10.4 — Association

BY ALEX ON AUGUST 19TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In the previous two lessons, we've looked at two types of object composition, composition and aggregation. Object composition is used to model relationships where a complex object is built from one or more simpler objects (parts).

In this lesson, we'll take a look at a weaker type of relationship between two otherwise unrelated objects, called an association. Unlike object composition relationships, in an association, there is no implied whole/part relationship.

Association

To qualify as an **association**, an object and another object must have the following relationship:

- The associated object (member) is otherwise unrelated to the object (class)
- The associated object (member) can belong to more than one object (class) at a time
- The associated object (member) does *not* have its existence managed by the object (class)
- The associated object (member) may or may not know about the existence of the object (class)

Unlike a composition or aggregation, where the part is a part of the whole object, in an association, the associated object is otherwise unrelated to the object. Just like an aggregation, the associated object can belong to multiple objects simultaneously, and isn't managed by those objects. However, unlike an aggregation, where the relationship is always unidirectional, in an association, the relationship may be unidirectional or bidirectional (where the two objects are aware of each other).

The relationship between doctors and patients is a great example of an association. The doctor clearly has a relationship with his patients, but conceptually it's not a part/whole (object composition) relationship. A doctor can see many patients in a day, and a patient can see many doctors (perhaps they want a second opinion, or they are visiting different types of doctors). Neither of the object's lifespans are tied to the other.

We can say that association models as "uses-a" relationship. The doctor "uses" the patient (to earn income). The patient uses the doctor (for whatever health purposes they need).

Implementing associations

Because associations are a broad type of relationship, they can be implemented in many different ways. However, most often, associations are implemented using pointers, where the object points at the associated object.

In this example, we'll implement a bi-directional Doctor/Patient relationship, since it makes sense for the Doctors to know who their Patients are, and vice-versa.

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  // Since Doctor and Patient have a circular dependency, we're going to forward declare Patien
6  class Patient;
7
8  class Doctor
9  {
10 private:
11     std::string m_name{};
12     std::vector<Patient *> m_patient{};
13
14 public:
15     Doctor(std::string name) :
16         m_name(name)
```

```
    {
    }

    void addPatient(Patient *pat);

    // We'll implement this function below Patient since we need Patient to be defined at tha
    friend std::ostream& operator<<(std::ostream &out, const Doctor &doc);

    std::string getName() const { return m_name; }
};

class Patient
{
private:
    std::string m_name{};
    std::vector<Doctor *> m_doctor{}; // so that we can use it here

    // We're going to make addDoctor private because we don't want the public to use it.
    // They should use Doctor::addPatient() instead, which is publicly exposed
    void addDoctor(Doctor *doc)
    {
        m_doctor.push_back(doc);
    }

public:
    Patient(std::string name)
        : m_name(name)
    {
    }

    // We'll implement this function below Doctor since we need Doctor to be defined at that
    friend std::ostream& operator<<(std::ostream &out, const Patient &pat);

    std::string getName() const { return m_name; }

    // We'll friend Doctor::addPatient() so it can access the private function Patient::addDo
    friend void Doctor::addPatient(Patient *pat);
};

void Doctor::addPatient(Patient *pat)
{
    // Our doctor will add this patient
    m_patient.push_back(pat);

    // and the patient will also add this doctor
    pat->addDoctor(this);
}

std::ostream& operator<<(std::ostream &out, const Doctor &doc)
{
    unsigned int length = doc.m_patient.size();
    if (length == 0)
    {
        out << doc.m_name << " has no patients right now";
        return out;
    }

    out << doc.m_name << " is seeing patients: ";
    for (unsigned int count = 0; count < length; ++count)
        out << doc.m_patient[count]->getName() << ' ';

    return out;
```

```
}

std::ostream& operator<<(std::ostream &out, const Patient &pat)
{
    unsigned int length = pat.m_doctor.size();
    if (length == 0)
    {
        out << pat.getName() << " has no doctors right now";
        return out;
    }

    out << pat.m_name << " is seeing doctors: ";
    for (unsigned int count = 0; count < length; ++count)
        out << pat.m_doctor[count]->getName() << ' ';

    return out;
}

int main()
{
    // Create a Patient outside the scope of the Doctor
    Patient *p1 = new Patient("Dave");
    Patient *p2 = new Patient("Frank");
    Patient *p3 = new Patient("Betsy");

