& in, Time& t) {
   // read in each variable from stream
   in >> t.hours >> t.minutes >> t.seconds;

   if( in.fail() ) { // if the stream is in a failed state
      throw std::runtime_error("read error");
   }

   t.reduce();
   return in;
}
```

Time Class Implementations: operator<

```
constexpr bool operator<(const Time& left, const Time& right) {
    return left.operator int() < right.operator int();
}
```

The **operator<** is a binary operator implemented as a non-member function. It invokes the **public**ly accessible conversion operator.

Time Class Implementations: binary operator+

From the definition of **operator+=**, we can define the binary subtraction operator. This does not need to be a member function, nor does it need special access privileges to class member variables. Outside of the class:

```
constexpr Time operator+(Time left, const Time& right) {
    return left += right;
}
```

Recall that **left** will be updated and its value will be returned by reference from **+=**; this reference will then be used to copy initialize the return value.

Note: it is not **constexpr** as it invokes a non-**constexpr** function.

Time Class Implementations: binary operator+

Note: a copy is made of the first variable! Despite **Time** being an object, we passed by value! If we passed it as a reference to const, we'd need to make a copy if it inside anyway because the binary addition isn't supposed to modify its input arguments.

```
// LESS EFFICIENT
constexpr Time operator+(const Time& left, const Time& right) {
    Time copy(left); // making a copy
    return copy += right;
}
```

This is another example of it sometimes being better to pass an object by value.

Time Class Implementations: binary operator*

We use this operator to multiply a **Time** either on the left or the right by a **double**.

Note: it would be impossible to write a member function ***=** to multiply a **double** on the left by a **Time** because the argument on the left would have to be the implicit object and a **double** is not a class type.

Time Class Implementations: binary operator*

Definitions:

```
constexpr Time operator*(Time left, double factor) {
    return left *= factor; // uses Time::operator*=
}

constexpr Time operator*(double factor, Time right) {
    // uses Time::operator*= with right as implicit argument
    return right *= factor;
}
```

User Defined Literals

A new feature of the C++ Standard is the support of user-defined literals. These are special functions a programmer can write to give a different meaning to the code that is written.

Recall how certain suffixes have meaning in C++:

```
unsigned x = 30u*1000u; // 30 and 1000 as an unsigned ints
1.234f; // 1.234 is a float
```

Now, we can give our own suffixes converting between different types and values.

```
// 1 mile could be converted to km
std::cout << 1._mi;
```

```
1.609344
```

User Defined Literals

The conversion from miles to km could have been accomplished with the definition:

```
constexpr long double operator ""_mi(long double miles)
{
    return miles * 1.609344L; // L suffix means long double
}
```

User Defined Literals

Annoying subtlety: to pass a value into a user defined literal that accepts **long double**, the argument must have a decimal in it.

1_mi; // ERROR, not okay. Should write 1._mi or 1.0_mi.

User Defined Literals

User defined literals can be invoked by calling their name, which must begin with an underscore _.

Not all inputs types are currently permissible; we will look at *two currently permissible numeric type inputs*.

Below, **T** represents an arbitrary return type and **_fun** is the function name. As an operator its name is **operator ""_fun**. Some declarations are below.

```
T operator "" _fun(unsigned long long int);
T operator "" _fun(long double);
```

Time Class Implementations: User Defined Literals

With

```
/**
Given input time in seconds and makes a Time object out of it
@param u a time in seconds as unsigned long long int
@return a Time object with that many seconds
*/
constexpr Time operator "" _sec(unsigned long long u) {
    return Time(static_cast<int>(u)); // cast to int for seconds
}
```

In **main**, we could write

```
std::cout << 63_sec;
```

and obtain

```
0:1:3
```

Time Class Implementations: operator==

If our class can be strongly ordered whereby
(!(A < B) & & !(B < A)) == (A==B)
then we only need **operator<** to define **operator==.** Here we can do just that, giving:

