Inheritance

Modern cpp Programming lecture 4

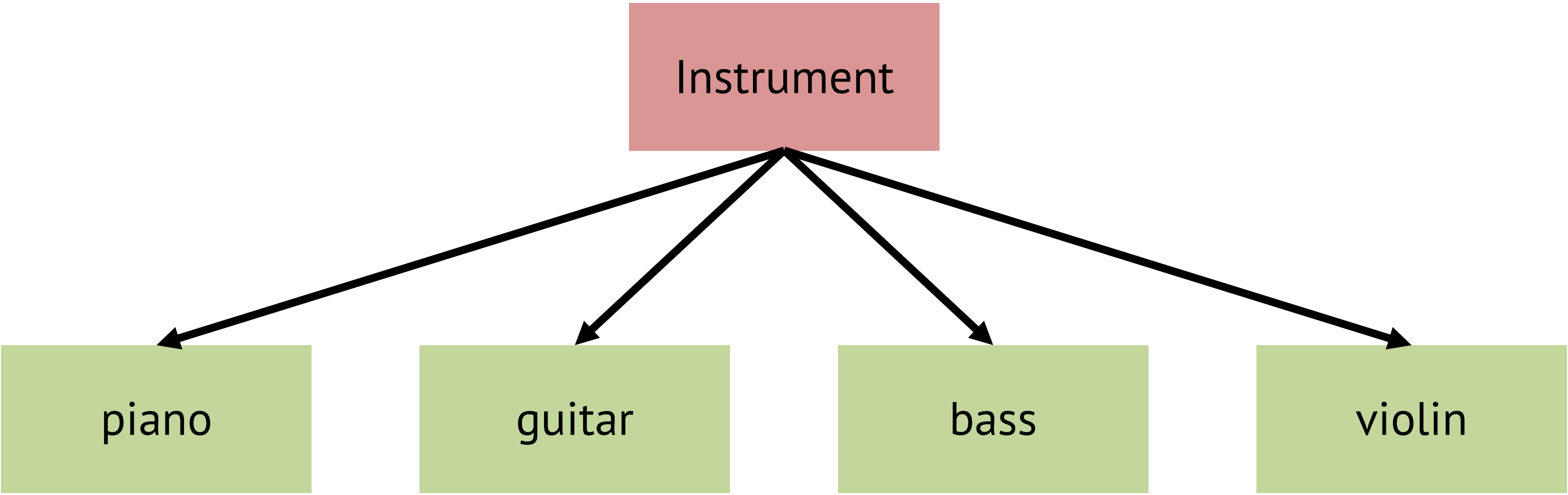
Inheritance

- What's inheritance?
  - The mechanism of basing an class upon another class, while retaining similar implementation
  - Deriving new classes from existing ones and then forming them i nto a hierarchy of classes
    - new class : *subclass, child class*
    - existing class : *superclass, base class, parent class*
  - Invented in 1969 for *Simula* and is now used throughout many OOP languages such as *Java*

Inheritance

- In OOP… inheritance
  - defines the relationship b/w classes
  - transfers the resources (methods, variables) of a class to another
- Why inheritance??
  - Intuitive class relationship (hierarchy)
  - Code reusing
  - easy & productive test
  - adds flexibility & extensibility to the class hierarchy

