Creative Software Design

8 – Inheritance, Const & Class

Yoonsang Lee
Fall 2022

Today's Topics

- Brief intro to "Fundamental Principles of Object-Oriented Programming"

- Inheritance
  - Concept
  - Overriding
  - Constructor & Destructor with Inheritance
  - Member initializer list with Inheritance
  - Multiple Inheritance

- Const & Class

Fundamental Principles of Object-Oriented Programming

- Encapsulation (already covered in 6-Class)
  - Binding the data with the code that manipulates it into a single unit
  - → Hiding details of the unit (data hiding).

- Inheritance (Today's topic)
  - When a class inherits another class, it has the same behavior or characteristics of another.

- Polymorphism (Next lecture)
  - The ability to create a variable, a function, or an object that has more than one form.

- Abstraction (closely related to polymorphism)
  - The principle of generalization - from a specific instance to a more generalized concept.

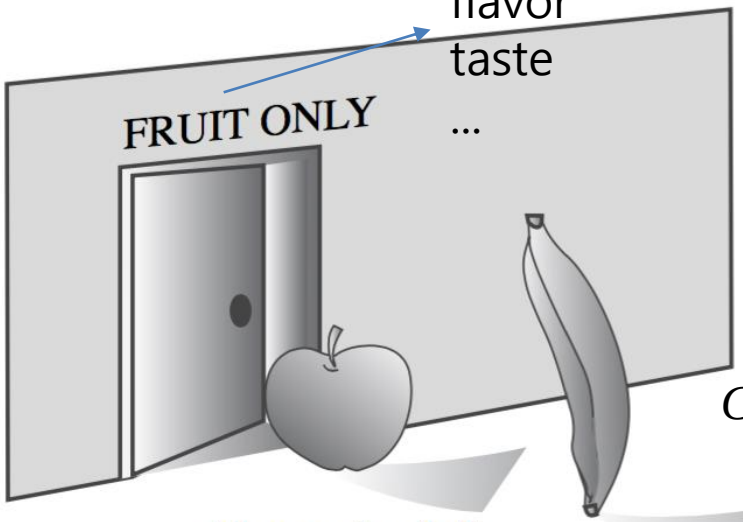
Inheritance

- Building a class on the top of an existing class.

- The goal is to
  - reuse the code for similar functionalities
  - and write new code only for additional functionalities.

- This allows you to establish **relationships** between classes.

Inheritance: is-a relationship

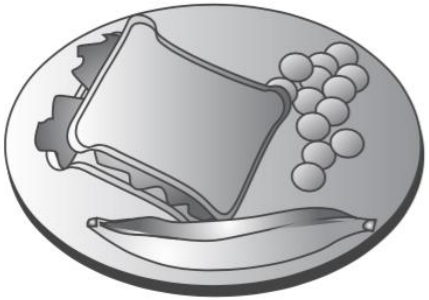


Class Banana inherits from *class Fruit.*

A banana *is a* fruit,
but
a lunch *has* a banana.

Inheritance: is-a relationship

Composition: has-a relationship



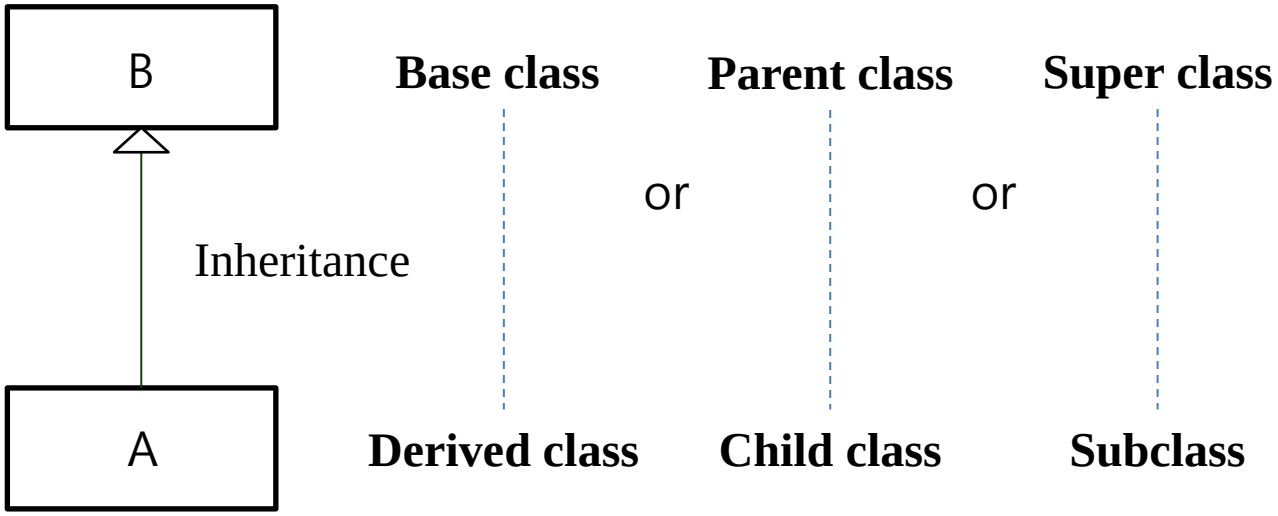
Class Lunch has a *class Banana* instance as a member variable.

