1 Operator Overloading

1.1 Some concepts

1.1.1 Unary and Binary

Operators can be either unary or binary. A unary operator has one parameter, while a binary
operator has two. If an operator is a member function, then implicitly its first parameter is the this
pointer to the object owning the member function. For example, the member function size of the
std::string class has the function prototype size_t std::string::size() const but can be thought
of as having the non-member function prototype size_t std::string::size(const std::string * this)
and a call to size of the form str.size() can be thought of as the call std::string::size(&str).

For an example of a binary operator member function, consider the function
void std::vector<T>::resize(size_t). This function can be thought of as having the non-member
function prototype void resize(std::vector<T> * this, size_t); calls of the form vec.resize(10) can
be thought of as calls resize(&vec,10).

1.1.2 Member or non-member

In many cases, it is possible to implement the same function as either a member or as a non-member
function. For example, the binary operator+ for a Complex number class could have either of the
following definitions:

Complex Complex::operator+(const Complex& rhs) const {
    Complex copy(*this);
    return copy += rhs;
}
Complex operator+(Complex lhs, const Complex& rhs) {
    return lhs += rhs;
}

However, it is preferable to use the second implementation. As a general rule, operators will be
made member functions when they have to be, which includes the following operators

1. subscript: operator[]
2. call: operator()
3. assignment: operator=
4. conversion

5. arrow: \texttt{operator\textasciitilde\textasciitilde}

when they are unary (except for literals), or when they are asymmetric. Asymmetric here means that exactly one of the parameters is modified, whereas a symmetric operator would not modify either of the two parameters or would modify both. Binary operators that are symmetric are implemented as non-member functions. Indeed, operator+ above is symmetric and binary, which would guide one to make it a non-member function.

1.2 \texttt{constexpr}

Functions/variables with the modifier \texttt{constexpr} are evaluated/allocated at compile-time rather than at run-time. The parameters and return values of \texttt{constexpr} functions must be of \texttt{LiteralType}. A variable is of \texttt{LiteralType} if it is a literal or if it is a \texttt{constexpr}. If a \texttt{constexpr} function is passed non-\texttt{LiteralType} parameters, then the function will be evaluated at runtime rather than at compile-time. Further information on \texttt{constexpr} functions can be found online; in particular, the body of a \texttt{constexpr} function cannot contain definitions of non-\texttt{constexpr} variables or contain try-blocks.

Remember that \texttt{constexpr} member functions, including constructors, and \texttt{constexpr} members must be defined inside of the class interface.

1.3 Summary of operators

We go over the common operators, indicating what they do, how they are evaluated, whether they should be members or non-members, what their return types and parameter types should be. To illustrate these operators, we consider a Vector2D class, with the member variables $x$ and $y$ holding the $x$- and $y$-coordinates respectively. First, we consider the constructor:

\begin{verbatim}
constexpr Vector2D (const double \_x = 0, const double \_y = 0) : x (\_x), y (\_y) {}
\end{verbatim}

The above constructor will be called as a \texttt{constexpr} only when the Vector2D object being created is marked as a \texttt{constexpr} and when the parameters passed in are of \texttt{LiteralType}:

\begin{verbatim}
int main() {
    constexpr double x = 1.5, y = 20.1;
    constexpr Vector2D vec(x, y); // evaluated at compile-time
    constexpr Vector2D vec1(1.4, 2.6); // evaluated at compile-time
    double w = 1.2;
    // constexpr Vector2D vec2(w, y); // ERROR: w not a constexpr
    Vector2D vec2(w, y); // OK: evaluated at run-time.
}
\end{verbatim}

Now we consider the other operators:
1. The unary `operator−` is a member function because it is unary. It is a constexpr because it will only call the constexpr constructor of the `2DVector`.

```cpp
constexpr Vector2D operator−() const {
    return Vector2D(−x, −y);
}
```

2. `operator+=` is binary and asymmetric, so should be implemented as a non-member function. It cannot be made constexpr because it modifies the object, which if made constexpr is necessarily const, and thus, cannot be modified.

```cpp
Vector2D& operator+=(const Vector2D& rhs) {
    x+=rhs.x;
    y+=rhs.y;
    return *this;
}
```

The return type of `operator+=` is a reference; indeed, expressions such as `(vec+=vec1)+=vec2` should modify `vec` to have the new `x` value `vec.x+vec1.x+vec2.x` and similarly for `y`. This is also why we return `*this`. `operator+=` is evaluated from right-to-left, so the line `vec+=vec1+=vec2` would be evaluated as `vec+=(vec1+(vec1+=vec2))` by default.

3. The unary `operator−−`, also called the prefix `operator−−`, is a member function because it is unary. It cannot be constexpr because it modifies the object.

```cpp
Vector2D& operator−−() {
    −−x;    −−y;
    return *this;
}
```

Expressions such as `−−−−−−vec` should subtract 3 from the vector `vec`; which is one reason we return by reference.

4. The binary `operator−−`, also called the postfix `operator−−`, is a member function because it is asymmetric. It cannot be constexpr for the usual reason.

```cpp
Vector2D operator−−(int) {
    Vector2D copy(*this);
    −−x;    −−y;
    return copy;
}
```

This operator should return what the object looked like before decrementing. Accordingly, a copy is created to hold the old data and then a copy is returned. It is necessary to return by value because the copy is a local member of the function.

5. The conversion `operator double` is a member function since it is a conversion operator. It can be constexpr because it does not try to modify any variables, and it only creates variables of `LiteralType`.

```cpp
constexpr operator double() const {
    return x*x + y*y;
}
```
We can make it so that this conversion operator cannot be called implicitly by adding the explicit modifier.

6. The user defined literal operator `'` _vec is a non-member function as an exception to the unary rule. It is a constexpr because it will only call the constexpr constructor of Vector2D. It should convert a double to a Vector2D with _y = 0:

```cpp
constexpr Vector2D operator `'` _vec(const double _x) {
    return Vector2D(_x);
}
```

Thus, the code 5.5 _vec will return a Vector2D object with _x = 5.5 and _y = 0.

7. operator< is a non-member friend function (if we defined the subscript operator, it would not be necessary to make it a friend). It is a constexpr. It is defined

```cpp
constexpr bool operator< (const Vector2D& lhs, const Vector2D& rhs) {
    if (lhs.x < rhs.x) {
        return true;
    } else if (lhs.x == rhs.x) {
        return lhs.y < rhs.y;
    }
    return false;
}
```

where we use the “lexicographical” ordering.

8. The other comparison operators are constexpr and are defined using the fact that !(a<b) && !(b<a) implies a==b.

9. operator<< is a non-member function because it is asymmetric and cannot be constexpr because the ostream object is not of LiteralType. It may need to be a friend function, which it does in our case. It is defined

```cpp
std::ostream& operator<< (std::ostream& os, const Vector2D& vec) {
    os << vec.x << vec.y;
    return os;
}
```

10. A constructor accepting an std::istream object would look like

```cpp
Vector2D::Vector2D (std::istream& is) : x(std::numeric_limits<double>::max()), y(is >> x >> y);
    if (!is) {
        throw std::runtime_error("bad format");
    }
```