10.2 — Composition

BY ALEX ON DECEMBER 4TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Object composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called **object composition**.

Broadly speaking, object composition models a “has-a” relationship between two objects. A car “has-a” transmission. Your computer “has-a” CPU. You “have-a” heart. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

In C++, you’ve already seen that structs and classes can have data members of various types (such as fundamental types or other classes). When we build classes with data members, we’re essentially constructing a complex object from simpler parts, which is object composition. For this reason, structs and classes are sometimes referred to as **composite types**.

Object Composition is useful in a C++ context because it allows us to create complex classes by combining simpler, more easily manageable parts. This reduces complexity, and allows us to write code faster and with less errors because we can reuse code that has already been written, tested, and verified as working.

Types of object composition

There are two basic subtypes of object composition: composition and aggregation. We’ll examine composition in this lesson, and aggregation in the next.

A note on terminology: the term “composition” is often used to refer to both composition and aggregation, not just to the composition subtype. In this tutorial, we’ll use the term “object composition” when we’re referring to both, and “composition” when we’re referring specifically to the composition subtype.

Composition

To qualify as a **composition**, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person’s body and a heart. Let’s examine these in more detail.

Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person’s body. The part in a composition can only be part of one object at a time. A heart that is part of one person’s body can not be part of someone else’s body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part’s lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the heart is created too. When a person’s body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a “death relationship”.

And finally, the part doesn’t know about the existence of the whole. Your heart operates blissfully unaware that it is part of a larger structure. We call this a **unidirectional** relationship, because the body knows about the heart, but not the other way around.

Note that composition has nothing to say about the transferability of parts. A heart can be transplanted from one body to another. However, even after being transplanted, it still meets the requirements for a composition (the heart is now owned by the recipient, and can only be part of the recipient object unless transferred again).

Our ubiquitous Fraction class is a great example of a composition:

```
class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction(int numerator=0, int denominator=1):
        m_numerator(numerator), m_denominator(denominator)
    {
        // We put reduce() in the constructor to ensure any fractions we make get reduced!
        // Since all of the overloaded operators create new Fractions, we can guarantee this
 will get called here
        reduce();
    }
};
```

This class has two data members: a numerator and a denominator. The numerator and denominator are part of the Fraction (contained within it). They can not belong to more than one Fraction at a time. The numerator and denominator don't know they are part of a Fraction, they just hold integers. When a Fraction instance is created, the numerator and denominator are created. When the fraction instance is destroyed, the numerator and denominator are destroyed as well.

While object composition models has-a type relationships (a body has-a heart, a fraction has-a denominator), we can be more precise and say that composition models "part-of" relationships (a heart is part-of a body, a numerator is part of a fraction). Composition is often used to model physical relationships, where one object is physically contained inside another.

The parts of a composition can be singular or multiplicative -- for example, a heart is a singular part of the body, but a body contains 10 fingers (which could be modeled as an array).

Implementing compositions

Compositions are one of the easiest relationship types to implement in C++. They are typically created as structs or classes with normal data members. Because these data members exist directly as part of the struct/class, their lifetimes are bound to that of the class instance itself.

Compositions that need to do dynamic allocation or deallocation may be implemented using pointer data members. In this case, the composition class should be responsible for doing all necessary memory management itself (not the user of the class).

In general, if you *can* design a class using composition, you *should* design a class using composition. Classes designed using composition are straightforward, flexible, and robust (in that they clean up after themselves nicely).

More examples

Many games and simulations have creatures or objects that move around a board, map, or screen. One thing that all of these creatures/objects have in common is that they all have a location. In this example, we are going to create a creature class that uses a point class to hold the creature's location.

First, let's design the point class. Our creature is going to live in a 2d world, so our point class will have 2 dimensions, X and Y. We will assume the world is made up of discrete squares, so these dimensions will always be integers.

Point2D.h:

```cpp
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_x;
    int m_y;

public:
    // A default constructor
    Point2D()
        : m_x(0), m_y(0)
    {
    }

    // A specific constructor
    Point2D(int x, int y)
        : m_x(x), m_y(y)
    {
    }

    // An overloaded output operator
    friend std::ostream& operator<<(std::ostream& out, const Point2D &point)
    {
        out << "(" << point.m_x << ", " << point.m_y << ")";
        return out;
    }

    // Access functions
    void setPoint(int x, int y)
    {
        m_x = x;
        m_y = y;
    }

};

#endif
```

Note that because we've implemented all of our functions in the header file (for the sake of keeping the example concise), there is no Point2D.cpp.