    Doctor *d1 = new Doctor("James");
    Doctor *d2 = new Doctor("Scott");

    d1->addPatient(p1);

    d2->addPatient(p1);
    d2->addPatient(p3);

    std::cout << *d1 << '\n';
    std::cout << *d2 << '\n';
    std::cout << *p1 << '\n';
    std::cout << *p2 << '\n';
    std::cout << *p3 << '\n';

    delete p1;
    delete p2;
    delete p3;

    delete d1;
    delete d2;

    return 0;
}
```

This prints:

```
James is seeing patients: Dave
Scott is seeing patients: Dave Betsy
Dave is seeing doctors: James Scott
Frank has no doctors right now
Betsy is seeing doctors: Scott
```

In general, you should avoid bidirectional associations if a unidirectional one will do, as they add complexity and tend to be harder to write without making errors.

Reflexive association

Sometimes objects may have a relationship with other objects of the same type. This is called a **reflexive association**. A good example of a reflexive association is the relationship between a university course and its prerequisites (which are also university courses).

Consider the simplified case where a Course can only have one prerequisite. We can do something like this:

```
#include <string>
class Course
{
private:
    std::string m_name;
    Course *m_prerequisite;

public:
    Course(std::string &name, Course *prerequisite=nullptr):
        m_name(name), m_prerequisite(prerequisite)
    {
    }

};
```

This can lead to a chain of associations (a course has a prerequisite, which has a prerequisite, etc...)

Associations can be indirect

In all of the above cases, we've used a pointer to directly link objects together. However, in an association, this is not strictly required. Any kind of data that allows you to link two objects together suffices. In the following example, we show how a Driver class can have a unidirectional association with a Car without actually including a Car pointer member:

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

class Car
{
private:
    std::string m_name;
    int m_id;

public:
    Car(std::string name, int id)
        : m_name(name), m_id(id)
    {
    }

    std::string getName() { return m_name; }
    int getId() { return m_id;  }
};

// Our CarLot is essentially just a static array of Cars and a lookup function to retrieve the
// Because it's static, we don't need to allocate an object of type CarLot to use it
class CarLot
{
private:
    static Car s_carLot[4];

public:
    CarLot() = delete; // Ensure we don't try to allocate a CarLot

    static Car* getCar(int id)
```

```
    {
        for (int count = 0; count < 4; ++count)
            if (s_carLot[count].getId() == id)
                return &(s_carLot[count]);

        return nullptr;
    }
};

Car CarLot::s_carLot[4] = { Car("Prius", 4), Car("Corolla", 17), Car("Accord", 84), Car("Matri

class Driver
{
private:
    std::string m_name;
    int m_carId; // we're associated with the Car by ID rather than pointer

public:
    Driver(std::string name, int carId)
        : m_name(name), m_carId(carId)
    {
    }

    std::string getName() { return m_name; }
    int getCarId() { return m_carId; }

};

int main()
{
    Driver d("Franz", 17); // Franz is driving the car with ID 17

    Car *car = CarLot::getCar(d.getCarId()); // Get that car from the car lot

    if (car)
        std::cout << d.getName() << " is driving a " << car->getName() << '\n';
    else
        std::cout << d.getName() << " couldn't find his car\n";

    return 0;
}
```

In the above example, we have a CarLot holding our cars. The Driver, who needs a car, doesn't have a pointer to his Car -- instead, he has the ID of the car, which we can use to get the Car from the CarLot when we need it.