```
constexpr bool operator==(const Time& lhs, const Time& rhs) {
    return !( (lhs < rhs) || (rhs < lhs) );
}
```

Time Class Implementations: operators <=, >, >=, !=

With **operator<** and **operator==** defined, we can define all the other boolean operators.

(t1 <= t2) is (t1 < t2) || (t1 == t2)

(t1 > t2) is (t2 < t1)

(t1 >= t2) is (t1 > t2) || (t1 == t2)

(t1 != t2) is !(t1 == t2)

Remarks on Boolean Operators

It can be frowned upon to define classes where **operator<** and **operator==** are in contradiction to each other. We would like to say that if **A == B** then neither **A < B** nor **B < A** hold, and vice versa. This isn't necessary, of course, but it can make the users of a class a little confused when that intuition is broken.

Having **operator==** allows for searching to work in generic algorithms such as with the **std::find** function, that iterates through a container to find the first element with equality to a given search value. With **<algorithm>** included, we can **std::find** iterators to an element.

operators <, ==

Having **operator<**, we can apply sorting as in **<algorithm>**:

```
std::vector<Time> vtime; // define vector and add times
vtime.push_back( Time(1,0,0) );
vtime.push_back( Time(0,2,0) );
vtime.push_back( Time(0,0,3) );

std::sort( std::begin(vtime), std::end(vtime) ); // Times will be sorted by <

for ( const auto& time : vtime ) { // for each time, print it
    std::cout << t << '\n';
}
```

```
0:0:3
0:2:0
1:0:0
```

Operators Are Just Functions

It's important to remember that operators are just functions. If **t1** and **t2** are **Time** variables, then the following are equivalent:

```
std::cout << t1; // this and...
operator<<(std::cout, t1); // this

t1 += t2; // this and...
t1.operator+=(t2); // this

std::cout << static_cast<int>(t1); // this and ...
std::cout <<t1.operator int(); // this

--t1; // this and...
t1.operator--(); // this

t1--; // this and...
t1.operator--(0); // this, oddly enough (postfix takes an int)
```

Declaration vs Definition

Note that to define binary **operator+**, we made use of the existence of the function **+=**.

It is essential that this binary **operator+** function *definition* appear *after the declaration* of the **operator+=** that it uses!

Since the **operator+=** is declared within the interface and the **operator+** appears after that interface, we are safe.

Declaration vs Definition

In C++, there is a difference between **declaration** and **definition**. Generally a declaration just tells the compiler that a particular variable/function/class exists; with a definition, that variable/function/class is given a place in memory.

As long as the compiler knows what everything is, it can suitably put placeholders for items that are undefined; then, the linker needs to fill in the details by finding the definitions.

Declaration vs Definition

The C++ Standard has some delicate rules in linking h- and cpp-files called the **One Definition Rule**.

"No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template."

“Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program... An inline function shall be defined in every translation unit in which it is odr-used.”

Declaration vs Definition

In practice: we should use header files to provided interfaces and function/variable declarations.

Only **constexpr** or **inline** functions (includes those defined inside of a class interface) should be defined in a header file.

Most functions should be defined in a separate .cpp file.

Declarations and Definitions

inline function: a function that the compiler allows (*and requires!*) to be defined in each cpp-file where it is used.

A **constexpr** function is implicitly **inline**.

Declarations and Definitions

Historically, the definition of an inline function meant that a function call could be replaced by a more localized set of instructions rather than passing values around and needing to track a return address for when a function call is over, etc.

Nowadays the compiler decides whether this happens and the programmer's use of the **inline** keyword only relates to the **one definition rule** as opposed to telling the compiler what to do.

Declarations and Definitions

Member functions are inline if defined within the class interface *or* declared with the **inline** keyword and defined outside.

Non-member functions are inline if defined with the **inline** keyword.

odr: one definition rule

translational unit: a single cpp-file that is being compiled (with whatever comes from the header files)

Declaration vs Definition

A function is *declared by its signature*, whether its inputs are named or not, without giving it a body. Giving it a *body constitutes a definition*.

```
void f(const std::vector<int>&); // Declares the function
void f(const std::vector<int>& v); // Declares the function again, okay!

void f(const std::vector<int>& v) { // Defines the function
    for (int i : v) {     std::cout << i << " "; }
}

void f(const std::vector<int>&); // Declares again, okay!

void f(const std::vector<int>& v) { } // ERROR: defining again!
```

Sometimes a function is declared and defined simultaneously: any function definition on its own could also be deemed a declaration.

Declarations and Definitions

A class is *declared* with its class-type keyword (**class**, **struct**, **union**).
Giving it an *interface defines the class*.