This Point2d class is a composition of its parts: location values x and y are part-of Point2D, and their lifespan is tied to that of a given Point2D instance.

Now let's design our Creature. Our Creature is going to have a few properties: a name, which will be a string, and a location, which will be our Point2D class.

Creature.h:

```cpp
#ifndef CREATURE_H
#define CREATURE_H

#include <iostream>
#include <string>
#include "Point2D.h"

class Creature
{
```

```cpp
private:
    std::string m_name;
    Point2D m_location;

public:
    Creature(const std::string &name, const Point2D &location)
        : m_name(name), m_location(location)
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Creature &creature)
    {
        out << creature.m_name << " is at " << creature.m_location;
        return out;
    }

    void moveTo(int x, int y)
    {
        m_location.setPoint(x, y);
    }
};
#endif
```

A composition may opt to use a part that has been given to it as input rather than create the part itself.-

This Creature is also a composition of its parts. The creature's name and location have one parent, and their lifetime is tied to that of the Creature they are part of.

And finally, Main.cpp:

```cpp
#include <string>
#include <iostream>
#include "Creature.h"
#include "Point2D.h"

int main()
{
    std::cout << "Enter a name for your creature: ";
    std::string name;
    std::cin >> name;
    Creature creature(name, Point2D(4, 7));

    while (1)
    {
        // print the creature's name and location
        std::cout << creature << '\n';

        std::cout << "Enter new X location for creature (-1 to quit): ";
        int x=0;
        std::cin >> x;
        if (x == -1)
            break;

        std::cout << "Enter new Y location for creature (-1 to quit): ";
        int y=0;
        std::cin >> y;
        if (y == -1)
            break;

        creature.moveTo(x, y);
        }

    return 0;
}
```

Here, "Point2D" object is created as a parameter of the Creature contructor.- "Po

Here's a transcript of this code being run:

```
Enter a name for your creature: Marvin
Marvin is at (4, 7)
Enter new X location for creature (-1 to quit): 6
Enter new Y location for creature (-1 to quit): 12
Marvin is at (6, 12)
Enter new X location for creature (-1 to quit): 3
Enter new Y location for creature (-1 to quit): 2
Marvin is at (3, 2)
Enter new X location for creature (-1 to quit): -1
```

Variants on the composition theme

Although most compositions directly create their parts when the composition is created and directly destroy their parts when the composition is destroyed, there are some variations of composition that bend these rules a bit.

For example:

- A composition may defer creation of some parts until they are needed. For example, a string class may not create a dynamic array of characters until the user assigns the string some data to hold.
- A composition may opt to use a part that has been given to it as input rather than create the part itself.
- A composition may delegate destruction of its parts to some other object (e.g. to a garbage collection routine).

The key point here is that the composition should manage its parts without the user of the composition needing to manage anything.

Composition and subclasses

One question that new programmers often ask when it comes to object composition is, "When should I use a subclass instead of direct implementation of a feature?". For example, instead of using the Point2D class to implement the Creature's location, we could have instead just added 2 integers to the Creature class and written code in the Creature class to handle the positioning. However, making Point2D its own class has a number of benefits:

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task well. This makes those classes easier to write and much easier to understand, as they are more focused. For example, Point2D only worries about point-related stuff, which helps keep it simple.
2. Each subclass can be self-contained, which makes them reusable. For example, we could reuse our Point2D class in a completely different application. Or if our creature ever needed another point (for example, a destination it was trying to get to), we can simply add another Point2D member variable.
3. The parent class can have the subclasses do most of the hard work, and instead focus on coordinating the data flow between the subclasses. This helps lower the overall complexity of the parent object, because it can delegate tasks to its children, who already know how to do those tasks. For example, when we move our Creature, it delegates that task to the Point class, which already understands how to set a point. Thus, the Creature class does not have to worry about how such things would be implemented.

A good rule of thumb is that each class should be built to accomplish a single task. That task should either be the storage and manipulation of some kind of data (e.g. Point2D, std::string), OR the coordination of subclasses (e.g. Creature). Ideally not both.