In this particular example, doing things this way is kind of silly, since getting the Car out of the CarLot requires an inefficient lookup (a pointer connecting the two is much faster). However, there are advantages to referencing things by a unique ID instead of a pointer. For example, you can reference things that are not currently in memory (maybe they're in a file, or in a database, and can be loaded on demand). Also, pointers can take 4 or 8 bytes -- if space is at a premium and the number of unique objects is fairly low, referencing them by an 8-bit or 16-bit integer can save lots of memory.

Composition vs aggregation vs association summary

Here's a summary table to help you remember the difference between composition, aggregation, and association:

Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes

Members existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a


10.5 -- Dependencies


Index


10.3 -- Aggregation

C++ TUTORIAL | PRINT THIS POST

106 comments to 10.4 — Association

« Older Comments 1 2

David
January 24, 2020 at 10:32 pm · Reply

Hi,I don't understand the meaning "The associated object (member) is otherwise unrelated to the object (class)".
Dose the meaning of "otherwise unrelated to the object" be the same with "unrelated to the object"?
Thanks for replying

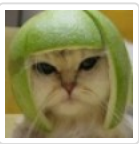
nascardriver
January 25, 2020 at 3:23 am · Reply

It means "Apart from being associated with the object, it's not related to the object in any other way".

ejamesr
November 15, 2019 at 1:42 pm · Reply

Hi nascardriver! You have a typo near the bottom of this lesson (10.4) where you say "... referencing them by an 8-bit or 16-bit integer can save lots of memory." I believe you meant to say "... referencing them by a 32-bit or 64-bit integer can save lots of memory.".

BTW, wonderful content, this is helping me get back up to speed on C++ after several years of focusing solely on C#.



Alex
November 16, 2019 at 2:01 pm · Reply

No, it's correct as written. If pointers took 4 or 8 bytes, there would be no space savings by replacing that with a 4 or 8 byte index. The space savings comes in if we can use an index that takes less memory (e.g. 8 or 16 bytes).

noobmaster
August 21, 2019 at 1:41 am · Reply

in Doctor-Patient example (line 86), we can just use pat.m_name instead of pat.getName() right?
And here's my full code if the patient is the one who add the doctors, not otherwise

```
1  #include<iostream>
2  #include<string>
3  #include<vector>
4
5  class Doctor;
6
7  class Patient
8  {
9  private:
10     std::string m_name;
11     std::vector<Doctor*> m_doctor;
12
13 public:
14     Patient(std::string name) : m_name{ name }
15     {
16     }
17
18     std::string getName() const { return m_name; }
19
20     void addDoctor(Doctor* doc);
21     friend std::ostream& operator<<(std::ostream& out, const Patient& pat);
22 };
23
24 class Doctor
25 {
26 private:
27     std::string m_name;
28     std::vector<Patient*> m_patient;
29
30     void addPatient(Patient* pat)
31     {
32         m_patient.push_back(pat);
33     }
34
```

```
public:
    Doctor(std::string name) : m_name{ name }
    {
    }

    std::string getName() const { return m_name; }

    friend void Patient::addDoctor(Doctor* doc);
    friend std::ostream& operator<<(std::ostream& out, const Doctor& doc);
};

void Patient::addDoctor(Doctor* doc)
{
    m_doctor.push_back(doc);
    doc->addPatient(this);
}

std::ostream& operator<<(std::ostream& out, const Patient& pat)
{
    unsigned int length{ pat.m_doctor.size() };
    if (length == 0)
    {
        out << pat.m_name << " has no doctors right now";
        return out;
    }

    out << pat.m_name << " is seeing doctors: ";
    for (unsigned int count{ 0 }; count < length; ++count)
        out << pat.m_doctor[count]->getName() << ' ';

    return out;
}

std::ostream& operator<<(std::ostream& out, const Doctor& doc)
{
    unsigned int length{ doc.m_patient.size() };
    if (length == 0)
    {
        out << doc.m_name << " has no patients right now";
        return out;
    }

    out << doc.m_name << " is seeing patients: ";
    for (unsigned int count{ 0 }; count < length; ++count)
        out << doc.m_patient[count]->getName() << ' ';

    return out;
}

int main()
{
    // Create a Patient outside the scope of the Doctor
    Patient* p1 = new Patient{ "Dave" };
    Patient* p2 = new Patient{ "Frank" };
    Patient* p3 = new Patient{ "Betsy" };

    Doctor* d1 = new Doctor{ "James" };
    Doctor* d2 = new Doctor{"Scott"};

    p1->addDoctor(d1);
    p1->addDoctor(d2);
    p2->addDoctor(d1);
```

```
97      p3->addDoctor(d2);
98
99      std::cout << *d1 << '\n';
100     std::cout << *d2 << '\n';
101     std::cout << *p1 << '\n';
102     std::cout << *p2 << '\n';
103     std::cout << *p3 << '\n';
104
105     delete p1;
106     delete p2;
107     delete p3;
108
109     delete d1;
110     delete d2;
111
112     return 0;
113 }
```