```
class X; // Declares that X is a class
class X; // Declares again that X is a class
struct Y; // Declares that Y is a class

class X { }; // Defines the class X, which does nothing and stores nothing
struct Y : public X { // Defines the class Y
    void foo() const;
};

struct Y; // Declares again, okay!

class X { }; // ERROR: defines again
struct Y { // ERROR: defines again
    double z;
};
```

Declarations and Definitions

Consider:

```
struct X {
    void f() const { } // inline because defined in class
    inline bool g() const; // will be inline
    double d() const; // will not be inline
};

bool X::g() const { return false; } // inline because declared as such

double X::d() const { return 3.14; } // not inline, not declared in X as such

int g( int x ) { return std::rand() % x; } // not inline

inline int h( int x ) { return std::rand() % x; } // inline

constexpr int seven();
constexpr int seven() { return 7; } // constexpr implicitly inline
```

Declarations and Definitions

Header guards are used to protect against multiple class definitions.

They protect against a class definition being included multiple times within a single cpp-file.

Note that unlike functions that are defined when given an implementation, a class is defined when given an interface, even if all the member functions are only declared but not defined.

Declarations and Definitions

The following:

X.h

```
struct X { };
```

A.h

```
#include "X.h"
struct A { X x; };
```

B.h

```
#include "X.h"
struct B { X x; };
```

main.cpp

```
#include "A.h"
#include "B.h"
int main() { return 0; }
```

Declarations and Definitions

Translates into:

processed_main.cpp

```
struct X { }; // from A.h included that includes X.h
struct A { X x; }; // from A.h defining A

struct X { }; // from B.h included that includes X.h
struct B { X x; }; // from B.h defining B

int main() { return 0; }
```

This is bad because we have multiple definitions!

Declarations and Definitions

But with header guards:

X.h

```
#ifndef _X_
#define _X_
struct X { };
#endif
```

A.h

```
#ifndef _A_
#define _A_
#include "X.h"
struct A{ X x; };
#endif
```

B.h

```
#ifndef _B_
#define _B_
#include "X.h"
struct B{ X x; };
#endif
```

main.cpp

```
#include "A.h"
#include "B.h"
int main() { return 0; }
```

Declarations and Definitions

This gets pasted together by the **preprocessor** as follows:

A.h is included. It checks whether the symbol **_A_** has been defined (it hasn't) so it defines that symbol and includes **X.h**.

From **X.h** it checks if the symbol **_X_** has been defined (it hasn't) so it defines that symbol and allows `struct X {};` to be dumped into the translational unit.

Then it dumps `struct A { X x;};` into the translational unit.

Declarations and Definitions

B.h is then included. It checks whether the symbol **_B_** has been defined (it hasn't) so it defines that symbol and includes **X.h**.

With **X.h** it checks if the symbol **_X_** has been defined (it has!) so it does nothing more.

Then it dumps struct B { X x;}; into the translational unit and proceeds to define the main routine.

The overall result obeys the one-definition rule:

```
struct X { };
struct A { X x; };
struct B { X x; };
int main() { return 0; }
```

Declarations and Definitions

A variable is *declared with the* **extern** *keyword without an initialization*; *otherwise* it is a *definition*. Declarations may come up in global variables.

```
double a; // a is defined and of type double: value is unknown

double b = 32.1; // b is defined and initialized, of type double, value 32.1

extern int x; // x is declared to be an int but has no place in memory

int x = 12; // x is now defined

extern int y = 43; // y is defined

double a = 0; // ERROR: re-defining a

std::string b = "cat"; // ERROR re-defining b

int y = 42; // ERROR: re-defining y
```

Declarations and Definitions

H.h

```
#ifndef __H__
#define __H__
extern int x; // x is declared to keep compiler happy in main.cpp
#endif
```

x.cpp

```
int x = 42; /* x is defined here to keep linker happy when x definition
    sought */
```

main.cpp

```
#include "H.h"
int main() // x global, declared in H.h, defined in x.cpp
{
    return x;
}
```

Declarations then Definitions

Consider the program below:

```
#include <iostream>

struct X {
    void f() const { g(); } // ERROR: what is g()?
    Y *yp; // ERROR: what is a Y?
};

struct Y { }; // read after X::yp declared

void g() { std::cout <<"hi"; } // read after X::f()'s definition is read

int main() {
    X().f();
    return 0;
}
```

Declarations then Definitions

The member function **f()** of **X** calls upon **g()**. Unfortunately, the compiler reads code sequentially and it does not know what **g()** is! This can be fixed by:

- declaring and/or defining **void g()** before the class
- only declaring **void X::f() const** and implementing **void X::f() const** after the declaration and/or definition of **void g()**

Declarations then Definitions

The data type **Y** (and hence **Y***) is unknown to the compiler by the time the **X** interface is read. This can be fixed by:

- declaring **Y** before the class with a simple struct Y; or
- defining **Y** above the definition of **X**.

Declarations then Definitions

In general, a function/variable/class should only be referenced if the compiler will have previously read its declaration or definition.

Within a class itself, the order of definitions of functions/member variables is more flexible (can declare the member variables after the member functions, for example).

More on Inlining and Linking

The only functions that can be defined more than once in a program are the inline functions, and they then must be defined once in every translational unit where they are used.

More on Inlining and Linking

The code below fails to run as the non-inline **S::foo()** is defined more than once.

H.h

```
#ifndef _H_
#define _H_
struct S { void foo() const; };
void S::foo() const { }
#endif
```

A.cpp

```
#include "H.h"
// definition of S::foo() const included here
```

Main.cpp

```
#include "H.h"
// definition of S::foo() const included here
int main() {          S().foo();          return 0;          }
```

More on Inlining and Linking

Note that by virtue of an inline function being defined in a header file, its definition will appear in each translational unit which includes that header.

And by only declaring non-inline functions in a header file and defining them in a .cpp file, we ensure only one definition appears throughout the entire program.

More on Inlining and Linking

H.h

```
#ifndef _H_
#define _H_
inline void bar() { } // inline, is defined!
void baz(); // not inline, just declared: defined in B.cpp
#endif
```

B.cpp

```
#include "H.h"
// baz is defined here but nowhere else
void baz() { bar(); } // inline bar defined here because of H.h
```

Main.cpp

```
#include "H.h"
int main () { // inline bar defined here because of H.h
    bar(); baz(); return 0;
}
```

More on Inlining and Linking

The C++ standard states that *“if the inline specifier is used in a friend declaration, that declaration shall be a definition or the function shall have previously been declared inline.”*

Note that it is still necessary the definition of the inline function appear in each translational unit.

The best practices here would be to just declare the function a friend and not worry about inlining (so define it in another cpp file) or to define the function where it is declared in the class.

More on Inlining and Linking

Warning: some programmers use the **static** keyword instead of **inline** to silence odr-based errors when defining free functions (i.e. functions that are not member functions). This is not good!

It can fix the error, but **static** means something different here. It actually tells the compiler to define a separate function (with identical set of instructions) in every translational unit. This is not what the **static** keyword was meant for.

operator,

There is an **operator,** - a comma operator in C++.

It evaluates a series of expressions in sequential order and returns the rightmost expression of a list.

The comma operator has the lowest precedence of all the operators.

```
int a = 7;
int b = 6;
a = 3, b = 4; // now a==3, b==4
```

operator,

More examples:

```
int x, y, z;

x = 3, 4, 5; // x==3 but as a whole this expression equals 5

y = (x = 6, 7, 8); // x==6 and y==8, and y is returned by reference

z = (0, 4, 14); // z == 14, and z is returned by reference

x = 0, y=10, z=20; // z==0, y==10, z==20, and z is returned by reference

std::cout <<3, 0, x; /* only prints “3”, and overall this returns x
    by reference */
```

Callable Classes and Lambdas

A class with the call operator is **callable** (also called a **functor**). Such classes, including **lambdas** (unnamed callable objects), can be used as parameters for different generic algorithms.

Suppose **raceTimes** is an **std::vector**<**Time**> storing the race times of runners in a marathon. We may wish to know if any runners had a run time between 4:0:0 and 5:0:0 inclusive.

We can use the generic algorithm **std::find_if** with an appropriate predicate. There are a number of ways to do this...

Callable Classes and Lambdas

If we have defined the function

