Inheritance

C++ Object Oriented Programming

Pei-yih Ting

NTOUCS

Contents

- Basic Inheritance
  - Why inheritance
  - How inheritance works
  - Protected members
  - Constructors and destructors
  - Derivation tree
  - Function overriding and hiding
  - Example class hierarchy

Contents

- Basic Inheritance
  - Why inheritance
  - How inheritance works
  - Protected members
  - Constructors and destructors
  - Derivation tree
  - Function overriding and hiding
  - Example class hierarchy
- Inheritance Design
  - Exploring different inheritance structure
  - Direct solution to reuse code
  - Alternative solutions
  - Better design
  - Final solutions
  - Design rules (IS-A relationship, Proper inheritance)
  - Dubious designs

Object-Oriented Analysis

- An object-orientated design provides a more **natural** and **systematic** framework for specifying and designing a programming solution.

Object-Oriented Analysis

- An object-orientated design provides a more **natural** and **systematic** framework for specifying and designing a programming solution.
- Program designs are almost always based on the **program specification**, i.e. a document describing the exact requirements a program is expected to achieve.

Object-Oriented Analysis

- An object-orientated design provides a more **natural** and **systematic** framework for specifying and designing a programming solution.
- Program designs are almost always based on the **program specification**, i.e. a document describing the exact requirements a program is expected to achieve.
- Four phases of the object-oriented analysis process:

Object-Oriented Analysis

- An object-orientated design provides a more **natural** and **systematic** framework for specifying and designing a programming solution.
- Program designs are almost always based on the **program specification**, i.e. a document describing the exact requirements a program is expected to achieve.

- Four phases of the object-oriented analysis process:
  - The identification of objects from the program specification.

Object-Oriented Analysis

- An object-orientated design provides a more **natural** and **systematic** framework for specifying and designing a programming solution.
- Program designs are almost always based on the **program specification**, i.e. a document describing the exact requirements a program is expected to achieve.

- Four phases of the object-oriented analysis process:
  - The identification of objects from the program specification.
  - The identification of the attributes and behaviours of these objects.

Object-Oriented Analysis

- An object-orientated design provides a more **natural** and **systematic** framework for specifying and designing a programming solution.
- Program designs are almost always based on the **program specification**, i.e. a document describing the exact requirements a program is expected to achieve.

- Four phases of the object-oriented analysis process:
  - The identification of objects from the program specification.
  - The identification of the attributes and behaviours of these objects.
  - The identification of any super-classes.

Object-Oriented Analysis

- An object-orientated design provides a more **natural** and **systematic** framework for specifying and designing a programming solution.
- Program designs are almost always based on the **program specification**, i.e. a document describing the exact requirements a program is expected to achieve.

- Four phases of the object-oriented analysis process:
  - The identification of objects from the program specification.
  - The identification of the attributes and behaviours of these objects.
  - The identification of any super-classes.
  - The specification of the behaviours of the identified classes.

Inheritance

- The distinction between an "object-**based** language" and an "object-**oriented** language" is the ability to support inheritance (or derivation).

Inheritance

- The distinction between an "object-**based** language" and an "object-**oriented** language" is the ability to support inheritance (or derivation).
- Composition/aggregation and inheritance are the most important two ways to construct object **hierarchies**.

Inheritance

- The distinction between an "object-**based** language" and an "object-**oriented** language" is the ability to support inheritance (or derivation).
- Composition/aggregation and inheritance are the most important two ways to construct object **hierarchies**.
- In the **OOA** process, after objects are identified from the problem domain and attributes and behaviors are modeled with classes in the analysis process, the next important phase is the identification of super-classes in the problem domain

Inheritance

- The distinction between an "object-**based** language" and an "object-**oriented** language" is the ability to support inheritance (or derivation).
- Composition/aggregation and inheritance are the most important two ways to construct object **hierarchies**.
- In the **OOA** process, after objects are identified from the problem domain and attributes and behaviors are modeled with classes in the analysis process, the next important phase is the identification of super-classes in the problem domain
- In the language level, a super-class defines the attributes and behaviors that are common to all its sub-classes.

Inheritance

- The distinction between an "object-**based** language" and an "object-**oriented** language" is the ability to support inheritance (or derivation).
- Composition/aggregation and inheritance are the most important two ways to construct object **hierarchies**.
- In the **OOA** process, after objects are identified from the problem domain and attributes and behaviors are modeled with classes in the analysis process, the next important phase is the identification of super-classes in the problem domain
- In the language level, a super-class defines the attributes and behaviors that are common to all its sub-classes.

Base class vs. **Derived class**

Inheritance

- The distinction between an "object-**based** language" and an "object-**oriented** language" is the ability to support inheritance (or derivation).
- Composition/aggregation and inheritance are the most important two ways to construct object **hierarchies**.
- In the **OOA** process, after objects are identified from the problem domain and attributes and behaviors are modeled with classes in the analysis process, the next important phase is the identification of super-classes in the problem domain
- In the language level, a super-class defines the attributes and behaviors that are common to all its sub-classes.

Base class		**Derived class**
Super-class	vs.	**Sub-class**

Inheritance

- The distinction between an "object-**based** language" and an "object-**oriented** language" is the ability to support inheritance (or derivation).
- Composition/aggregation and inheritance are the most important two ways to construct object **hierarchies**.
- In the **OOA** process, after objects are identified from the problem domain and attributes and behaviors are modeled with classes in the analysis process, the next important phase is the identification of super-classes in the problem domain
- In the language level, a super-class defines the attributes and behaviors that are common to all its sub-classes.

Base class		**Derived class**
Super-class	vs.	**Sub-class**
Parent class		**Child class**

Basic Inheritance

The Basic Problem: Extension

- Imagine you have a class for describing students

The Basic Problem: Extension

- Imagine you have a class for describing students

```
class Student {
public:
   Student();
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

The Basic Problem: Extension

- Imagine you have a class for describing students

```
class Student {
public:
    Student();
    ~Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

- Want to **add fields** to handle the requirements for graduate students

The Basic Problem: Extension

- Imagine you have a class for describing students

```
class Student {
public:
    Student();
    ~Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

```
class Student {
public:
    Student();
    ~Student();
    void setData(char *name, int age,
                 int stipend);
    int getAge() const;
    const char *getName() const;
    int getStipend() const;
private:
    char *m_name;
    int m_age;
    int m_stipend;
};
```

- Want to **add fields** to handle the requirements for graduate students

The Basic Problem: Extension

- Imagine you have a class for describing students

```
class Student {
public:
    Student();
    ~Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

```
class Student {
public:
    Student();
    ~Student();
    void setData(char *name, int age,
                 int stipend);
    int getAge() const;
    const char *getName() const;
    int getStipend() const;
private:
    char *m_name;
    int m_age;
    int m_stipend;
};
```

- Want to **add fields** to handle the requirements for graduate students

What is the problem of this design?

Not Good!

Not Good!

- In the above design
  - **Student** becomes a general purpose class, a set of attributes and interfaces are used for undergraduate students, while another set of attributes and interfaces are used for graduate students … a form with many redundant fields

Not Good!

- In the above design
  - **Student** becomes a general purpose class, a set of attributes and interfaces are used for undergraduate students, while another set of attributes and interfaces are used for graduate students … a form with many redundant fields
  - In the process of this change, all previously developed programs, including those implementations of the Student class and those codes that are the client programs of the Student class, have to be recompiled…. This change is global, not limited to the part you plan to add.

Not Good!

- In the above design
  - **Student** becomes a general purpose class, a set of attributes and interfaces are used for undergraduate students, while another set of attributes and interfaces are used for graduate students … a form with many redundant fields
  - In the process of this change, all previously developed programs, including those implementations of the Student class and those codes that are the client programs of the Student class, have to be recompiled…. This change is global, not limited to the part you plan to add.

OCP: open-closed principle

Not Good!

- In the above design
  - **Student** becomes a general purpose class, a set of attributes and interfaces are used for undergraduate students, while another set of attributes and interfaces are used for graduate students … a form with many redundant fields
  - In the process of this change, all previously developed programs, including those implementations of the Student class and those codes that are the client programs of the Student class, have to be recompiled…. This change is global, not limited to the part you plan to add.

OCP: open-closed principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

A Solution – Separate Classes

- No redundant members, old codes for Student need only change the name to UnderGraduate.

A Solution – Separate Classes

- No redundant members, old codes for Student need only change the name to UnderGraduate.

```
class Undergraduate {
public:
   Undergraduate();
   ~Undergraduate();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

A Solution – Separate Classes

- No redundant members, old codes for Student need only change the name to UnderGraduate.

```
class Undergraduate {
public:
    Undergraduate();
    ~Undergraduate();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

```
class Graduate {
public:
    Graduate();
    ~Graduate();
    void setData(char *name,
                 int age, int stipend);
    int getAge() const;
    const char *getName() const;
    int getStipend() const;
private:
    char *m_name;
    int m_age;
    int m_stipend;
};
```

A Solution – Separate Classes

- No redundant members, old codes for Student need only change the name to UnderGraduate.

```
class Undergraduate {
public:
   Undergraduate();
   ~Undergraduate();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

```
class Graduate {
public:
   Graduate();
   ~Graduate();
   void setData(char *name,
                int age, int stipend);
   int getAge() const;
   const char *getName() const;
   int getStipend() const;
private:
   char *m_name;
   int m_age;
   int m_stipend;
};
```

Why is this still a poor solution?

A Solution – Separate Classes

- No redundant members, old codes for Student need only change the name to UnderGraduate.

```
class Undergraduate {
public:
    Undergraduate();
    ~Undergraduate();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

```
class Graduate {
public:
    Graduate();
    ~Graduate();
    void setData(char *name,
                 int age, int stipend);
    int getAge() const;
    const char *getName() const;
    int getStipend() const;
private:
    char *m_name;
    int m_age;
    int m_stipend;
};
```

Why is this still a poor solution?

- A client program cannot treat both classes of objects in a uniform way, ex. The library circulation system wants to check which students are holding books overdue, it has to handle undergraduate and graduate students with separate pieces of codes.

A Solution – Separate Classes

- No redundant members, old codes for Student need only change the name to UnderGraduate.

```
class Undergraduate {
public:
    Undergraduate();
    ~Undergraduate();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

```
class Graduate {
public:
    Graduate();
    ~Graduate();
    void setData(char *name,
                 int age, int stipend);
    int getAge() const;
    const char *getName() const;
    int getStipend() const;
private:
    char *m_name;
    int m_age;
    int m_stipend;
};
```

Why is this still a poor solution?

- A client program cannot treat both classes of objects in a uniform way, ex. The library circulation system wants to check which students are holding books overdue, it has to handle undergraduate and graduate students with separate pieces of codes.
- Also, a lot of redundancy.

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

new member functions

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

new member functions

new data member

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

new member functions

new data member

Student is called the base class, **Graduate** is called the derived class

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

new member functions

new data member

Student is called the base class, **Graduate** is called the derived class

- All the data members (m_name and m_age) and most the member functions (setData(), getAge(), getName()) of class Student are automatically inherited by the Graduate class

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

new member functions

new data member

Student is called the base class, **Graduate** is called the derived class

- All the data members (m_name and m_age) and most the member functions (setData(), getAge(), getName()) of class Student are automatically inherited by the Graduate class
- New member functions

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
    Graduate(char *name, int age, int stipend);
    int getStipend() const;
private:
    int m_stipend;
};
```

new member functions

new data member

Student is called the base class, **Graduate** is called the derived class

- All the data members (m_name and m_age) and most the member functions (setData(), getAge(), getName()) of class Student are automatically inherited by the Graduate class
- New member functions

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
    setData(name, age); // this is inherited from Student
}
```

Basic Inheritance in C++

- Declare a class Graduate that is derived from Student

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

new member functions

new data member

Student is called the base class, Graduate is called the derived class

- All the data members (m_name and m_age) and most the member functions (setData(), getAge(), getName()) of class Student are automatically inherited by the Graduate class
- New member functions

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}

int Graduate::getStipend() const {
   return m_stipend;
}
```

Basic Inheritance (cont'd)

✧ **Usages:**

Basic Inheritance (cont'd)

- **Usages:**

```
Student student;
student.setData("Mel", 19);
Graduate gradStudent("Ron", 24, 3000);
```

Basic Inheritance (cont'd)

- **Usages:**

```
Student student;
student.setData("Mel", 19);
Graduate gradStudent("Ron", 24, 3000);
```

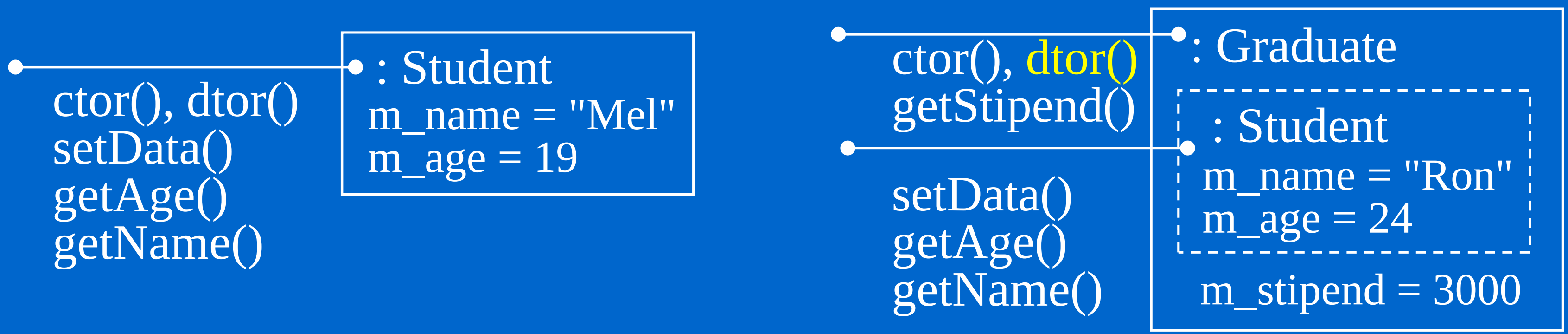
ctor(), dtor()
setData()
getAge()
getName()

: Student
m_name = "Mel"
m_age = 19

Basic Inheritance (cont'd)

✧ **Usages:**

```
Student student;
student.setData("Mel", 19);
Graduate gradStudent("Ron", 24, 3000);
```



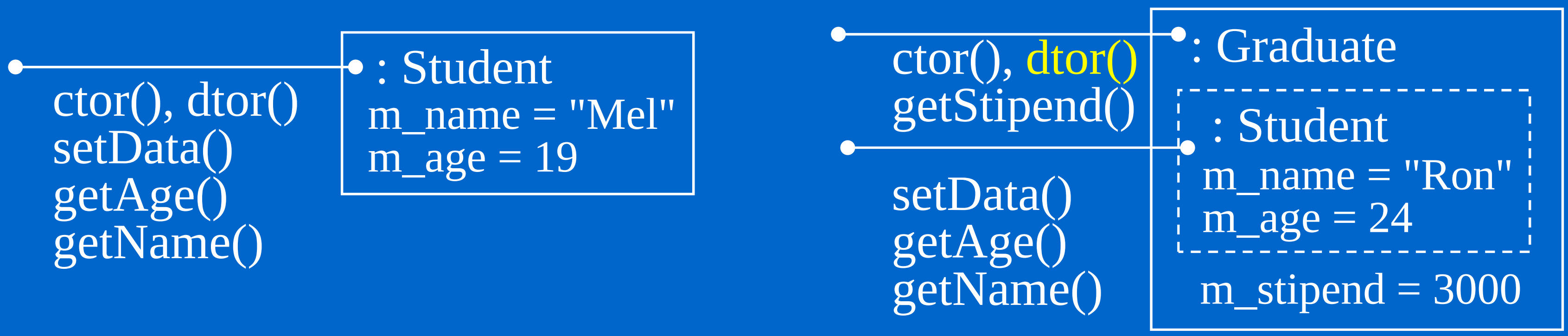
Basic Inheritance (cont'd)

- **Usages:**

Student student;

student.setData("Mel", 19);

Graduate gradStudent("Ron", 24, 3000);

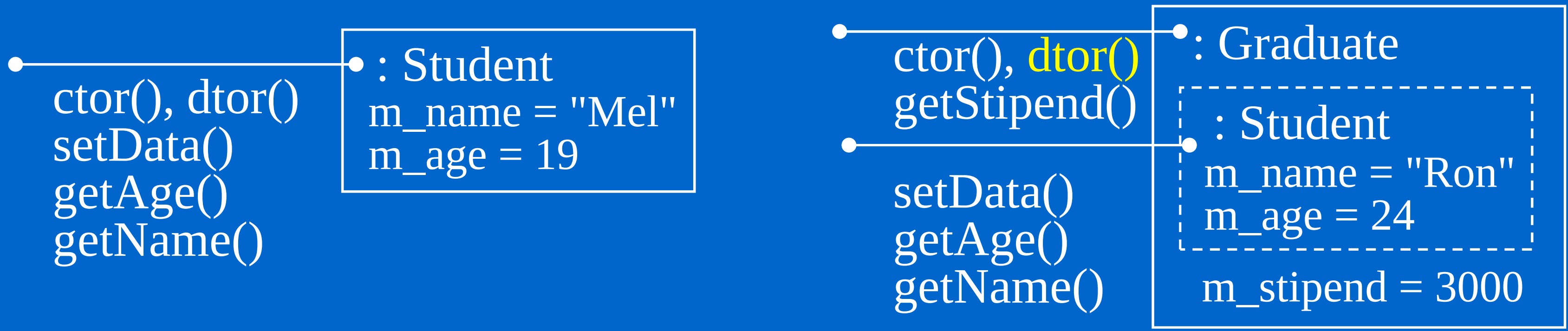



```
cout << student.getName() << " is " << student.getAge()
     << " years old undergraduate student\n";
```

Basic Inheritance (cont’d)

- **Usages:**

```
Student student;
student.setData("Mel", 19);
Graduate gradStudent("Ron", 24, 3000);
```




```
cout << student.getName() << " is " << student.getAge()
     << " years old undergraduate student\n";
cout << gradStudent.getName() << " is  " << gradStudent.getAge()
     << " years old and has a stipend of " << gradStudent.getStipend()
     << " dollars.\n";
```

Basic Inheritance (cont'd)

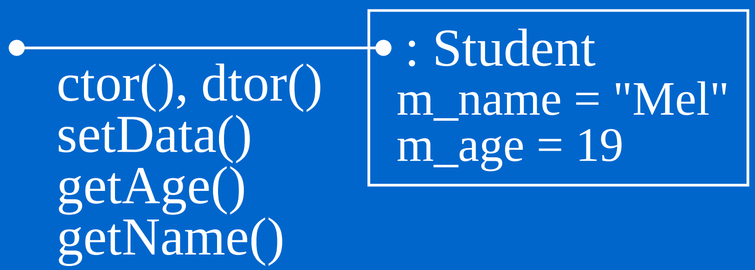
Note: A Graduate object **is a** Student object because a Graduate object provides the complete set of **interface** functions of a Student object, i.e., they looks the same from the outside.

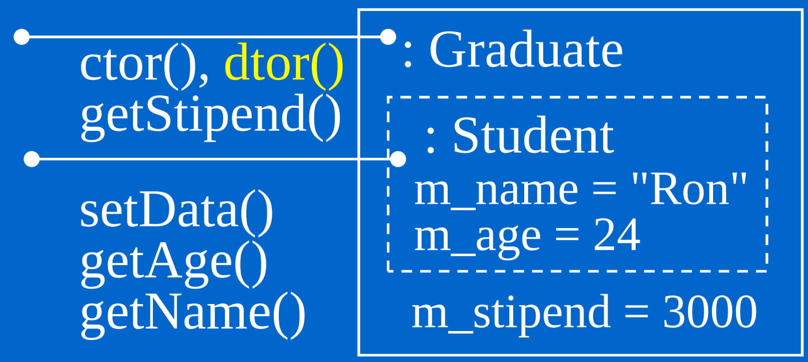
- **Usages:**

Student student;

student.setData("Mel", 19);

Graduate gradStudent("Ron", 24, 3000);






```
cout << student.getName() << " is " << student.getAge()
     << " years old undergraduate student\n";
cout << gradStudent.getName() << " is  " << gradStudent.getAge()
     << " years old and has a stipend of " << gradStudent.getStipend()
     << " dollars.\n";
```

Basic Inheritance (cont'd)

Note: A Graduate object **is a** Student object because a Graduate object provides the complete set of **interface** functions of a Student object, i.e., they looks the same from the outside.