nascardriver
August 21, 2019 at 5:30 am · Reply

You could, but you shouldn't. If a class offers you an access function, you should use it unless you're worried about performance loss of the function call.
The class might use the access function for more than just accessing the member, eg. counting calls or returning a modified member.

noobmaster
August 21, 2019 at 8:51 pm · Reply

oh okay, thanks!

Behzad
August 1, 2019 at 6:26 am · Reply

I have a slightly modified version of Doctor-Patient example in which I have tried to change the Patient::addDoctor function such that the Patient can also modify the doctor object and update its patient list. But when I run the code, it gives me Segmentation fault: 11
If I comment out line 96, the code runs just fine. Could you please help me find what's causing the Segmentation fault?

```
1   #include <iostream>
2   #include <string>
3   #include <vector>
4
5   class Patient;
6
7   class Doctor
8   {
9   private:
10      std::string m_name {};
11      std::vector<Patient*> m_patient {};
12
13  public:
14      Doctor(std::string name) : m_name(name)
15      {
16      }
17
```

```
18      void addPatient(Patient *pat);
19
20      friend std::ostream& operator << (std::ostream& out, const Doctor& doc);
21
22      std::string getName() const
23      {
24          return m_name;
25      }
26  };
27
28  class Patient
29  {
30  private:
31      std::string m_name {};
32      std::vector<Doctor*> m_doctor {};
33
34  public:
35      Patient(std::string name) : m_name(name)
36      {
37      }
38
39      void addDoctor(Doctor *doc);
40
41      friend std::ostream& operator << (std::ostream& out, const Patient& pat);
42
43      std::string getName() const
44      {
45          return m_name;
46      }
47  };
48
49
50  std::ostream& operator << (std::ostream& out, const Patient& pat)
51  {
52      unsigned int length = pat.m_doctor.size();
53
54      out << pat.getName();
55      if (length==0)
56      {
57          out << " is a patient and he/she is not seeing any doctors";
58      }
59      else
60      {
61          out << " is a patient and he/she is seeing doctors: ";
62          for (Doctor* const &doc: pat.m_doctor)
63              out << doc->getName() << " ";
64      }
65      return out;
66  }
67
68  std::ostream& operator << (std::ostream& out, const Doctor& doc)
69  {
70      unsigned int length = doc.m_patient.size();
71
72      out << doc.getName();
73      if (length==0)
74      {
75          out << " is a doctor and he/she is not seeing any patients";
76      }
77      else
78      {
79          out << " is a doctor and he/she is seeing patients: ";
```

```
        for (Patient* const &pat: doc.m_patient)
            out << pat->getName() << " ";
    }

    return out;
}

void Doctor::addPatient(Patient *pat)
{
    m_patient.push_back(pat);
    pat->addDoctor(this);
}

void Patient::addDoctor(Doctor *doc)
{
    m_doctor.push_back(doc);
    doc->addPatient(this); // IF YOU COMMENT OUT THIS LINE, THE CODE WORKS FINE. WHY?
}

int main()
{
    // Create a Patient outside the scope of the Doctor
    Patient *p1 = new Patient("Dave");
    Patient *p2 = new Patient("Frank");
    Patient *p3 = new Patient("Betsy");

    Doctor *d1 = new Doctor("James");
    Doctor *d2 = new Doctor("Scott");

    d1->addPatient(p1);
    d2->addPatient(p1);
    d2->addPatient(p3);

    std::cout << *d1 << '\n';
    std::cout << *d2 << '\n';
    std::cout << *p1 << '\n';
    std::cout << *p2 << '\n';
    std::cout << *p3 << '\n';

    delete p1;
    delete p2;
    delete p3;

    delete d1;
    delete d2;

    return 0;
}
```

