Introduction to classes

Returning data members by lvalue reference

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

class Employee
{
    std::string m_name{};

public:
    void setName(std::string_view name) { m_name = name; }
    const std::string& getName() const { return m_name; } //  getter returns
by const reference
};

int main()
{
    Employee joe{}; // joe exists until end of function
    joe.setName("Joe");

    std::cout << joe.getName(); // returns joe.m_name by reference

    return 0;
}
```

Now when joe.getName() is invoked, joe.m_name is returned by reference to the caller, avoiding having to make a copy. The caller then uses this reference to print joe.m_name to the console.

An rvalue object is destroyed at the end of the full expression in which it is created. Any references to members of the rvalue object are left dangling at that point.

A reference to a member of an rvalue object can only be safely used within the full expression where the rvalue object is created.

Using member functions that return by reference safely

- Prefer to use the return value of a member function that returns by reference immediately (illustrated in case 1). Since this works with both lvalue and rvalue objects, if you always do this, you will avoid trouble.

- Do not “save” a returned reference to use later (illustrated in case 2), unless you are sure the implicit object is an lvalue. If you do this with an rvalue implicit object, undefined behavior will result when you use the now-dangling reference.
- If you do need to persist a returned reference for use later and aren’t sure that the implicit object is an lvalue, using the returned reference as the initializer for a non-reference local variable, which will make a copy of the member being referenced into the local variable (illustrated in case 3).

Declare public members first, protected members next, and private members last. This spotlights the public interface and de-emphasizes implementation details.

```cpp
#include <iostream>
#include <string>

class Yogurt
{
    std::string m_flavor{ "vanilla" };

public:
    void setFlavor(std::string_view flavor)
    {
        m_flavor = flavor;
    }

    const std::string& getFlavor() const { return m_flavor; }
};

// Best: non-member function print() is not part of the class interface
void print(const Yogurt& y)
{
        std::cout << "The yogurt has flavor " << y.getFlavor() << '\n';
}

int main()
{
    Yogurt y{};
    y.setFlavor("cherry");
    print(y);

    return 0;
}
```

Delegating constructors

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

class Employee
{
private:
    std::string m_name{};
    int m_id{ 0 };

public:
    Employee(std::string_view name)
        : Employee{ name, 0 } // delegate initialization to
Employee(std::string_view, int) constructor
    {
    }

    Employee(std::string_view name, int id)
        : m_name{ name }, m_id{ id } // actually initializes the members
    {
        std::cout << "Employee " << m_name << " created\n";
    }

};

int main()
{
    Employee e1{ "James" };
    Employee e2{ "Dave", 42 };
}
```

Reducing constructors using default arguments

```cpp
Employee(std::string_view name, int id = 0) // default argument for id
        : m_name{ name }, m_id{ id }
```

Best practise: Members for which the user must provide initialization values should be defined first (and as the leftmost parameters of the constructor). Members for which the user can optionally provide

initialization values (because the default values are acceptable) should be defined second (and as the rightmost parameters of the constructor).

Copy constructor

```
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
        : m_numerator{numerator}, m_denominator{denominator}
    {
    }

    // Copy constructor
    Fraction(const Fraction& fraction)
        // Initialize our members using the corresponding member of the
parameter
        : m_numerator{ fraction.m_numerator }
        , m_denominator{ fraction.m_denominator }
    {
        std::cout << "Copy constructor called\n"; // just to prove it works
    }

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator <<
")\n";
    }
};

int main()
{
    Fraction f { 5, 3 };  // Calls Fraction(int, int) constructor
    Fraction fCopy { f }; // Calls Fraction(const Fraction&) copy
constructor
```

```
    f.print();
    fCopy.print();

    return 0;
}
```