Divisor:

```
#include <iostream>
#include <list>

using namespace std;

// Forward declaration of Subject
class Subject;

// Parent Observer
class Observer
{
private:
    Subject *model;
    int denominator;

public:
    Subject *getSubject()
    {
        return model;
    }
    int getDenominator()
    {
        return denominator;
    }

    virtual void notify() = 0;

    Observer(Subject *mod, int divisor);
};

// Concrete Subject or Publisher
class Subject
{
public:
    void setValue(int val)
    {
        value = val;
        notifyObservers();
    }
    int getValue()
    {
        return value;
    }
```

```
    void registerObserver(Observer *observer)
    {
        observers_.push_back(observer);
    }

    void unregisterObserver(Observer *observer)
    {
        observers_.remove(observer);
    }

    void notifyObservers()
    {
        for (auto &observer : observers_)
        {
            observer->notify();
        }
    }

private:
    list<Observer *> observers_;
    int value;
};

Observer::Observer(Subject *mod, int divisor)
{
    model = mod;
    denominator = divisor;
    // Now this is valid because Subject is fully defined
    model->registerObserver(this);
}

// Child Observer 1
class DivObserver : public Observer
{
public:
    DivObserver(Subject *mod, int divisor) : Observer(mod, divisor) {}

    void notify() override
    {
        int value = getSubject()->getValue();
        int d = getDenominator();
        cout << value << " division " << d << " is " << value / d << endl;
    }
};
```

```
class ModObserver : public Observer
{
public:
   ModObserver(Subject *mod, int divisor) : Observer(mod, divisor) {}

   void notify() override
   {
      int value = getSubject()->getValue();
      int d = getDenominator();
      cout << value << " mod " << d << " is " << value % d << endl;
   }
};

int main()
{
   Subject subject;
   DivObserver divObj1(&subject, 4);
   ModObserver modObj1(&subject, 3);
   subject.setValue(14);
}
```

Messenger:

```
#include "Friends.h"

Friends ::Friends(const string &name) : name_(name) {}

void Friends ::update(const string &newMessage)
{
   cout << name_ << " received a message: " << newMessage << endl;
}
#include "iObserver.h"
#include <string>
#include <iostream>
using namespace std;

class Friends : public IObserver
{
public:
   Friends(const string &name);
   void update(const string &newMessage) override;

private:
   string name_;
```

```
};
#include "groupChat.h"

void GroupChat::subscribe(IObserver *observer)
{
   observers_.push_back(observer);
}

void GroupChat::unsubscribe(IObserver *observer)
{
   auto it = find(observers_.begin(), observers_.end(), observer);
   if (it != observers_.end())
   {
      observers_.erase(it);
   }
}
void GroupChat::notify()
{
   for (auto &observer : observers_)
   {
      observer->update(messages.back());
   }
}

void GroupChat::sendMessage(const string &msg)
{
   messages.push_back(msg);
   notify();
}
#include "iSubject.h"
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

// Concrete Subject
class GroupChat : public ISubject
{
public:
   void subscribe(IObserver *observer) override;
   void unsubscribe(IObserver *observer) override;
   void notify() override;

   void sendMessage(const string &msg);
```

```
private:
    vector<IObserver *> observers_;
    vector<string> messages;
};
#pragma once
#include <string>
using namespace std;

// Abstract class or Interface
class IObserver
{
public:
    virtual ~IObserver() = default;
    virtual void update(const string &newMessage) = 0;
};
#pragma once
#include "iObserver.h"

// Abstract Subject Or Interface
class ISubject
{
public:
    virtual ~ISubject() = default;
    virtual void subscribe(IObserver *observer) = 0;
    virtual void unsubscribe(IObserver *observer) = 0;
    virtual void notify() = 0;
};
#include <iostream>
#include <string>
#include "groupChat.cpp"
#include "Friends.cpp"