Inheritance

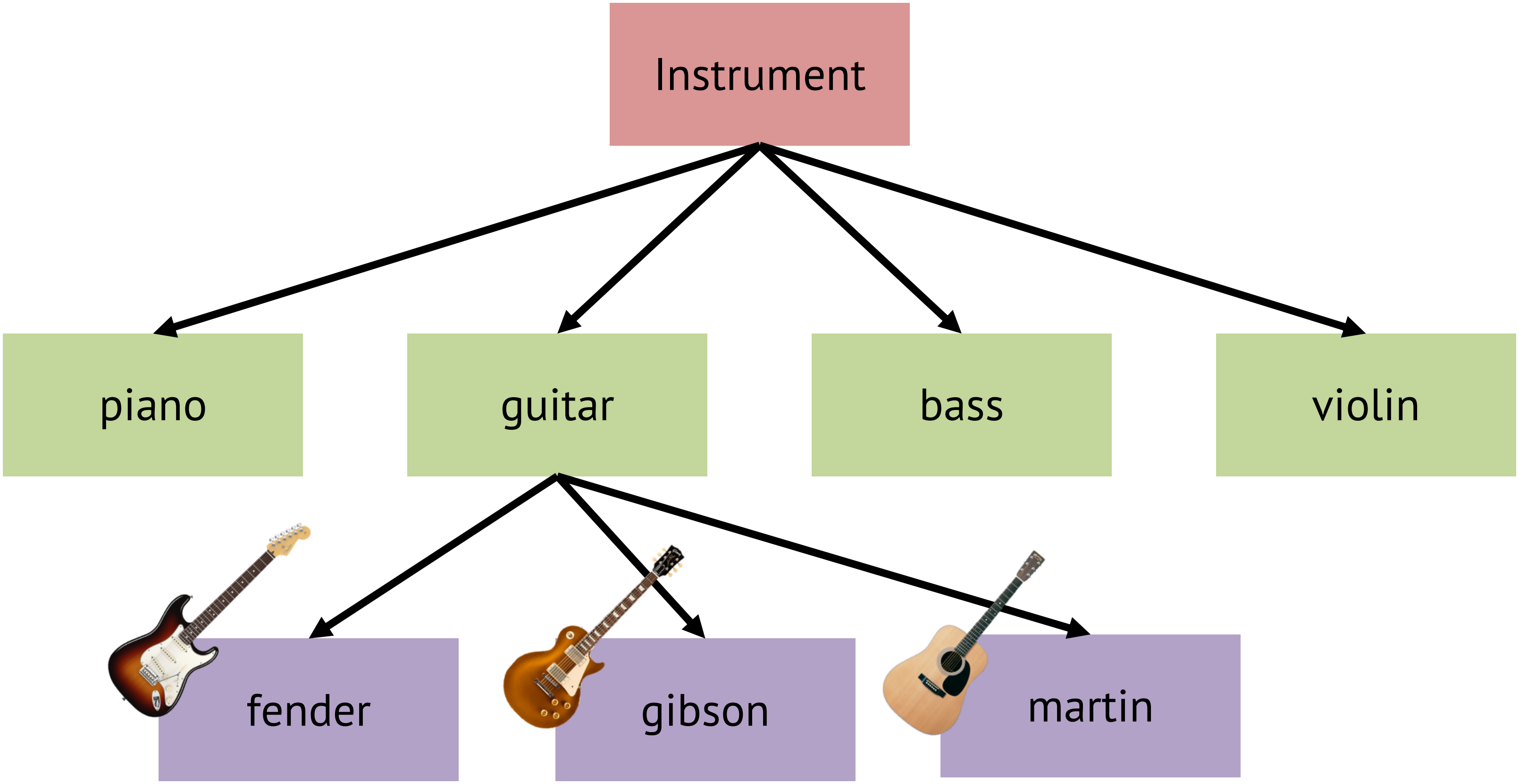


Piano, guitar, bass, and *violin* have some common features…

- generate sound
- Follow equal temperament
- and else…

Instrument **class is defined to store these commonalities**
Piano, guitar, bass, violin classes **only need to store the differences**
=> Code reusing!!

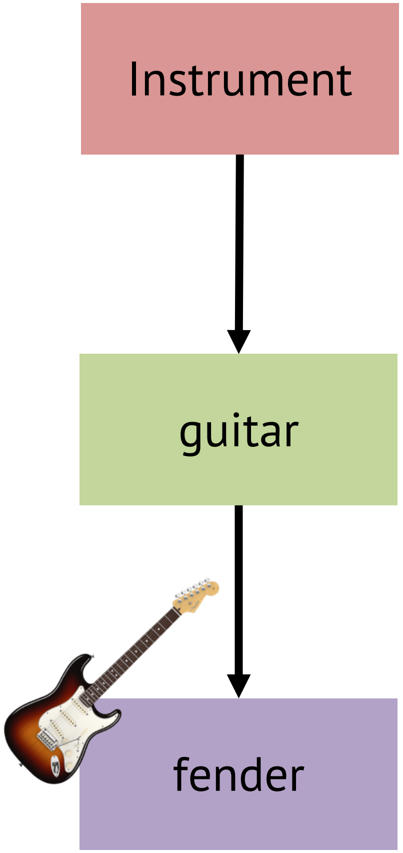
Inheritance



Able to provide any hierarchy structure!!

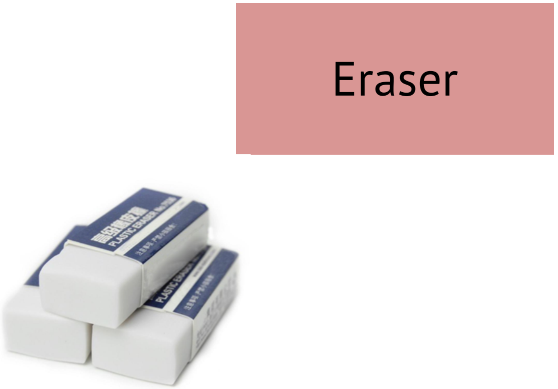
Inheritance

- is-a relationship
  - *child class(subclass)* is a *parent class(superclass)*
  - ***guitar* is a *instrument***
  - ***fender* is a *guitar***
- Quite evident!!