Student

Graduate

- **Usages:**

```
Student student;
student.setData("Mel", 19);
Graduate gradStudent("Ron", 24, 3000);
```

ctor(), dtor()
setData()
getAge()
getName()

: Student
m_name = "Mel"
m_age = 19

ctor(), dtor()
getStipend()

setData()
getAge()
getName()

: Graduate
: Student
m_name = "Ron"
m_age = 24
m_stipend = 3000

```
cout << student.getName() << " is " << student.getAge()
     << " years old undergraduate student\n";
cout << gradStudent.getName() << " is  " << gradStudent.getAge()
     << " years old and has a stipend of " << gradStudent.getStipend()
     << " dollars.\n";
```

Basic Inheritance (cont'd)

- This would be illegal

```
int Graduate::getStipend() const {
    if (m_age > 30)
        return 0;
    return m_stipend;
}
```

Basic Inheritance (cont'd)

- This would be illegal

```
int Graduate::getStipend() const {
    if (m_age > 30)
        return 0;
    return m_stipend;
}
```

- Private data member of the base class is implicitly declared/defined but is still kept private from its derived class. (the boundary of base class is maintained)

Basic Inheritance (cont'd)

- This would be illegal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

- Private data member of the base class is implicitly declared/defined but is still kept private from its derived class. (the boundary of base class is maintained)

- This is legal

```
int Graduate::getStipend() const {
   if (getAge() > 30)
      return 0;
   return m_stipend;
}
```

Basic Inheritance (cont'd)

- This would be illegal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

- Private data member of the base class is implicitly declared/defined but is still kept private from its derived class. (the boundary of base class is maintained)

- This is legal

```
int Graduate::getStipend() const {
   if (getAge() > 30)
      return 0;
   return m_stipend;
}
```

- **Back to OCP**:

Basic Inheritance (cont'd)

- This would be illegal

```
int Graduate::getStipend() const {
    if (m_age > 30)
        return 0;
    return m_stipend;
}
```

- Private data member of the base class is implicitly declared/defined but is still kept private from its derived class. (the boundary of base class is maintained)
- This is legal

```
int Graduate::getStipend() const {
    if (getAge() > 30)
        return 0;
    return m_stipend;
}
```

- **Back to OCP**: Did you extend the functionality of the class Student?

Basic Inheritance (cont’d)

- This would be illegal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

- Private data member of the base class is implicitly declared/defined but is still kept private from its derived class. (the boundary of base class is maintained)

- This is legal

```
int Graduate::getStipend() const {
   if (getAge() > 30)
      return 0;
   return m_stipend;
}
```

- **Back to OCP**: Did you extend the functionality of the class Student?
  Did you edit student.h or student.cpp?

Protected Data and Functions

- Can we give the derived class access to "private" data of base class?

Protected Data and Functions

✧ Can we give the derived class access to "private" data of base class?

```
class Student {
public:
   Student();
   ~Student();
   void setData(char *name, int
   age);
   int getAge() const;
   const char *getName() const;
protected:
   char *m_name;
   int m_age;
};
```

Protected Data and Functions

- Can we give the derived class access to "private" data of base class?

```
class Student {
public:
   Student();
   ~Student();
   void setData(char *name, int
  age);
   int getAge() const;
   const char *getName() const;
protected:
   char *m_name;
   int m_age;
};
```

- The following is now legal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

Protected Data and Functions

- Can we give the derived class access to "private" data of base class?

```
class Student {
public:
   Student();
   ~Student();
   void setData(char *name, int
   age);
   int getAge() const;
   const char *getName() const;
protected:
   char *m_name;
   int m_age;
};
```

- The following is now legal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

- Who can access protected fields?

Protected Data and Functions

- Can we give the derived class access to "private" data of base class?

```
class Student {
public:
   Student();
   ~Student();
   void setData(char *name, int
  age);
   int getAge() const;
   const char *getName() const;
protected:
   char *m_name;
   int m_age;
};
```

- The following is now legal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

- Who can access protected fields?
  - base class and friends of base class

Protected Data and Functions

- Can we give the derived class access to "private" data of base class?

```
class Student {
public:
   Student();
   ~Student();
   void setData(char *name, int
   age);
   int getAge() const;
   const char *getName() const;
protected:
   char *m_name;
   int m_age;
};
```

- The following is now legal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

- Who can access protected fields?
  - base class and friends of base class
  - derived class and friends of derived classes

Protected Data and Functions

- Can we give the derived class access to "private" data of base class?

```
class Student {
public:
   Student();
   ~Student();
   void setData(char *name, int
   age);
   int getAge() const;
   const char *getName() const;
protected:
   char *m_name;
   int m_age;
};
```

- The following is now legal

```
int Graduate::getStipend() const {
   if (m_age > 30)
      return 0;
   return m_stipend;
}
```

Note: the encapsulation perimeter is enlarged a great deal with "protected" in your design

- Who can access protected fields?
  - base class and friends of base class
  - derived class and friends of derived classes

Basic Inheritance (cont'd)

- Most of the member functions of the base class are implicitly inherited by the derived class except

Basic Inheritance (cont'd)

- Most of the member functions of the base class are implicitly inherited by the derived class except
  - The constructor (including copy ctor)

Basic Inheritance (cont'd)

- Most of the member functions of the base class are implicitly inherited by the derived class except
  - The constructor (including copy ctor)
  - The assignment operator

Basic Inheritance (cont'd)

- Most of the member functions of the base class are implicitly inherited by the derived class except
  - The constructor (including copy ctor)
  - The assignment operator
  - The destructor

Basic Inheritance (cont'd)

- Most of the member functions of the base class are implicitly inherited by the derived class except
  - The constructor (including copy ctor)
  - The assignment operator
  - The destructor
- They are synthesized by the complier again if not explicitly defined. The synthesized ctor, dtor, and assignment operator would chain automatically to the function defined in the base class.

Inheritance and Constructors

- Rewrite Student using constructor

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
    Student(char *name, int age);
    ~Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
    Student(char *name, int age);
    ~Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

- In this case, the constructor for Graduate fails

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
   Student(char *name, int age);
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

- In this case, the constructor for Graduate fails

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
   Student(char *name, int age);
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

- In this case, the constructor for Graduate fails

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

error C2512: 'Student' : no appropriate default constructor available

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
    Student(char *name, int age);
    ~Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

- In this case, the constructor for Graduate fails

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
    setData(name, age); // this is inherited from Student
}
```

error C2512: 'Student' : no appropriate default constructor available

- Why??

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
   Student(char *name, int age);
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

- In this case, the constructor for Graduate fails

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

error C2512: 'Student' : no appropriate default constructor available

- Why??

```
Graduate::Graduate(char *name, int age, int stipend)
            : Student(), m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
   Student(char *name, int age);
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

- In this case, the constructor for Graduate fails

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

error C2512: 'Student' : no appropriate default constructor available

- Why??

```
Graduate::Graduate(char *name, int age, int stipend)
            : Student(), m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

Compiler insert this automatically

Inheritance and Constructors

- Rewrite Student using constructor

```
class Student {
public:
   Student(char *name, int age);
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

- In this case, the constructor for Graduate fails

```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

error C2512: 'Student' : no appropriate default constructor available

- Why??

```
Graduate::Graduate(char *name, int age, int stipend)
            : Student(), m_stipend(stipend) {
   setData(name, age); // this is inherited from Student
}
```

chaining

Compiler insert this automatically

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

```
Graduate::Graduate(char *name, int age, int stipend)
              : Student(name, age), m_stipend(stipend) {
   setData(name, age); // setData() is inherited from Student
}
```

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

```
Graduate::Graduate(char *name, int age, int stipend)
              : Student(name, age), m_stipend(stipend) {
   setData(name, age); // setData() is inherited from Student
}
```

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

```
Graduate::Graduate(char *name, int age, int stipend)
                : Student(name, age), m_stipend(stipend) {
    setData(name, age); // setData() is inherited from Student
}

Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

```
Graduate::Graduate(char *name, int age, int stipend)
                  : Student(name, age), m_stipend(stipend) {
    setData(name, age); // setData() is inherited from Student
}

Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```

- You cannot initialize base class members directly in the initialization list even if they are public or protected, i.e.

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

```
Graduate::Graduate(char *name, int age, int stipend)
                : Student(name, age), m_stipend(stipend) {
    setData(name, age); // setData() is inherited from Student
}

Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```

- You cannot initialize base class members directly in the initialization list even if they are public or protected, i.e.

```
Graduate::Graduate(char *name, int age, int stipend)
                : m_age(age), m_stipend(stipend)
```

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

```
Graduate::Graduate(char *name, int age, int stipend)
              : Student(name, age), m_stipend(stipend) {
   setData(name, age); // setData() is inherited from Student
}

Student::Student(char *name, int age) : m_age(age) {
   m_name = new char[strlen(name)+1];
   strcpy(m_name, name);
}
```

- You cannot initialize base class members directly in the initialization list even if they are public or protected, i.e.

```
Graduate::Graduate(char *name, int age, int stipend)
              : m_age(age), m_stipend(stipend)
```

error C2614: 'Graduate' : illegal member initialization: 'm_age' is not a base or member

Inheritance and Ctors (cont'd)

- In this case, the correct form of the constructor for Graduate is

```
Graduate::Graduate(char *name, int age, int stipend)
                : Student(name, age), m_stipend(stipend) {
    setData(name, age); // setData() is inherited from Student
}

Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```

- You cannot initialize base class members directly in the initialization list even if they are public or protected, i.e.

```
Graduate::Graduate(char *name, int age, int stipend)
                : m_age(age), m_stipend(stipend)
```

error C2614: 'Graduate' : illegal member initialization: 'm_age' is not a base or member

- Base class guarantee

  The base class will be fully constructed before the body of the derived class constructor is entered

Copy Constructor

- Copy constructor is also a constructor. Member objects and base class **must** be initialized through **initialization list**

Copy Constructor

- Copy constructor is also a constructor. Member objects and base class **must** be initialized through **initialization list**
- For example:

Copy Constructor

- Copy constructor is also a constructor. Member objects and base class **must** be initialized through **initialization list**
- For example:

```
class Derived: public Base {
public:
    …
    Derived(Derived &src);
    …
private:
    Component m_obj;
};
```

Copy Constructor

- Copy constructor is also a constructor. Member objects and base class **must** be initialized through **initialization list**

- For example:

```
class Derived: public Base {
public:
    …
    Derived(Derived &src);
    …
private:
    Component m_obj;
};
Derived::Derived(Derived &src): Base(src), m_obj(src.m_obj) {
    …
}
```

Copy Constructor

- Copy constructor is also a constructor. Member objects and base class **must** be initialized through **initialization list**

- For example:

```
class Derived: public Base {
public:
   …
   Derived(Derived &src);
   …
private:
   Component m_obj;
};
Derived::Derived(Derived &src): Base(src), m_obj(src.m_obj) {
   …
}
```

If you do not define a copy ctor, the compiler would generate one exactly like this.

Copy Constructor

- Copy constructor is also a constructor. Member objects and base class **must** be initialized through **initialization list**
- For example:

```
class Derived: public Base {
public:
    …
    Derived(Derived &src);
    …
private:
    Component m_obj;
};
Derived::Derived(Derived &src): Base(src), m_obj(src.m_obj) {
    …
}
```

Note:

```
Derived::Derived(Derived &src):
        m_obj(src.m_obj)
{
    …
}
```

If you do not define a copy ctor, the compiler would generate one exactly like this.

Copy Constructor

- Copy constructor is also a constructor. Member objects and base class **must** be initialized through **initialization list**

- For example:

Compiler adds **Base()** invocation automatically

```
class Derived: public Base {
public:
   …
   Derived(Derived &src);
   …
private:
   Component m_obj;
};
```

Note:

```
Derived::Derived(Derived &src):
      m_obj(src.m_obj)
{
   …
}
```

```
Derived::Derived(Derived &src): Base(src), m_obj(src.m_obj) {
   …
}
```

If you do not define a copy ctor, the compiler would generate one exactly like this.

Inheritance and Destructors

- If we add a dynamically allocated string data member to Graduate to store the student's home address, then Graduate requires a destructor

Inheritance and Destructors

- If we add a dynamically allocated string data member to Graduate to store the student's home address, then Graduate requires a destructor

```
Student::Student(char *name, int age) : m_age(age) {
   m_name = new char[strlen(name)+1];
   strcpy(m_name, name);
   cout << "In Student ctor\n";
}
```

Inheritance and Destructors

- If we add a dynamically allocated string data member to Graduate to store the student's home address, then Graduate requires a destructor

```
Student::Student(char *name, int age) : m_age(age) {
   m_name = new char[strlen(name)+1];
   strcpy(m_name, name);
   cout << "In Student ctor\n";
}
```

```
Student::~Student() {
   delete[] m_name;
   cout << "In Student dtor\n";
}
```

Inheritance and Destructors

- If we add a dynamically allocated string data member to Graduate to store the student's home address, then Graduate requires a destructor

```
Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
    cout << "In Student ctor\n";
}
```

```
Student::~Student() {
    delete[] m_name;
    cout << "In Student dtor\n";
}
```

```
Graduate::Graduate(char *name, int age, int stipend, char *address)
    : Student(name, age), m_stipend(stipend) {
    m_address = new char[strlen(address)+1];
    strcpy(m_address, address);
    cout << "In Graduate ctor\n";
}
```

Inheritance and Destructors

- If we add a dynamically allocated string data member to Graduate to store the student's home address, then Graduate requires a destructor

```
Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
    cout << "In Student ctor\n";
}
```

```
Student::~Student() {
    delete[] m_name;
    cout << "In Student dtor\n";
}
```

```
Graduate::Graduate(char *name, int age, int stipend, char *address)
    : Student(name, age), m_stipend(stipend) {
    m_address = new char[strlen(address)+1];
    strcpy(m_address, address);
    cout << "In Graduate ctor\n";
}
```

```
Graduate::~Graduate() {
    delete[] m_address;
    cout << "In Graduate dtor\n";
}
```

Inheritance and Dtors (cont'd)

- What happens in main()

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
         << "has a stipend of " <, student.getStipend() << "dollars.\n"
         << "His address is " << student.getAddress() << "\n";
}
```

Inheritance and Dtors (cont'd)

✧ What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
         << "has a stipend of " <, student.getStipend() << "dollars.\n"
         << "His address is " << student.getAddress() << "\n";
}
```

The output is:

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
         << "has a stipend of " <, student.getStipend() << "dollars.\n"
         << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
```

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
         << "has a stipend of " <, student.getStipend() << "dollars.\n"
         << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
```

Inheritance and Dtors (cont'd)

✧ What happens in main()

```
void main() {
    Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
    cout << student.getName() << " is " << student.getAge() << " years old and "
            << "has a stipend of " <, student.getStipend() << "dollars.\n"
            << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
```

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
         << "has a stipend of " <, student.getStipend() << "dollars.\n"
         << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
His address is 8899 Storkes Rd.
```

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
    Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
    cout << student.getName() << " is " << student.getAge() << " years old and "
            << "has a stipend of " <, student.getStipend() << "dollars.\n"
            << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
His address is 8899 Storkes Rd.
In Graduate dtor
```

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
          << "has a stipend of " <, student.getStipend() << "dollars.\n"
          << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
His address is 8899 Storkes Rd.
In Graduate dtor
In Student dtor
```

chaining

Inheritance and Dtors (cont’d)

- What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
         << "has a stipend of " <, student.getStipend() << "dollars.\n"
         << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
His address is 8899 Storkes Rd.
In Graduate dtor
In Student dtor
```

chaining

- The compiler automatically calls each dtor when the object dies.

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
    Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
    cout << student.getName() << " is " << student.getAge() << " years old and "
            << "has a stipend of " <, student.getStipend() << "dollars.\n"
            << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
His address is 8899 Storkes Rd.
In Graduate dtor
In Student dtor
```

chaining

- The compiler automatically calls each dtor when the object dies.
- The dtors are invoked in the opposite order of the ctors

Inheritance and Dtors (cont'd)

- What happens in main()

```
void main() {
   Graduate student("Michael", 24, 6000, " 8899 Storkes Rd.");
   cout << student.getName() << " is " << student.getAge() << " years old and "
         << "has a stipend of " <, student.getStipend() << "dollars.\n"
         << "His address is " << student.getAddress() << "\n";
}
```

The output is:

```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
His address is 8899 Storkes Rd.
In Graduate dtor
In Student dtor
```

chaining

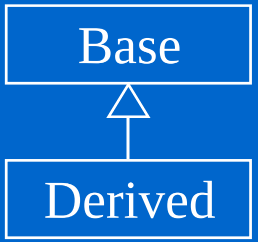
- The compiler automatically calls each dtor when the object dies.
- The dtors are invoked in the opposite order of the ctors
  - In destructing the derived object, the base object is still in scope and functioning correctly.

Chaining of Assignment Operator

- By default, the compiler adds a "**bit-wise copy**" assignment operator for every class which you do not define an assignment operator

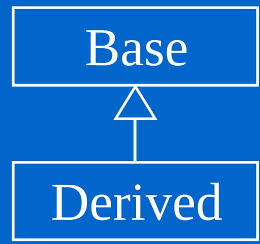
Chaining of Assignment Operator

- By default, the compiler adds a "**bit-wise copy**" assignment operator for every class which you do not define an assignment operator
- If you have a class hierarchy where a class Derived inherits from a class Base. There are 4 possibilities in defining their assignment operators:



Chaining of Assignment Operator

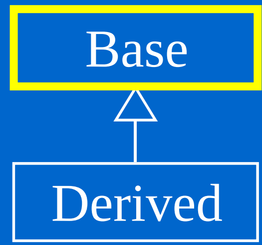
- By default, the compiler adds a "**bit-wise copy**" assignment operator for every class which you do not define an assignment operator
- If you have a class hierarchy where a class Derived inherits from a class Base. There are 4 possibilities in defining their assignment operators:



1. If both classes do not have assignment operator: both are bit-wise copy

Chaining of Assignment Operator

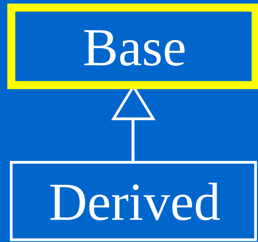
- By default, the compiler adds a "**bit-wise copy**" assignment operator for every class which you do not define an assignment operator
- If you have a class hierarchy where a class Derived inherits from a class Base. There are 4 possibilities in defining their assignment operators:



1. If both classes do not have assignment operator: both are bit-wise copy
2. If you define Base& Base::operator=(Base &) but not Derived& Derived::operator=(Derived &), then compiler synthesizes

Chaining of Assignment Operator

- By default, the compiler adds a "**bit-wise copy**" assignment operator for every class which you do not define an assignment operator
- If you have a class hierarchy where a class Derived inherits from a class Base. There are 4 possibilities in defining their assignment operators:

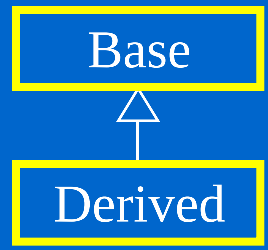


1. If both classes do not have assignment operator: both are bit-wise copy
2. If you define Base& Base::operator=(Base &) but not Derived& Derived::operator=(Derived &), then compiler synthesizes

```
Derived& Derived::operator=(Derived &rhs) {
    Base::operator=(rhs); // calling your function
        ….
    return *this;
}
```

Chaining of Assignment Operator

- By default, the compiler adds a "**bit-wise copy**" assignment operator for every class which you do not define an assignment operator
- If you have a class hierarchy where a class Derived inherits from a class Base. There are 4 possibilities in defining their assignment operators:



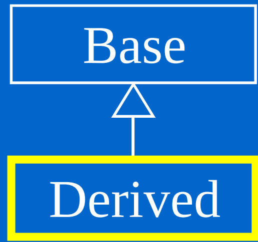
1. If both classes do not have assignment operator: both are bit-wise copy
2. If you define Base& Base::operator=(Base &) but not Derived& Derived::operator=(Derived &), then compiler synthesizes

```
Derived& Derived::operator=(Derived &rhs) {
    Base::operator=(rhs); // calling your function
        ….
    return *this;
}
```

3&4. If you define Derived& Derived::operator=(Derived &rhs) yourself, you have to call Base::operator=(rhs); in Derived::operator=(Derived) no matter it is synthesized or not; otherwise the Base part of the object would not be copied.