Inheritance: is-a relationship

- "is-a" relationship: use (public) inheritance when "A" is a "B".
  - A car is a vehicle.
    A truck is a vehicle.
    A cart is a vehicle.
    …
  - A student is a person.
    A professor is a person.
    …
  - A person is an animal.
    A dog is an animal.
    …

Inheritance

- If a class A inherits from another class B,
  - Class A implicitly "has" the member variables and functions of class B.
  - Class A can have additional member variables and functions.

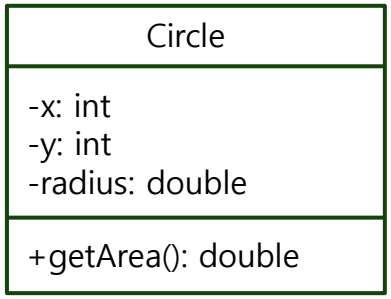



```
class A : public B
{
  ...
};
```

(UML class diagram)

UML Class Diagram Example

Unified Modeling Language (UML): for visualizing the design of a software system.



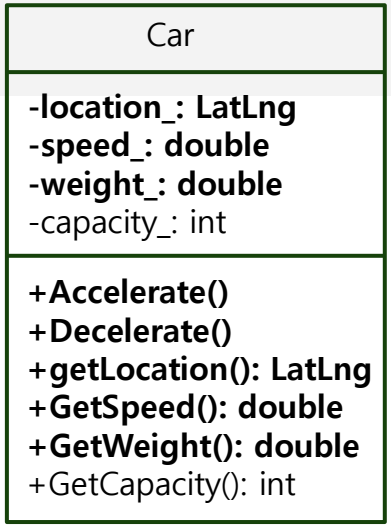
+: public
-: private
#: protected

variable: data type
method(parameter): return type

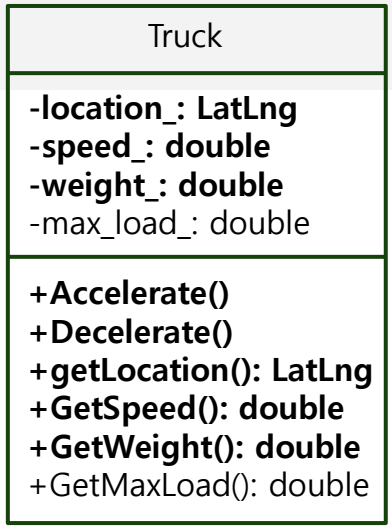
```
#include <iostream>
using namespace std;

class Circle {

private:
    int x, y;
    double radius;
public:
    Circle(int px, int py, double pradius) {
        x=px, y=py, radius=pradius;}
    double getArea() { return 3.14*radius*radius; }
};

int main()
{
    Circle c(2,3,4);
    cout << c.getArea() << endl;

}
```

An Inheritance Example

```
class Car {
 public:
  Car() {}
  void Accelerate();
  void Decelerate();
  LatLng GetLocation();
  double GetSpeed();
  double GetWeight();
  int GetCapacity();
 private:
  LatLng location_;
  double speed_;
  double weight_;
  int capacity_;
};
```

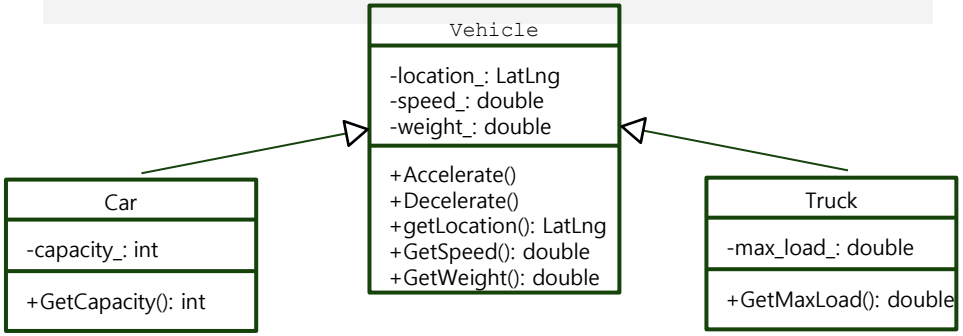



```
class Truck {
 public:
  Truck() {}
  void Accelerate();
  void Decelerate();
  LatLng GetLocation();
  double GetSpeed();
  double GetWeight();
  double GetMaxLoad();
 private:
  LatLng location_;
  double speed_;
  double weight_;
  double max_load_;
};
```



An Inheritance Example

```
// Vehicle class.

class Vehicle {
 public:
  Vehicle() {}
  void Accelerate();
  void Decelerate();

  LatLng GetLocation();
  double GetSpeed();
  double GetWeight();

 private:
  LatLng location_;
  double speed_;
  double weight_;
};
```