nascardriver
August 1, 2019 at 6:33 am · Reply

`addDoctor` and `addPatient` are calling each other. This causes an infinite loop and a segmentation fault, because your call stack is full.

Atas
July 11, 2019 at 1:29 am · Reply

Wanted to be pedantic and make the parameter const in

```
void addDoctor(Doctor *doc)
{
    m_doctor.push_back(doc);
}
```

but as it turns out you cant push_back a pointer to const in a vector of regular pointers. Makes sense, I guess.

Louis Cloete
April 18, 2019 at 8:20 am · Reply

Hi @Alex. I had a hunch that rearranging the two classes will enable you to friend only void Doctor::addPatient(Patient *pat) instead of the whole class Doctor. Here is my adjusted code:

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

class Patient;

class Doctor
{
private:
    std::string m_name;
    std::vector<Patient *> m_patient {};

public:
    Doctor(std::string name):
        m_name(name)
    {
    }

    void addPatient(Patient *pat);

    friend std::ostream& operator<<(std::ostream &out, const Doctor &doc);

    std::string getName() const { return m_name; }
};

class Patient
{
private:
    std::string m_name;
    std::vector<Doctor *> m_doctor {};

    void addDoctor(Doctor *doc)
    {
        m_doctor.push_back(doc);
    }

public:
    Patient(std::string name)
        : m_name(name)
    {
    }

    friend std::ostream& operator<<(std::ostream &out, const Patient &pat)
    {
        unsigned int length = pat.m_doctor.size();
        if (length == 0)
        {
            out << pat.getName() << " has no doctors right now";
```

```
            return out;
        }

        out << pat.m_name << " is seeing doctors: ";
        for (unsigned int count = 0; count < length; ++count)
            out << pat.m_doctor[count]->getName() << ' ';

        return out;
    }

    std::string getName() const { return m_name; }

    // We can friend only the relevant function from Doctor
    // by rearranging the two classes.
    friend void Doctor::addPatient(Patient *pat);
};

void Doctor::addPatient(Patient *pat)
{
    // Our doctor will add this patient
    m_patient.push_back(pat);

    // and the patient will also add this doctor
    pat->addDoctor(this);
}

std::ostream& operator<<(std::ostream &out, const Doctor &doc)
{
    unsigned int length = doc.m_patient.size();
    if (length == 0)
    {
        out << doc.m_name << " has no patients right now";
        return out;
    }

    out << doc.m_name << " is seeing patients: ";
    for (unsigned int count = 0; count < length; ++count)
        out << doc.m_patient[count]->getName() << ' ';

    return out;
}


int main()
{
    // Create a Patient outside the scope of the Doctor
    Patient *p1 = new Patient("Dave");
    Patient *p2 = new Patient("Frank");
    Patient *p3 = new Patient("Betsy");

    Doctor *d1 = new Doctor("James");
    Doctor *d2 = new Doctor("Scott");

    d1->addPatient(p1);

    d2->addPatient(p1);
    d2->addPatient(p3);

    std::cout << *d1 << '\n';
    std::cout << *d2 << '\n';
    std::cout << *p1 << '\n';
    std::cout << *p2 << '\n';
```