Chaining of Assignment Operator

- By default, the compiler adds a "**bit-wise copy**" assignment operator for every class which you do not define an assignment operator
- If you have a class hierarchy where a class Derived inherits from a class Base. There are 4 possibilities in defining their assignment operators:



1. If both classes do not have assignment operator: both are bit-wise copy
2. If you define Base& Base::operator=(Base &) but not Derived& Derived::operator=(Derived &), then compiler synthesizes

```
Derived& Derived::operator=(Derived &rhs) {
    Base::operator=(rhs); // calling your function
        ….
    return *this;
}
```

3&4. If you define Derived& Derived::operator=(Derived &rhs) yourself, you have to call Base::operator=(rhs); in Derived::operator=(Derived) no matter it is synthesized or not; otherwise the Base part of the object would not be copied.

Layers of Inheritance

- Let us add a new type of graduate student

Layers of Inheritance

- Let us add a new type of graduate student

```
class Student {
public:
    Student(char *name, int age);
    ~Student();
    void setData(char *name, int age);
    int getAge() const;
    const char *getName() const;
private:
    char *m_name;
    int m_age;
};
```

Layers of Inheritance

- Let us add a new type of graduate student

```
class Student {
public:
   Student(char *name, int age);
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

Layers of Inheritance

- Let us add a new type of graduate student

```
class Student {
public:
   Student(char *name, int age);
   ~Student();
   void setData(char *name, int age);
   int getAge() const;
   const char *getName() const;
private:
   char *m_name;
   int m_age;
};
```

```
class ForeignGraduate: public Graduate {
public:
   ForeignGraduate(char *name, int age,
                   int stipend,
                   char *nationality);
   ~ForeignGraduate()
   const char *getNationality();
private:
   char *m_nationality;
};
```

```
class Graduate: public Student {
public:
   Graduate(char *name, int age, int stipend);
   int getStipend() const;
private:
   int m_stipend;
};
```

Layers of Inheritance (cont'd)

Student

Layers of Inheritance (cont'd)

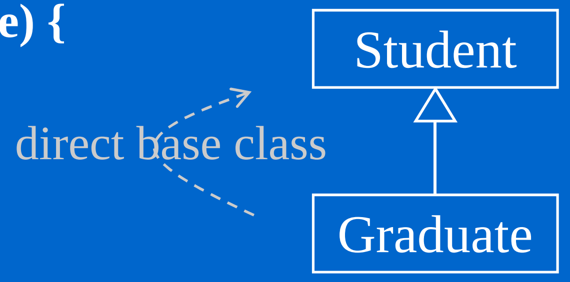
- **ctor of Student**

```
Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```

Layers of Inheritance (cont'd)

* **ctor of Student**

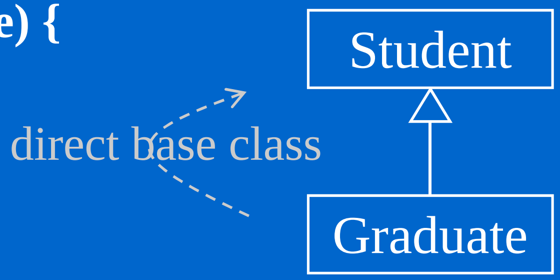
```
Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```



Layers of Inheritance (cont'd)

- **ctor of Student**

```
Student::Student(char *name, int age) : m_age(age) {
   m_name = new char[strlen(name)+1];
   strcpy(m_name, name);
}
```



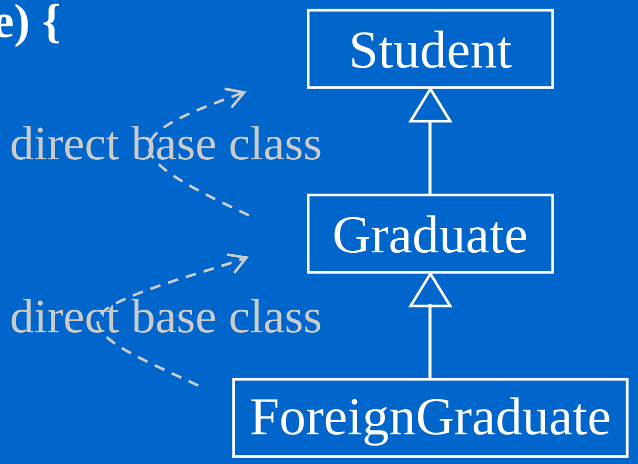
- **ctor of Graduate invokes the ctor of its direct base class - Student**

```
Graduate::Graduate(char *name, int age, int stipend)
   : Student(name, age), m_stipend(stipend) {
}
```

Layers of Inheritance (cont'd)

- ctor of Student

```
Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```



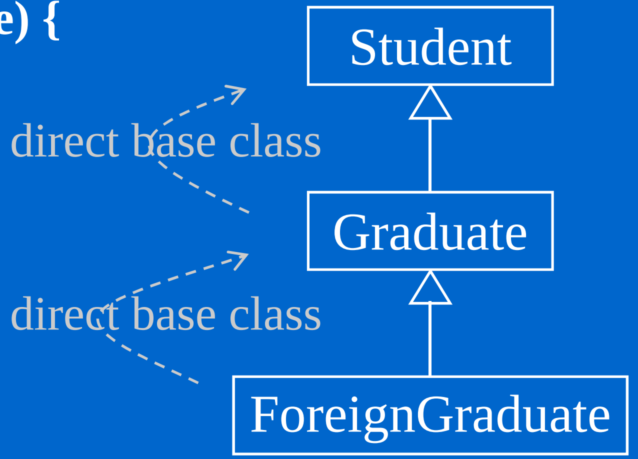
- ctor of Graduate invokes the ctor of its direct base class - Student

```
Graduate::Graduate(char *name, int age, int stipend)
    : Student(name, age), m_stipend(stipend) {
}
```

Layers of Inheritance (cont'd)

- ctor of Student

```
Student::Student(char *name, int age) : m_age(age) {
   m_name = new char[strlen(name)+1];
   strcpy(m_name, name);
}
```



- ctor of Graduate invokes the ctor of its direct base class - Student

```
Graduate::Graduate(char *name, int age, int stipend)
   : Student(name, age), m_stipend(stipend) {
}
```

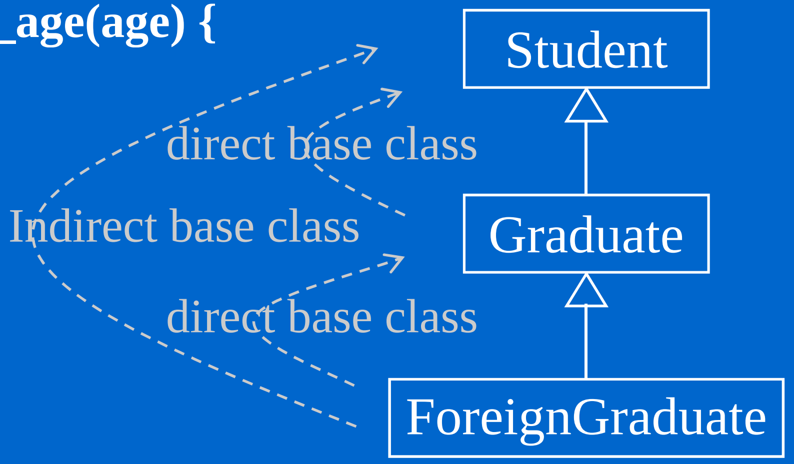
- ctor of ForeignGraduate invokes the ctor of its direct base class - Graduate

```
ForeignGraduate::ForeignGraduate(char *name,
              int age, int stipend, char *nationality)
     : Graduate(name, age, stipend) {
   m_nationality = new char[strlen(nationality)+1];
   strcpy(m_nationality, nationality);
}
```

Layers of Inheritance (cont'd)

- ctor of Student

```
Student::Student(char *name, int age) : m_age(age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
}
```



- ctor of Graduate invokes the ctor of its direct base class - Student

```
Graduate::Graduate(char *name, int age, int stipend)
    : Student(name, age), m_stipend(stipend) {
}
```

- ctor of ForeignGraduate invokes the ctor of its direct base class - Graduate

```
ForeignGraduate::ForeignGraduate(char *name,
              int age, int stipend, char *nationality)
      : Graduate(name, age, stipend) {
    m_nationality = new char[strlen(nationality)+1];
    strcpy(m_nationality, nationality);
}
```

Behavior Changing (Hiding)

- In the previous example, suppose we would like to have a display() member function in the Student class that shows the details of a Student object on the screen, ex.

Behavior Changing (Hiding)

- In the previous example, suppose we would like to have a display() member function in the Student class that shows the details of a Student object on the screen, ex.

```
void Student::display() const {
    cout << m_name << " is " << m_age << "years old.\n";
}
```

Behavior Changing (Hiding)

- In the previous example, suppose we would like to have a display() member function in the Student class that shows the details of a Student object on the screen, ex.

```
void Student::display() const {
    cout << m_name << " is " << m_age << "years old.\n";
}
```

- The Graduate class **automatically inherits this member function**. However, the output of this function for a Graduate object is in a way short of many important data.

Behavior Changing (Hiding)

- In the previous example, suppose we would like to have a display() member function in the Student class that shows the details of a Student object on the screen, ex.

```
void Student::display() const {
    cout << m_name << " is " << m_age << "years old.\n";
}
```

- The Graduate class **automatically inherits this member function**. However, the output of this function for a Graduate object is in a way short of many important data.
- We would like to **redefine this function** in the derived class – Graduate, such that it will show the stipend and address together.

Behavior Changing (Hiding)

- In the previous example, suppose we would like to have a display() member function in the Student class that shows the details of a Student object on the screen, ex.

```
void Student::display() const {
   cout << m_name << " is " << m_age << "years old.\n";
}
```

- The Graduate class **automatically inherits this member function**. However, the output of this function for a Graduate object is in a way short of many important data.
- We would like to **redefine this function** in the derived class – Graduate, such that it will show the stipend and address together.

```
void Graduate::display() const {    // masks the inherited version of display()
   cout << getName() << " is " << getAge() << " years old.\n";
   cout << "He has a stipend of " << m_stipend << " dollars.\n";
   cout << "His address is " << m_address << ".\n";
}
```

Behavior Changing (Hiding)

- In the previous example, suppose we would like to have a display() member function in the Student class that shows the details of a Student object on the screen, ex.

```
void Student::display() const {
    cout << m_name << " is " << m_age << "years old.\n";
}
```

- The Graduate class **automatically inherits this member function**. However, the output of this function for a Graduate object is in a way short of many important data.
- We would like to **redefine this function** in the derived class – Graduate, such that it will show the stipend and address together.

```
void Graduate::display() const {    // masks the inherited version of display()
    cout << getName() << " is " << getAge() << " years old.\n";
    cout << "He has a stipend of " << m_stipend << " dollars.\n";
    cout << "His address is " << m_address << ".\n";
}
```

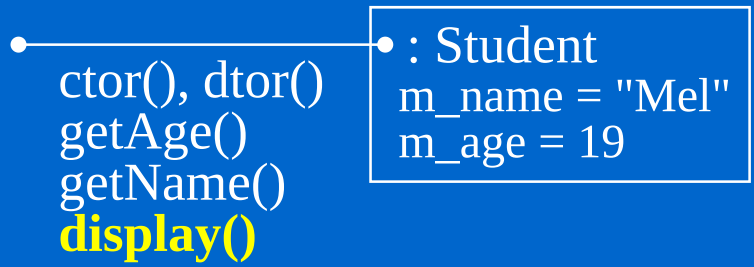
- Note: **function signature** is exactly the same as in the base class.

Behavior Changing (cont'd)

- Example **usage** of the previous design:

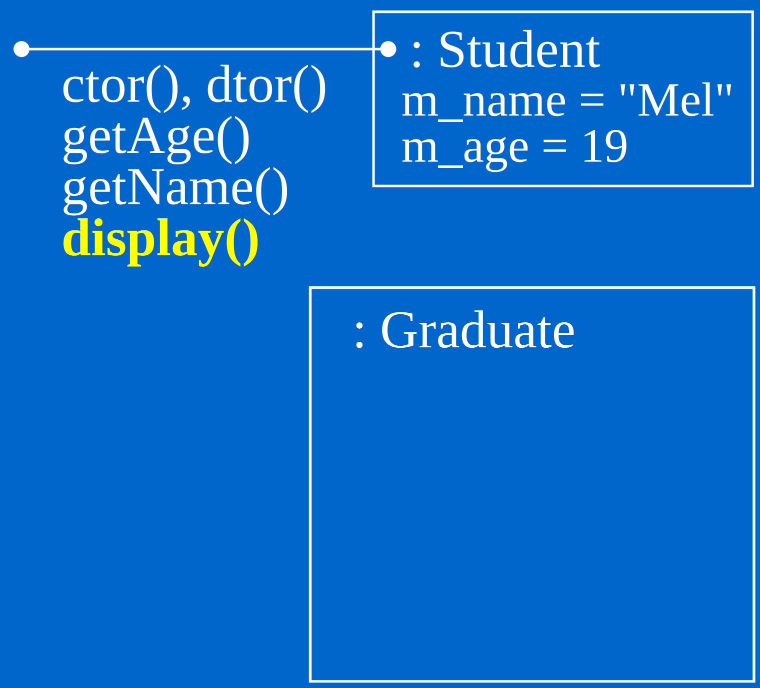
Behavior Changing (cont’d)

- Example **usage** of the previous design:



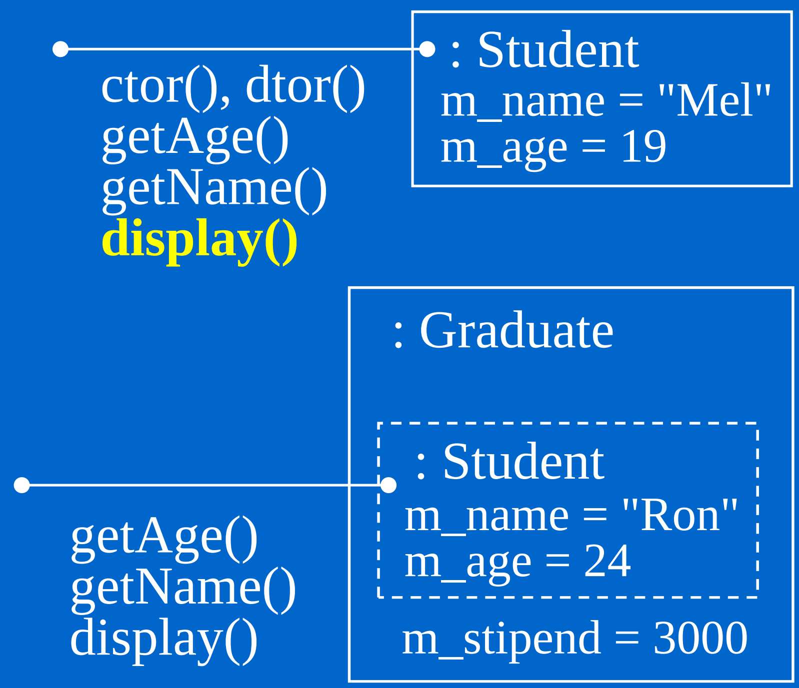
Behavior Changing (cont'd)

✧ Example **usage** of the previous design:



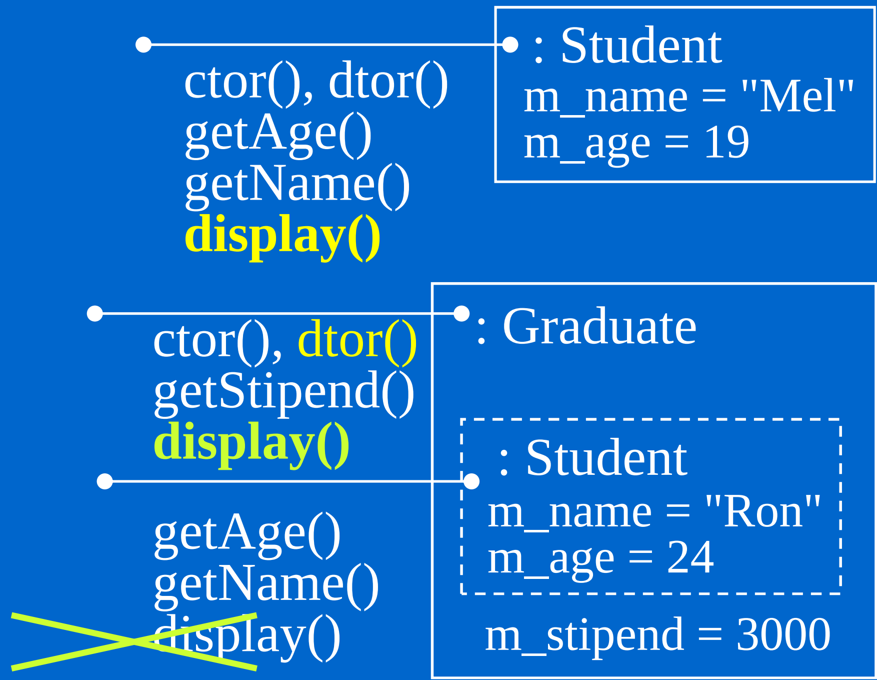
Behavior Changing (cont'd)

- Example **usage** of the previous design:



Behavior Changing (cont'd)

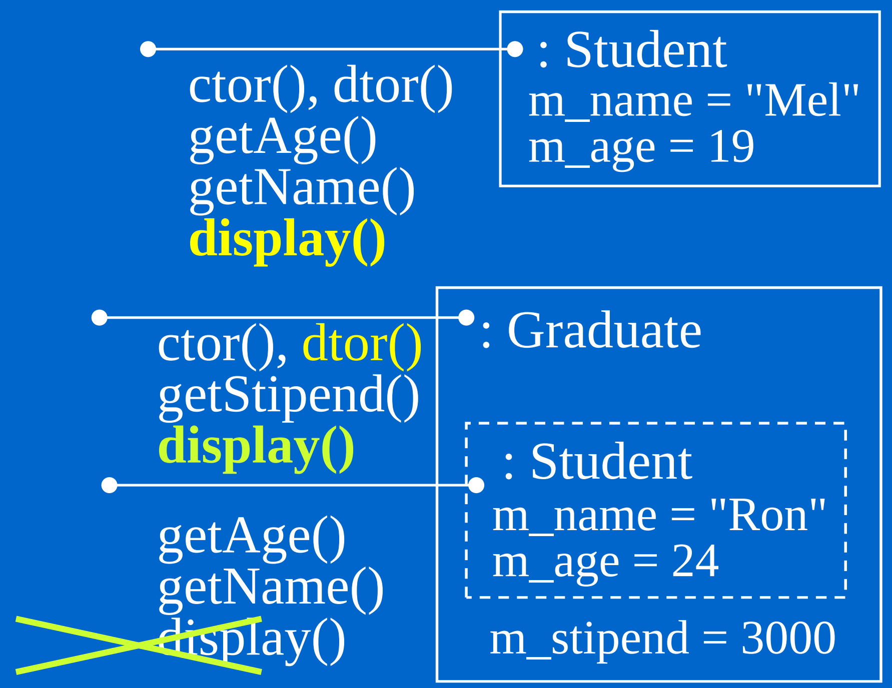
- Example **usage** of the previous design:



Behavior Changing (cont'd)

- Example **usage** of the previous design:

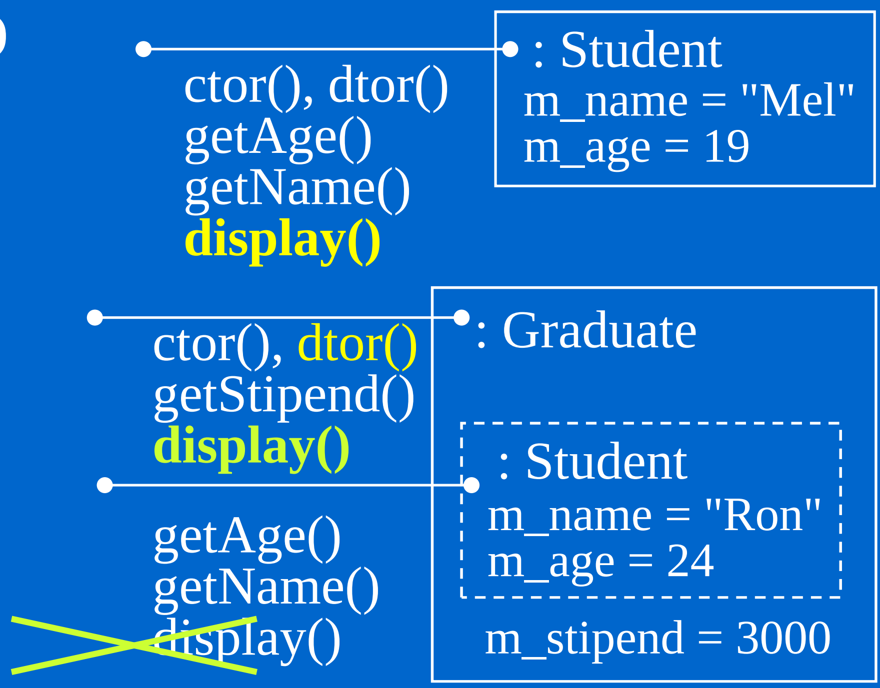
Student student1("Alice", 20);
Graduate student2("Michael", 24, 6000, "8899 Storkes Rd.");