In this case of our example, it makes sense that Creature shouldn't have to worry about how Points are implemented, or how the name is being stored. Creature's job isn't to know those intimate details. Creature's job is to worry about how to coordinate the data flow and ensure that each of the subclasses knows *what* it is supposed to do. It's up to the individual subclasses to worry about *how* they will do it.

10.3 -- Aggregation

Index

10.1 -- Object relationships

C++ TUTORIAL | PRINT THIS POST

115 comments to 10.2 — Composition

« Older Comments 1 2

song
September 1, 2019 at 6:13 pm · Reply

Hi, Alex.
Thanks for your sharing this great knowledge.
I've been learning so much from your tutorial.

I have a seemingly stupid question. It seems stupid because no one has questioned yet...

I would like to know how the overloaded operator " << " can print the ""creature location"". I can't find any invoking " << " operator in the "main.cpp".

Thanks to you again.
Sincerely, Song

nascardriver
September 2, 2019 at 3:01 am · Reply

main.cpp line 16

song
September 2, 2019 at 7:48 pm · Reply

Thank you nascardriver.
It's stupid of me not to notice that obvious line.
Believe me, I read the block over and over and there I was.
I see the line clearly now.
Thanks... ^^;

Mukesh Kumar
June 4, 2019 at 10:46 pm · Reply

If I don't want to use overload operator how to write this piece of code.

friend std::ostream& operator<<(std::ostream& out, const Point2D &point)
{
out << "(" << point.m_x << ", " << point.m_y << ")";
return out;
}

nascardriver
June 5, 2019 at 3:47 am · Reply

```
friend std::ostream& printToStream(std::ostream& out, const Point2D &point)
{
  out << '(' << point.m_x << ", " << point.m_y << ')';
  return out;
}

// ...

printToStream(std::cout, point);
```

Bruno Luiz Demarchi
March 17, 2019 at 8:50 am · Reply

Hi Alex!

I have a question. Can I still access normally an object wich I passed to another one to make a composition?

Ex:

```
class Encoder
{
  private:
  uint32_t ticks;

  public:
  Encoder(uint32_t ticks): ticks(ticks)
  { };

  uint32_t getTicks() return ticks;
}

class Motor
{
  private:
  Encoder enc;

  public:
  Motor(Encoder enc): enc(enc)
  { };

  uint32_t getSpeed()
  {
     return enc.getTicks;
  }
}

void main()
{
  uint32_t test1, test2;
31
32     Encoder encTest(0);
33     Motor motorTest(encTest);
34
35     test1 = encTest.getTicks();
36     test2 = motorTest.getSpeed();
37 }
```

This is just a test code as example. My question is, are test1 == test2? Can I keep using my encTest normally even after passing it to my motorTest?

nascardriver
March 17, 2019 at 9:42 am · Reply

Hi Bruno!

* Line 7, 19, 32, 33: Initialize your variables with brace initializers. You used direct initialization.
* Line 4, 16, 30: Initialize your variables with brace initializers.
* Line 10: Missing braces.
* Don't use @uint32_t. Use @std::uint_fast32_t or @std::uint_least32_t.
* @main invalid declaration.

Neither @encTest or @motorTest modified their copy the ticks, so @test1 and @test2 compare equal.
But @Motor isn't storing @enc by pointer or reference. Modifying @encTest or @motorTest does not affect the other.

```
// ...

test1 == test2; // = true

encTest.ticks = 1; // Assuming @Encode::ticks was public.

encTest.getTicks(); // = 1
motorTest.getSpeed(); // = 0
```

Bruno Luiz Demarchi
March 17, 2019 at 5:03 pm · Reply

Thanks man!

Actually I didn't know that i could (and should) use {} instead of (), I'm kind of new in c++.

What are the diferences between uint32_t and uint_fast32_t? Because into the stdint.h they are define as:

typedef unsigned   int uint32_t;
typedef unsigned   int uint_fast32_t;

To me they work as the same, that's why I choose to work with uint32_t.

How could I store @enc by point or reference?

Again, thanks!!

nascardriver
March 18, 2019 at 3:35 am · Reply

> I didn't know that i could (and should) use {} instead of ()
Lesson 1.4
Brace initializers disallow implicit casts and can be used almost everywhere.

> What are the diferences between uint32_t and uint_fast32_t?
Lesson D.2.4a
@std::int*_t isn't guaranteed to be available.

