C++ Program Design
-- Composition & Inheritance

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

Composition

- In real-life, complex objects are often built from smaller, simpler objects.
- *has-a* relationship
  - *PC has-a* CPU, a motherboard

```
#include "CPU.h"
#include "Motherboard.h"
#include "RAM.h"

class PersonalComputer{
private:
    CPU m_cCPU;    Motherboard m_cMotherboard;
    RAM m_cRAM;
};
```

Initializing class member variables

```
PersonalComputer::PersonalComputer(int nCPUSpeed,
                                   char *strMotherboardModel,
                                   int nRAMSize)
    : m_cCPU(nCPUSpeed),
     m_cMotherboard(strMotherboardModel),
     m_cRAM(nRAMSize)
{
}
```

Why use composition?

- Each individual class can be kept relatively simple and straightforward, focused on performing one task.
- Each subobject can be self-contained, which makes them reusable.
  - reuse our Point2D

- each class should be built to accomplish a single task. That task should either be
  - **the storage and manipulation of some kind of data (eg. Point2D),**
  - **OR the coordination of subclasses (eg. Creature).**
  - **Not both.**

Aggregation

- in a composition, the complex object "**owns**" all of the subobjects it is composed of.
- When a composition is destroyed, all of the **subobjects are destroyed** as well.
  - If you destroy a PC, you would expect it's RAM and CPU to be destroyed as well.

- An **aggregation** is a specific type of composition where **no ownership** between the complex object and the subobjects is implied.
- When an aggregate is destroyed, the subobjects are **not destroyed**.
  - math department of a school, made up of teachers.
  - The department should be an aggregate.
  - When the department is destroyed, the teachers should still exist independently (they can go get jobs in other departments).

Aggregation

```
class Teacher
{
private:
    string m_strName;
public:
    Teacher(string strName)
        : m_strName(strName)
    {}

    string GetName()
    { return m_strName; }
};
```

```
class Department
{
private:
    Teacher *m_pcTeacher; // Thi
s dept holds only one teacher

public:
    Department(Teacher *pcTeache
r=NULL) : m_pcTeacher(pcTeacher)
    {}
};
```

```
int main(){
    // Create a teacher outside the scope of the Department
    Teacher *pTeacher = new Teacher("Bob"); // create a teacher
    {
        // Create a department and use the constructor parameter to pass
        // the teacher to it.
        Department cDept(pTeacher);

    } // cDept goes out of scope here and is destroyed

    // pTeacher still exists here because cDept did not destroy it
    delete pTeacher;
}
```