Behavior Changing (cont'd)

- Example **usage** of the previous design:

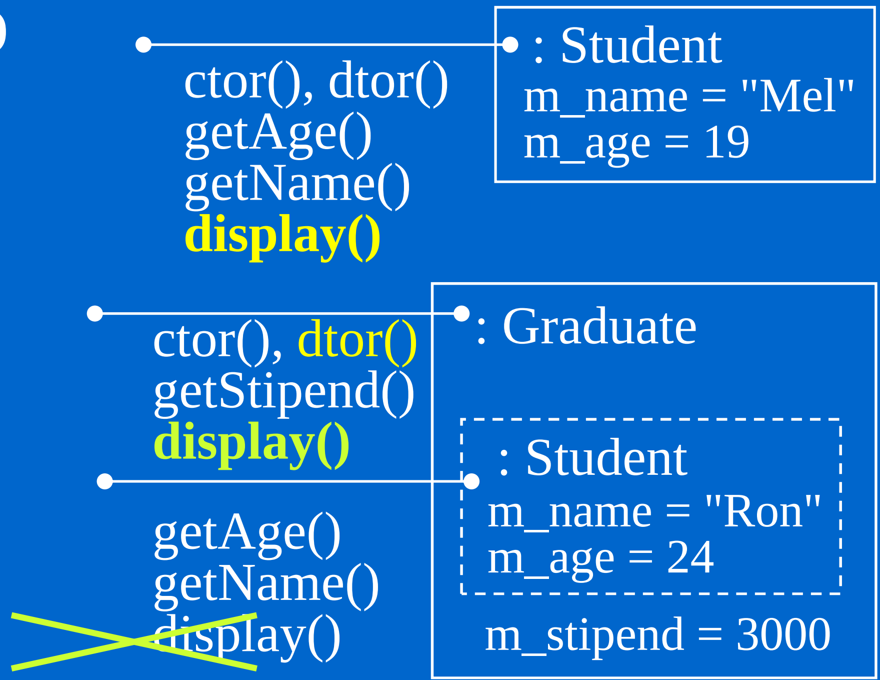
```
Student student1("Alice", 20);
Graduate student2("Michael", 24, 6000, "8899 Storkes Rd.");

student1.display();    // Student::display()
cout << "\n";
```



Behavior Changing (cont'd)

- Example **usage** of the previous design:

```
Student student1("Alice", 20);
Graduate student2("Michael", 24, 6000, "8899 Storkes Rd.");

student1.display();    // Student::display()
cout << "\n";
```



Output:

Alice is 20 years old.

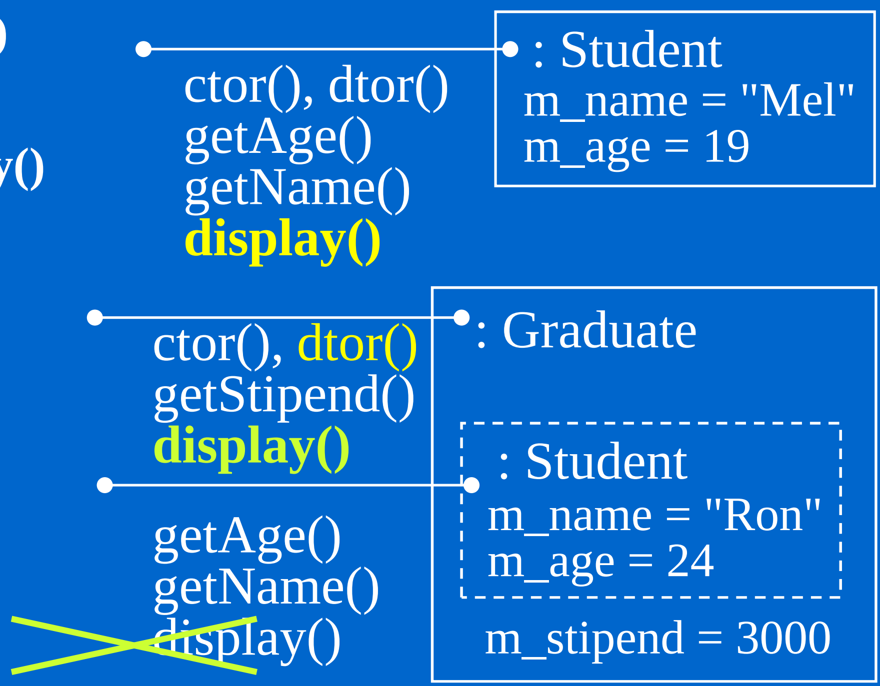
Behavior Changing (cont'd)

- Example **usage** of the previous design:

```
Student student1("Alice", 20);
Graduate student2("Michael", 24, 6000, "8899 Storkes Rd.");

student1.display();    // Student::display()
cout << "\n";
student2.display();    // Graduate::display()
```

Output:

```
Alice is 20 years old.
```



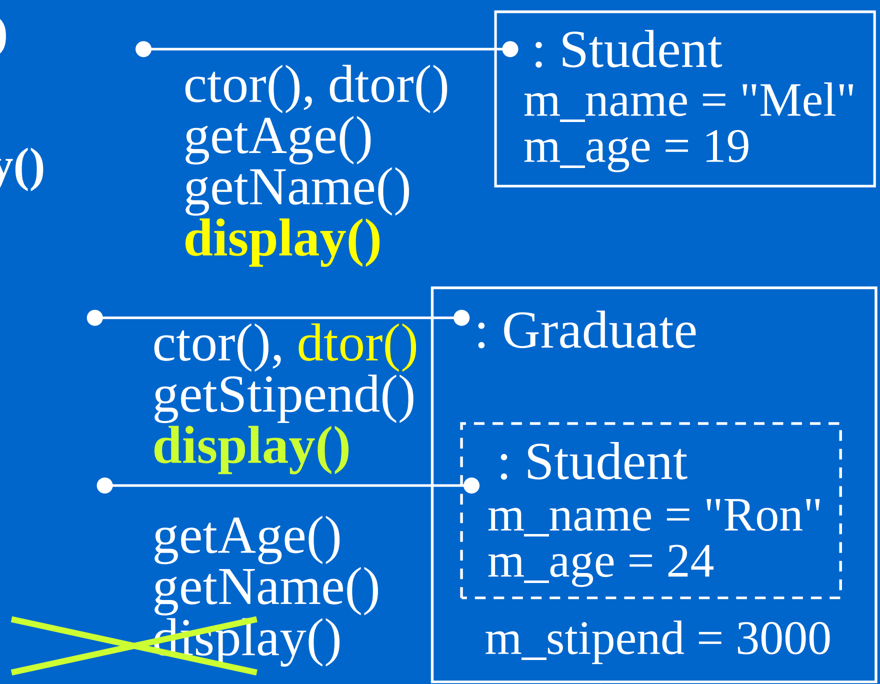
Behavior Changing (cont'd)

✧ Example **usage** of the previous design:

```
Student student1("Alice", 20);
Graduate student2("Michael", 24, 6000, "8899 Storkes Rd.");

student1.display();    // Student::display()
cout << "\n";
student2.display();    // Graduate::display()
```

Output:

```
Alice is 20 years old.

Michael is 24 years old.
He has a stipend of 6000 dollars.
His address is 8899 Storke Rd.
```



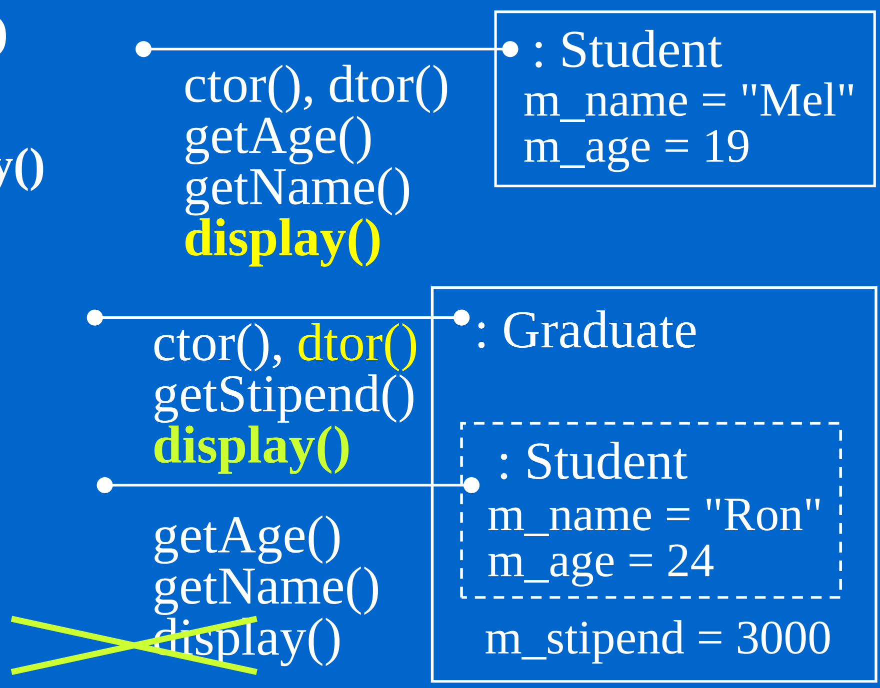
Behavior Changing (cont'd)

- Example **usage** of the previous design:

```
Student student1("Alice", 20);
Graduate student2("Michael", 24, 6000, "8899 Storkes Rd.");

student1.display();    // Student::display()
cout << "\n";

student2.display();    // Graduate::display()
```

Output:

```
Alice is 20 years old.

Michael is 24 years old.
He has a stipend of 6000 dollars.
His address is 8899 Storke Rd.
```



- Note: display() interface usually can enhance the encapsulation, replacing the functionality of trivial accessor functions

Behavior Changing (cont'd)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

Behavior Changing (cont’d)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

```
void Graduate::display() const // masks the inherited version of display() {

   cout << "He has a stipend of " << m_stipend << " dollars.\n";
   cout << "His address is " << m_address << ".\n";
}
```

Behavior Changing (cont'd)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

```
void Graduate::display() const // masks the inherited version of display() {
   Student::display(); // invoke the inherited codes
   cout << "He has a stipend of " << m_stipend << " dollars.\n";
   cout << "His address is " << m_address << ".\n";
}
```

Behavior Changing (cont'd)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

```
void Graduate::display() const // masks the inherited version of display() {
    Student::display(); // invoke the inherited codes
    cout << "He has a stipend of " << m_stipend << " dollars.\n";
    cout << "His address is " << m_address << ".\n";
}
```

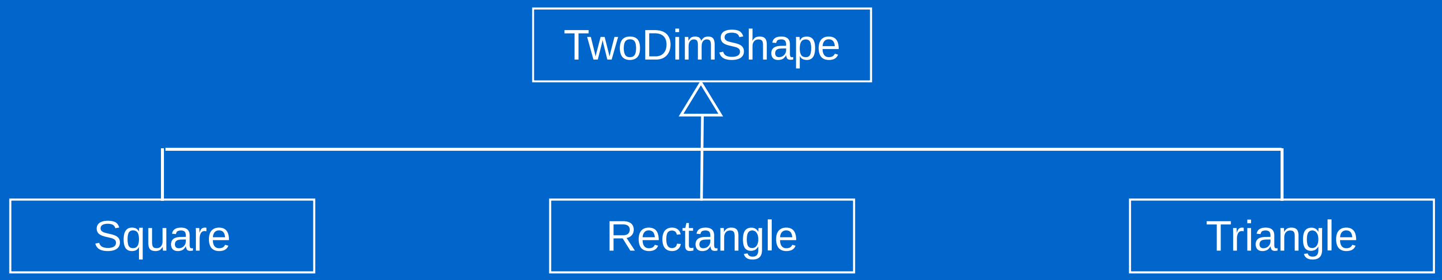
- The functions defined in the base class are OK for most derived classes. Only some of them need to be changed in the derived classes. Ex.

Behavior Changing (cont'd)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

```
void Graduate::display() const // masks the inherited version of display() {
   Student::display(); // invoke the inherited codes
   cout << "He has a stipend of " << m_stipend << " dollars.\n";
   cout << "His address is " << m_address << ".\n";
}
```

- The functions defined in the base class are OK for most derived classes. Only some of them need to be changed in the derived classes. Ex.

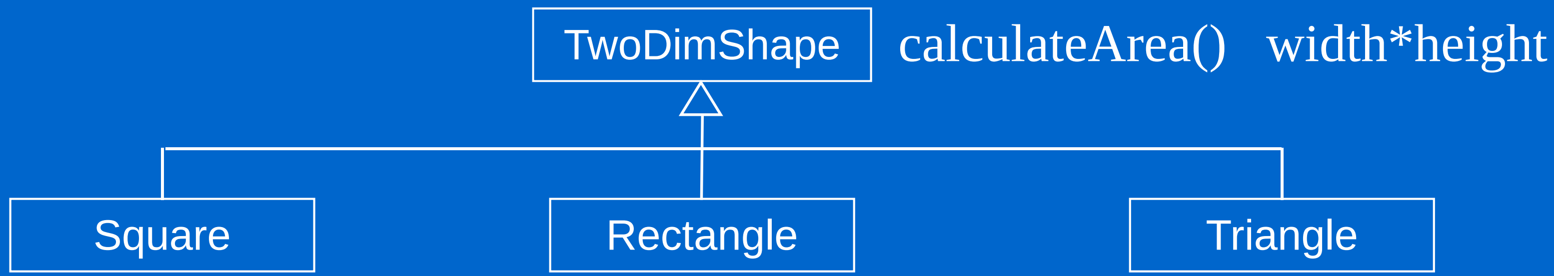


Behavior Changing (cont'd)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

```
void Graduate::display() const // masks the inherited version of display() {
    Student::display(); // invoke the inherited codes
    cout << "He has a stipend of " << m_stipend << " dollars.\n";
    cout << "His address is " << m_address << ".\n";
}
```

- The functions defined in the base class are OK for most derived classes. Only some of them need to be changed in the derived classes. Ex.

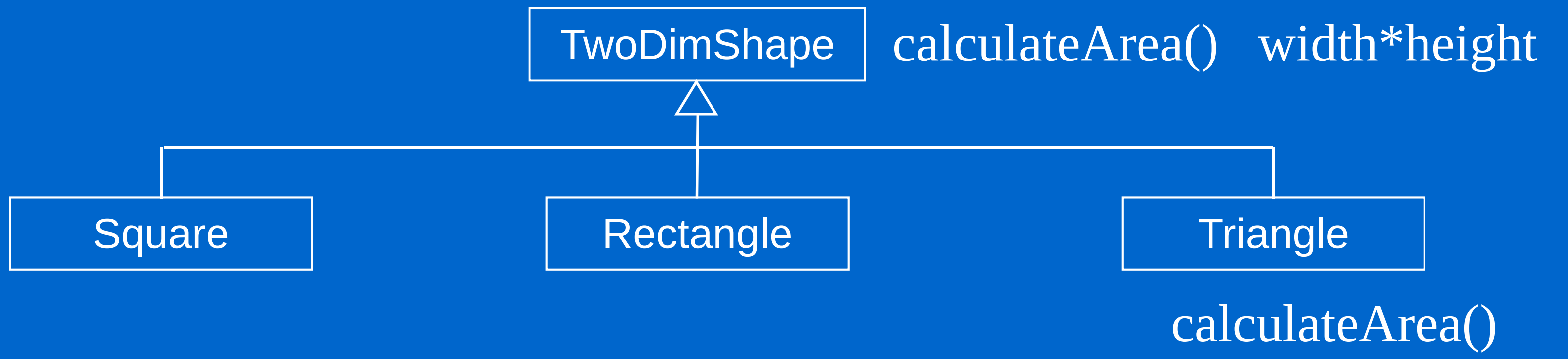


Behavior Changing (cont'd)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

```
void Graduate::display() const // masks the inherited version of display() {
   Student::display(); // invoke the inherited codes
   cout << "He has a stipend of " << m_stipend << " dollars.\n";
   cout << "His address is " << m_address << ".\n";
}
```

- The functions defined in the base class are OK for most derived classes. Only some of them need to be changed in the derived classes. Ex.

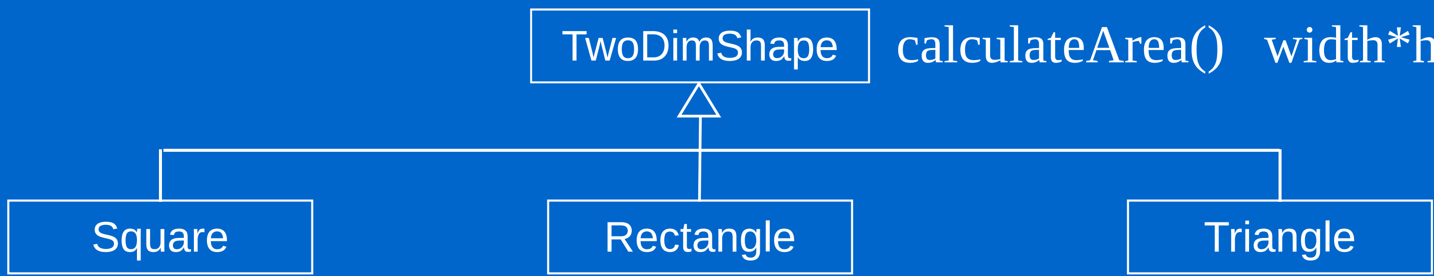


Behavior Changing (cont'd)

- Avoid the **redundancy** of the common code, **Student::display()**, in the inherited version of display(), **Graduate::display()**, by

```
void Graduate::display() const // masks the inherited version of display() {
   Student::display(); // invoke the inherited codes
   cout << "He has a stipend of " << m_stipend << " dollars.\n";
   cout << "His address is " << m_address << ".\n";
}
```

- The functions defined in the base class are OK for most derived classes. Only some of them need to be changed in the derived classes. Ex.



calculateArea() width*height

calculateArea()

1/2*TwoDimShape::calculateArea()

Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

  Example:

Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

Example:

Mini

Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

Example:

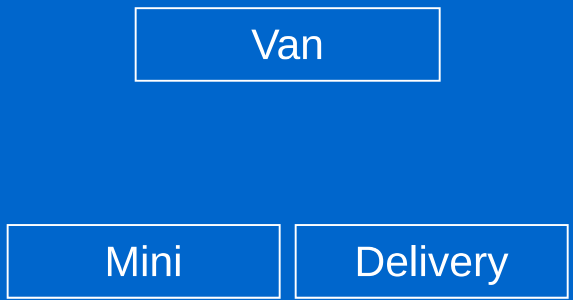
Mini

Delivery

Class Hierarchy

✧ sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

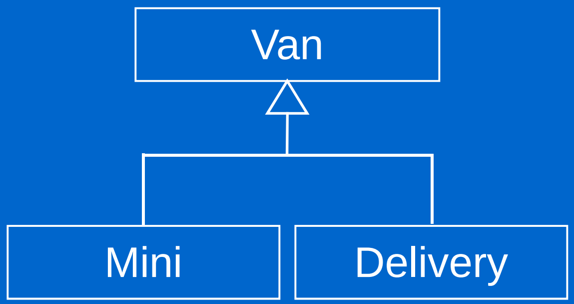
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