> How could I store @enc by point or reference?
Lesson 6.7 to 7.8

Abhishek
March 3, 2019 at 11:50 pm · Reply

It would be better if we leave 'parent' and 'child' words for Inheritance; for example in "The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.", I think. We can call the whole to be the 'Owner' and component to be 'Ownee'.

Alex
March 4, 2019 at 9:17 pm · Reply

That terminology makes sense for compositions, but not for other types of relationships that are non-owning. Parent-child isn't intended to imply anything about ownership.

Rimal Tahir
December 12, 2018 at 6:35 am · Reply

Is operator overloading compulsory here?

Alex
December 14, 2018 at 7:22 pm · Reply

If I'm understanding your question correctly, no. It's just used to allow our user-defined class to be printed using std::cout and operator<<. If you don't care about that, then there's no need to overload anything.

beginnerCoder
December 30, 2018 at 5:27 am · Reply

Hello! When defining the specific constructor in the Creature class, is it possible to initialize the parameters to default values like this:

```
Creature(const std::string &name="unnamed", const Point2D &location())
        : m_name(name), m_location(location)
```

to insert a creature with no name at the origin? I am in doubt about the syntax for initializing the location parameter. If I'm not mistaken location() should simply call the default constructor of Point2D if no parameter is specified by the user.

nascardriver
December 30, 2018 at 6:56 am · Reply

References have to reference something. @location doesn't reference anything at there's nothing it could reference. Use @std::reference_wrapper and set it to a nullptr or overload to constructor with a version that doesn't take a location.

AJ
November 22, 2018 at 11:23 pm · Reply

why we should use default constructor and setters both in creature program?? because both have same meanings.

Alex
November 27, 2018 at 12:46 pm · Reply

The Creature program doesn't have any setters, so I'm not sure what you're actually trying to ask...

seriouslysupersonic
August 25, 2018 at 3:02 am · Reply

Hi!

Regarding the #include directives @main, is it better practice to explicitly include the child class?

Alex
August 25, 2018 at 8:48 am · Reply

I don't understand the question. Can you clarify?

seriousltsupersonic
August 25, 2018 at 8:54 am · Reply

Since Creature uses Point2D is there a reason to include both @main?

Alex
August 25, 2018 at 11:58 am · Reply

It's a good practice to #include every header that contains classes that your .cpp file uses, even if another header may already include it.

Since main.cpp creates a Point2D to pass into the constructor, it should #include "Point2D.h".

hrmn
July 9, 2018 at 4:46 am · Reply

Point2D m_location; inside class is composition not aggregation right?, as it's not a pointer or reference

Alex
July 9, 2018 at 4:47 pm · Reply

Correct.

Abdul mannan
June 5, 2018 at 5:45 am · Reply

Can we do composition if parent class is abstract?

Alex
June 6, 2018 at 4:55 pm · Reply

Yes. You can even do composition in an abstract class (you just won't be able to instantiate that class directly).

boredprogrammer
May 14, 2018 at 2:45 pm · Reply

I think, a good thing to add to these parts of the tutorial are the UML diagrams where the ideas are extracted from. Just to post a picture of the UML diagram used to represent compositions, aggregations, etc. I think the direct connection from modelling a piece of software via UML to the implementation in ANY language, would make more evident the explained here. Ie, aggregations exists even when some lenguages don't use pointers or even references, but there are ways to have the same model of parts.

Just an idea... and also, boxes, arrows, diagrams are nice!

Aniket
May 5, 2018 at 3:53 am · Reply

You haven't answered the question "When should I use a subclass instead of direct implementation of a feature?"
Instead, after that question you have been answering "Why should I use a subclass instead of direct implementation of a feature?"

I want to understand that when part clearly.

Alex
May 6, 2018 at 9:36 pm · Reply

You should generally make a member a class type when the storage or manipulation of that member is non-trivial. For example, if a member needs to represent a point or a fraction, you should use a class, because the storage and manipulation of that data isn't trivial. If you need to store an employee's ID or age, use an int, because that is trivial.

