User-defined data types

Using `struct` in C++, we can define a new data-type as collection of variables of different types. For example, the following `struct` defines a `student` type with three variables: `name` of type `string`, `age` of type `int`, and `gpa` of type `double`.

```
struct student {
    string name;
    int age;
    double gpa;
};
```

The above define a new type `student`, which can be used just like any other type. We can then declare variables of type `student` and access their fields as follows:

```
student s1, s2;
s1.name = "Ali";
s1.age = 20;
s1.gpa = 3.5;
s2.name = "Ahmed";
s2.age = 21;
s2.gpa = 3.2;
```

A initialization list can be used to initialize the fields of a `struct`:

```
student s1 = {"Ali", 20, 3.5};
student s2 {"Ahmed", 21, 3.2}; // = sign is optional since C++11
student s3 {"Na-laiq", 20}; // s3.gpa will be 0.0 (default value for double)
```

Since C++20, *designated initializers* can be used to initialize specific fields of a `struct`:

```
student s1 {.name = "Ali", .age = 20, .gpa = 3.5};
student s2 {.name {"Ahmed"}, .age {21}, .gpa {3.2} }; // can also use braces
                                                      // instead of = sign
student s3 {.name = "Prodigy", .gpa = 3.7}; // s3.age will be 0
student s4 {.gpa = 2.2, .name = "Nobody"}; // error, designator order must
                                           // match declaration order
```

Passing `struct` to functions

Just like other data types, a `struct` can be passed to functions by value or by reference. When passed by value, a copy of the `struct` is made and passed to the function. When passed by reference, the function receives a reference to the original `struct` and can modify it.

```
// pass by value
void print(student s) {
    cout << s.name << " " << s.age << " " << s.gpa << endl;
```

```
}
// pass by reference
void print(const student& s) {
    cout << s.name << " " << s.age << " " << s.gpa << endl;
}
```

Returning `struct` from functions

`struct` can be returned from functions. The following function returns a `student` object:

```
student create_student(string name, int age, double gpa) {
    student s;
    s.name = name;
    s.age = age;
    s.gpa = gpa;
    return s;
}
```

Lab exercises

A data type for (mathematical) vectors in 2D

We will implement a 2D-vector data-type using `struct` and implement some functions to implement operations on 2D-vectors. The `struct` definition is given below:

```
struct vector2D {
    double x;
    double y;
};
```

We will implement the following operations on 2D-vectors:

Function	Description	Usage example
`print`	prints a vector	`print(v1)`
`add`	returns the sum of two vectors	`vector2D v3 = add(v1, v2)`
`operator+`	returns the sum of two vectors	`vector2D v3 = v1 + v2`
`operator*`	returns the scalar product of a vector and a double	`vector2D v3 = v1 * 2.0`
`operator*=`	updates the vector with its scalar product with a double	`v1 *= 2.0`
`dot`	returns the dot product of two vectors	`double d = dot(v1, v2)`

Implement the above operations on 2D-vectors and test them with following code:

```
int main() {
    vector2D v1 {1.2, 3.4};
```

```
    vector2D v2 {5.6, 7.8};
    print(v1); cout << '\n';
    print(v2); cout << '\n';
    vector2D v3 = add(v1, v2);
    print(v3); cout << '\n';
    vector2D v4 = v1 + v2;
    print(v4); cout << '\n';
    vector2D v5 = v1 * 2.0;
    print(v5); cout << '\n';
    v1 *= 2.0;
    print(v1); cout << '\n';
    double d = dot(v1, v2);
    cout << d << '\n';
}
```

The above code should print the following:

```
(1.2, 3.4)
(5.6, 7.8)
(6.8, 11.2)
(6.8, 11.2)
(2.4, 6.8)
(2.4, 6.8)
66.48
```

The following exercises will help you implement these operations.

Exercise 1

Create a function that takes a `vector2D` as input and prints its fields. The function should be declared as follows:

```
void print(const vector2D& v);
```

The function should print the vector in the following format:

```
(1.2, 3.4)
```

where the first number is the value of `x` field and the second number is the value of `y` field.

Exercise 2

Write a function that takes two `vector2D` as input and returns the sum of the two vectors. The function should be declared as follows:

```
vector2D add(const vector2D& v1, const vector2D& v2);
```

The function should return a new vector whose `x` field is the sum of the `x` fields of the two input vectors, and `y` field is the sum of the `y` fields of the two input vectors.

Exercise 3

Overloading + operator for `vector2D`.

In C++, we can also use + operator to add two `vector2D` values, if we use the following signature:

```
vector2D operator+(const vector2D& v1, const vector2D& v2);
```

This way we can use the + operator to add two vectors as follows:

```
vector2D v1 {1.2, 3.4};
vector2D v2 {5.6, 7.8};
vector2D v3 = v1 + v2;
```

Exercise 4

*Scalar multiplication using * operator.*

Write a function that takes a `vector2D` and a `double` as input and returns a new vector whose `x` and `y` fields are multiplied by the input `double`. The function should be declared as follows:

```
vector2D operator*(const vector2D& v, double s);
```

The function should return a new vector whose `x` field is the product of the `x` field of the input vector and the input `double`, and `y` field is the product of the `y` field of the input vector and the input `double`.