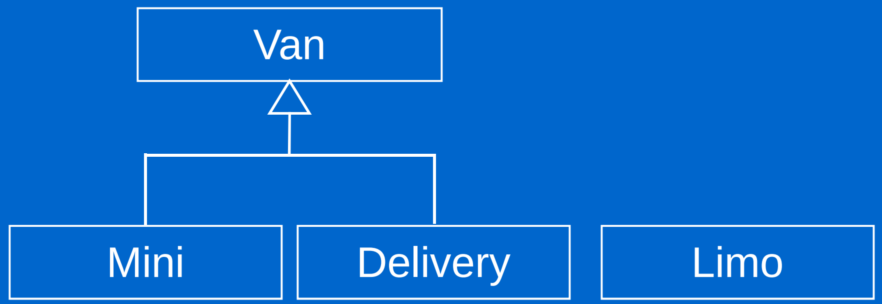
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

Example:

Class Hierarchy

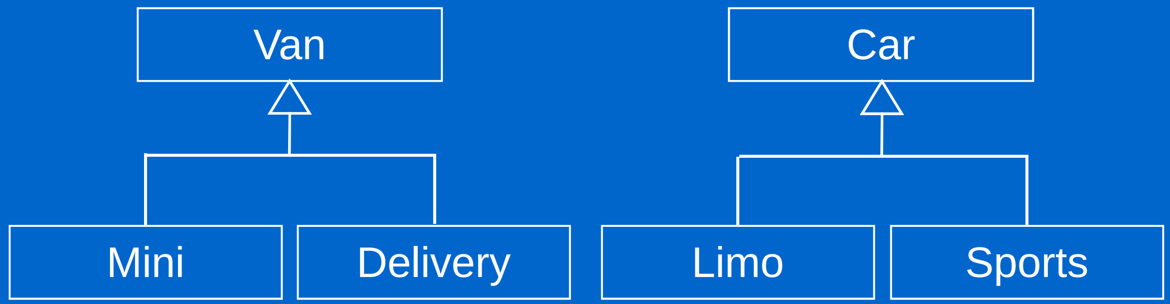
- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

Example:

Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

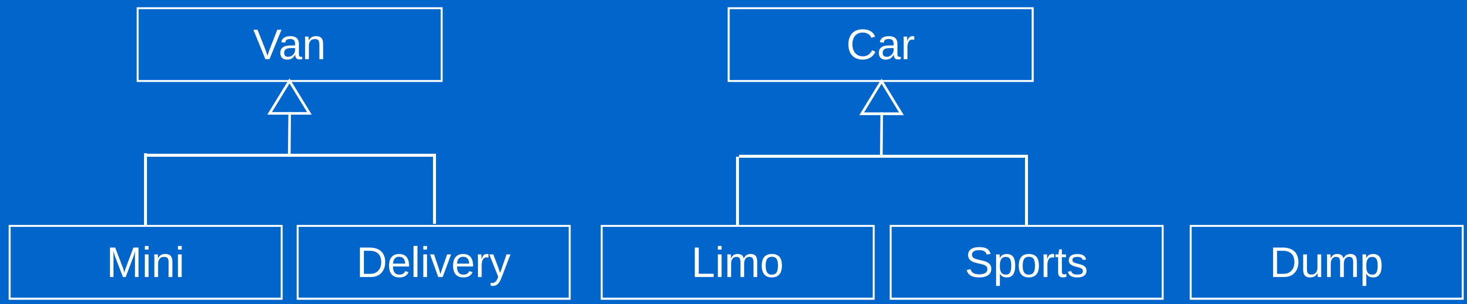
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

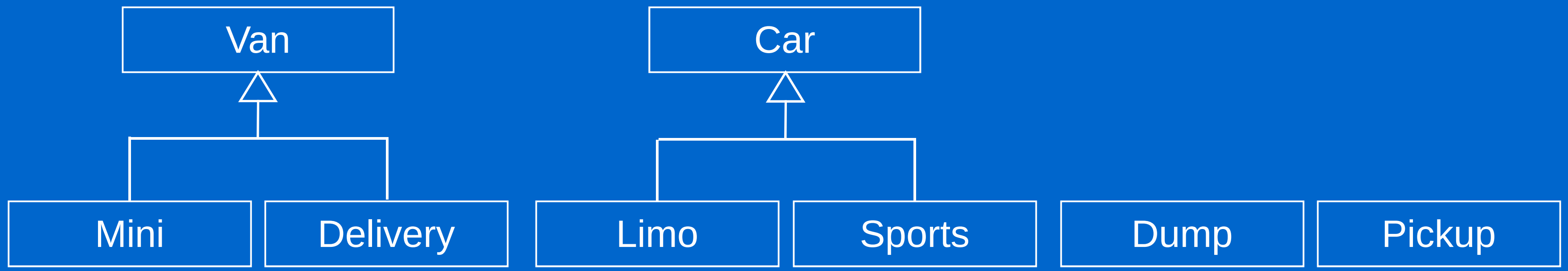
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

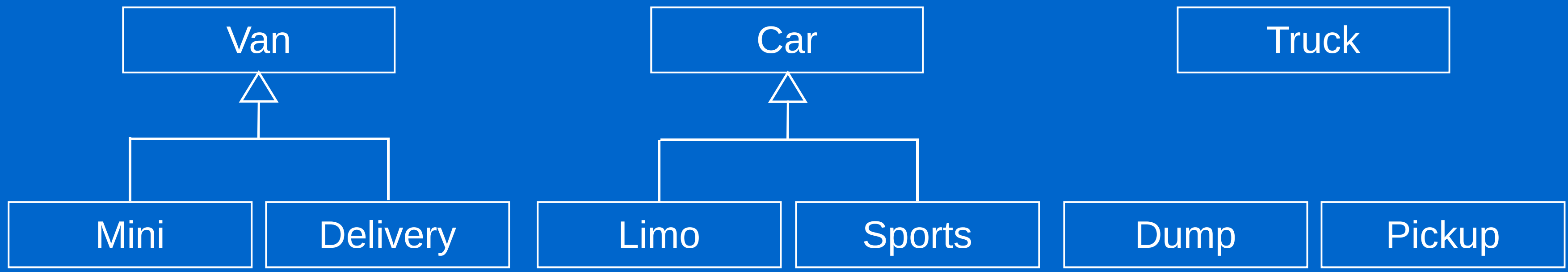
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

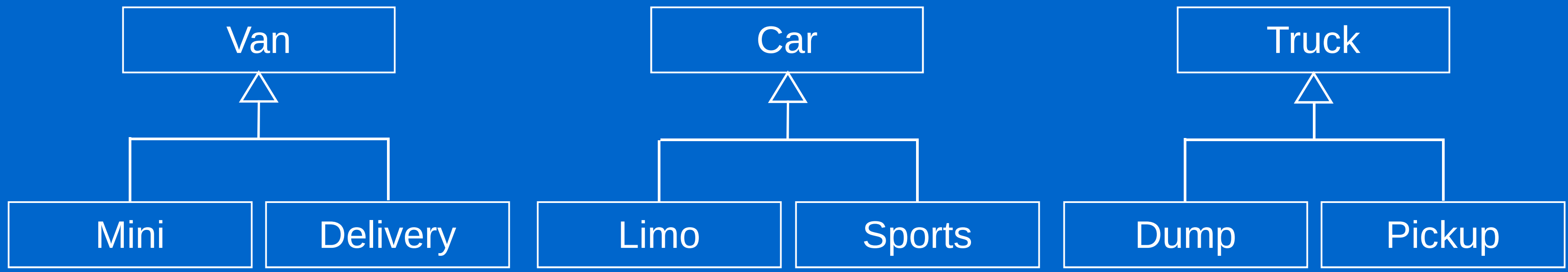
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

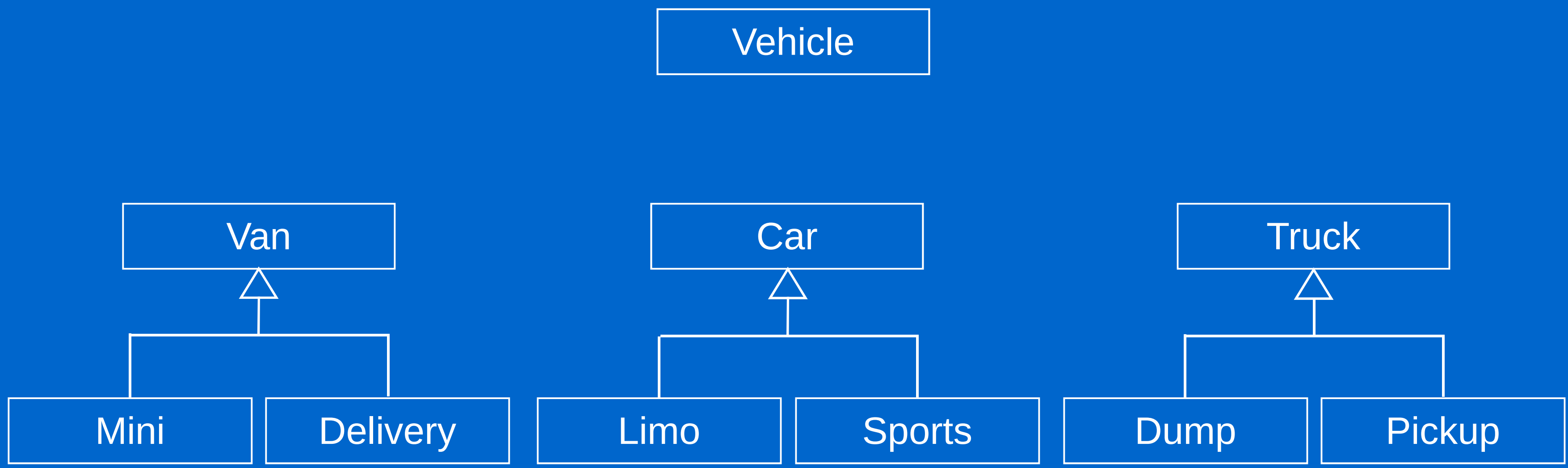
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

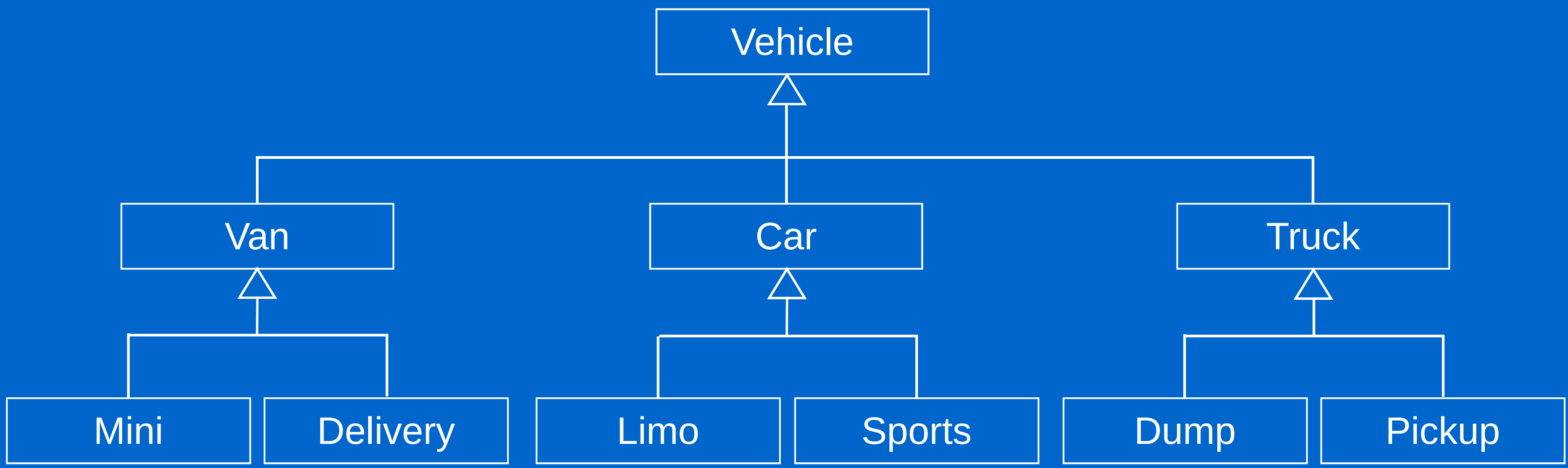
Example:



Class Hierarchy

✧ sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

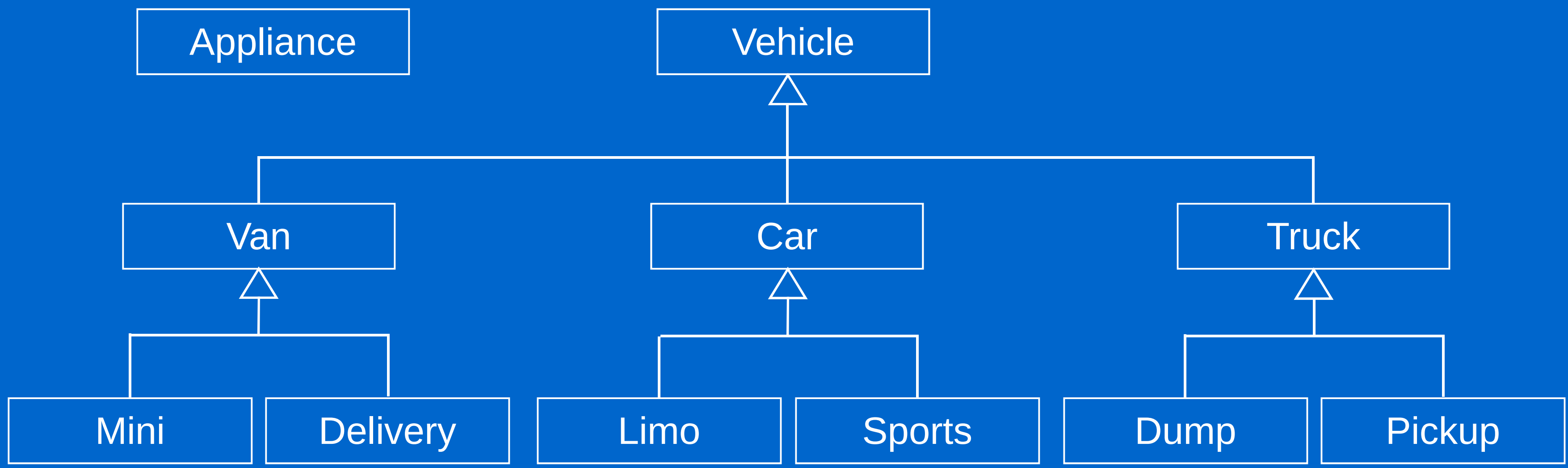
Example:



Class Hierarchy

✧ sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

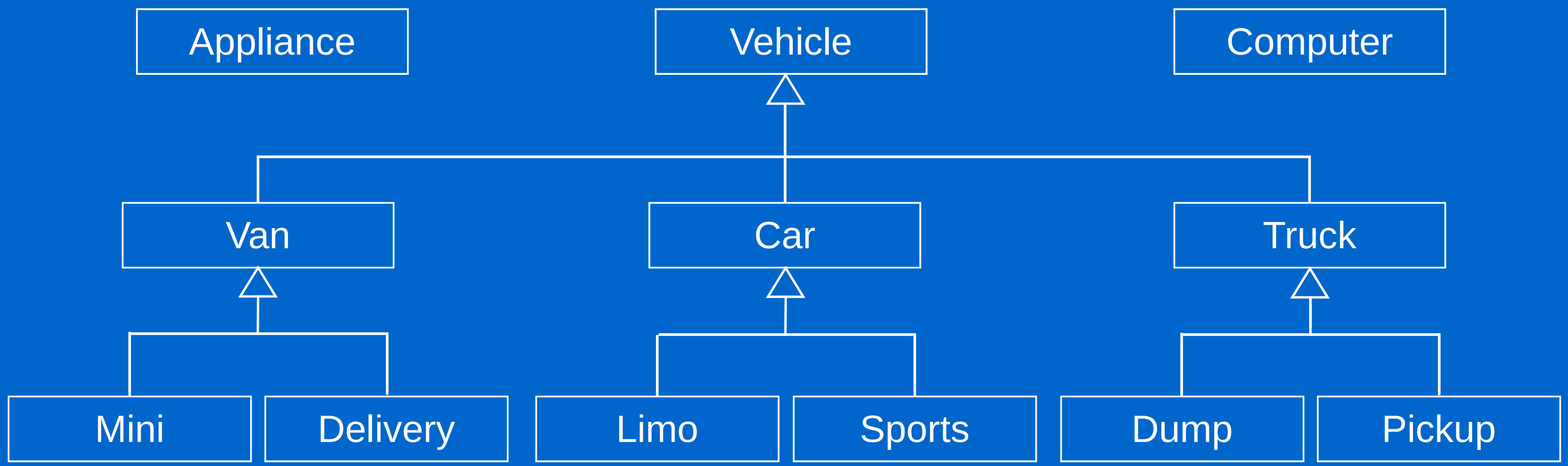
Example:



Class Hierarchy

✧ sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

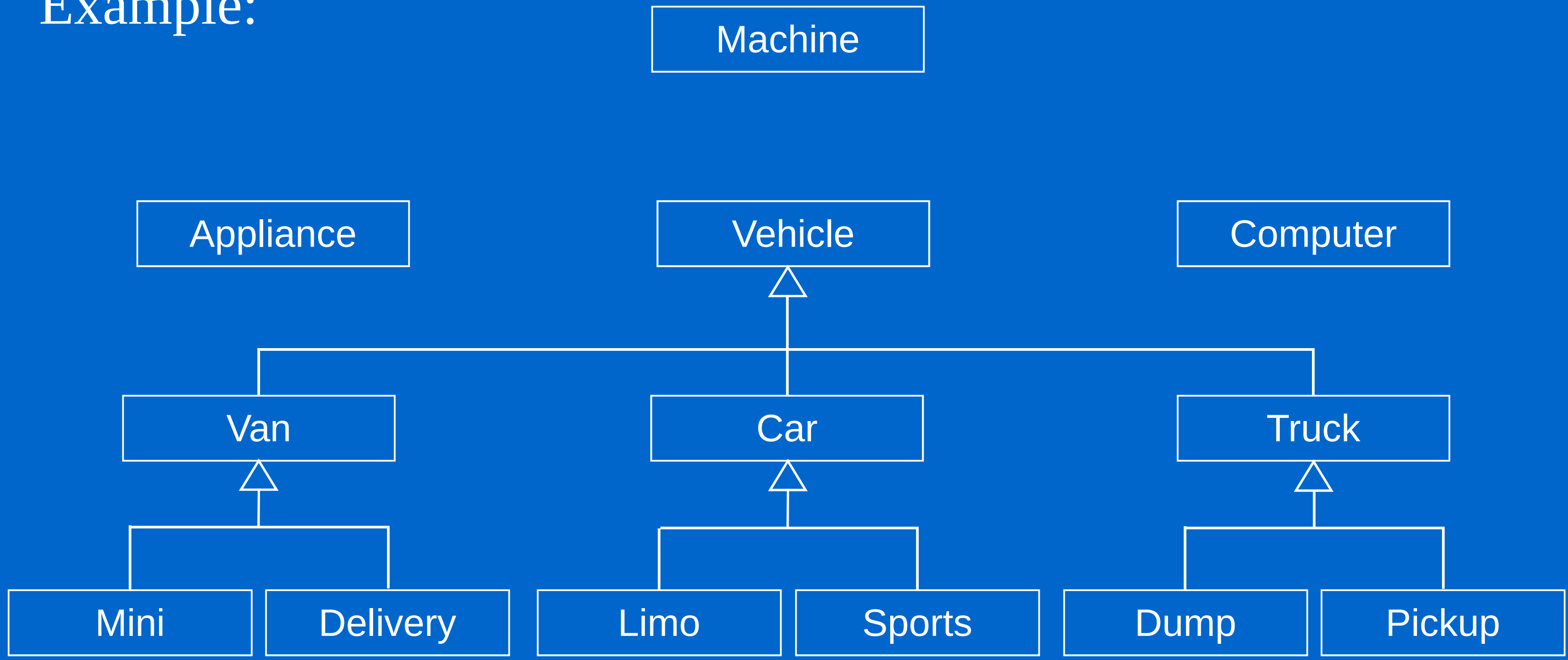
Example:



Class Hierarchy

✧ sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

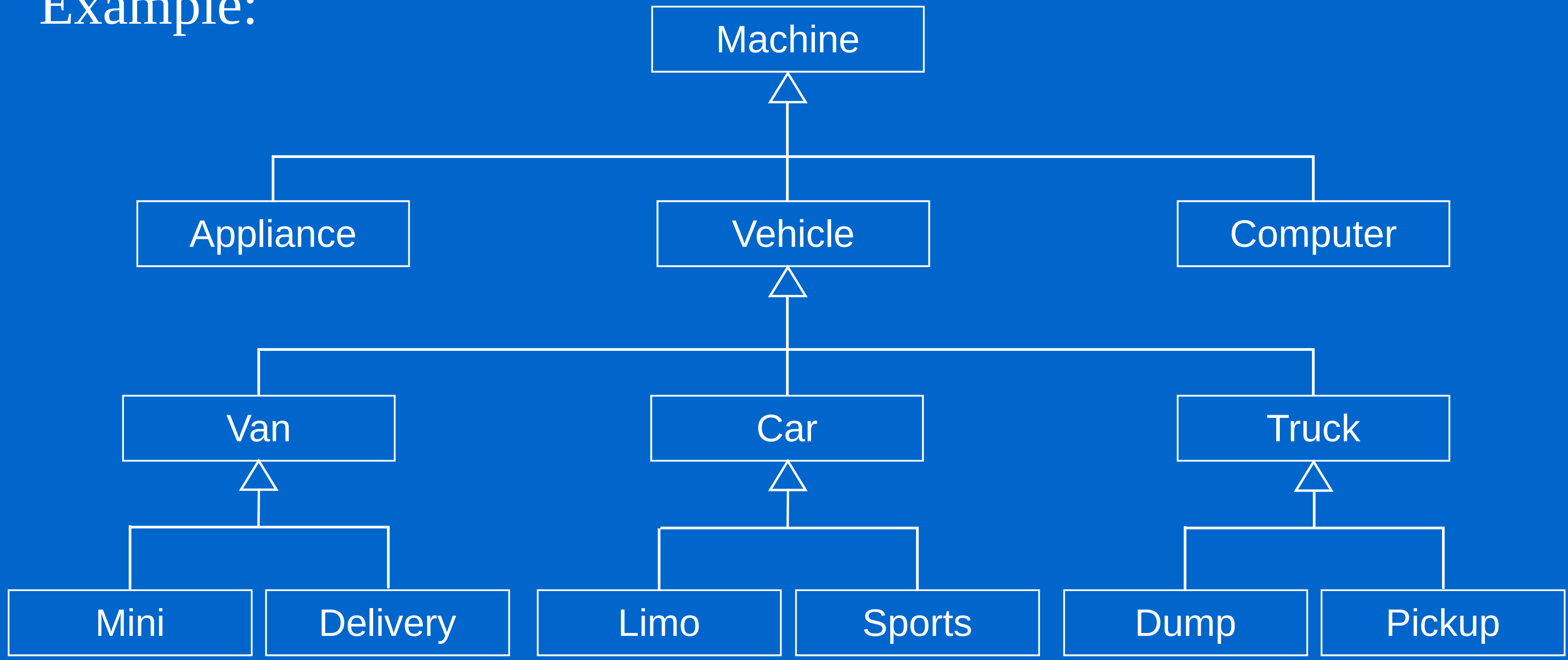
Example:



Class Hierarchy

- sub-class super-class relationship can lead to a **class hierarchy** or **inheritance hierarchy**.

Example:



Real-World Examples Of Inheritance

- Microsoft Foundation Class Version 6.0

Real-World Examples Of Inheritance

- Microsoft Foundation Class Version 6.0
  - A tree-style class hierarchy

Real-World Examples Of Inheritance

- Microsoft Foundation Class Version 6.0
  - A tree-style class hierarchy
- Java Class Library
- …

Microsoft Foundation Class Library Version 6.0

CObject

- **Application Architecture**
  - **CCmdTarget**
    - CWinThread
      - CWinApp
        - COleControlModule
          - user application
    - CDocTemplate
      - CSingleDocTemplate
      - CMultiDocTemplate
    - COleObjectFactory
      - COleTemplateServer
    - COleDataSource
    - COleDropSource
    - COleDropTarget
    - COleMessageFilter
    - CConnectionPoint
    - CDocument
      - COleDocument
        - COleLinkingDoc
          - COleServerDoc
            - CRichEditDoc
      - user documents
    - CDocItem
      - COleClientItem
        - COleDocObjectItem
        - CRichEditCntrItem
        - user client items
      - COleServerItem
        - CDocObjectServerItem
        - user server items
    - CDocObjectServer
    - **Window Support**
    - **CWnd**
      - **Frame Windows**
      - CFrameWnd
        - CMDIChildWnd
          - user MDI windows
        - CMDIFrameWnd
          - user MDI workspaces
        - CMiniFrameWnd
        - user SDI windows
        - COleIPFrameWnd
      - CSplitterWnd
      - **Dialog Boxes**
      - CDialog
        - CCommonDialog
          - CColorDialog
          - CFileDialog
          - CFindReplaceDialog
          - CFontDialog
          - COleDialog
            - COleBusyDialog
            - COleChangeIconDialog
      - **Views**
      - CView
        - CCtrlView
          - CEditView
          - CListView
          - CRichEditView
          - CTreeView
        - CScrollView
          - user scroll views
          - CFormView
      - **Controls**
      - CAnimateCtrl
      - CButton
        - CBitmapButton
      - CComboBox
        - CComboBoxEx
      - CDateTimeCtrl
      - CEdit
      - CHeaderCtrl
      - CHotKeyCtrl
- user objects
- **Exceptions**
- CException
  - CArchiveException
  - CDaoException
  - CDBException
  - CFileException
  - CInternetException
  - CMemoryException
  - CNotSupportedException
  - COleException
  - COleDispatchException
  - CResourceException
  - CUserException
- **File Services**
- CFile
  - CMemFile
    - CSharedFile
  - COleStreamFile
    - CMonikerFile
      - CAsyncMonikerFile
        - CDataPathProperty
          - CCachedDataPathProperty
  - CSocketFile
  - CStdioFile
    - CInternetFile
      - CGopherFile
      - CHttpFile
- CRecentFileList
- **Gra**
- **Cor**
- **Gra Dra**
- **Me**
- **Cor**
- **ODI Sup**
- **DA Sup**

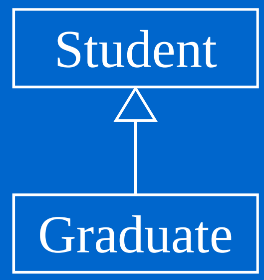
Inheritance Design

Exploring Solutions to Inheritance

- The University database program

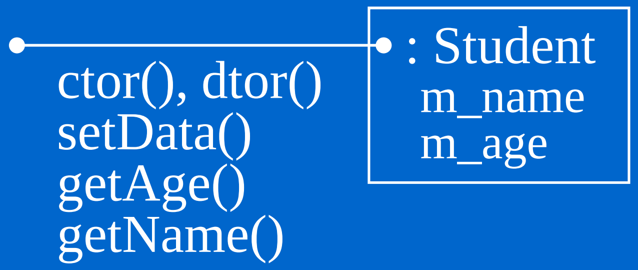
Exploring Solutions to Inheritance

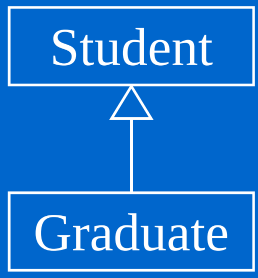
✧ The University database program



Exploring Solutions to Inheritance

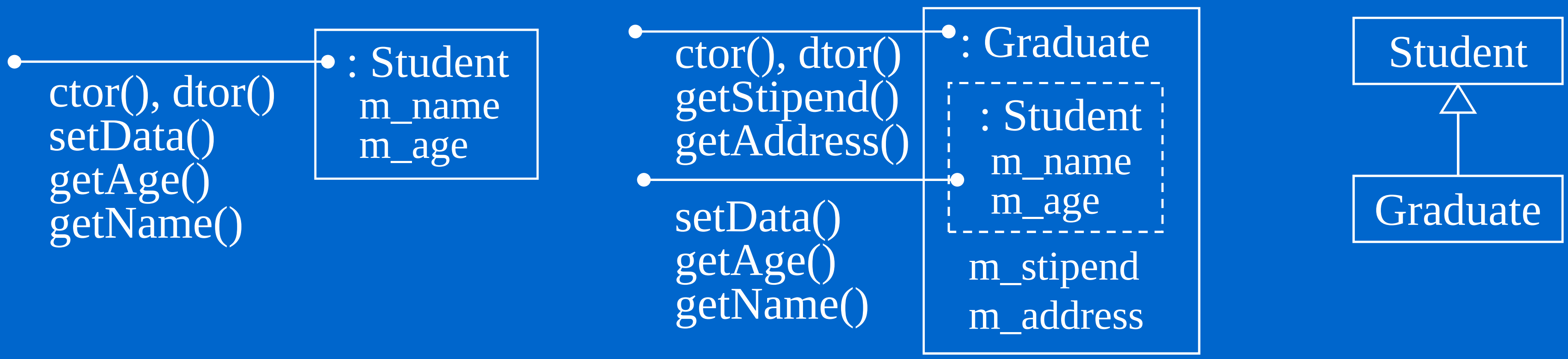
- The University database program





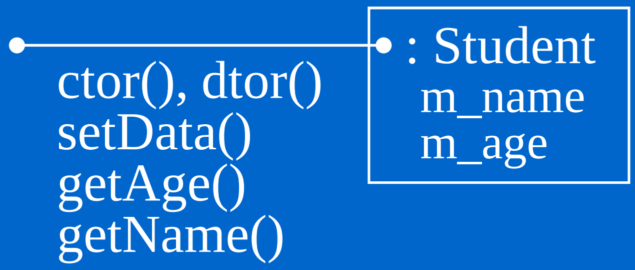
Exploring Solutions to Inheritance

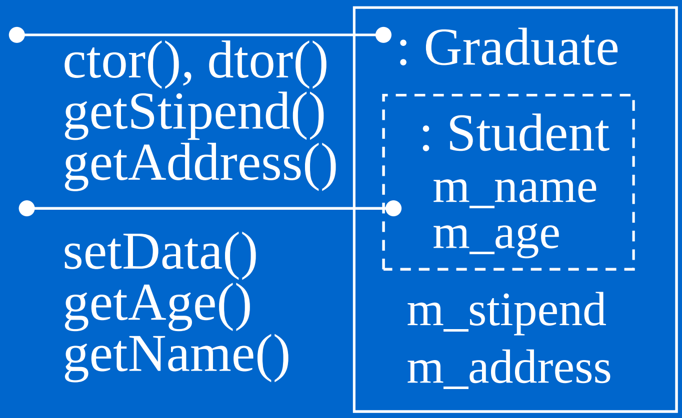
- The University database program

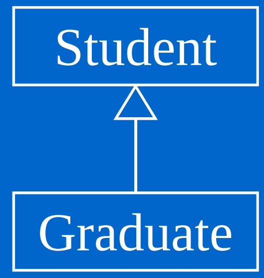


Exploring Solutions to Inheritance

- The University database program



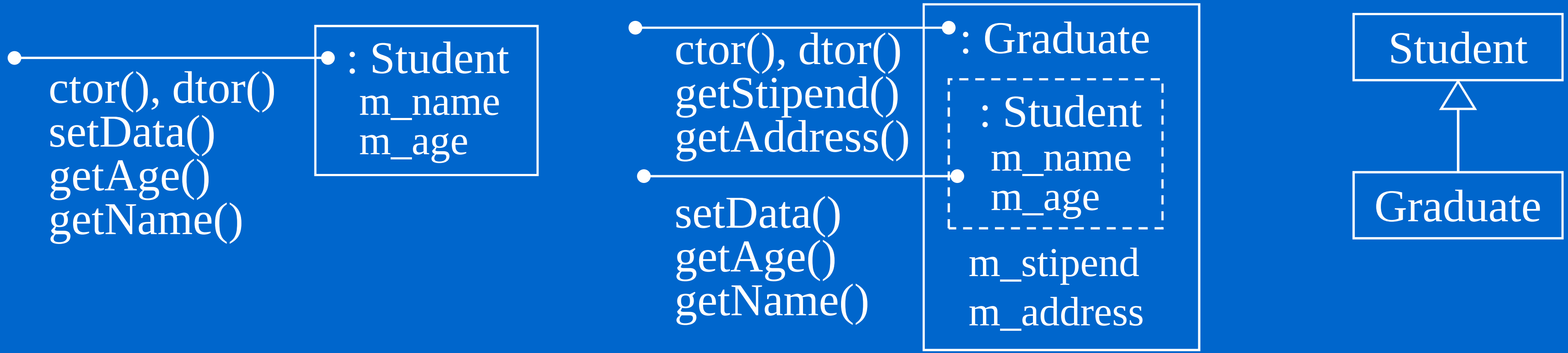




- We would like to add a class Faculty, whose attributes include

Exploring Solutions to Inheritance

- The University database program

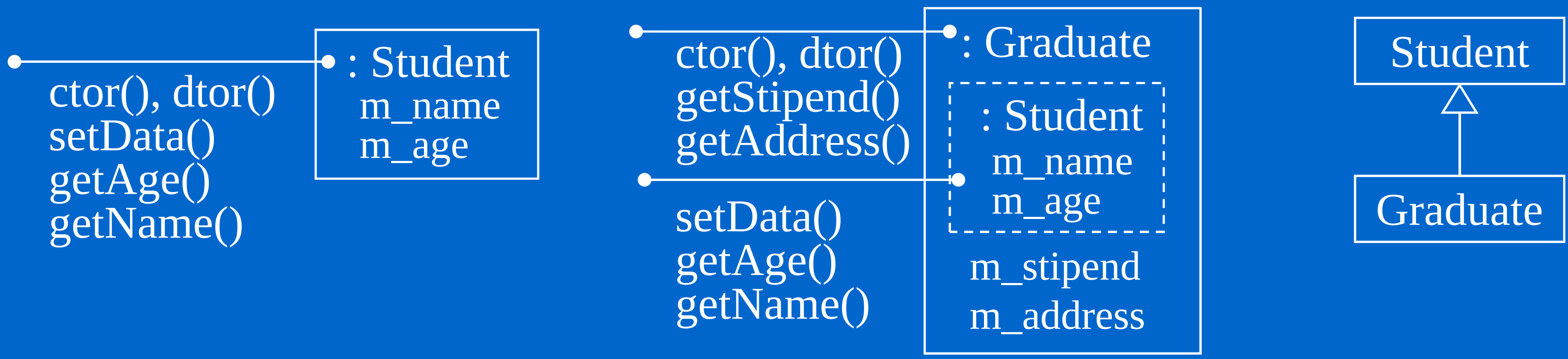


- We would like to add a class Faculty, whose attributes include

  m_name
  m_age
  m_address
  m_rank

Exploring Solutions to Inheritance

- The University database program

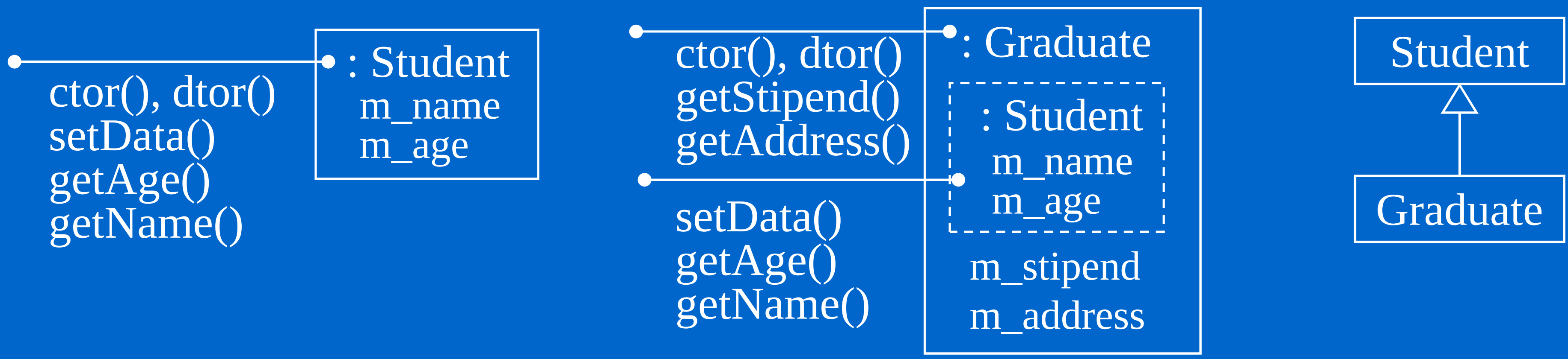


- We would like to add a class Faculty, whose attributes include

m_name
m_age
m_address **room # and building id of the office**
m_rank

Exploring Solutions to Inheritance

- The University database program



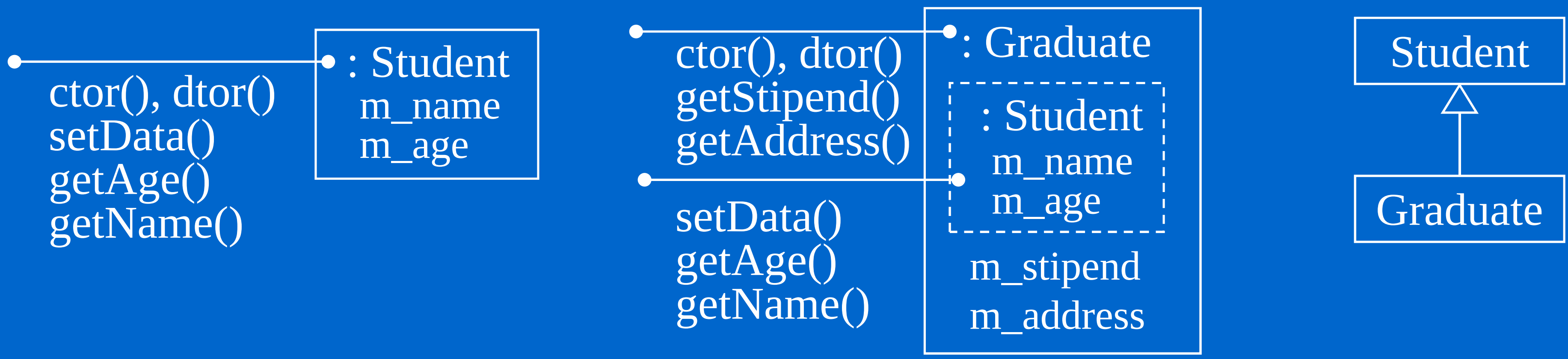
- We would like to add a class Faculty, whose attributes include

m_name
m_age
m_address **room # and building id of the office**
m_rank

Note that there is no stipend.

Exploring Solutions to Inheritance

- The University database program



- We would like to add a class Faculty, whose attributes include

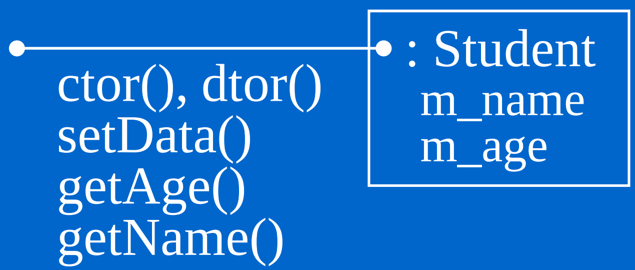
  m_name
  m_age
  m_address ← **room # and building id of the office**
  m_rank

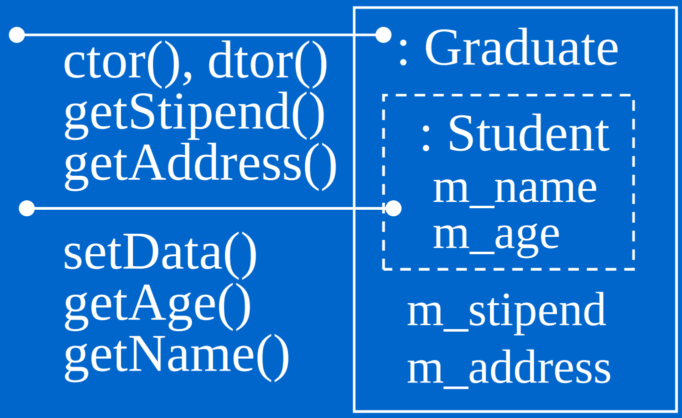
  Note that there is no stipend.