Ritesh
February 20, 2018 at 11:00 am · Reply

Hey Alex,
I wrote the above program of creatures and point2d, by creating a Creature header and its cpp and same for Point2D I created both its header and cpp
I defined the friend function and set functions in the cpp files. heres the code

```
Point2d.h
#pragma once
#include<iostream>
class Point2D
{
  int m_x;
```

```
    int m_y;
public:
    Point2D(int x = 0, int y = 0) :m_x(x), m_y(y)
    {

    }
    friend std::ostream& operator<<(std::ostream &out, const Point2D &obj);
    void setPoint(int x = 0, int y = 0);
};
```

Point2D.cpp

```
#include "Point2D.h"
std::ostream& operator<<(std::ostream &out, const Point2D &obj)
{
    out << "(" << obj.m_x << "," << obj.m_y << ")\n";
    return out;
}
void Point2D::setPoint(int x, int y)
{
    m_x = x;
    m_y = y;
}
```

Creature.h

```
#pragma once
#include<string>
#include<iostream>
#include "Point2D.h"
class Creature
{
    std::string m_name;
    Point2D m_location;
public:
    Creature(const std::string &name, const Point2D &obj) :m_name(name), m_location(obj)
    {

    }
    friend std::ostream& operator<<(std::ostream &out, const Creature &obj);
    void moveTo(int x = 0, int y = 0);
};
```

Creature.cpp

```
#include "Creature.h"
std::ostream& operator<<(std::ostream &out, const Creature &obj)
{
    out << obj.m_name << " is at " << obj.m_location<<'\n';
    return out;
}
void Creature::moveTo(int x, int y)
{
    m_location.setPoint(x, y);
}
```

and Finally main.cpp

```
#include "stdafx.h"
#include<iostream>
#include "Point2D.h"
#include "Creature.h"
```

using namespace std;
int main()
{
    cout << "Enter the name of your creature";
    string name;
    cin >> name;
    Creature creature(name, Point2D(4, 12));
    cout << creature; // this line gives linker error
    return 0;
}
and the linker error is LNK2019, unresolved externals referenced in main, and when I'll comment this line cout<< creature, then code compiles and links fine. I am not understanding what's going wrong in this code, why my linker is not linking the definition of the overloaded cout << creature;

Alex
February 22, 2018 at 9:42 am · Reply

Not sure. When I put this all in one file, it compiled fine. Therefore, it appears like the issue here is that the linker is not able to find the definition of operator<< for Creature. That suggests to me that maybe you forgot to add Creature.cpp to your project, so it's not actually being compiled/linked when you compile your program.

Ryder
February 18, 2018 at 7:11 pm · Reply

What I learn from this section.

Composition makes programmers less worry about the deletion of the object. In most of the case, this type of relation guarantee the existance of the member in such a way it automatically knows when to birth and to death.

Some questions that I do not understand:

What is the difference between a default constructor and a specific one? Why we need two here? What is the meaning of the default constructor? What kind of date type is m_x and m_y in the default constructor?

```
    Point2D()
        : m_x(0), m_y(0)
    {
    }

    // A specific constructor
    Point2D(int x, int y)
        : m_x(x), m_y(y)
    {
    }
```

What does it mean? If I don't want to use overload operator how to write this piece of code?

```
    friend std::ostream& operator<<(std::ostream& out, const Point2D &point)
    {
        out << "(" << point.m_x << ", " << point.m_y << ")";
        return out;
}
```

nascardriver
February 19, 2018 at 3:08 am · Reply

Hi Ryder!

> What is the difference between a default constructor and a specific one? Why we need two here?

```
Point2d pt1{ }; // Calls default constructor, without it this wouldn't be allowed
Point2d pt2; // Same as above

Point2d pt3{ 22, 12 }; // Calls the Point2d(int, int) constructor which sets the poin
t's x and y values
Point2d pt4(22, 12); // Same as above
Point2d pt5 = Point2d(22, 12) // Same as above
```

> What is the meaning of the default constructor?
The default constructor gets called whenever you create an object without passing arguments to the constructor.

> What kind of date type is m_x and m_y in the default constructor?
The are the local variables @m_x and @m_y

```
class Point2D
{
private:
  int m_x; // Here
  int m_y; // And here
  ...
```

> What does it mean? If I don't want to use overload operator how to write this piece of code?
You can't do that without overloading the << operator.

