More on classes

The hidden “this” pointer and member function chaining

Returning *this

If we make each function return *this by reference, we can chain the calls together. Here is the new version of Calc with “chainable” functions:

```
class Calc
{
private:
    int m_value{};

public:
    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() const { return m_value; }
};
```

Note that add(), sub() and mult() are now returning *this by reference. Consequently, this allows us to do the following:

```
calc.add(5).sub(3).mult(4); // method chaining
```

Resetting a class back to default state

```
void reset()
{
    *this = {}; // value initialize a new object and overwrite the implicit
object
}
```

```
class Calc
{
private:
    int m_value{};

public:
    Calc& add(int value) { m_value += value; return *this; }
```

```
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() const { return m_value; }

    void reset() { *this = {}; }
};
```

Classes and header files

#include

```
class Date
{
private:
    int m_year{};
    int m_month{};
    int m_day{};

public:
    Date(int year, int month, int day); // constructor declaration

    void print() const; // print function declaration

    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const  { return m_day; }
};

Date::Date(int year, int month, int day) // constructor definition
    : m_year{ year }
    , m_month{ month }
    , m_day{ day }
{
}

void Date::print() const // print function definition
{
    std::cout << "Date(" << m_year << ", " << m_month << ", " << m_day <<
")\n";
};

int main()
```

```
{
    const Date d{ 2015, 10, 14 };
    d.print();

    return 0;
}
```

Best practise: Prefer to put your class definitions in a header file with the same name as the class. Trivial member functions (such as access functions, constructors with empty bodies, etc…) can be defined inside the class definition. Prefer to define non-trivial member functions in a source file with the same name as the class.

Nested types (member types)

Best practice: Define any nested types at the top of your class type.

```
#include <iostream>

class Fruit
{
public:
    // FruitType has been moved inside the class, under the public access specifier
        // We've also renamed it Type and made it an enum rather than an enum class
    enum Type
    {
        apple,
        banana,
        cherry
    };

private:
    Type m_type {};
    int m_percentageEaten { 0 };

public:
    Fruit(Type type) :
        m_type { type }
    {
    }
```

```
    Type getType() { return m_type;  }
    int getPercentageEaten() { return m_percentageEaten;  }

    bool isCherry() { return m_type == cherry; } // Inside members of Fruit, we no longer need to prefix enumerators with FruitType::
};

int main()
{
    // Note: Outside the class, we access the enumerators via the Fruit:: prefix now
    Fruit apple { Fruit::apple };

    if (apple.getType() == Fruit::apple)
        std::cout << "I am an apple";
    else
        std::cout << "I am not an apple";

    return 0;
}
```

User-defined conversions

```
#include <iostream>

class Foo
{
private:
    int m_x{};
public:
    Foo(int x)
        : m_x{ x }
    {
    }

    int getX() const { return m_x; }
};

void printFoo(Foo f) // has a Foo parameter
{
    std::cout << f.getX();
}
```

```
int main()
{
    printFoo(5); // we're supplying an int argument

    return 0;
}
```

In this version, printFoo has a Foo parameter but we're passing in an argument of type int. Because these types do not match, the compiler will try to implicitly convert int value 5 into a Foo object so the function can be called.

Unlike the first example, where our parameter and argument types were both fundamental types (and thus can be converted using the built-in numeric promotion/conversion rules), in this case, one of our types is a program-defined type. The C++ standard doesn't have specific rules that tell the compiler how to convert values to (or from) a program-defined type.

Instead, the compiler will look to see if we have defined some function that it can use to perform such a conversion. Such a function is called a user-defined conversion.

Nested typedefs and type aliases

```
public:
    using IDType = int;

private:
    std::string m_name{};
    IDType m_id{};
    double m_wage{};
```

Nested classes and access to outer class members

```
#include <iostream>
#include <string>
#include <string_view>

class Employee
{
public:
    using IDType = int;

    class Printer
    {
    public:
```

```
        void print(const Employee& e) const
        {
            // Printer can't access Employee's `this` pointer
            // so we can't print m_name and m_id directly
            // Instead, we have to pass in an Employee object to use
            // Because Printer is a member of Employee,
            // we can access private members e.m_name and e.m_id directly
            std::cout << e.m_name << " has id: " << e.m_id << '\n';
        }
    };

private:
    std::string m_name{};
    IDType m_id{};
    double m_wage{};

public:
    Employee(std::string_view name, IDType id, double wage)
        : m_name{ name }
        , m_id{ id }
        , m_wage{ wage }
    {
    }

    // removed the access functions in this example (since they aren't used)
};

int main()
{
    const Employee john{ "John", 1, 45000 };
    const Employee::Printer p{}; // instantiate an object of the inner class
    p.print(john);

    return 0;
}
```