```
bool four_to_five(const Time& theTime) {
    return (theTime >= Time(4,0,0)) && (theTime <= Time(5,0,0));
}
```

Then we can write:

```
auto timeInRange = std::find_if( std::begin(raceTimes),
    std::end(raceTimes), four_to_five);

if( timeInRange != std::end(raceTimes)) {
    std::cout << "At least one time in range.";
}
else {
    std::cout << "No time in range.";
}
```

Callable Classes and Lambdas

std::find_if runs through each element of the container from **begin** to just before **end** and feeds it into the **four_to_five** function.

If the output is **true**, it returns an iterator to that position and looks no further; otherwise, it moves to the next element.

If no elements are found where the function returns **true**, it returns the past-the-end iterator.

But maybe we don't want to have to write such a precise function to begin with. Instead, we can replace the function we pass with a callable class...

Callable Classes and Lambdas

We write the class

```
struct inRange {
    Time min, max;

    inRange(const Time& _min, const Time& _max) : min(_min),
        max(_max) { }

    bool operator()(const Time& test) const {
        return (min <= test) & & (test <= max);
    }
};
```

Note how depending upon the input parameters to its constructor, this class can store any min or max time.

Its call operator is a function that can be invoked...

Callable Classes and Lambdas

Now we have more freedom about what time range we pick as we are not tied to a single function definition.

```
Time minTime = getAMinimumTime(); // spits out some lower bound
Time maxTime = getAMaximumTime(); // spits out some upper bound

auto timeInRange = std::find_if( std::begin(raceTimes),
    std::end(raceTimes), inRange(minTime, maxTime) );

if( timeInRange != std::end(raceTimes)) {
    std::cout << "At least one time in range.";
}
else {
    std::cout << "No time in range.";
}
```

Callable Classes and Lambdas

An unnamed instance of **inRange** is constructed, **inRange(minTime, maxTime)**, with the given min and max times and this object is passed to **std::find_if**.

The call operator is used as a predicate to check each time value in the **raceTimes** vector.

This gives us more freedom in the range than with writing the **four_to_five** function, but we are still restricted to consider the inclusive range of values. If instead we wanted to see if any times were above **5:0:0** or below **4:0:0**, we'd need to write another callable class or function...

Callable Classes and Lambdas

A **lambda** is an unnamed callable object that can be constructed via:

```
[capture values] (call operator arguments) ->
   call return type { call body; }
```

This parallels the construction of a class with a call operator:

```
class something{
private:
   member variables;
public:
   // CONSTRUCTOR
   something(values for variables) : /* initialize them */ {}

   // CALL OPERATOR
   call return type operator() (call operator arguments) const
      { call body }
};
```

Callable Classes and Lambdas

The capture list and call operator arguments should be comma separated but can also be empty.

The **capture values** are variables in the scope where the lambda is defined that it needs to use. Think of these as constructor parameters.

The **call arguments** are the arguments the lambda requires for its call operator: a lambda is a callable class, so we need to be able to call it!

The **return type** follows the arrow; on some compilers it is not required if the body is short enough but for robustness, it should be included!

The instructions to carry out are in the **body**.

Callable Classes and Lambdas

```
Time lower(4,0,0);
Time upper(5,0,0);

// find if there is a time below 4h or above 5h...
auto timeInRange = std::find_if( std::begin(raceTimes),
    std::end(raceTimes),
        [lower,upper](const Time& test)->bool
            { return (test < lower) || (test > upper); } );
```

The lambda needs access to **lower** and **upper** for comparison: they are unknown within its body unless they are captured!

It behaves as a predicate, taking in each time from the vector by reference and returning a **bool** as to whether the condition is upheld.

Callable Classes and Lambdas

Basically with a lambda we can give a callable class inline. This inline capacity is nice if we want to do a "one off" operation.

```
// count times above 6:0:0
auto above6count = std::count_if( std::begin(raceTimes),
    std::end(raceTimes),
        [](const Time& test)->bool
        { return test > Time(6,0,0); } );

// subtract 1 minute from each time
const Time oneMin(0,1,0);
std::for_each( std::begin(raceTimes),
    std::end(raceTimes),
        [oneMin](Time& _time)->void
        { _time -= oneMin; } );
```

Note: **std::count_if** is also found in the **<algorithm>** header. It counts items when the predicate is true.

Callable Classes and Lambdas

We can also store a lambda as a variable.