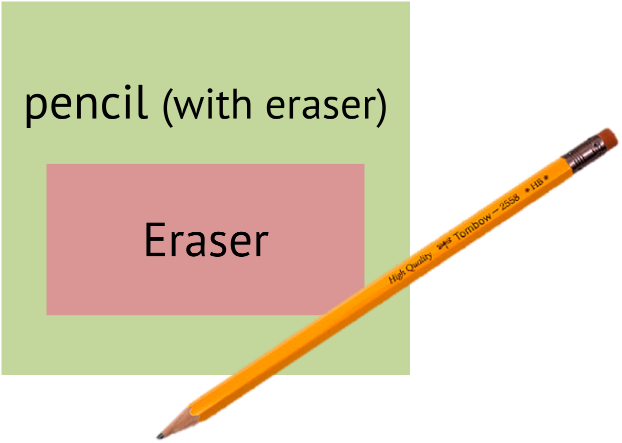


Inheritance

- cf) has-a relationship
  - *child class* has a *parent class*
  - a.k.a. *child class* contains a *parent class* as its variable
  - **no explicit keyword** for has-a relationship
  - just set parent class as the variable of a child class



Eraser



pencil (with eraser)

Eraser

Inheritance

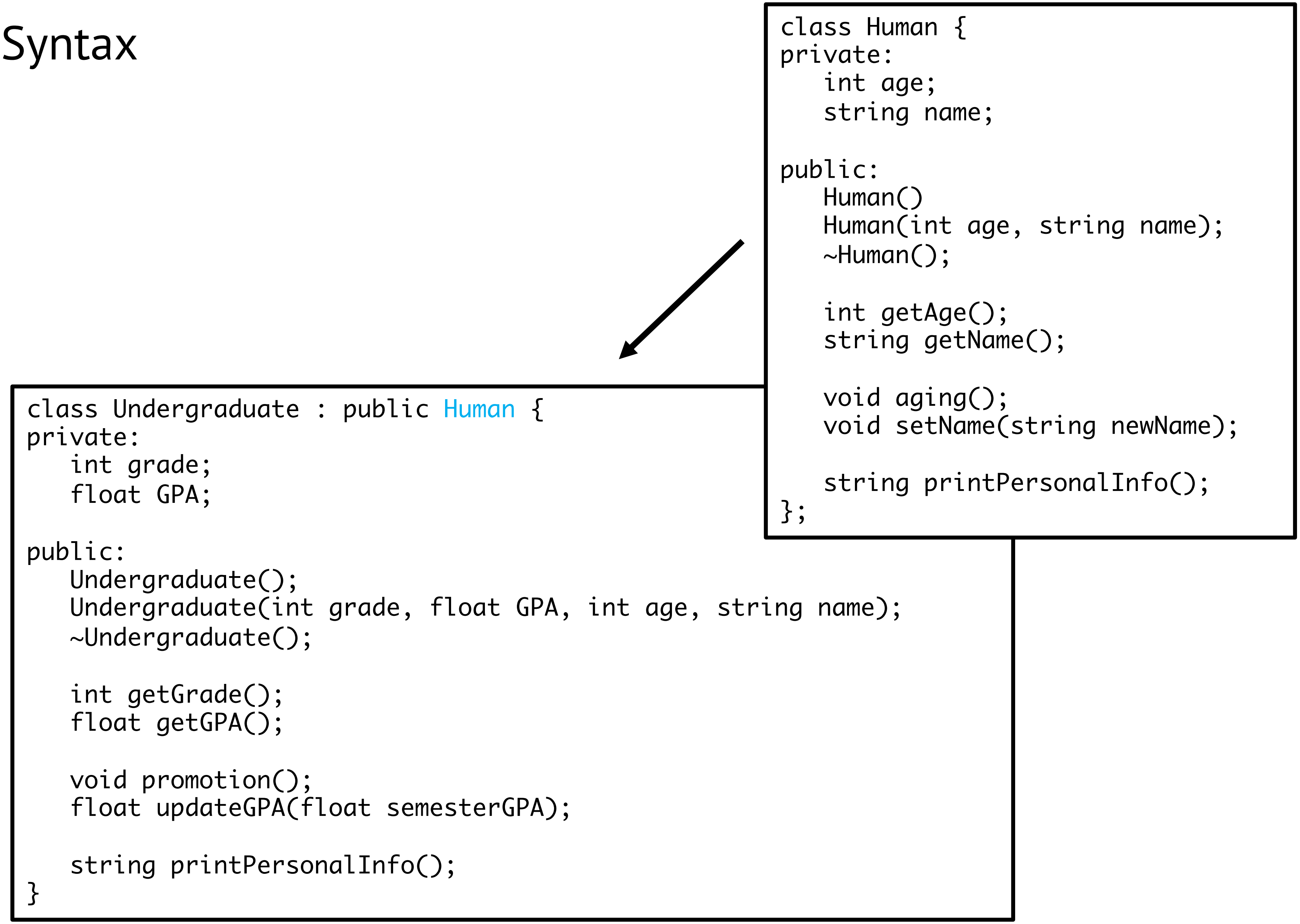
- In most modern programming languages
  - cpp, Java, kotlin, javascript, typescript, python, ruby, c#, swift, go, R, rust, scala, ocaml…
  - contain oop & inheritance as their feature
  - why you must learn inheritance