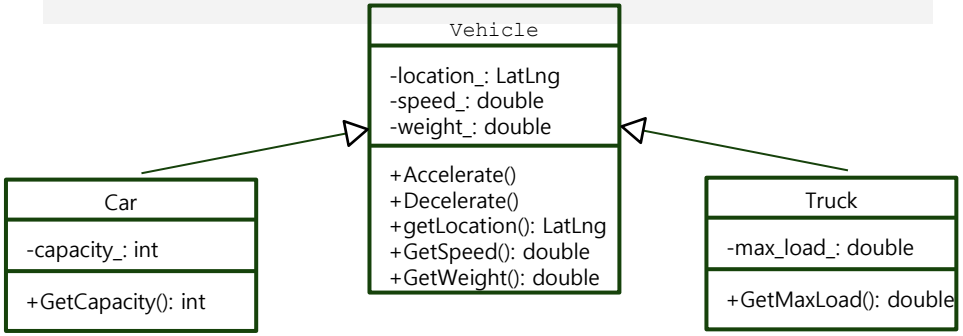



```
// Car class.

class Car : public Vehicle {
 public:
  Car() : Vehicle() {}

  int GetCapacity();

 private:
  int capacity_;
};
```

```
// Truck class.

class Truck : public Vehicle {
 public:
  Truck() : Vehicle() {}

  double GetMaxLoad();

 private:
  double max_load_;
};
```

An Inheritance Example

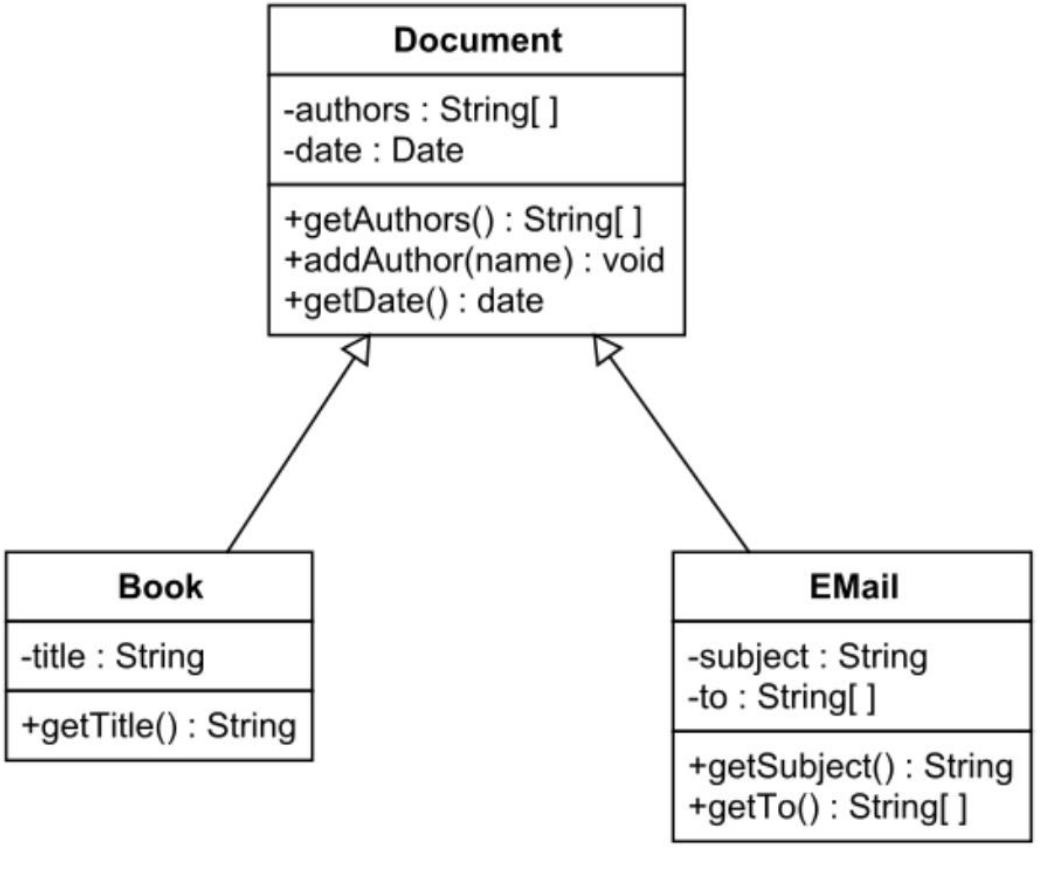
```
// Vehicle class.

class Vehicle {
 public:
  Vehicle() {}
  void Accelerate();
  void Decelerate();

  LatLng GetLocation();
  double GetSpeed();
  double GetWeight();

 private:
  LatLng location_;
  double speed_;
  double weight_;
};
```




```
// Car class.

class Car : public Vehicle {
 public:
  Car() : Vehicle() {}

  int GetCapacity();

 private:
  int capacity_;
};
```

```
// Main routine.

int main() {
  Car car;
  cout << car.GetCapacity() << endl;
  cout << car.GetSpeed() << endl;
  cout << car.GetWeight() << endl;
  return 0;
}
```

Another inheritance example



Quiz #1

- Go to https://www.slido.com/
- Join #**csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2022123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Overriding vs. Overloading

- Function overloading (함수 중복정의)
  - provides multiple definitions of function by changing signatures (i.e. changing the number, order, or data type of parameters but leaving the function name the same)
  - has nothing to do with inheritance
  - should be used in the same scope

```
int print(int a) { … }

int print(int a, int b) { … }
```

- **Function overriding (함수 재정의)**
  - **Redefinition of base class function** in the derived class with same signatures

Overriding Member Function

- You can override a member function to provide a custom functionality of the derived class.