```
auto greet = []()->void { std::cout << "Hello world!"; };

greet(); // invokes the lambda object to print message
```

We can only store the lambda with the **auto** keyword because each lambda we create is created inline as its own unique type of class.

Callable Classes and Lambdas

The **capture values** are **const** by default for a lambda.

```
int n = 11;

auto bad_lam = [n]()->void { std::cout <<++n; }; // ERROR: n changed!
```

The idea is that every time a lambda is called, it should by default exhibit the same behaviour.

Callable Classes and Lambdas

We can use the **mutable** keyword after the call parameters and allow for modification of captured values. Note that the lambda will only modify its local copy of the variable.

```
int n = 11;

auto good_lam = [n]() mutable ->void { std::cout <<++n; };

good_lam(); // prints 12: its local value is 12 but n stays as 11
good_lam(); // prints 13: its local value is 13 but n stays as 11
```

Callable Classes and Lambdas

The **mutable** keyword also applies to general class member variables. Although seldom used, it allows a variable to be modified in a const member function:

```
struct Bah {
    mutable int i;

    // okay to change i because it was marked mutable
    void blaaargh() const { ++i; }
};
```

Callable Classes and Lambdas

The use of **mutable** is rare, but think, for example of the assignment operator for **std::shared_ptr**s:

```
shared_ptr &operator=(const shared_ptr&);
```

Because the internal reference count would change here, the assigned from object is modified, but it can still be marked as const for all other intents and purposes.

Callable Classes and Lambdas

Aside: it is possible to automatically capture all variables in a lambda's closure (surroundings) that are used via **=**:

```
int x = 1, y = 2;
auto lam = [=]()->void{ std::cout <<x + y; }; // okay
```

Lambdas can even capture references to arguments using **&**, but we won't discuss it.

std::function Template

We also consider the **std::function** template class.

The **std::function** template is included in the **<functional>** header. The template parameter is the **signature** of the function it should be storing. Some examples are below:

```
void g(int, double);
// stuff

std::function<void(int, double)> f = &g; // store a function
std::function<double()> f2 = []()->double{ return 0.98; }; // store a lambda
```

So a **function** object can be initialized with another function or a callable class, etc. Calling **f2()** outputs **0.98**.

In general, we refer to function pointers, lambdas, callable classes, function objects, and the likes as **functors**.

What Is Compiler Generated?

There are a number of conditions in which the compiler may or may not generate a default constructor, copy/move constructor, copy/move assignment operator, or destructor.

The compiler will not overrule the decision of the programmer, so any version of the above a programmer writes is the “official” version of the function.

What Is Compiler Generated?

These concerns are more pronounced when we consider **the big five:** destrutor, copy constuctor, copy assignment operator, move constructor, and move assignment operator.

In many classes, we don't need to write any of them, however, and can allow the compiler to generate them for us. As long as our individual member variables can be copied/moved and obey RAII, there's no need to write them!

What Is Compiler Generated?

Programmer Declares	Compiler Provides: Default Constructor	Destructor	Copy Constructor	Copy Assignment	Move Constructor	Move Assignment
Some Constructor	no	yes	yes	yes	yes	yes
Default Constructor	-	yes	yes	yes	yes	yes
Destructor	yes	-	yes	yes	no	no
Copy Constructor	no	yes	-	yes	no	no
Copy Assignment	yes	yes	yes	-	no	no
Move Constructor	no	yes	no	no	-	no
Move Assignment	yes	yes	no	no	no	-

What Is Compiler Generated?

From the table, we see that, for example, if a class has a destructor declared then move semantics are gone. But by re-enabling their default behaviour as we did in polymorphism, the copy semantics are not generated by default unless we re-enable them as well.

Assignment operator= (copy)

Recall we wrote a copy assignment for **basic::string** that involved writing all the work for the copy constructor. Here, we get around this by letting the copy constructor do its job...