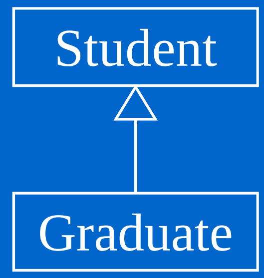
- Should Faculty be derived from Student or Graduate or none of both?

Exploring Solutions to Inheritance

- The University database program







- We would like to add a class Faculty, whose attributes include

m_name
m_age
m_address — **room # and building id of the office**
m_rank

Note that there is no stipend.

- Should Faculty be derived from Student or Graduate or none of both?

- Let us first try inheriting Faculty from Graduate since the two groups have so much data in common

Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

Exploring Solutions (cont'd)

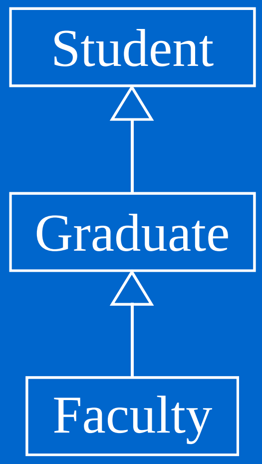
- Deriving Faculty from Graduate makes a very efficient reuse of codes

```
class Faculty: public Graduate {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```

Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

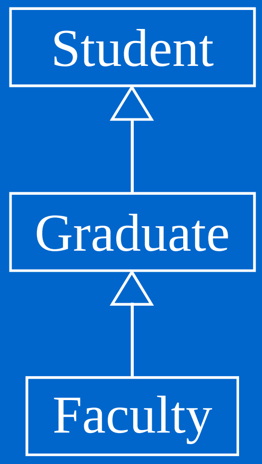
```
class Faculty: public Graduate {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```



Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

```
class Faculty: public Graduate {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```

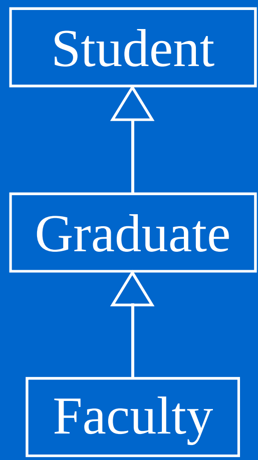


- We are forced to ignore Graduate::m_stipend in ctor

Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

```
class Faculty: public Graduate {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```



- We are forced to ignore Graduate::m_stipend in ctor

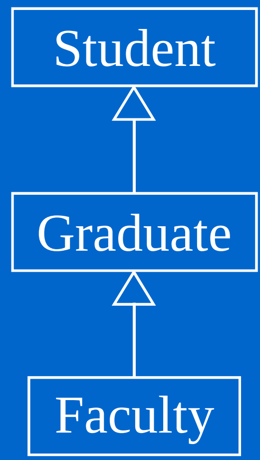
```
Faculty::Faculty(char *name, int age, char *address, char *rank)
               : Graduate(name, age, 0, address) {
   m_rank = new char[strlen(rank)+1];
   strcpy(m_rank, rank);
}
```

Zero is a dummy value for the stipend

Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

```
class Faculty: public Graduate {
public:
    Faculty(char *name, int age, char *address, char *rank);
    ~Faculty();
    const char *getRank() const;
private:
    char *m_rank;
};
```



- We are forced to ignore Graduate::m_stipend in ctor

```
Faculty::Faculty(char *name, int age, char *address, char *rank)
            : Graduate(name, age, 0, address) {
    m_rank = new char[strlen(rank)+1];
    strcpy(m_rank, rank);
}
```

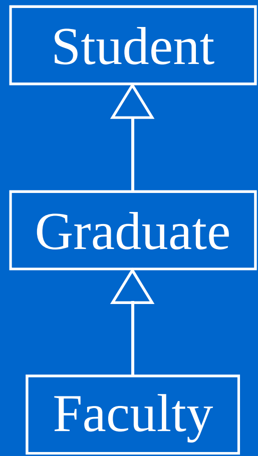
Zero is a dummy value for the stipend

- However, the client can still do this

Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

```
class Faculty: public Graduate {
public:
    Faculty(char *name, int age, char *address, char *rank);
    ~Faculty();
    const char *getRank() const;
private:
    char *m_rank;
};
```



- We are forced to ignore Graduate::m_stipend in ctor

```
Faculty::Faculty(char *name, int age, char *address, char *rank)
            : Graduate(name, age, 0, address) {
    m_rank = new char[strlen(rank)+1];
    strcpy(m_rank, rank);
}
```

Zero is a dummy value for the stipend

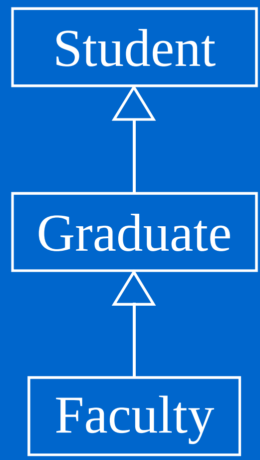
- However, the client can still do this

```
Faculty prof("Lin", 40, "#2 Bei-Ning", "Associate Professor");
cout << prof.getStipend();
```

Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

```
class Faculty: public Graduate {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```



- We are forced to ignore Graduate::m_stipend in ctor

```
Faculty::Faculty(char *name, int age, char *address, char *rank)
            : Graduate(name, age, 0, address) {
   m_rank = new char[strlen(rank)+1];
   strcpy(m_rank, rank);
}
```

Zero is a dummy value for the stipend

- However, the client can still do this

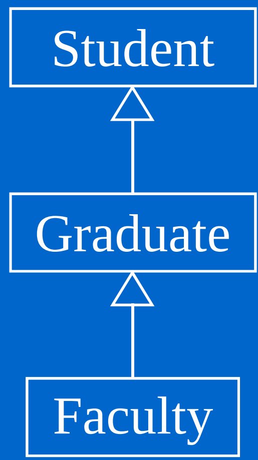
```
Faculty prof("Lin", 40, "#2 Bei-Ning", "Associate Professor");
cout << prof.getStipend();
```

You can spare a data member but cannot turn off an interface of the base class.

Exploring Solutions (cont'd)

- Deriving Faculty from Graduate makes a very efficient reuse of codes

```
class Faculty: public Graduate {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```



- We are forced to ignore Graduate::m_stipend in ctor

```
Faculty::Faculty(char *name, int age, char *address, char *rank)
              : Graduate(name, age, 0, address) {
   m_rank = new char[strlen(rank)+1];
   strcpy(m_rank, rank);
}
```

Zero is a dummy value for the stipend

- However, the client can still do this

```
Faculty prof("Lin", 40, "#2 Bei-Ning", "Associate Professor");
cout << prof.getStipend();
```

This is NOT a good solution!

You can spare a data member but cannot turn off an interface of the base class.

Another Possible Solution

- How about deriving Faculty from Student because Faculty requires all of the data from Student

Another Possible Solution

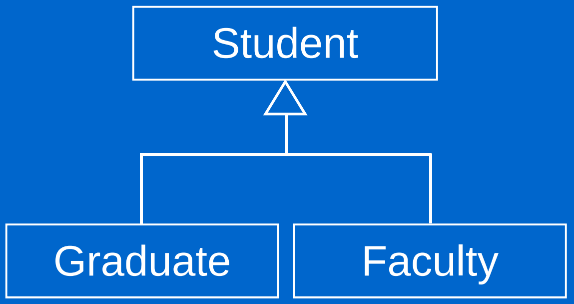
- How about deriving Faculty from Student because Faculty requires all of the data from Student

```
class Faculty: public Student {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
   const char *getAddress() const;
private:
   char *m_address;
   char *m_rank;
};
```

Another Possible Solution

- How about deriving Faculty from Student because Faculty requires all of the data from Student

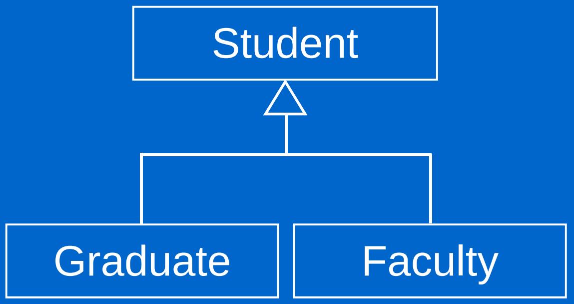
```
class Faculty: public Student {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
   const char *getAddress() const;
private:
   char *m_address;
   char *m_rank;
};
```



Another Possible Solution

- How about deriving Faculty from Student because Faculty requires all of the data from Student

```
class Faculty: public Student {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
   const char *getAddress() const;
private:
   char *m_address;
   char *m_rank;
};
```

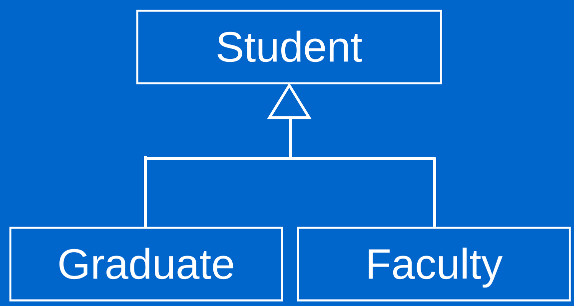


- What is the problem now?

Another Possible Solution

- How about deriving Faculty from Student because Faculty requires all of the data from Student

```
class Faculty: public Student {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
   const char *getAddress() const;
private:
   char *m_address;
   char *m_rank;
};
```

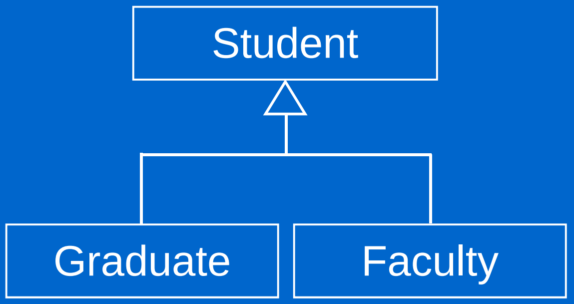


- What is the problem now?
  - Faculty duplicates some codes in Graduate: m_address related

Another Possible Solution

- How about deriving Faculty from Student because Faculty requires all of the data from Student

```
class Faculty: public Student {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
   const char *getAddress() const;
private:
   char *m_address;
   char *m_rank;
};
```

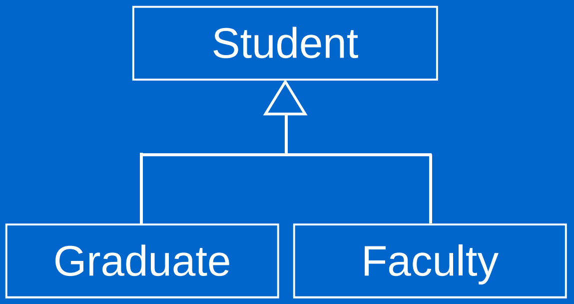


- What is the problem now?
  - Faculty duplicates some codes in Graduate: m_address related
  - What happens if Student adds a field for "undergraduate advisor"?

Another Possible Solution

- How about deriving Faculty from Student because Faculty requires all of the data from Student

```
class Faculty: public Student {
public:
    Faculty(char *name, int age, char *address, char *rank);
    ~Faculty();
    const char *getRank() const;
    const char *getAddress() const;
private:
    char *m_address;
    char *m_rank;
};
```

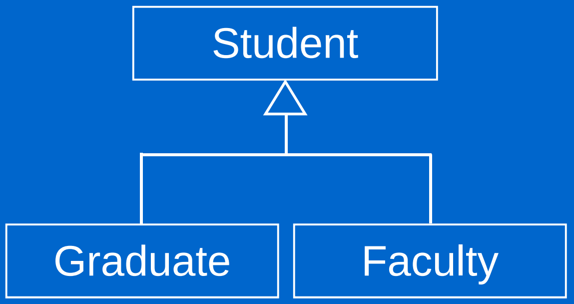


- What is the problem now?
    - Faculty duplicates some codes in Graduate: m_address related
    - What happens if Student adds a field for "undergraduate advisor"?
    - The problem is that Faculty is intrinsically **not** a Student.

Another Possible Solution

- How about deriving Faculty from Student because Faculty requires all of the data from Student

```
class Faculty: public Student {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
   const char *getAddress() const;
private:
   char *m_address;
   char *m_rank;
};
```



- What is the problem now?
  - Faculty duplicates some codes in Graduate: m_address related
  - What happens if Student adds a field for "undergraduate advisor"?
  - The problem is that Faculty is intrinsically **not** a Student.

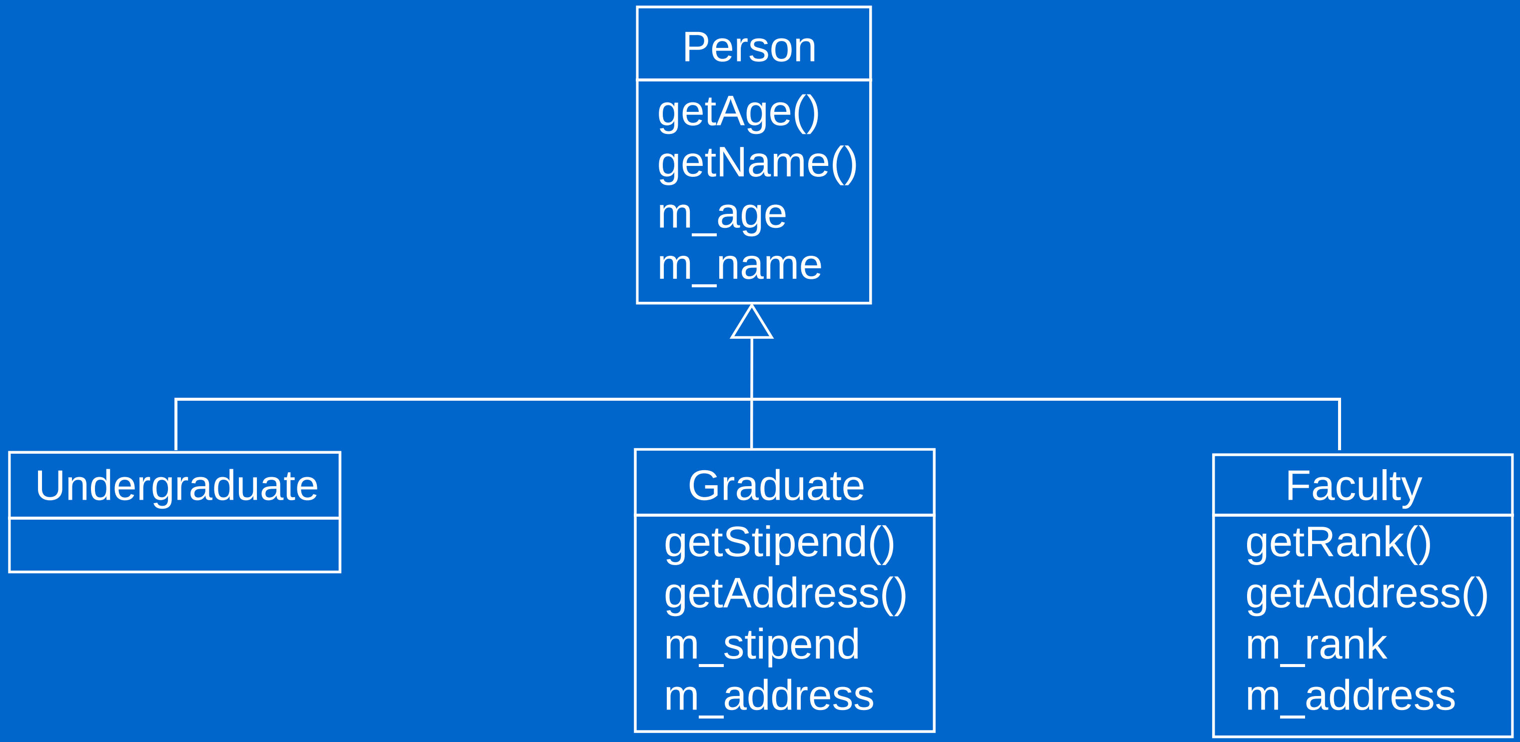
"Inheritance SHOULD **NOT** be designed based on solely implementation considerations – eg. code reuse."

A Better Design

- Create a **Person** class and put everything common to all people in that class, all other classes are derived from this class.

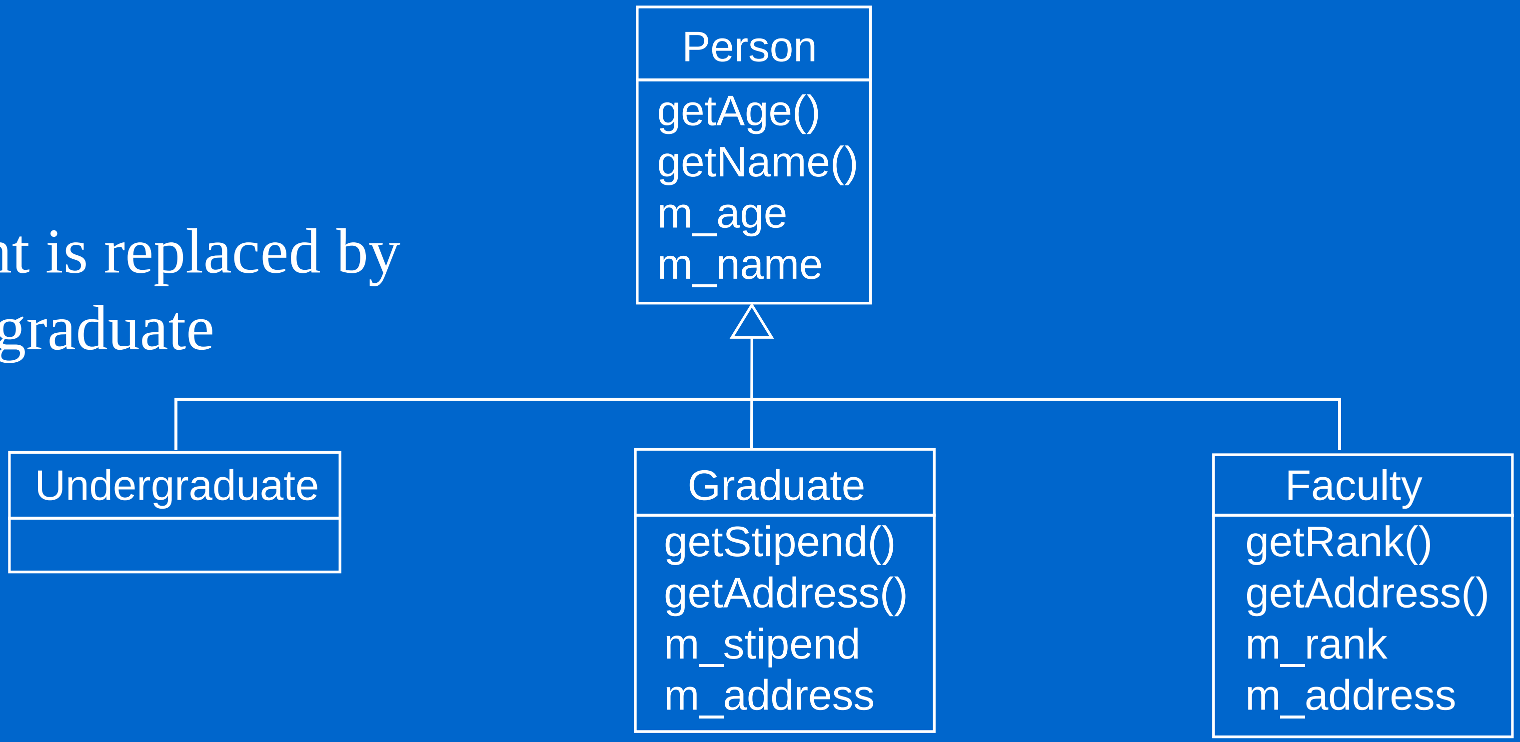
A Better Design

- Create a **Person** class and put everything common to all people in that class, all other classes are derived from this class.