```
    std::cout << *p3 << '\n';

    delete p1;
    delete p2;
    delete p3;

    delete d1;
    delete d2;

    return 0;
}
```

Now my question is, is there any benefit doing things the one way or another? I would guess that my adjusted code is better, since you expose less of the private innards of class Patient to the outside via Doctor's public interface. It is also expressing intent more clearly and directly in code. Am I reasoning correctly?

Alex
April 22, 2019 at 2:09 pm · Reply

For this particular example, your solution is better because it exposes less. If Patient and Doctor were more intertwined, having to friend each individual function might be burdensome -- but that's not the case here. I've updated the lesson incorporating your adjustment. Thanks!

lucieon
February 2, 2019 at 2:31 am · Reply

Hello.

I tried to code the Doctor-Patient program using header files but it doesn't compile.
My code:

1) Doctor.h

```
#include <string>
#include <vector>
#include "Patient.h"

class Doctor
{
private:
    std::string m_name;
    std::vector<Patient *> m_patient;
public:
    Doctor(std::string name);
    void addPatient(Patient *p);
    friend std::ostream& operator<<(std::ostream &out, const Doctor &d);
    std::string getName() const;
};
```

2) Patient.h
Note: Here I made the addDoctor(Doctor *) function public because I don't know how to friend a Patient class' function in Doctor class.

```
#include <string>
#include <vector>
#include "Doctor.h"

class Patient
{
```

```
private:
    std::string m_name;
    std::vector<Doctor *> m_doctor;
    //void addDoctor(Doctor *doc);
public:
    void addDoctor(Doctor *d);
    Patient(std::string name);
    friend std::ostream& operator<<(std::ostream &out, const Patient &p);
    std::string getname() const;
};
```

3) Doctor.cpp

```
#include "Doctor.h"

Doctor::Doctor(std::string name)
    : m_name{ name }
{
}

void Doctor::addPatient(Patient * p)
{
    // our doctor will add this patient
    m_patient.push_back(p);

    // and the patient will also add this doctor
    p->addDoctor(this);
}

std::string Doctor::getName() const
{
    return m_name;
}

std::ostream & operator<<(std::ostream & out, const Doctor & d)
{
    unsigned int length = d.m_patient.size();

    if (length == 0)
        return out << d.m_name << " has no patients right now.";

    out << d.m_name << " is seeing patients: ";
    for (unsigned int count = 0; count < length; ++count)
        out << d.m_patient[count]->getname() << ' ';

    return out;
}
```

4) Patient.cpp

```
#include "Patient.h"

void Patient::addDoctor(Doctor * d)
{
    m_doctor.push_back(d);
}

Patient::Patient(std::string name)
    : m_name{ name }
{
}

std::string Patient::getname() const
```

```
{
    return m_name;
}

std::ostream & operator<<(std::ostream & out, const Patient & p)
{
    unsigned int length = p.m_doctor.size();

    if (length == 0)
        return out << p.getname() << " has no doctors right now.";
    out << p.getname() << " is seeing doctors: ";
    for (unsigned int count = 0; count < length; ++count)
        out << p.m_doctor[count]->getName() << ' ';

    return out;
}
```

5) association.cpp

```
#include <iostream>
#include "Patient.h"
#include "Doctor.h"

int main()
{
    Patient *p1 = new Patient("Dave");
    Patient *p2 = new Patient("Frank");
    Patient *p3 = new Patient("Betsy");

    Doctor *d1 = new Doctor("James");
    Doctor *d2 = new Doctor("Scott");

    return 0;
}
```

And the output I get after building the project:

```
1>------ Build started: Project: association, Configuration: Debug Win32 ------
1>association.cpp
1>c:\users\lucieon\source\repos\learncpp\association\association\doctor.h(11): error C2065:
1>c:\users\lucieon\source\repos\learncpp\association\association\doctor.h(11): error C2059:
1>c:\users\lucieon\source\repos\learncpp\association\association\doctor.h(15): error C2061:
1>Doctor.cpp
1>c:\users\lucieon\source\repos\learncpp\association\association\patient.h(11): error C2065
1>c:\users\lucieon\source\repos\learncpp\association\association\patient.h(11): error C2059
1>c:\users\lucieon\source\repos\learncpp\association\association\patient.h(18): error C2061
1>c:\users\lucieon\source\repos\learncpp\association\association\doctor.cpp(20): error C266
1>c:\users\lucieon\source\repos\learncpp\association\association\patient.h(18): note: see d
1>Patient.cpp
1>c:\users\lucieon\source\repos\learncpp\association\association\doctor.h(11): error C2065:
1>c:\users\lucieon\source\repos\learncpp\association\association\doctor.h(11): error C2059:
1>c:\users\lucieon\source\repos\learncpp\association\association\doctor.h(15): error C2061:
1>Generating Code...
1>Done building project "association.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Is it possible to do resolve the circular dependency between Doctor and Patient this way? If not then can you please explain why?
And is there any other alternative than the forward declaration option because it just feels unnatural to code it like that.