Exercise 5

*Overloading *= operator for updating the `vector2D` with its scalar product.*

Write a function with the following signature:

```
void operator*=(vector2D& v, double s);
```

Unlike the previous exercise, this function should update the input vector with its scalar product. For example, the following code should update `v1` with its scalar product with 2.0:

```
vector2D v1 {1.2, 3.4};
v1 *= 2.0;
```

Exercise 6

Write a function that takes two `vector2D` as input and returns the dot product of the two vectors. The function should be declared as follows:

```
double dot(const vector2D& v1, const vector2D& v2);
```

The dot product of two vectors is defined as follows:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = x_1 \times x_2 + y_1 \times y_2$$

A data type for fractional numbers

Implement a data-type to handle fraction values. A fraction value is a number of the form $\frac{p}{q}$ where p and q are integers and $q \neq 0$.

The following `struct` holds the the two fields of a fraction: `num` and `denom` to represent the numerator and denominator of the fraction.

```
struct fraction {
    int num;
    int denom;
};
```

We will implement the following operations on fractions.

Function	Description
`operator«`	prints a fraction value
`simplify`	simplifies a fraction
`operator+`	returns the sum of two fractions
`operator+=`	updates the first fraction with the sum of the two fractions
`operator*`	returns the product of two fractions
`operator*=`	updates the first fraction with the product of the two fractions
`equal`	returns `true` if two fractions are equal, and `false` otherwise

After implementing the above operations, the following code should work:

```
int main() {
    fraction f1 {1, 2};
    cout << f1 << endl; // prints 1/2

    fraction f2 {3, 4};
    f2 += f1;
    cout << f2 << endl; // should prints 5/4

    fraction f3 = f1 * f2;
    cout << f3 << endl; // should prints 5/8

    fraction f4 {1, 3};
    f3 *= (f1 * f2);
    cout << f3 << endl; // should prints 25/64

    fraction f5 {2, 4}; // same as f1
    if (f1 == f5)
        cout << "f1 and f5 are equal" << endl;
    else
```

```
            cout << "f1 and f5 are not equal" << endl;
        // the above should print "f1 and f5 are equal"
    }
```

First, we overload « operator for `fraction` type to be able to print a fraction using `cout`. As `cout` is an object of type `ostream`, we need to overload the « operator for `ostream`. The overloaded « operator is defined as follows:

```
ostream& operator<<(ostream& os, const fraction& f) {
    os << f.num << "/" << f.denom;
    return os;
}
```

Using the above definition, we can print a fraction value as follows:

```
fraction f {1, 2};
cout << f << endl; // prints 1/2
```

Define the following functions to implement more operations on `fraction` values:

Exercise 7 .....

Overloading + operator for `fraction`.

Write a function that takes two `fraction` as input and returns the sum of the two fractions. The sum should be a fraction in its simplest form. For example, the sum of $\frac{1}{2}$ and $\frac{1}{4}$ is $\frac{3}{4}$.

The function should be declared as follows:

```
fraction operator+(const fraction& f1, const fraction& f2);
```

This way we can use the + operator to add two fractions as follows:

```
fraction f1 {1, 2};
fraction f2 {3, 4};
fraction f3 = f1 + f2;
cout << f3 << endl; // should prints 5/4
```

Exercise 8 .....

Overloading += operator for `fraction`.

Write a function that takes two `fraction` as input and updates the first fraction with the sum of the two fractions.

The function should be declared as follows:

```
void operator+=(fraction& f1, const fraction& f2);
```

This way we can use the += operator to add two fractions as follows:

```
fraction f1 {1, 2};
fraction f2 {3, 4};
f1 += f2;
cout << f1 << endl; // should prints 5/4
```

Exercise 9

*Overloading * operator for* `fraction`.

Write a function that takes two `fraction` as input and returns the product of the two fractions. The product should be a fraction in its simplest form. For example, the product of $\frac{1}{2}$ and $\frac{1}{4}$ is $\frac{1}{8}$.

The function should be declared as follows:

```
fraction operator*(const fraction& f1, const fraction& f2);
```

This way we can use the * operator to multiply two fractions as follows:

```
fraction f1 {1, 2};
fraction f2 {3, 4};
fraction f3 = f1 * f2;
cout << f3 << endl; // should prints 3/8
```

Exercise 10

*Overloading *= operator for* `fraction`.

Write a function that takes two `fraction` as input and updates the first fraction with the product of the two fractions.

The function should be declared as follows:

```
void operator*=(fraction& f1, const fraction& f2);
```

This way we can use the *= operator to multiply two fractions as follows:

```
fraction f1 {1, 2};
fraction f2 {3, 4};
f1 *= f2;
cout << f1 << endl; // should prints 3/8
```

Exercise 11

Overloading == operator for `fraction`.

Write a function that takes two `fraction` as input and returns `true` if the two fractions are equal, and `false` otherwise.

The function should be declared as follows:

```
bool operator==(const fraction& f1, const fraction& f2);
```

This way we can use the == operator to compare two fractions as follows:

```
fraction f1 {1, 2};
fraction f2 {2, 4}; // same as f1
if (f1 == f2)
    cout << "f1 and f2 are equal" << endl;
else
    cout << "f1 and f2 are not equal" << endl;
// the above should print "f1 and f2 are equal"
```