Inheritance in cpp

- mainly supports is-a relationship (quite obvious!!)
- Child class “inherits” the features from parent class
- you need to understand following keywords…
  - Inheritance
  - Access identifier (public, private, protected)
  - overriding
  - friend
  - Polymorphism (next lecture)
  - virtual (next lecture)

Inheritance in cpp

- Syntax

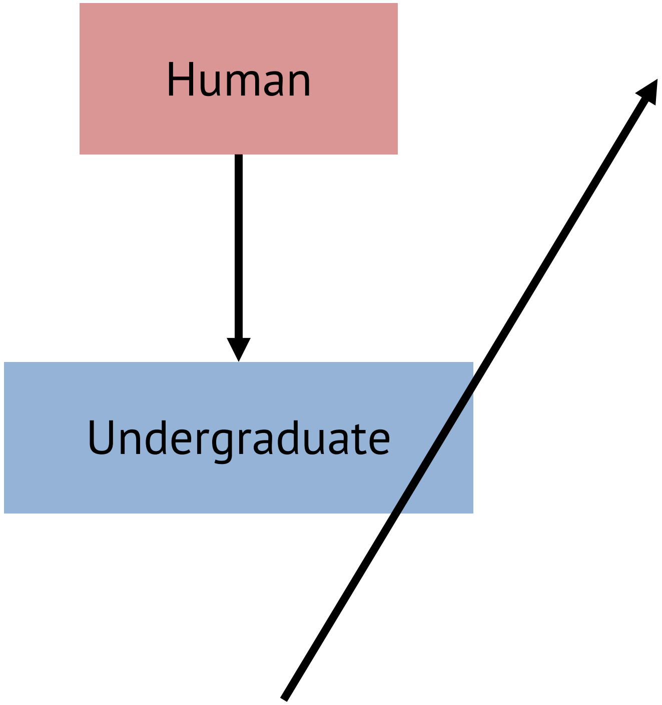


```
class Human {
private:
   int age;
   string name;

public:
   Human()
   Human(int age, string name);
   ~Human();

   int getAge();
   string getName();

   void aging();
   void setName(string newName);

   string printPersonalInfo();
};
```

```
class Undergraduate : public Human {
private:
   int grade;
   float GPA;

public:
   Undergraduate();
   Undergraduate(int grade, float GPA, int age, string name);
   ~Undergraduate();

   int getGrade();
   float getGPA();

   void promotion();
   float updateGPA(float semesterGPA);

   string printPersonalInfo();
}
```

Inheritance in cpp

- Constructor
  - function call flow for the constructors
  - basically, *child constructor* calls *default parent constructor* first



Human

Undergraduate

Default constructor

```
Human() {
    cout << “Human” << endl;
}

Undergraduate() {
    cout << “Undergraduate” << endl;
}
```

```
Output:
Human
Undergraduate
```

Inheritance in cpp

- Constructor
  - What about *non-default constructor??*
  - You must declare it explicitly!!

```
Human :: Human(int age, string name) {
   this->age = age;
   this->name = name;
   cout << "Age: " << this->age << endl;
   cout << "Name: " << this->name << endl;
}

Undergraduate :: Undergraduate(int grade, float GPA, int age, string name) : Human(age, name) {
   this->grade = grade;
   this->GPA = GPA;
   cout << "Grade: " << this->grade << endl;
   cout << "GPA: " << this-> GPA << endl;
}

Undergraduate(3, 4.2, 20, "Bob");
```

explicitly calling non-default constructor

Output

```
Age: 20
Name: Bob
Grade: 3
GPA: 4.2
```

Inheritance in cpp

- Constructor
  - What about *non-default constructor??*
  - You must declare it explicitly!!

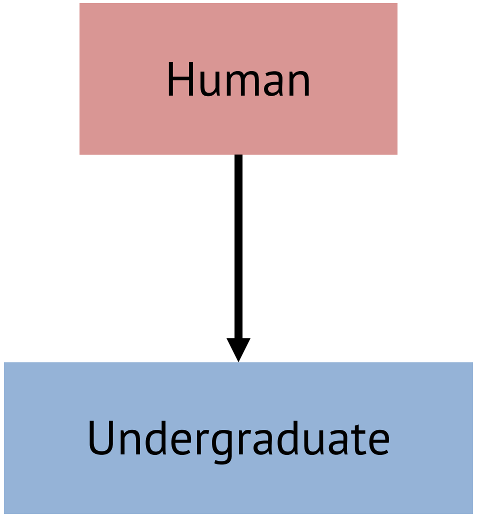
```
Human :: Human() {
   this->age = 10;
   this->name = “Alice”;
   cout << "Age: " << this->age << endl;
   cout << "Name: " << this->name << endl;
}

Undergraduate :: Undergraduate(int grade, float GPA, int age, string name) {
   this->grade = grade;
   this->GPA = GPA;
   cout << "Grade: " << this->grade << endl;
   cout << "GPA: " << this-> GPA << endl;
}

Undergraduate(3, 4.2, 20, "Bob");
```

implicitly calling default constructor

```
Output
Age: 10
Name: Alice
Grade: 3
GPA: 4.2
```