Thank you!

nascardriver
February 3, 2019 at 3:58 am · Reply

Hi Lucieon!

> Is it possible to do resolve the circular dependency between Doctor and Patient this way? If not then can you please explain why?
No, now you have circular includes.

> is there any other alternative than the forward declaration
No, you need forward declarations. Use as few includes as possible in header files.

Louis Cloete
April 18, 2019 at 8:58 am · Reply

You need to forward declare class Doctor; in Patient.h and class Patient; in Doctor.h and then #include both Patient.h and Doctor.h in both of Patient.cpp and Doctor.cpp. You should also declare void Doctor::addPatient(Patient *pat) as a friend of class Patient. Then your code should be equivalent to the example given in the text.

beginnerCoder
December 30, 2018 at 6:29 am · Reply

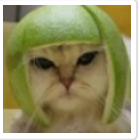
Hello! I wonder why we don't also add this line of code to the function addDoctor:

doc->addPatient(this)

since if a patient sees a doctor then the doctor has a new patient!

nascardriver
December 30, 2018 at 7:08 am · Reply

Nope, @addPatient would call @addDoctor, resulting in an infinite recursive loop.



Alex
December 30, 2018 at 11:38 pm · Reply

The interface for these classes is set up in such a way that users of the code would call Patient::addDoctor, which handles the bidirectional adding. Doctor::addPatient is just a private helper function to assist with that.

Gaurav Arya
November 30, 2018 at 7:38 pm · Reply

Hi Alex,

Taking the example of university course and its prerequisite course, will the relationship between them still be called reflexive if there are multiple prerequisites to a single course? A trivial example would be physics, for which one may have 2 prerequisites maths and english.

Alex
December 2, 2018 at 3:53 pm · Reply

Yes, in your example the physics course would have two reflexive associations.

Michael Stef
October 12, 2018 at 8:37 am · Reply

So linked list is implemented using reflexive association relation...

Nitin
August 14, 2018 at 11:20 am · Reply

Hi Alex,

```
Patient(std::string name)
        : m_name(name)
    {
    }


Course(std::string &name, Course *prerequisite=nullptr):
        m_name(name), m_prerequisite(prerequisite)
    {
    }
```

I see that for functions taking in std::string as an argument, you sometimes pass by value and at other times pass by reference (even though it isn't being modified.) I'd really like to know what the best practice is.

If the modification is to be seen in the calling function, then yes, pass by reference is the way to go. If not, what's the best practice? Pass by value, or a (const) reference?

Thanks,
Nitin

nascardriver
August 14, 2018 at 11:33 am · Reply

Hi Nitin!
If the argument isn't being modified, pass by const reference, unless it's a built-in type (int, float, double, ...).
If the argument is being modified, pass by pointer if the caller is supposed to see the change, otherwise pass by value.
Only pass by non-const reference, when passing by pointer is not possible or too complicated (eg. operators).

Nitin
August 14, 2018 at 12:50 pm · Reply

Thanks nascardriver! Appreciate the clarity.

danskin
February 1, 2019 at 8:32 am · Reply

hi nascardriver,

Can you please elaborate what does "caller is supposed to see the change" means?