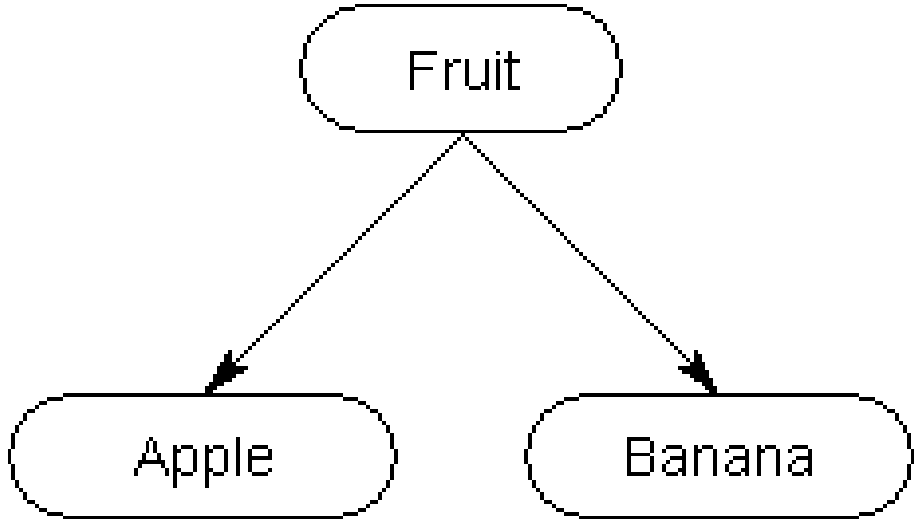
Compositions vs. Aggregations

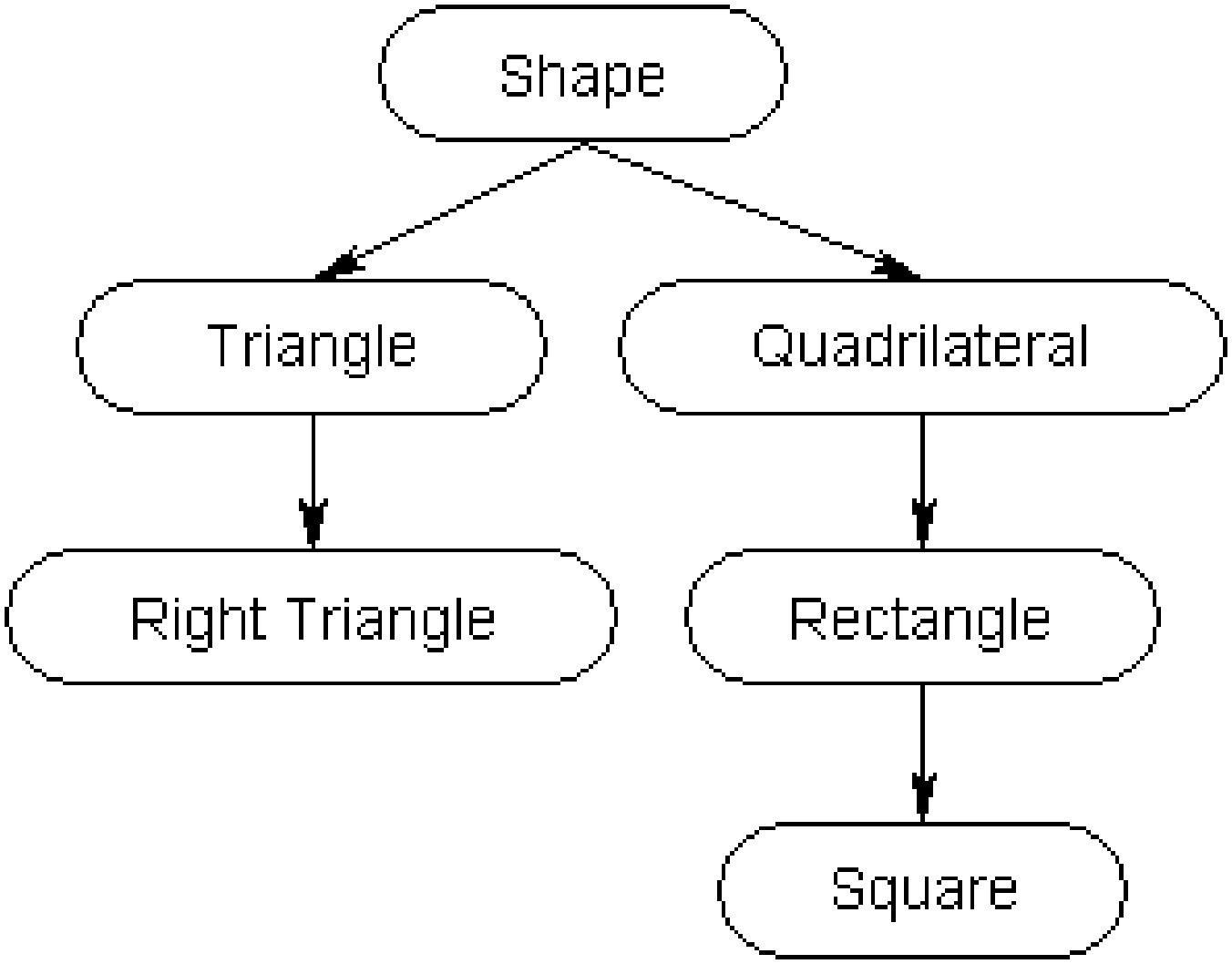
- Compositions:
  - Typically use normal member variables
  - Can use pointer values if the composition class automatically handles allocation/deallocation
  - Responsible for creation/destruction of subclasses
- Aggregations:
  - Typically use pointer variables that point to an object that lives outside the scope of the aggregate class
  - Can use reference values that point to an object that lives outside the scope of the aggregate class
  - Not responsible for creating/destroying subclasses

inheritance

How to construct complex classes

- Has-a
  - Composition
  - Aggregation
- Is-a: Inheritance
  - **parent** or **base**
  - **child** or **derived** object





Why the need for inheritance in C++?