Inheritance in cpp

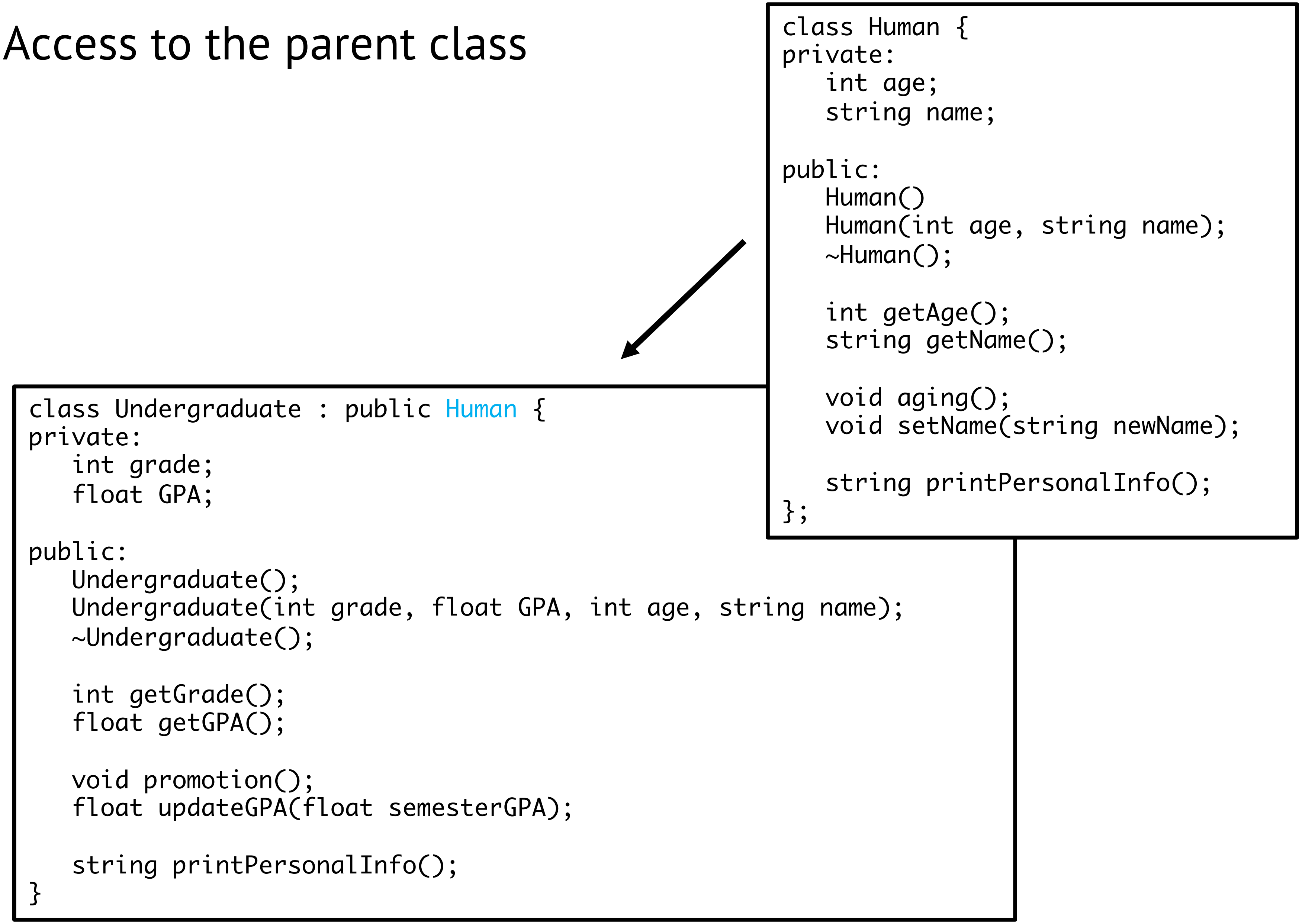
- Destructor
  - function call flow
  - *child class destructor -> parent class destructor*



```
~Human() {
   cout << “Human” << endl;
}

~Undergraduate() {
   cout << “Undergraduate” << endl;
}

Output:
Undergraduate
Human
```

Inheritance in cpp

- Access to the parent class



```cpp
class Human {
private:
   int age;
   string name;

public:
   Human()
   Human(int age, string name);
   ~Human();

   int getAge();
   string getName();

   void aging();
   void setName(string newName);

   string printPersonalInfo();
};
```