using namespace std;

int main()
{
    GroupChat chat;

    Friends friend1("Alice");
    Friends friend2("Bob");
    Friends friend3("Charlie");

    chat.subscribe(&friend1);
    chat.subscribe(&friend2);
```

```
    chat.subscribe(&friend3);

    chat.sendMessage("Hey Everyone, let's plan our trip!");
}
```

WeatherStationClient

```
#include "Client.h"
#include <iostream>
using namespace std;

Client ::Client(int id) : id_(id) {}

void Client ::update(float temp, float humidity, float pressure)
{
    // print the changed values
    cout << "---Client (" << id_ << ") Data---\tTemperature: "
         << temp << "\thumidity: " << humidity
         << "\tPressure: " << pressure << endl;
}
#ifndef CLIENT_H
#define CLIENT_H
#include "Observer.h"
#include <iostream>

/*
A concrete client that implements the Observer interface
*/
class Client : public Observer
{
public:
    Client(int id);
    void update(float temp, float humidity, float pressure) override;

private:
    int id_;
};
#endif // CLIENT_H
#include <iostream>
#include "WeatherData.h"
#include "Client.h"
```

```
using namespace std;

int main()
{
   // Concrete Subject
   WeatherStation weatherStation;

   // Concrete Observers
   Client one(1), two(2), three(3);
   float temp, humidity, pressure;

   weatherStation.registerObserver(&one);
   weatherStation.registerObserver(&two);
   weatherStation.registerObserver(&three);

   cout << "Enter Temperature, Humidity, Pressure (seperated by spaces) << ";
   cin >> temp >> humidity >> pressure;

   weatherStation.setState(temp, humidity, pressure);

   weatherStation.unregisterObserver(&two);

   cout << "Enter Temperature, Humidity, Pressure (seperated by spaces) << ";
   cin >> temp >> humidity >> pressure;

   weatherStation.setState(temp, humidity, pressure);
}
#ifndef OBSERVER_H
#define OBSERVER_H
/*
Interface for the Observer
Observer.h
*/

class Observer
{
public:
   virtual void update(float temp, float humidity, float pressure) = 0;
};
#endif // OBSERVER_H
```

```
#ifndef SUBJECT_H
#define SUBJECT_H
#include "Observer.h"

/*
Interface for the Subject
*/

class Subject
{
public:
   // Register an Observer
   virtual void registerObserver(Observer *observer) = 0;

   // Unregister an Observer
   virtual void unregisterObserver(Observer *observer) = 0;

   // Notify all the registered observers when a change happens
   virtual void notifyObservers() = 0;
};
#endif // SUBJECT_H
#include "WeatherData.h"

void WeatherStation ::setState(float temp, float humidity, float pressure)
{
   this->temp = temp;
   this->humidity = humidity;
   this->pressure = pressure;
   cout << endl;
   notifyObservers();
}

void WeatherStation ::registerObserver(Observer *observer)
{
   observers_.push_back(observer);
}

void WeatherStation ::unregisterObserver(Observer *observer)
{
   // Find the observer
```

```
    auto iterator = find(observers_.begin(), observers_.end(), observer);

    if (iterator != observers_.end())
    {
        observers_.erase(iterator); // remove the observer
    }
}

void WeatherStation ::notifyObservers()
{
    for (auto &observer : observers_) // Notify all the observers
    {
        observer->update(temp, humidity, pressure);
    }
}
#ifndef WEATHERSTATION_H
#define WEATHERSTATION_H

#include <iostream>
#include <vector>
#include <algorithm>
#include "Subject.h"
#include "Observer.h"
using namespace std;

/*
A concrete implementation of the Subject interface
WeatherData.h
*/

class WeatherStation : public Subject
{
public:
    void registerObserver(Observer *observer) override;
    void unregisterObserver(Observer *observer) override;

    void notifyObservers() override;

    // Set the new state of the weather station
    void setState(float temp, float humidity, float pressure);
```

```
private:
    vector<Observer *> observers_;
    float temp = 0.0f;
    float humidity = 0.0f;
    float pressure = 0.0f;
};
#endif // WEATHERSTATION_H
```