- Redefine a member function with the same name as the inherited functio n.
  - All ancestor's member functions with the same name will be occluded.
  - To access the ancestor's member functions, use `Ancestor::MemberFunction()`.

An example of overriding

```
// Vehicle class.

class Vehicle {
 public:
  Vehicle() {}
  void Accelerate();
  void Decelerate();

  LatLng GetLocation();
  double GetSpeed();
  double GetWeight();

 private:
  LatLng location_;
  double speed_;
  double weight_;
};
```

```
// Car class.
class Car : public Vehicle {
 public:
  Car() : Vehicle() {}

  int GetCapacity();

  // Override the parent's GetWeight().
  double GetWeight() {
    return Vehicle::GetWeight()+passenger_weight_;
  }
 private:
  int capacity_;
  double passenger_weight_;
};
```

```
// Main routine.

int main() {
  Car car;
  cout << car.GetCapacity() << endl;
  cout << car.GetSpeed() << endl;
  cout << car.GetWeight() << endl;
  return 0;
}
```

An example of overriding

```
// Vehicle class.

class Vehicle {
 public:
  Vehicle() {}
  void Accelerate();
  void Decelerate();

  LatLng GetLocation();
  double GetSpeed();
  double GetWeight();

 private:
  LatLng location_;
  double speed_;
  double weight_;
};
```

```
// Car class.
class Car : public Vehicle {
 public:
  Car() : Vehicle() {}

  int GetCapacity();

  // Override the parent's GetWeight().
  double GetWeight() {
    return Vehicle::GetWeight()+passenger_weight_;
  }
 private:
  int capacity_;
  double passenger_weight_;
};
```

=weight_?

```
// Main routine.

int main() {
  Car car;
  cout << car.GetCapacity() << endl;
  cout << car.GetSpeed() << endl;
  cout << car.GetWeight() << endl;
  return 0;
}
```

An example of overriding

```
// Vehicle class.

class Vehicle {
 public:
  Vehicle() {}
  void Accelerate();
  void Decelerate();

  LatLng GetLocation();
  double GetSpeed();
  double GetWeight();

 protected:
  LatLng location_;
  double speed_;
  double weight_;
};
```

public: everyone can access.
private: only its member functions can access.
protected: its member functions and the member functions of descendant classes can access.

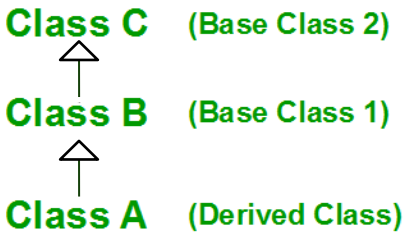
```
// Car class.
class Car : public Vehicle {
 public:
  Car() : Vehicle() {}

  int GetCapacity();
  // Override the parent's GetWeight().
  double GetWeight() {
    return weight_ + passenger_weight_;
  }
 private:
  int capacity_;
  double passenger_weight_;
};
```

```
// Main routine.
int main() {
  Car car;
  cout << car.GetCapacity() << endl;
  cout << car.GetSpeed() << endl;
  cout << car.GetWeight() << endl;
  return 0;
}
```

Constructor & Destructor with Inheritance

- Constructor and destructor call order:
  - Constructors are called from base class to derived class.
  - Destructors are called in reverse order.

Order of Inheritance



Order of Constructor Call

1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

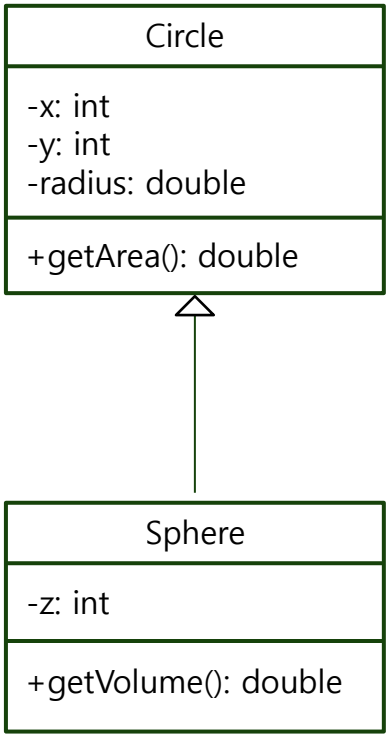
Constructor & Destructor with Inheritance: Example 1

```
class Parent {
 public:
  Parent() { cout << " Parent"; }
  ~Parent() { cout << " ~Parent"; }
};

class Child : public Parent {
 public:
  Child() { cout << " Child"; }
  ~Child() { cout << " ~Child"; }
};

class Test : public Child {
 public:
  Test() { cout << " Test"; }
  ~Test() { cout << " ~Test"; }
};
```

```
int main() {
  {
    Child child;
    cout << endl;
  }
  cout << endl;
  {
    Test test;
    cout << endl;
  }
  cout << endl;
  return 0;
}
```

```
 Parent Child
 ~Child ~Parent
 Parent Child Test
 ~Test ~Child ~Parent
```