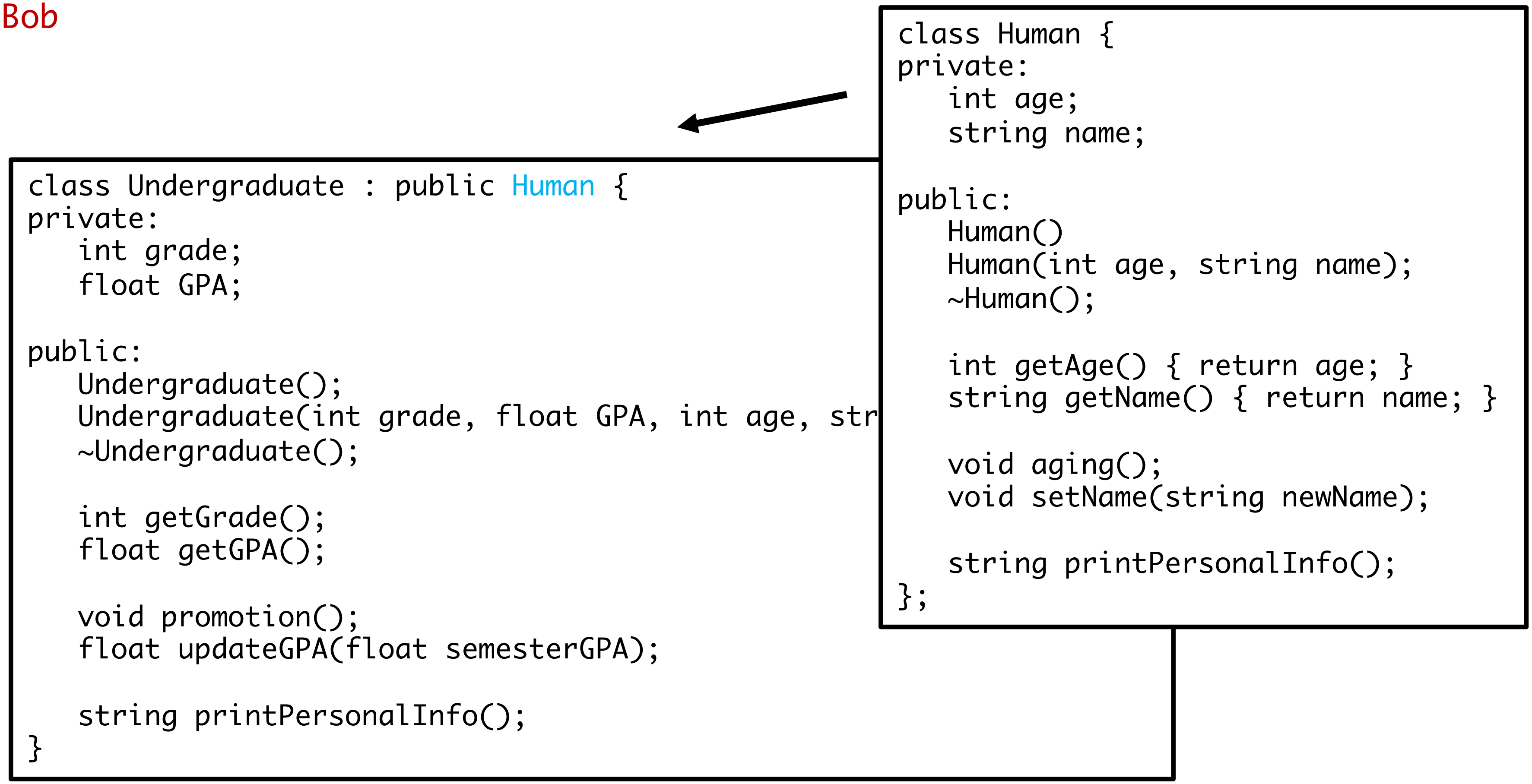
```cpp
class Undergraduate : public Human {
private:
   int grade;
   float GPA;

public:
   Undergraduate();
   Undergraduate(int grade, float GPA, int age, string name);
   ~Undergraduate();

   int getGrade();
   float getGPA();

   void promotion();
   float updateGPA(float semesterGPA);

   string printPersonalInfo();
}
```

Inheritance in cpp

```
Undergraduate* bob = new Undergraduate(3, 4.2, 20, "Bob");
bob->getAge();
bob->getName();
```

```
Output:
20
Bob
```

able to access the public functions in Human class



```
class Human {
private:
   int age;
   string name;

public:
   Human()
   Human(int age, string name);
   ~Human();

   int getAge() { return age; }
   string getName() { return name; }

   void aging();
   void setName(string newName);

   string printPersonalInfo();
};
```

```
class Undergraduate : public Human {
private:
   int grade;
   float GPA;

public:
   Undergraduate();
   Undergraduate(int grade, float GPA, int age, str
   ~Undergraduate();

   int getGrade();
   float getGPA();

   void promotion();
   float updateGPA(float semesterGPA);

   string printPersonalInfo();
}
```

Inheritance in cpp

```
Undergraduate* bob = new Undergraduate(3, 4.2, 20, "Bob");
bob->getName();
```

```
Compiler Output:
error: 'getName' is a private member of 'Human'
   cout << bob->getName() << endl;
                ^
note: declared private here
   string getName() { return name; }
          ^
1 error generated.
```

cannot access private function
=> Compile error!!

```
class Human {
private:
   int age;
   string name;
   string getName() { return name; }

public:
   Human()
   Human(int age, string name);
   ~Human();

   int getAge() { return age; }

   void aging();
   void setName(string newName);

   string printPersonalInfo();
};
```

```
class Undergraduate : public Human {
private:
   int grade;
   float GPA;

public:
   Undergraduate();
   Undergraduate(int grade, float GPA, int age, str
   ~Undergraduate();

   int getGrade();
   float getGPA();

   void promotion();
   float updateGPA(float semesterGPA);

   string printPersonalInfo();
}
```

Inheritance in cpp

- Access identifier in inheritance
- Recall access identifier!!
  - public
    - accessible from itself / outside the class / **child classes**
  - private
    - accessible from itself
  - protected
    - accessible from itself / **child classes**
- we now know what *child class* is!!