Nur
November 4, 2017 at 12:42 am · Reply

Hello Alex,
Hope you are doing well.
I have a question regarding to above given Creature class.
My question:

Can we consider internally (or separately) the relationship between Creature class's object and std::ostream's object std::cout as Dependencies, although you have given as a composition between Creature and Point2D classes?

If what I said is correct then it is worth noting that the concepts of composition and dependencies are not mutually exclusive, and can be mixed freely within the same class.
Am I right?. If not, please could you clarify it?

Thanks for a great tutorial!I have never seen such a tutorial in C++. It is the most precious one.

Alex
November 5, 2017 at 8:34 am · Reply

Yes, I'd say std::cout is a dependency in this case, as the class uses it temporarily. You're correct in that the various relationships aren't exclusive -- a single class might contain an composition element, an association element, and a dependency.

Nurlan
October 19, 2017 at 9:00 am · Reply

Hello Alex,
I hope you are fine.
As you said to qualify as a composition, an object and a part must have the following relationship:
One of them is The part (member) can only belong to one object (class) at a time.

And you gave the Fraction class as a great example of a composition:

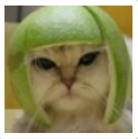
```
class Fraction
{
private:
   int m_numerator;
   int m_denominator;

public:
   Fraction(int numerator=0, int denominator=1):
   m_numerator(numerator), m_denomin1ator(denominator)
   {
   }
};

//Now, let's write the main() function by myself
int main()
{
  Fraction f1(3, 4);
  Fraction &f2=f1;//is this still composition since now part (members)belongs two //objects, which has the same
address,i.e. f2 is reference to f1;

}
```

Is this composition even it is belongs to two objects which has the same address, or it is considered to one object since it is alias name of f1?
Thanks in advance!



Alex
October 21, 2017 at 10:43 am · Reply

Fraction is a composition since the parts (m_numerator and m_denominator) belong to the whole (the Fraction object).

The fact that you have a second variable referencing the first variable is immaterial. f1 owns its members. f2 is just a reference to f1, and has no bearing on ownership.

Zoran
July 5, 2017 at 12:57 am · Reply

In Creature constructor, why do you pass name by value, unlike location?

That is, instead of

```
Creature(std::string name, const Point2D &location)
        : m_name(name), m_location(location)
{
}
```

wouldn't this be better:

```
Creature(const std::string &name, const Point2D &location)
        : m_name(name), m_location(location)
{
}
```

Is there a reason?

Alex
July 5, 2017 at 3:45 pm · Reply

Oversight on my part. I've updated it to use a const reference.

Omri
June 17, 2017 at 9:31 pm · Reply

"the heart is now owned by the donor, and can only be part of the donor object unless transferred again"
Perhaps:
"the heart is now owned by the receiver only, and can only be part of the receiver object (no more belongs to the doner, no mutual ownership) until transferred again"

Alex
June 18, 2017 at 8:18 pm · Reply

Fixed. Thanks for pointing that typo out.

Omri
June 17, 2017 at 9:22 pm · Reply

Typo:
"For example, a heart is an part of a person's body."
=>
"For example, a heart is *a* part of a person's body."

Rohit
March 19, 2017 at 7:39 am · Reply

Hi Alex!

1) Suppose we create a .h file which contain function declarations and another .cpp file which contains function definition. Now if we add .h file to our program, does it itself include .cpp file or we need to add .cpp file also and how?

2) Is file handling discussed in these tutorials?

Alex
March 20, 2017 at 8:42 am · Reply

1) You have to add the .cpp file into your project yourself. This is discussed back in chapter 1.
2) Yep, in chapter 18.

robbe
March 4, 2017 at 9:49 am · Reply

I'm probably wrong but since a programmer has no way of knowing a Point2D object contains merely fundamental type member variables, shouldn't the m_location variable (of type Point2D) in the Creature class be dynamically allocated and destroyed according to RAII?

Alex
March 4, 2017 at 1:38 pm · Reply

No. Classes are expected to clean up after themselves. Point2D should clean up itself when it is destroyed regardless of how it is implemented. Any class using Point2D should not need to worry about this, regardless of whether the class is dynamically allocated or not.

Michiel
February 15, 2017 at 2:57 am · Reply

In main.cpp you forgot to add std:: before the first cout

```
cout << "Enter a name for your creature";
```

Alex
February 15, 2017 at 11:28 am · Reply

Fixed, thanks!

« Older Comments 1 2