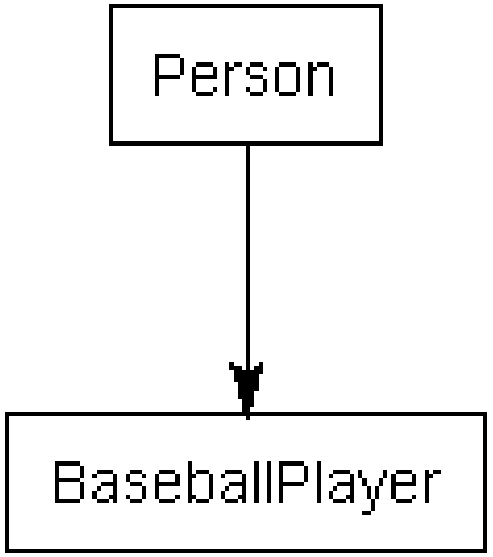
- Reusable
- However, existing code often does not do EXACTLY what you need it to.
  - what if you have a triangle and you need a right triangle?

a) change the existing code to do what you want.
   - no longer be able to use it for its original purpose

b) make a copy of some or all of the existing code and change it to do what we want.
   - maintenance problem: Improvements or bug fixes have to be added to multiple copies of functions

Basic inheritance in C++

```
class Person{
public:
    std::string m_strName;
    int m_nAge;    bool m_bIsMale;

    std::string GetName() { return m_strName; }
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }

    Person(std::string strName = "", int nAge = 0, bool bIsMale
= false) : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    { }  };
```

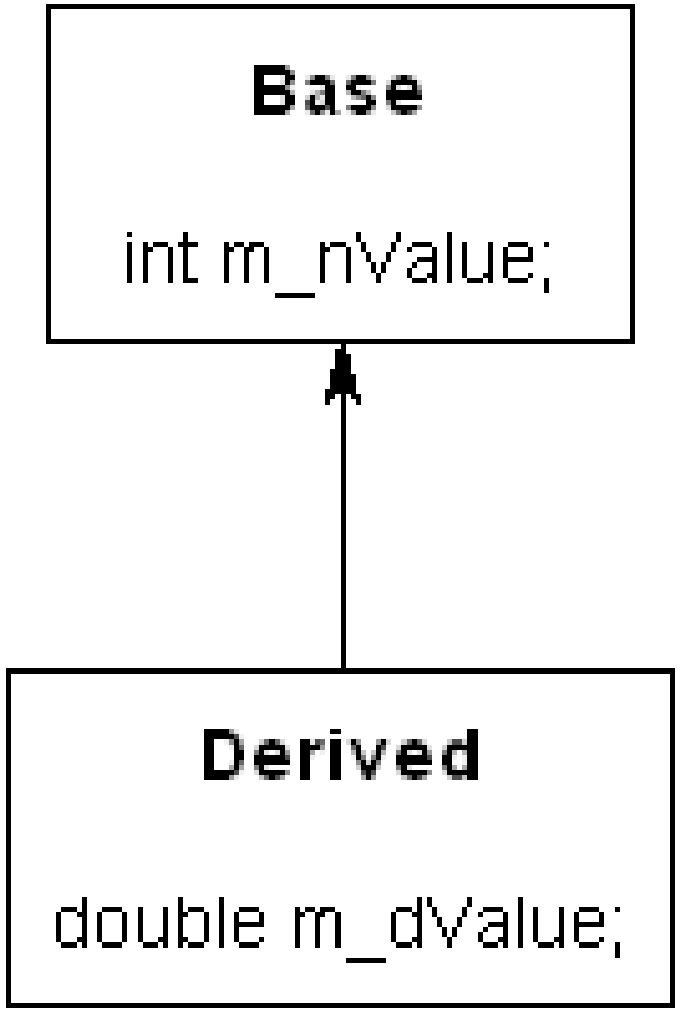



```
class BaseballPlayer : public Person{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns =
0) : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    { }
};
```

- BaseballPlayer cJoe;
- cJoe.m_strName = "Joe";
- std::cout << cJoe.GetName() << std::endl;