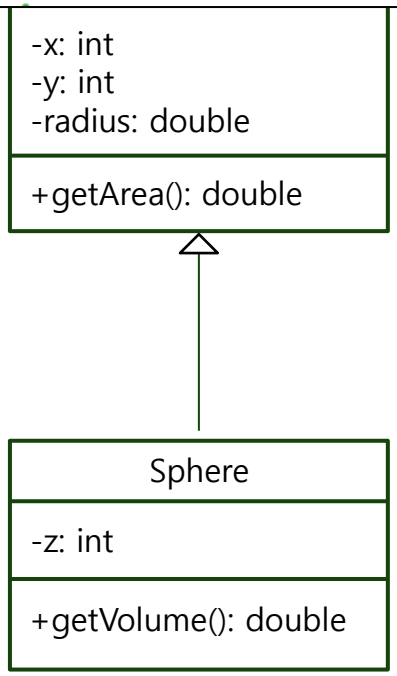
Constructor & Destructor with Inheritance: Example 2




```cpp
#include <iostream>
using namespace std;

class Circle {
private:
    int x, y;
    double radius;
public:
    Circle(int px, int py, double pradius) {
        x=px; y=py; radius=pradius;}
    double getArea() { return 3.14*radius*radius; }
};

class Sphere: public Circle{
private:
    int z;
public:
    Sphere(int px, int py, double pradius, int pz){
        cout << "Sphere" << endl;
        x=px; y=py; radius=pradius; z=pz;}
    double getVolumn(){
        return 4.0/3*3.14*radius*radius*radius;
    }
};

int main()
{
    Circle c(2,3,4.0);
    cout << c.getArea() << endl;
    Sphere s(2,3,4.0,5);
    cout << s.getVolumn() << endl;
    return 0;
}
```

: Example 2

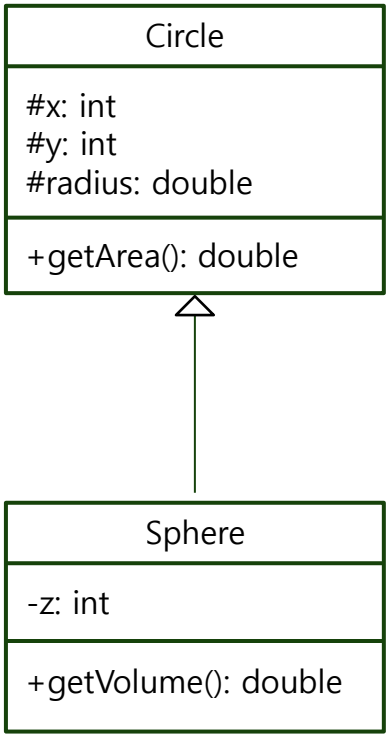
```
8_10.cc:18:5: error: constructor for 'Sphere' must explicitly initialize the
      base class 'Circle' which does not have a default constructor
    Sphere(int px, int py, double pradius, int pz){
    ^
8_10.cc:4:7: note: 'Circle' declared here
class Circle {
      ^
8_10.cc:20:9: error: 'x' is a private member of 'Circle'
        x=px; y=py; radius=pradius; z=pz;}
        ^
8_10.cc:6:9: note: declared private here
    int x, y;
```




```
public:
    Circle(int px, int py, double pradius) {
        x=px; y=py; radius=pradius;}
    double getArea() { return 3.14*radius*radius; }
};

class Sphere: public Circle{
private:
    int z;
public:
    Sphere(int px, int py, double pradius, int pz){
        cout << "Sphere" << endl;
        x=px; y=py; radius=pradius; z=pz;}
    double getVolumn(){
        return 4.0/3*3.14*radius*radius*radius;
    }
};

int main()
{
    Circle c(2,3,4.0);
    cout << c.getArea() << endl;
    Sphere s(2,3,4.0,5);
    cout << s.getVolumn() << endl;
    return 0;
}
```

implicitly calls Circle's default constructor which is not defined

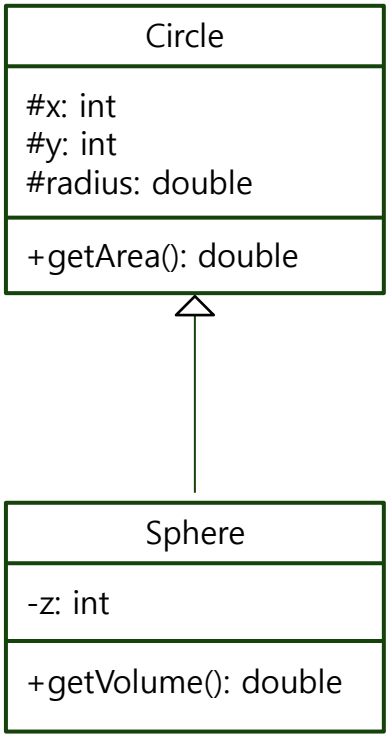
Constructor & Destructor with Inheritance: Example 2




```cpp
#include <iostream>
using namespace std;

class Circle {
protected:
    int x, y;
    double radius;
public:
    Circle(){ cout << "Circle: no parameter" << endl;  }
    Circle(int px, int py, double pradius) {
        cout << "Circle: with parameters" << endl;
        x=px; y=py; radius=pradius;}
    double getArea() { return 3.14*radius*radius; }
};

class Sphere: public Circle{
private:
    int z;
public:
    Sphere(int px, int py, double pradius, int pz){
        cout << "Sphere" << endl;
        x=px; y=py; radius=pradius; z=pz;}
    double getVolumn(){
        return 4.0/3*3.14*radius*radius*radius;
    }
};

int main()
{
    Circle c(2,3,4.0);
    cout << c.getArea() << endl;
    Sphere s(2,3,4.0,5);
    cout << s.getVolumn() << endl;
    return 0;
}
```

```
Circle: with parameters
50.24
Circle: no parameter
Sphere
267.947
```