Access identifier

- Simple example

```
class Google {
private:
   void projectZero() {
      cout << "Resolves zero-day vulnerabilities!!" << endl;
   }

protected:
   void alphaGo() {
      cout << "AI Go player" << endl;
   }

public:
   void android() {
      cout << "Operating system for mobile devices" << endl;
   }
};

class GoogleKorea : public Google {
public:
   void projects() {
      android();
      alphaGo();
   }
};

GoogleKorea* gk = new GoogleKorea();
gk->projects();
```

```
Output:
Operating system for mobile devices
AI Go player
```

Access identifier

- Simple example

```
class Google {
private:
   void projectZero() {
      cout << "Resolves zero-day vulnerabilities!!" << endl;
   }

protected:
   void alphaGo() {
      cout << "AI Go player" << endl;
   }

public:
   void android() {
      cout << "Operating system for mobile devices" << endl;
   }
};

class GoogleKorea : public Google {
public:
   void projects() {
      android();
      alphaGo();
      projectZero();    // generates compile error!! (tries to access private element)
   }
};

GoogleKorea* gk = new GoogleKorea();
gk->projects();
```

Access identifier

- Simple example

```
class Google {
private:
   void projectZero() {
      cout << "Resolves zero-day vulnerabilities!!" << endl;
   }

protected:
   void alphaGo() {
      cout << "AI Go player" << endl;
   }

public:
   void android() {
      cout << "Operating system for mobile devices" << endl;
   }
};

class GoogleKorea : public Google {
public:
   void projects() {
      android();
      alphaGo();
   }
};

GoogleKorea* gk = new GoogleKorea();
gk->android();       // OK
gk->alphaGo();       // error!! protected element!!
gk->projectZero();   // error!! private element!!
```

Access identifier

- Ok…then what does it mean??

```
class GoogleKorea : public Google {
public:
   void projects() {
      android();
      alphaGo();
   }
};
```

- quite easy!! simple rule::
  - Access identifier of the child class makes the parent class’ access identifiers which are weaker same as itself.

Access identifier

the power of access identifiers: public < protected < private

```
class Google {
private:
   void projectZero();
protected:
   void alphaGo();
public:
   void android();
};

class GoogleKorea : public Google
{ };
```

projectZero is now private
alphaGo is now protected
android is now public

```
class Google {
private:
   void projectZero();
protected:
   void alphaGo();
public:
   void android();
};

class GoogleKorea : protected Google
{ };
```

projectZero is now private
alphaGo is now protected
android is now protected

```
class Google {
private:
   void projectZero();
protected:
   void alphaGo();
public:
   void android();
};

class GoogleKorea : private Google
{ };
```

projectZero is now private
alphaGo is now private
android is now private

Access identifier

- If you work alone..
  - private and public is enough!!
- but while collaborating…
  - you should carefully design the inheritance structure
  - and should also carefully select access identifier!!

Overriding

- Definition (from Wikipedia)
  - allows a child class to provide a specific implementation of a met hod that is already provided by one of its parent classes
- Easy version
  - *overwrites* method from parent class
    - which has same method signature
      - same name
      - same arguments
  - re-defines the method from parent class

Overriding

- Simple example

```
class Google {
public:
    void android() {
        cout << "Global ver: Operating system for mobile devices" << endl;
    }
};

class GoogleKorea : public Google {
public:
    void android() {     // overriding
        cout << "Korean ver: 모바일 디바이스를 위한 운영체제" << endl;
    }
};

Google* g = new Google();
GoogleKorea* gk = new GoogleKorea();
g->android();
gk->android();
```

```
Output:
Operating system for mobile devices
Korean ver: 모바일 디바이스를 위한 운영체제
```

Overriding

- Why overriding
  - able to define a specific behavior for the subclass
  - subclass can implement a superclass method based on its requirements
  - provides multiple implementation of same method
    => simplifies code!!