Order of construction of derived classes

```
class Base{
public:  int m_nValue;
    Base(int nValue=0)
        : m_nValue(nValue){}
};

class Derived: public Base{
public:  double m_dValue;
    Derived(double dValue=0.0)
        : m_dValue(dValue){}
};
```

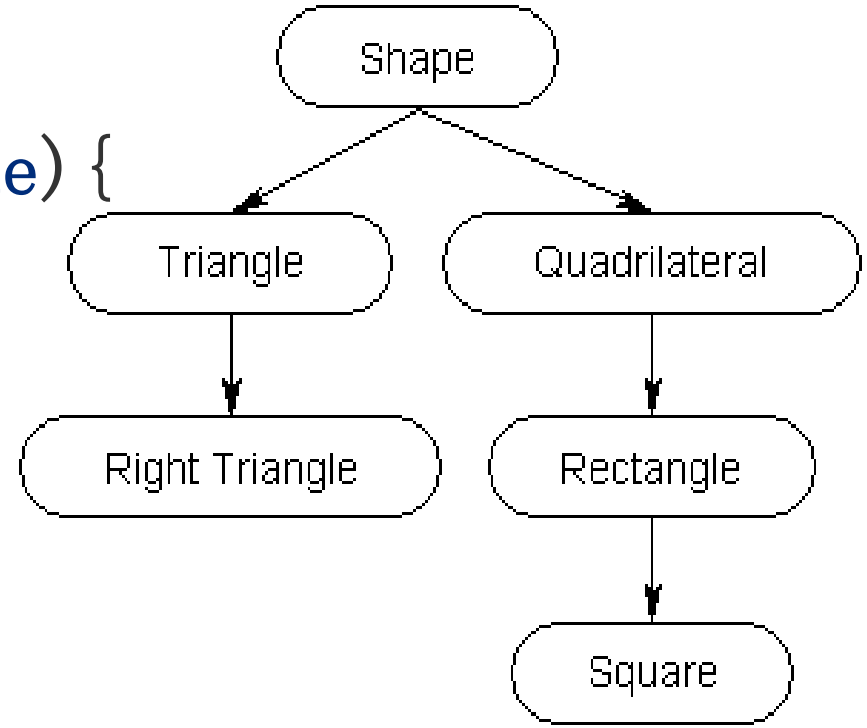


what happens when we instantiate a derived class

```
int main()
{
    Derived cDerived;

    return 0;
}
```

```
Base(int nValue=0): m_nValue(nValue){
        cout << "Base" << endl;}

Derived(double dValue=0.0): m_dValue(dValue){
        cout << "Derived" << endl;}

Derived cDerived;
```

Base
Derived



C++ always constructs the “first” or “most base” class first. It then walks through the inheritance tree in order and constructs each suc cessive derived class.

what actually happens when cDerived is instantiated?

1. Memory for cDerived is set aside (enough for both the Base and Derived portions).
2. The appropriate Derived constructor is called
3. **The Base object is constructed first using the appropriate Base constructor**
4. The initialization list initializes variables
5. The body of the constructor executes
6. Control is returned to the caller

Initializing base class members

```
class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0, int nValue=0)
        // does not work
        : m_dValue(dValue), m_nValue(nValue)
    {
    }
};
```

what would happen if m_nValue were const

Initializing base class members

```
class Derived: public Base{
public:
    double m_dValue;

    Derived(double dValue=0.0, int nValue=0)
        : Base(nValue), // Call Base(int) constructor with value nValue!
            m_dValue(dValue)
    {
    }
};
Derived cDerived(1.3, 5); // use Derived(double) constructor
```

Initializing base class members

1. Memory for cDerived is allocated.
2. The Derived(double, int) constructor is called, where dValue = 1.3, and nValue = 5
3. The compiler looks to see if we've asked for a particular Base class constructor. We have! So it calls Base(int) with nValue = 5.
4. The base class constructor initialization list sets m_nValue to 5
5. The base class constructor body executes
6. The base class constructor returns
7. The derived class constuctor initialization list sets m_dValue to 1.3
8. The derived class constructor body executes
9. The derived class constructor returns

```
Person(std::string strName = "", int nAge = 0, bool bIsMale = fa
lse) : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    {}

class BaseballPlayer : public Person{
public:
    double m_dBattingAverage;    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
    : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    { }
```

it makes sense to give our BaseballPlayer a name and age when we cre ate them, How?

```
BaseballPlayer(std::string strName = "", int nAge = 0,
                                        bool bIsMale = false,
        double dBattingAverage = 0.0, int nHomeRuns = 0)
        : Person(strName, nAge, bIsMale),
          m_dBattingAverage(dBattingAverage),
          m_nHomeRuns(nHomeRuns)
    { }

BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);
```

destructor

- When a derived class is destroyed, each destructor is called in the reverse order of construction.