Constructor & Destru

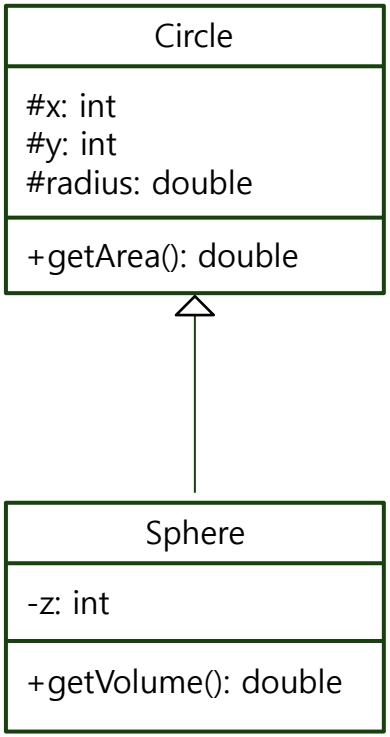



```
#include <iostream>
using namespace std;

class Circle {
protected:
    int x, y;
    double radius;
public:

    Circle(int px, int py, double pradius) {
        cout << "Circle: with parameters" << endl;
        x=px; y=py; radius=pradius;}
    double getArea() { return 3.14*radius*radius; }
};

class Sphere: public Circle{
private:
    int z;
public:
    //Sphere(int px, int py, double pradius, int pz){
    //  cout << "Sphere" << endl;
    //      x=px; y=py; radius=pradius; z=pz;}
    Sphere(int px, int py, double pradius, int pz):
      Circle(px, py, pradius), z(pz){
        cout << "Sphere" << endl;
      }
    double getVolumn(){
        return 4.0/3*3.14*radius*radius*radius;
    }
};

int main()
{
    Circle c(2,3,4.0);
    cout << c.getArea() << endl;
    Sphere s(2,3,4.0,5);
    cout << s.getVolumn() << endl;
    return 0;
}
```

explicitly calls Circle's constructor

```
Circle: with parameters
50.24
Circle: with parameters
Sphere
267.947
```

Constructor & Dest




```cpp
#include <iostream>
using namespace std;

class Circle {
protected:
    int x, y;
    double radius;
public:
    //Circle(){ cout << "Circle: no parameter" << endl;  }
    //Circle(int px, int py, double pradius) {
    //    cout << "Circle: with parameters" << endl;
    //    x=px; y=py; radius=pradius;}
    double getArea() { return 3.14*radius*radius; }
};

class Sphere: public Circle{
private:
    int z;
public:
    Sphere(int px, int py, double pradius, int pz){
        cout << "Sphere" << endl;
          x=px; y=py; radius=pradius; z=pz;}
    //Sphere(int px, int py, double pradius, int pz):
    //  Circle(px, py, pradius), z(pz){
    //    cout << "Sphere" << endl;
    //  }
    double getVolumn(){
        return 4.0/3*3.14*radius*radius*radius;
    }
};

int main()
{
  //Circle c(2,3,4.0);
  //  cout << c.getArea() << endl;
    Sphere s(2,3,4.0,5);
    cout << s.getVolumn() << endl;
    return 0;
}
```

```
Sphere
267.947
```

Quiz #2

- Go to https://www.slido.com/
- Join #**csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Member initializer list with Inheritance

- You can't initialize (by a member initializer list) a parent class member in the child class.

- The child class can indirectly initialize the parent's members by calling the parent's constructor in its member initializer list.

```
class A
{
public:
    int memberA;
    A(int n):memberA(n) { }
};

class B: public A
{
public:
    B():memberA(10) {}  // error
};
```

```
class A
{
public:
    int memberA;
    A(int n):memberA(n) { }
};

class B: public A
{
public:
    B():A(10) {}  // Ok
};
```

Person Example - outline

```
// Person class.

class Person {
 public:
  Person(const string& name);

  const string& name();
  const string& address();
  void ChangeAddress(const string& addr);
};

// Student class.

class Student : public Person {
 public:
  Student(const string& name);

  void RegisterClass(int class_id);
  int GetNumClasses();
  int ComputeTuition();
};
```

```
// Employee class

class Employee : public Person {
 public:
  Employee(const string& name,int salary);

  int salary();
  int ComputeIncomeTax();
  void SetSalary(int salary);
};

// Faculty class

class Faculty : public Employee {
 public:
  Faculty(const string& name, int salary);

  void TeachClass(int class_id);
};
```