```
class string {
   // stuff ...

   string& operator=(const string& that){
      if(this == &that) { // check if same object (self-assignent)
         return *this;
      }
      string copy(that); // invoke copy constructor

      std::swap(ptr, copy.ptr); // exchange resources
      std::swap(sz, copy.sz); // and sizes
      return *this;
   }
};
```

Assignment operator= (copy)

If we are not self-assigning, we make a completely separate copy of **that**. Then, we swap the resources the two manage so that ***this** manages the memory formerly held by **copy** and **copy** holds the memory formerly held by ***this**.

At the end of the function scope, **copy** is destructed thereby freeing up the memory formerly held by ***this**.

Assignment operator= (just one!)

We can be very slick and just *write a single assigment operator that manages both copy and move assignments* (in which case we would not write **string operator=(const string&)** or **string operator=(string&&)** and only write the assignment below).

```
class string {
    // stuff

    string& operator=(string that){ // store the right-hand-side

        std::swap(ptr, that.ptr); // exchange resources
        std::swap(sz, that.sz); // and sizes

        return *this;
    }
};
```

Assignment operator= (just one!)

Note that **that** is neither an lvalue nor rvalue reference.

Since the copy and move constructors are already written, if we are assigning from an lvalue then **that** will be constructed through the copy constructor, making it a separate independent copy.

If we are assigning from an rvalue then **that** will be constructed efficiently through the move constructor and it will also be a separate independent copy of the original rvalue.

We then swap the resources so that ***this** has the correct memory and value and allow **that** to be destroyed as it holds the data of ***this** we wanted to overwrite.

Assignment operator= (just one!)

The work we've done here falls under the term of the **copy and swap idiom** in C++.

With this, we don't even worry about self-assignment because it is very rare in general.

Ref-Qualifiers

There is yet one more level of control a programmer has in when member functions are invoked. We can specify that a function can only be invoked on **lvalues** or only on **rvalues** by appending an **&** or **&&**, respectively, at the end of a member function name.

```
struct Foo {
    // ...
    Foo& operator=(Foo) &; // note the extra &
};

Foo f1, f2;
f1 = f2; // okay, f1 is an lvalue

// this is pretty much never what someone wants to do, thankfully...
Foo{} = f2; // ERROR: doing assignment on rvalue!
```

Aside: std::swap

The **std::swap** function efficiently swaps the Standard C++ class types and fundamental types. It can be thought of as the templated function:

```
namespace std {
   /**
   Function swaps two objects of type T: both are lvalues
   @tparam T the type of the two objects
   @param first the first argument
   @param second the second argument
   */
   template<typename T>
   void swap(T& first, T& second) {
   T temp( std::move(first) );
   first = std::move(second);    second = std::move(temp);
   }
}
```

The **std::swap** function invokes the move constructor and move assignment operator of the types being swapped.

Aside: constexpr Pointers/References

Only **static** objects have their pointer locations determined at compile time. As such, only addresses of static objects can be constant expressions and thus pointers can only be **constexpr** if they are **nullptr** or address a static object and references can only be **constexpr** if they reference a static object.

```
constexpr int a = 6; // EVIL global, but a is static so useful as an example

int main() {
    constexpr const int *ap = &a; // okay
    constexpr const int &ar = a; // okay

    constexpr int i = 7;
    constexpr const int *ip = &i; // ERROR: &i not constant expression here
    constexpr const int &ir = i; // ERROR: needs &i
    return 0;
}
```

Aside: Unions

A **union** is a C++ class that behaves like a **struct** (everything public by default) but where only one member can be active at once. They are useful in "saving space": the member variables overlap in their bits.

```
union A {
    int i;
    double d;
};

// ...

A x;
x.i = 3; // so x.i == 3, x.b == ???
x.d = 1.7; // so x.d == 1.7, x.i == ???
```

Aside: Preprocessor Marcos

There are lots of functions we can do with the preprocessor alone with its **#define** commands, etc. **#define** defines a symbol as whatever comes next to it in a line (with some function syntax, too). Just as a few examples:

```
#define ONE 1
#define TWO 2

#define foo(x,y) ((x<y) ? y : x)

auto main() -> int { return foo(ONE, TWO); } // returns 2
```

Summary

- We can give our own meaning to operators for class objects by overloading operators.
- The **call**, **subscript**, **assignment**, and **conversion** operators must be member functions.
- For a member function, the first unspecified argument is always the class object itself.
- The **friend** keyword grants a function/class access to the member variables of the class that lists that function/class as a friend.
- **this** points to the class object itself.
- A **declaration** only tells the compiler what something is in general; the **definition** is what gives it a place in memory.
- The **One Definition Rule** must be followed for proper linking and compilation.

Summary

- There are a variety of cases when the compiler will or will not synthesize a default constructor, move/copy constructor, move/copy assignment operator, or destructor.
- A **lambda** is an unnamed callable class; callable classes can be used in generic algorithms.
- **User-defined literals** can act as shorthand.
- The **copy and swap** idiom allows for easy-to-implement copy and move assignments by copying/moving from the temporary operator argument.
- **Ref qualifiers** allow for a member function to only be invoked on lvalues or rvalues.

Exercises I

1. What is **operator overloading** and how is it useful?
2. What conventions/requirements apply for **operator overloading**?
3. On increments/decrements
   - How does the compiler differentiate the operators invoked for prefix vs postfix **++** and **--**?
   - What would happen if binary **operator++** were implemented as a free function, i.e., non-member function?

Exercises II

4. Write a **Vector3D** class, making use of **constexpr**, that behaves like mathematical vectors. It should:
   - store 3 **double**s: **x**, **y**, **z**;
   - have a default constructor setting all members to 0;
   - have a constructor accepting a value for **x**, **y**, and **z**;
   - overload **unary+** and **unary-** to return a copy of the object or one where all its entries are negated, respectively;
   - overload **operator+=** and **operator-=** to update the left argument by adding/subtracting the corresponding values from the right;
   - overload binary **+** and -;
   - overload $<<$ so that a stream can print the class in the format: "<x, y, z>";
   - overload $>>$ so that 3 consecutive value can be read into **x**, **y**, and **z** from an input stream; and
   - overload the subscript operator (on const) to accept a char 'x', 'y', or 'z' and return those values.

Exercises III

5. Write **Newtons** and **Pounds** classes, making use of **constexpr**, that store a **double** representing an object's weight in N or lbs, respectively. Note that 1 N = 0.225 lbs. Each class should:
   - have a constructor accepting the value for its weight;
   - have a conversion to **std::string** operator producing strings such as "3.8 N" or "4.44 lbs";
   - have a conversion operator to the other class: convert a **Newtons** object to a **Pounds** object; and
   - have a user-defined literal such that **_N** and **_lbs**, when suffixed after a **long double** create a **Newtons** or **Pounds** class out of those numbers;
   - overload the binary and unary versions of **++** and **--** to increment/decrement by 1 N/lb; and
   - overload all the comparison operators **<**, **>**, **<=**, **>=**, **==**, and **!=**.

   Test the classes by creating a **std::vector** storing either **Newtons** or **Pounds** objects and adding a bunch of values to it, using user-define literals and objects of the other type (so conversions can take place). Then **std::sort** it or use **std::find** to look for a value.

Exercises IV

6. The **std::accumulate** function is a generic algorithm. Its job is to "add" up all the elements of a range of values and give the "sum". Suppose the sum we desire is of type **S** and all the elements in the range are of type **E**. We can add up all the elements provided there is some meaningful way to add an **S** and an **E** to get back an **S**.
   The function can accept 4 arguments:
   (i) a starting iterator;
   (ii) a past-end iterator;
   (iii) an initial value of the sum (of type **S**); and
   (iv) a callable class that tells us how to add an **S** on the left and an **E** on the right to get an **S**.
   Use **std::accumulate** to add up all values in a **std::vector**<**Vector3D**> by making use of:
   - a free function
   - a class type
   - a lambda
   - a **std::function**

Exercises V

7.
- Explain the **one definition rule** and how it relates to variables, functions, and classes.
- Now explain why *header guards* are necessary in large programs.
- Consider defining member functions outside of the class interface but within the same header file as the class is defined. When is it okay / not okay to do so?
- Why must **constexpr** functions be defined in header files?
- Besides the fact that global variables are generally bad, is it generally safe to *define* a global variable in a header file?
- Explain why the code below fails to **compile** and fix it.

```
 struct A {
  B b;
  A() { foo(); }
};
struct B { A* ap = nullptr; };
inline int foo() { B(); return x; }
extern int x;
int main() { A a; return 0; }
int x = 4;
```