Destructors

Destructor naming

```
#include <iostream>

class Simple
```

```
{
private:
    int m_id {};

public:
    Simple(int id)
        : m_id { id }
    {
        std::cout << "Constructing Simple " << m_id << '\n';
    }

    ~Simple() // here's our destructor
    {
        std::cout << "Destructing Simple " << m_id << '\n';
    }

    int getID() const { return m_id; }
};

int main()
{
    // Allocate a Simple
    Simple simple1{ 1 };
    {
        Simple simple2{ 2 };
    } // simple2 dies here

    return 0;
} // simple1 dies here
```

Class templates with member functions

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

// Here's a deduction guide for our Pair (required in C++17 or older)
// Pair objects initialized with arguments of type T and T should deduce to
```

```
Pair<T>
template <typename T>
Pair(T, T) -> Pair<T>;

int main()
{
    Pair<int> p1{ 5, 6 };        // instantiates Pair<int> and creates
object p1
    std::cout << p1.first << ' ' << p1.second << '\n';

    Pair<double> p2{ 1.2, 3.4 }; // instantiates Pair<double> and creates
object p2
    std::cout << p2.first << ' ' << p2.second << '\n';

    Pair<double> p3{ 7.8, 9.0 }; // creates object p3 using prior definition
for Pair<double>
    std::cout << p3.first << ' ' << p3.second << '\n';

    return 0;
}
```

Static member variables

```
#include <iostream>

int generateID()
{
    static int s_id{ 0 }; // static local variable
    return ++s_id;
}

int main()
{
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';

    return 0;
}
```

This program prints:

1
2

Friend non-member functions

```
#include <iostream>

class Accumulator
{
private:
    int m_value { 0 };

public:
    void add(int value) { m_value += value; }

    // Here is the friend declaration that makes non-member function void print(const Accumulator& accumulator) a friend of Accumulator
    friend void print(const Accumulator& accumulator);
};

void print(const Accumulator& accumulator)
{
    // Because print() is a friend of Accumulator
    // it can access the private members of Accumulator
    std::cout << accumulator.m_value;
}

int main()
{
    Accumulator acc{};
    acc.add(5); // add 5 to the accumulator

    print(acc); // call the print() non-member function

    return 0;
}
```

Defining a friend non-member inside a class

```
#include <iostream>

class Accumulator
{
private:
    int m_value { 0 };
```

```
public:
    void add(int value) { m_value += value; }

    // Friend functions defined inside a class are non-member functions
    friend void print(const Accumulator& accumulator)
    {
        // Because print() is a friend of Accumulator
        // it can access the private members of Accumulator
        std::cout << accumulator.m_value;
    }
};

int main()
{
    Accumulator acc{};
    acc.add(5); // add 5 to the accumulator

    print(acc); // call the print() non-member function

    return 0;
}
```

Multiple friends

```
#include <iostream>

class Humidity; // forward declaration of Humidity

class Temperature
{
private:
    int m_temp { 0 };
public:
    explicit Temperature(int temp) : m_temp { temp } { }

    friend void printWeather(const Temperature& temperature, const Humidity&
humidity); // forward declaration needed for this line
};

class Humidity
{
private:
```

```
    int m_humidity { 0 };
public:
    explicit Humidity(int humidity) : m_humidity { humidity } {  }

    friend void printWeather(const Temperature& temperature, const Humidity&
humidity);
};

void printWeather(const Temperature& temperature, const Humidity& humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
       " and the humidity is " << humidity.m_humidity << '\n';
}

int main()
{
    Humidity hum { 10 };
    Temperature temp { 12 };

    printWeather(temp, hum);

    return 0;
}
```

Best practise:1) A friend function should prefer to use the class interface over direct access whenever possible. 2)Prefer to implement a function as a non-friend when possible and reasonable.

Friend classes and friend member functions

A friend class is a class that can access the private and protected members of another class.

```
#include <iostream>

class Storage
{
private:
    int m_nValue {};
    double m_dValue {};
public:
    Storage(int nValue, double dValue)
       : m_nValue { nValue }, m_dValue { dValue }
```

```
    { }

    // Make the Display class a friend of Storage
    friend class Display;
};

class Display
{
private:
    bool m_displayIntFirst {};

public:
    Display(bool displayIntFirst)
         : m_displayIntFirst { displayIntFirst }
    {
    }

    // Because Display is a friend of Storage, Display members can access
the private members of Storage
    void displayStorage(const Storage& storage)
    {
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue <<
'\n';
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue <<
'\n';
    }

    void setDisplayIntFirst(bool b)
    {
         m_displayIntFirst = b;
    }
};

int main()
{
    Storage storage { 5, 6.7 };
    Display display { false };

    display.displayStorage(storage);

    display.setDisplayIntFirst(true);
```

```
    display.displayStorage(storage);

    return 0;
}
```

1. Friendship is not reciprocal. Just because Display is a friend of Storage does not mean Storage is also a friend of Display. If you want two classes to be friends of each other, both must declare the other as a friend.
2. Nor is friendship inherited. If class A makes B a friend, classes derived from B are not friends of A.