Overriding

- Overriding vs. Overloading

Method Overloading	Method Overriding
Provides functionality to reuse method name for different arguments	Provides functionality to override a behavior which the class have inherited from parent class
Occurs usually within a single class (may also occur in child/parent classes)	Occurs in two classes that have child-parent or is-a relationship
Must have different argument list (signature)	Must have the same argument list
May have different return types	Must have the same or covariant return type
May have different access modifiers	Must not have a more restrictive access modifier but may have less restrictive access modifier

Why? DIY...

reference: https://laptrinhx.com/everything-about-method-overloading-vs-method-overriding-2837559910/

Friend

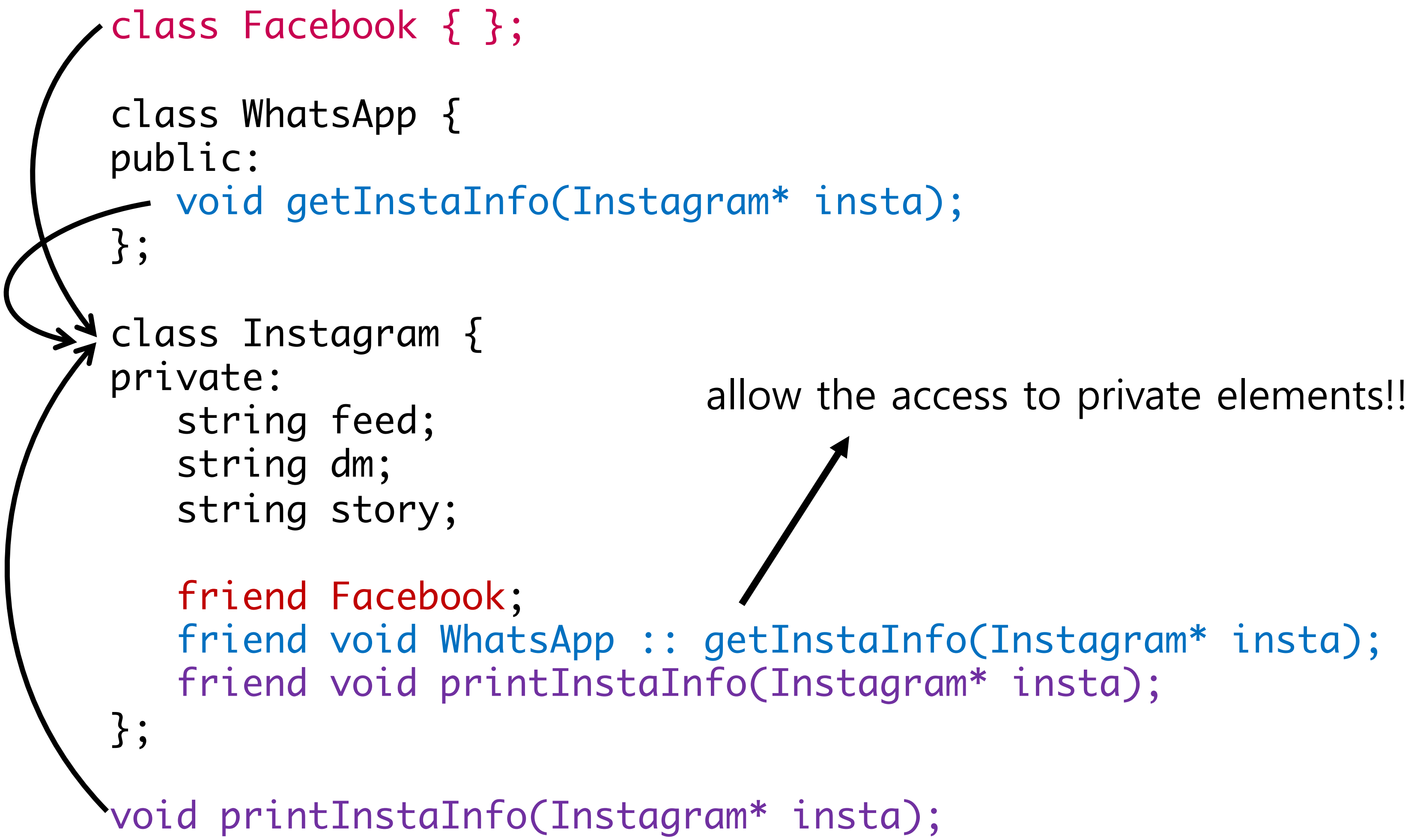
- Not exactly related to inheritance…
- But a quite important feature!!
  - so let's just talk about it here
- Recall…
  - private / protected elements provides limited access permission
  - important feature for *information hiding*!!
  - *friend* keyword offers an *exceptional* way to access those elements

Friend

- Simple rule
  1. *Friends can access private / protected elements freely (like public)*
  2. *Friends can have 3 types: **class**, **member function**, **global function***
- Any class can declare other ***class, member function, global function*** as its friend.

Friend

- Friend exmplae



```
class Facebook { };

class WhatsApp {
public:
   void getInstaInfo(Instagram* insta);
};

class Instagram {
private:
   string feed;
   string dm;
   string story;

   friend Facebook;
   friend void WhatsApp :: getInstaInfo(Instagram* insta);
   friend void printInstaInfo(Instagram* insta);
};

void printInstaInfo(Instagram* insta);
```

allow the access to private elements!!

Thank you!!

contact: jeonhyun97@postech.ac.kr