Person Example - implementation

person.h

```
#ifndef _PERSON_H_
#define _PERSON_H_

#include <string>

class Person {
 public:
  Person(const std::string& name)
      : name_(name) {}

  const std::string& name() {
    return name_;
  }
  const std::string& address() {
    return address_;
  }

  void ChangeAddress(const std::string& addr) {
    address_ = addr;
  }

 private:
  std::string name_, address_;
};

#endif
```

student.h

```
#ifndef _STUDENT_H_
#define _STUDENT_H_

#include <set>
#include "person.h"

class Student : public Person {
 public:
  Student(const std::string& name)
      : Person(name) {}

  void RegisterClass(int class_id) {
    registered_classes_.insert(class_id);
  }

  int GetNumClasses() {
    return registered_classes_.size();
  }

  int ComputeTuition() {
    return registered_classes_.size() * 100
        + 500;
  }

 private:
  std::set<int> registered_classes_;
};

#endif
```

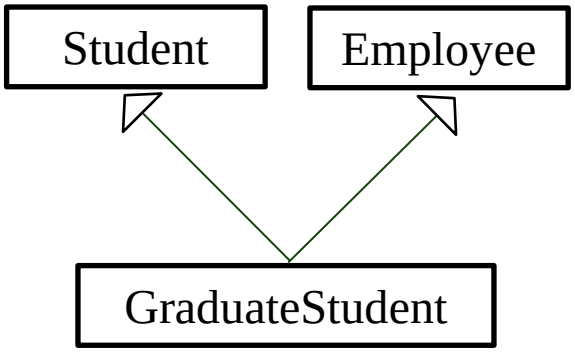
Person Example - implementation

main.cc

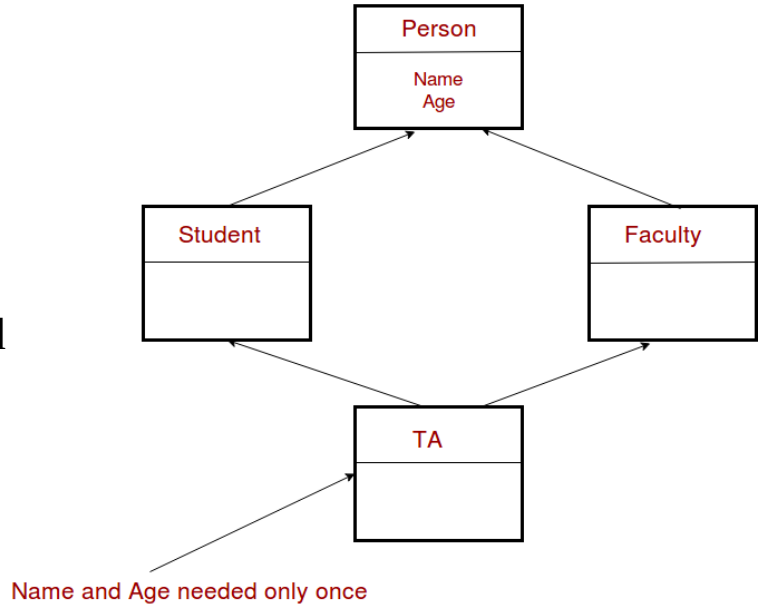
```
#include "employee.h"
#include "faculty.h"
#include "student.h"
using namespace std;

int main() {
  Student john("John"), david("David");
  Employee susan("Susan", 200);
  Faculty daniel("Daniel", 100);

  john.ChangeAddress("New York");
  david.RegisterClass(101);
  daniel.TeachClass(101);
  daniel.TeachClass(102);

  return 0;
}
```

Multiple Inheritance

- Inheriting from two or more base classes.
  - The derived class has all the members of base classes



- Issues
  - Ambiguity
    - What happens if base classes has same-named members?
  - The diamond problem
    - What happens if parent classes are derived from the same grandparent class?



Multiple Inheritance: Example

```
class Person {
 public:
 // ...
};
class Student : public Person {
 public:
 // ...
};
class Employee : public Person {
 public:
 // ...
};

// Multiple inheritance example.
class GraduateStudent
    : public Student, public Employee
{
 public:
  GraduateStudent(const string& name,
                  int salary)
      : Student(name),
        Employee(name + "*", salary) {
}
};
```

```
int main() {
  GraduateStudent mark("Mark", 50);

  cout << mark.GetNumClasses() << endl;
  cout << mark.salary() << endl;
  return 0;
}
```

Multiple Inheritance: Example

```
class Person {
 public:
 // ...
};
class Student : public Person {
 public:
 // ...
  void DoSomething();
};
class Employee : public Person {
 public:
 // ...
  void DoSomething();
};
// Multiple inheritance example.
class GraduateStudent
    : public Student, public Employee {
public:
  GraduateStudent(const string& name,
                  int salary)
      : Student(name),
        Employee(name + "*", salary) {}
};
```

```
int main() {
  GraduateStudent mark("Mark", 50);

  // Eror - ambiguous function DoSomething
  mark.DoSomething();

  return 0;
}
```

Multiple Inheritance

- Actually, you can avoid these problem by using `virtual` inheritance in C++.