Thank you

nascardriver
February 1, 2019 at 8:50 am · Reply

"caller is supposed to see that the variable is being modified" is what I meant to say.

```
int i{ 0 };

myFunction(i); // Does this modify @i? I'd say no, but it could.
myFunction(&i); // Does this modify @i? Probably, why else would it take a poi
```

David
July 8, 2018 at 12:39 pm · Reply

Hi Alex! I wanted to implement the Patient/Doctor classes with each class' member functions defined in separate cpp files. Is the following code the best way to do this:

patient.hpp:

```
#ifndef patient_hpp
#define patient_hpp

#include <iostream>
#include <string>
#include <vector>

class Doctor; // forward declaration because of circular dependence

class Patient
{
private:
    std::string m_name;
    std::vector<Doctor *> m_doctor;

    void addDoctor(Doctor *doc);

public:
    Patient(std::string name)
    : m_name(name)
    {
    }

    friend std::ostream& operator<<(std::ostream &out, const Patient &pat);

    std::string getName() const { return m_name; }

    friend class Doctor;
};

#include "doctor.hpp" // include associated class definition here

#endif /* patient_hpp */
```

patient.cpp:

```
#include "patient.hpp"

void Patient::addDoctor(Doctor *doc)
{
    m_doctor.push_back(doc);
}

std::ostream& operator<<(std::ostream &out, const Patient &pat)
{
    unsigned long length = pat.m_doctor.size();
    if (length == 0)
    {
        out << pat.getName() << " has no doctors right now";
        return out;
    }

    out << pat.m_name << " is seeing doctors: ";
    for (unsigned int count = 0; count < length; ++count)
        out << pat.m_doctor[count]->getName() << ' ';

    return out;
}
```

doctor.hpp:

```
#ifndef doctor_hpp
#define doctor_hpp

#include <iostream>
#include <string>
#include <vector>

class Patient; // forward declaration

class Doctor
{
private:
    std::string m_name;
    std::vector<Patient *> m_patient;

public:
    Doctor(std::string name):
    m_name(name)
    {
    }

    void addPatient(Patient *pat);

    friend std::ostream& operator<<(std::ostream &out, const Doctor &doc);

    std::string getName() const { return m_name; }
};

#include "patient.hpp" // include associated class definition here

#endif /* doctor_hpp */
```

doctor.cpp:

```
#include "doctor.hpp"

```

```
void Doctor::addPatient(Patient *pat)
{
    m_patient.push_back(pat);

    pat->addDoctor(this);
}

std::ostream& operator<<(std::ostream &out, const Doctor &doc)
{
    unsigned long length = doc.m_patient.size();
    if (length == 0)
    {
        out << doc.m_name << " has no patients right now";
        return out;
    }

    out << doc.m_name << " is seeing patients: ";
    for (unsigned int count = 0; count < length; ++count)
        out << doc.m_patient[count]->getName() << ' ';

    return out;
}
```

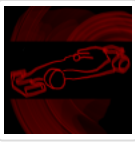
nascardriver
July 9, 2018 at 4:22 am · Reply

Hi David!

* @Patient::getName and @Doctor::getName should be defined in source files and should return const references.
* @Patient::Patient and @Doctor::Doctor should be defined in source files.
* @Patient doesn't ever edit it's doctors, you could store const doctors.
* @patient.hpp:20, @patient.cpp:10,18, @doctor.hpp:18, @doctor.cpp:12,20: Use uniform initization.
* The name printing loops could be for-each loops, because you don't need to index.
* @std::vector::size returns a @std::size_t, use that instead of unsigned long.
* Both operator<<s: Use @std::vector::empty.

David
July 9, 2018 at 6:56 am · Reply

Hi nascardriver! Thanks for all the great feedback. One question - if I define @Patient::Patient and @Doctor::Doctor in source files, how will the definitions "see" one another? Would I need to #include the source files?



nascardriver
July 9, 2018 at 7:20 am · Reply

Definition != Declaration

patient.hpp

```
...
const std::string &getName() const;
...
```

patient.cpp

```
...
```

```
const std::string &Patient::getName() const
{
  return m_name;
}
...
```

« Older Comments 1 2