protected

```
class Base
{
public:
    int m_nPublic; // can be accessed by anybody
private:
    int m_nPrivate; // can only be accessed by Base member functions (but not derived classes)
protected:
    int m_nProtected; // can be accessed by Base member functions, or derived classes.
};
```

inheritance type – a complex topic

```
// Inherit from Base publicly
class Pub: public Base{};

// Inherit from Base privately
class Pri: private Base{};

// Inherit from Base protectedly
class Pro: protected Base{};

class Def: Base // Defaults to private inheritance{};
```

Adding, changing, and hiding members in a derived class

Adding new functionality

```
class Base{
protected:
int m_nValue;

public:
Base(int nValue)
    : m_nValue(nValue)
{}

void Identify() { cout << "I am a Base" << endl; }
};
```

```
class Derived: public Base
{
public:
    Derived(int nValue)
        :Base(nValue)
    { }

    int GetValue() { return m_nValue; }
};
```

Redefining functionality

```
class Derived: public Base{
public:
    // Here's our modified function
    void Identify() { cout << "I am a Derived" << endl; }
};
```

- Base cBase(5);
- cBase.Identify();
-
- Derived cDerived(7);
- cDerived.Identify()

I am a Base
I am a Derived

Adding to existing functionality

```
class Derived: public Base{
public:
    void Identify(){
        Base::Identify(); // call Base::Identify() first
        cout << "I am a Derived"; // then identify ourselves
    }
};

void Identify(){
   Identify(); // would be Derived::Identify() => infinite loop!
   cout << "I am a Derived"; // then identify ourselves
}
```

Hiding functionality

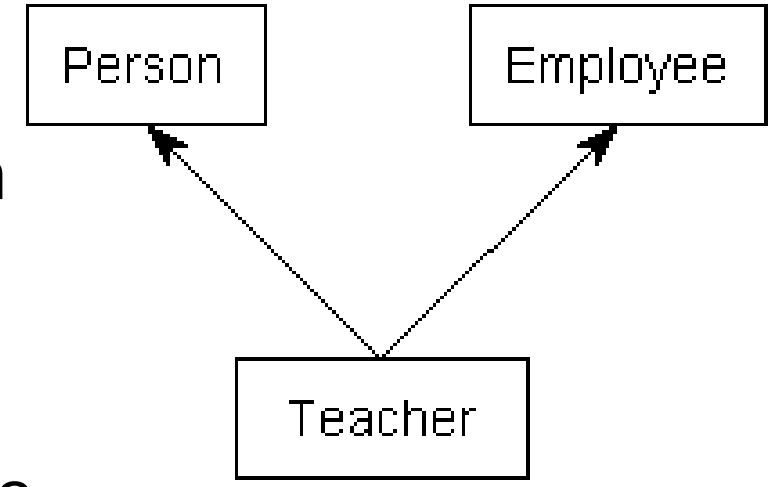
- In C++, it is not possible to remove functionality from a class. However, it is possible to hide existing functionality.

```
class Base{
protected:
    void PrintValue() { cout << m_nValue; }
};
class Derived: public Base{
public:
    Base::PrintValue;
};
            // PrintValue is public in Derived, so this is okay
            cDerived.PrintValue(); // prints 7
```

```
class Base{
public:    int m_nValue;
};

class Derived: public Base{
private:    Base::m_nValue;
};

int main(){
    Derived cDerived(7);

    // The following won't work because m_nValue has been redefi
ned as private
    cout << cDerived.m_nValue;
```

Multiple inheritance

- multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare.



- most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well.

- Many object-oriented languages (eg. Smalltalk, PHP) do not even support multiple inheritance.

- Many relatively modern languages such as Java and C# restricts classes to single inheritance of normal classes, but allow multiple inheritance of interface classes