- General advice: Avoid using multiple inheritance as much as possible.
  - It is commonly believed that multiple inheritance tends to mass things up.
  - That's why Java forbids multiple inheritance.

- Note that multiple inheritance from *interfaces* (pure abstract classes in C++) can be very helpful.
  - Java only allows multiple inheritance from *interfaces* (“implements” multiple interfaces in Java)

Const: review

- Const variables

  const int MAX = 100;

- Const parameters

  int sum(const int x, const int y) { . . . }

- Pointer to const and const pointer

  const int *pNum = &num

  int* const p = &a;

Const & Class

- Const member variables
  - **should be initialized in *member initializer list* of a constructor**
- Const member functions
  - can read the value of member variables
  - **cannot change the value of member variables**
- Const object
  - cannot change the value of member variables on a const object
  - **cannot call non-const member functions on a const object**

Const: member variables

```
#include <iostream>
using namespace std;

class Circle {
private:
  double Radius;
  const double PI;
public:
  Circle(double r=0, double p){Radius = r; PI=p;}
  void SetRadius(double r) { Radius = r;}
  double GetArea() const { return PI*Radius*Radius;}
};

int main(){
  Circle cir(2,4);
  cout << cir.GetArea() << endl;
  return 0;
}
```

???

Const: member variables

```
#include <iostream>
using namespace std;

class Circle {
private:
  double Radius;
  const double PI;
public:
  //Circle(double r=0, double p){Radius = r; PI=p;}
  Circle(double r, double p): Radius(r), PI(p){}
  void SetRadius(double r) { Radius = r;}
  double GetArea() const { return PI*Radius*Radius;}
};

int main(){
  Circle cir(2,4);
  cout << cir.GetArea() << endl;
  return 0;
}
```

- Const member variables
  - **should be initialized in *member initializer list* of a constructor**

Const: member function

```
#include <iostream>
using namespace std;

class Circle {
private:
  double Radius;
  const double PI;
public:
  //Circle(double r=0, double p){Radius = r; PI=p;}
  Circle(double r, double p): Radius(r), PI(p){}
  void SetRadius(double r) const { Radius = r;}
  double GetArea() const { return PI*Radius*Radius;}
};

int main(){
  Circle cir(2,4);
  cir.SetRadius(5.0);
  cout << cir.GetArea() << endl;
  return 0;
}
```

???

Const: member function

```
#include <iostream>
using namespace std;

class Circle {
private:
  double Radius;
  const double PI;
public:
  //Circle(double r=0, double p){Radius = r; PI=p;}
  Circle(double r, double p): Radius(r), PI(p){}
  void SetRadius(double r)        { Radius = r;}
  double GetArea() const { return PI*Radius*Radius;}
};

int main(){
  Circle cir(2,4);
  cir.SetRadius(5.0);
  cout << cir.GetArea() << endl;
  return 0;
}
```

- Const member functions
  - can read member variables, **cannot update member variables**

Const: object

```
#include <iostream>
using namespace std;

class Circle {
    private:
        double Radius;
        const double PI;
    public:
        Circle(double r = 0): Radius(r), PI(3.14){ }
        void SetRadius(double r) {Radius = r;}
        double GetArea() const { return (PI*Radius*Radius);}
};

int main()
{
  Circle cir(2);
  cout << cir.GetArea() << endl;

  const Circle cir2(3);
  cout << cir2.GetArea() << endl;
  //cir2.SetRadius(5);    //compile error

  return 0;
}
```

- Const object
  - cannot update member variables
  - **cannot call non-const member functions**

Quiz #3

- Go to https://www.slido.com/
- Join #**csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Class Inheritance Types

- Types of inheritance: `public`, `protected`, and `private`.
  - Depending on the inheritance types, the parent's member has diffe rent access control IN the child class.
  - Most commonly used is **public inheritance** (and probably it's the only useful inheritance).

Type of inheritance	Parent's public member	Parent's protected member	Parent's private member
public	public	protected	x (not accessible)
protected	protected	protected	x (not accessible)
private	private	private	x (not accessible)

Example of Private Inheritance

```
class A {
 public:
  void APublic() {}
 protected:
  void AProtected() {}
 private:
  void APrivate() {}
};

// Private inheritance.
class CA : private A {
 public:
  void CAPublic() {
    APublic();      // OK.
    AProtected();   // OK.
    APrivate();     // Error.
  }
  void CAPublic2() {}
 protected:
  void CAProtected() {
  }
 private:
  void CAPrivate() {
  }
};
```

```
class Client : public CA {
  void Function() {
    APublic();       // Error.
    AProtected();    // Error.
    APrivate();      // Error.

    CAPublic();      // Error.
    CAPublic2();     // OK.
    CAProtected();   // OK.
    CAPrivate();     // Error.
  }
};
```

```
// Main routine.

int main() {
  CA ca;
  ca.APublic();    // Error.
  ca.CAPublic();   // Error
  ca.CAPublic2(); // OK.
  ...
}
```

Next Time

- Labs for this lecture:
  - Lab1 (next Tue): Assignment 8-1
  - Lab2 (next Thur): Assignment 8-2

- Next lecture:
  - 9 - Polymorphism 1