A Better Design

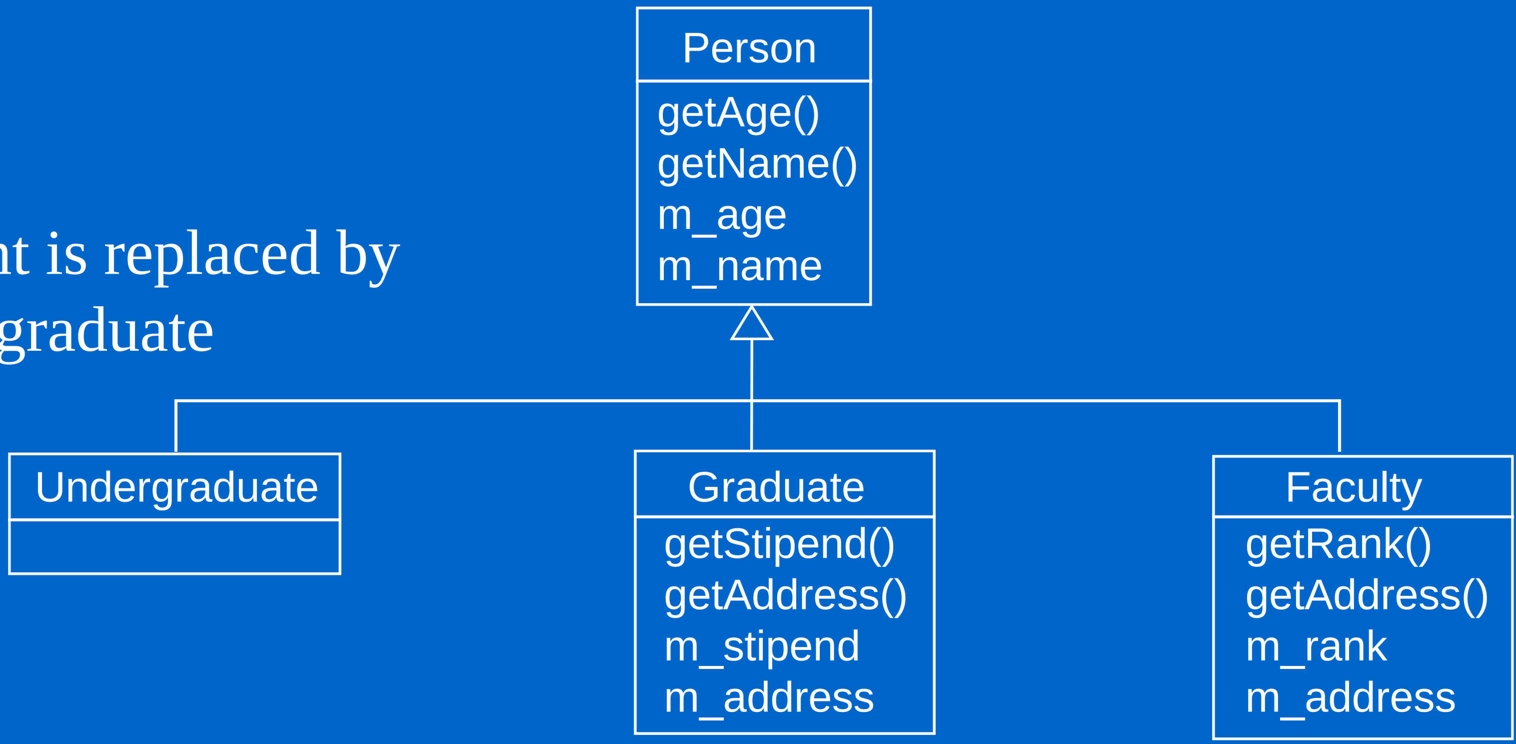
- Create a **Person** class and put everything common to all people in that class, all other classes are derived from this class.
- Student is replaced by Undergraduate



A Better Design

- Create a **Person** class and put everything common to all people in that class, all other classes are derived from this class.

- Student is replaced by Undergraduate

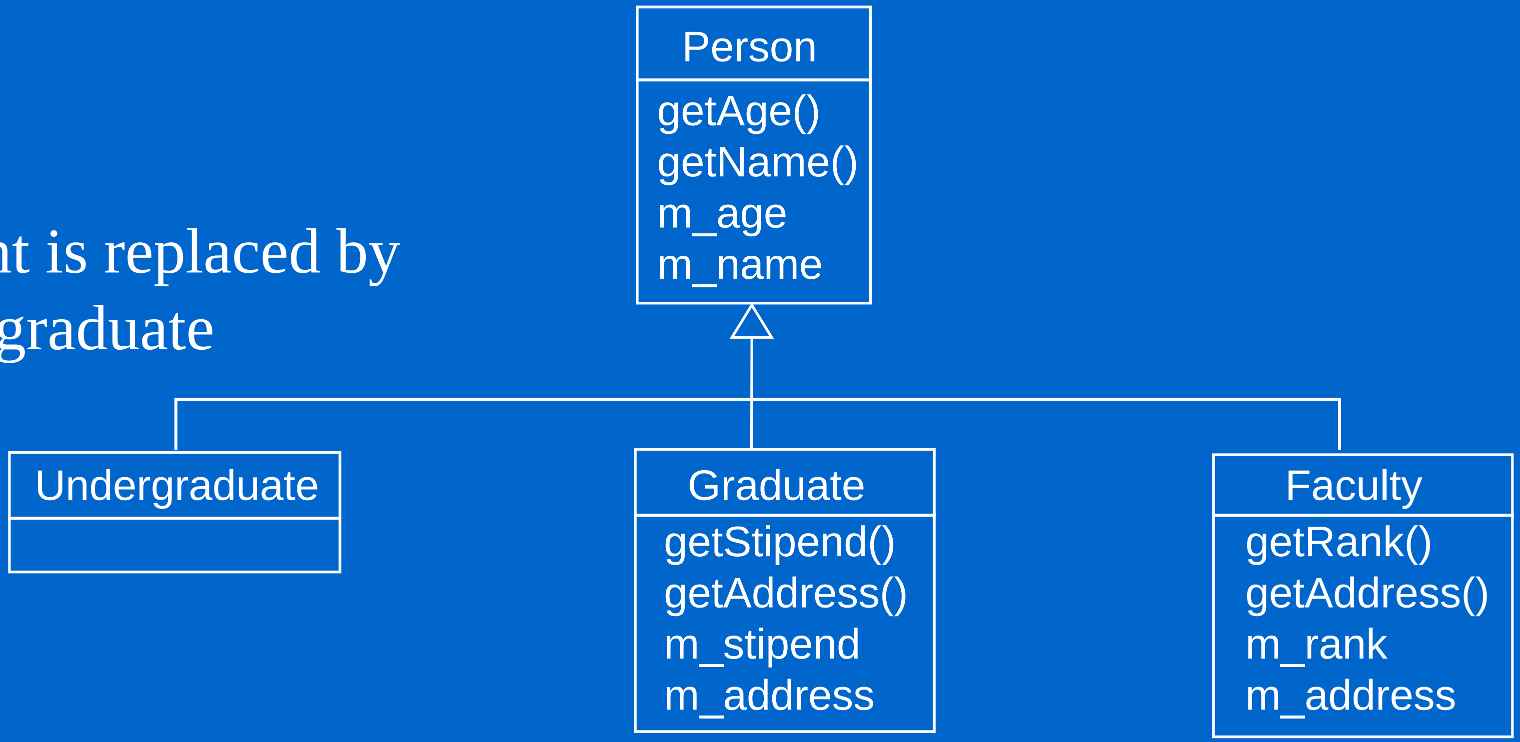


- Should we eliminate UnderGraduate and use only Person in its place?

A Better Design

- Create a **Person** class and put everything common to all people in that class, all other classes are derived from this class.

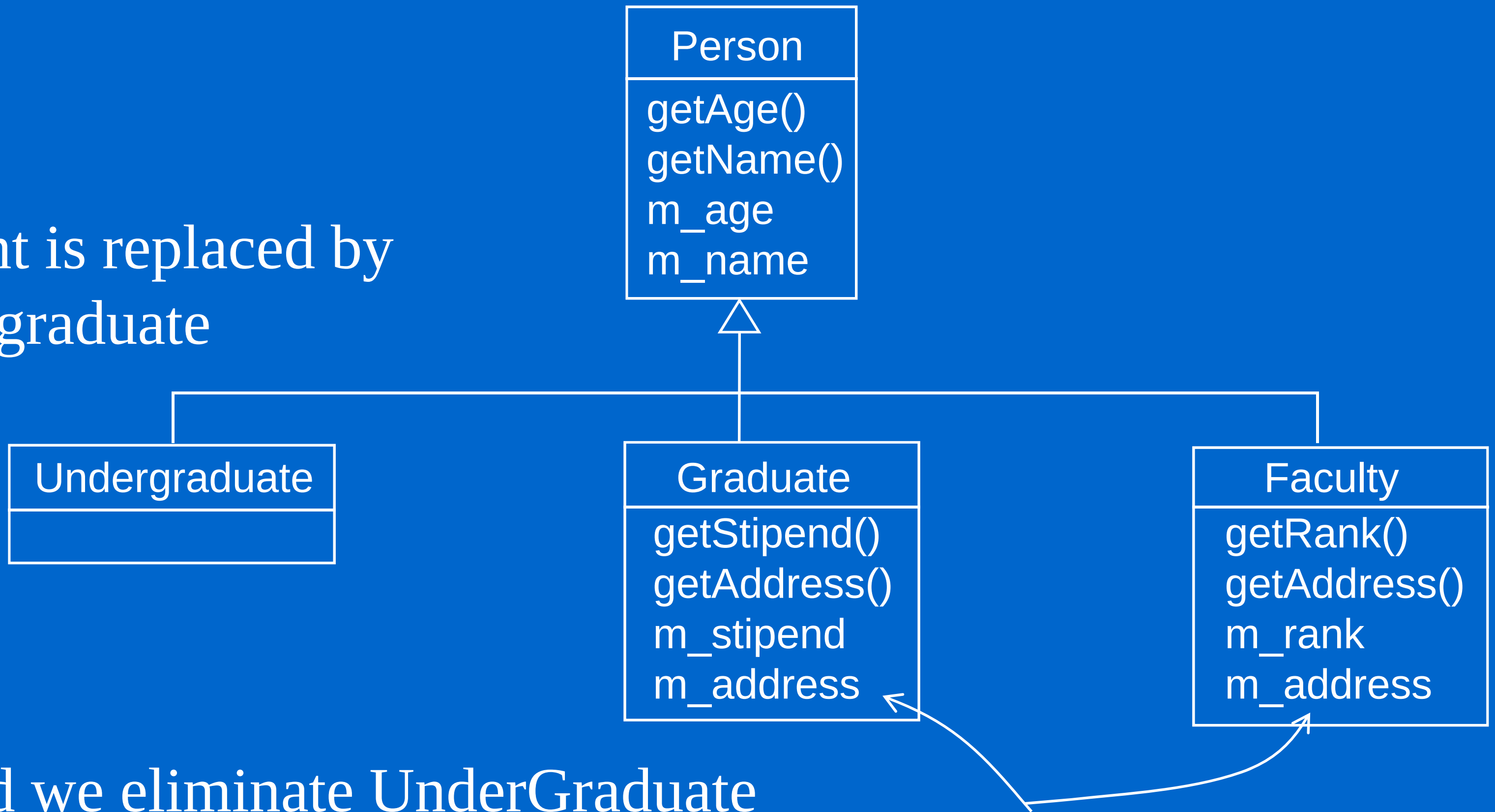
- Student is replaced by Undergraduate



- Should we eliminate UnderGraduate and use only Person in its place?
- Should Graduate be derived from Undergraduate?

A Better Design

- Create a **Person** class and put everything common to all people in that class, all other classes are derived from this class.
- Student is replaced by Undergraduate



Is there any redundancy?

- Should we eliminate UnderGraduate and use only Person in its place?
- Should Graduate be derived from Undergraduate?

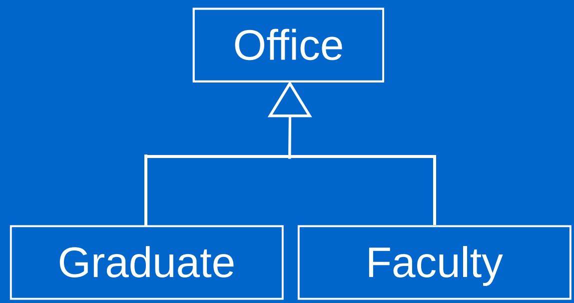
Adding an Office Class

Adding an Office Class

- Codes related to address could be merged into a single copy. How about encapsulating all data related to the address in the **Office** class?

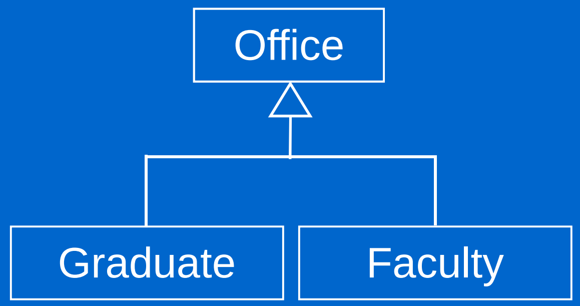
Adding an Office Class

- Codes related to address could be merged into a single copy. How about encapsulating all data related to the address in the **Office** class?
- Anyone who needs an office can then **inherit** from Office.



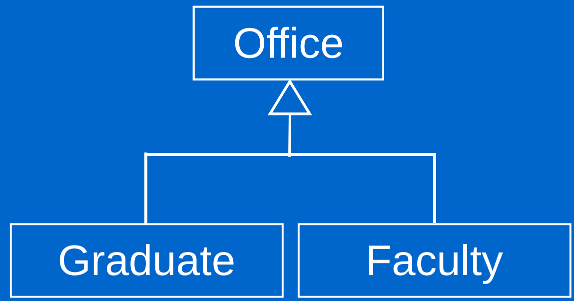
Adding an Office Class

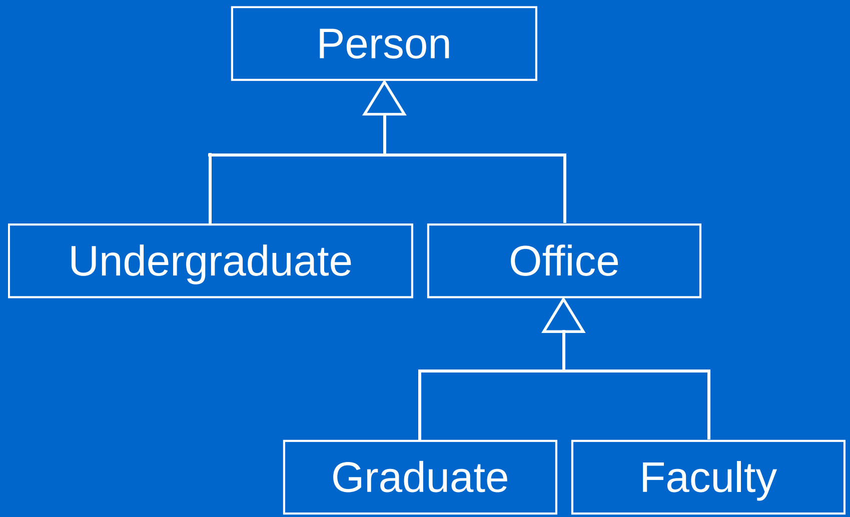
- Codes related to address could be merged into a single copy. How about encapsulating all data related to the address in the **Office** class?
- Anyone who needs an office can then **inherit** from Office.
- But Graduate and Faculty still need to inherit name and age categories so this design forces us to this inheritance



Adding an Office Class

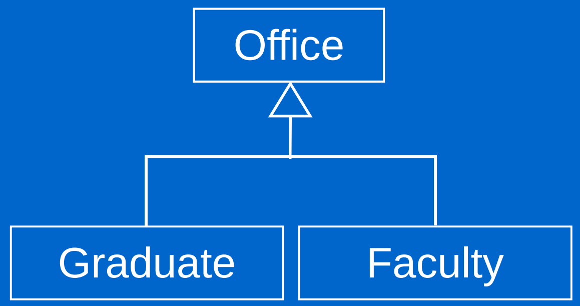
- Codes related to address could be merged into a single copy. How about encapsulating all data related to the address in the **Office** class?
- Anyone who needs an office can then **inherit** from Office.
- But Graduate and Faculty still need to inherit name and age categories so this design forces us to this inheritance

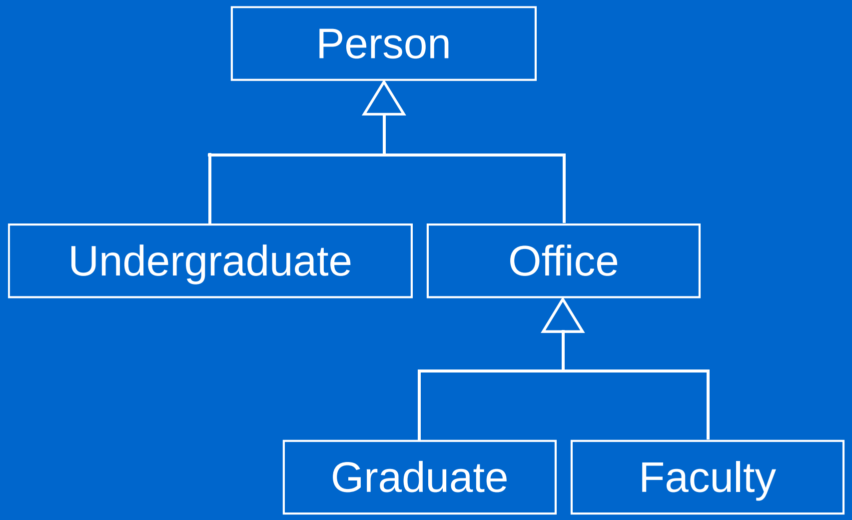




Adding an Office Class

- Codes related to address could be merged into a single copy. How about encapsulating all data related to the address in the **Office** class?
- Anyone who needs an office can then **inherit** from Office.
- But Graduate and Faculty still need to inherit name and age categories so this design forces us to this inheritance

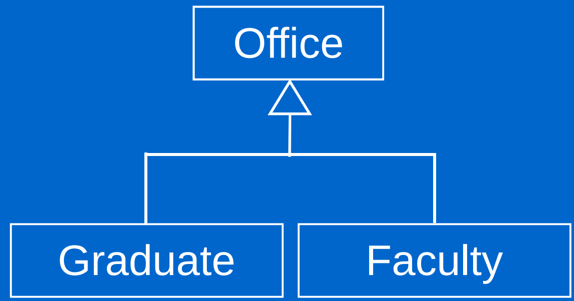


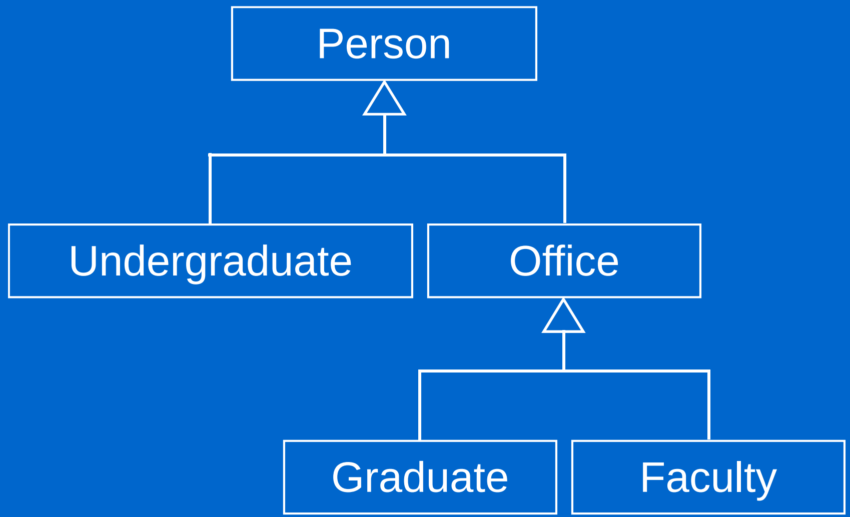


Bad design!! Problematic!!?

Adding an Office Class

- Codes related to address could be merged into a single copy. How about encapsulating all data related to the address in the **Office** class?
- Anyone who needs an office can then **inherit** from Office.
- But Graduate and Faculty still need to inherit name and age categories so this design forces us to this inheritance



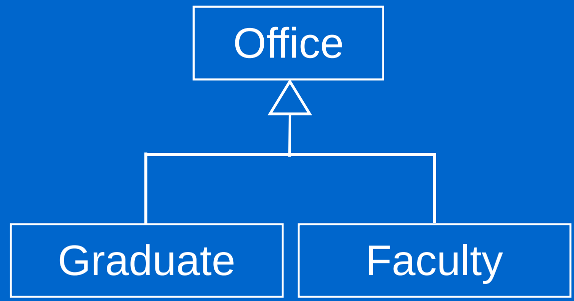


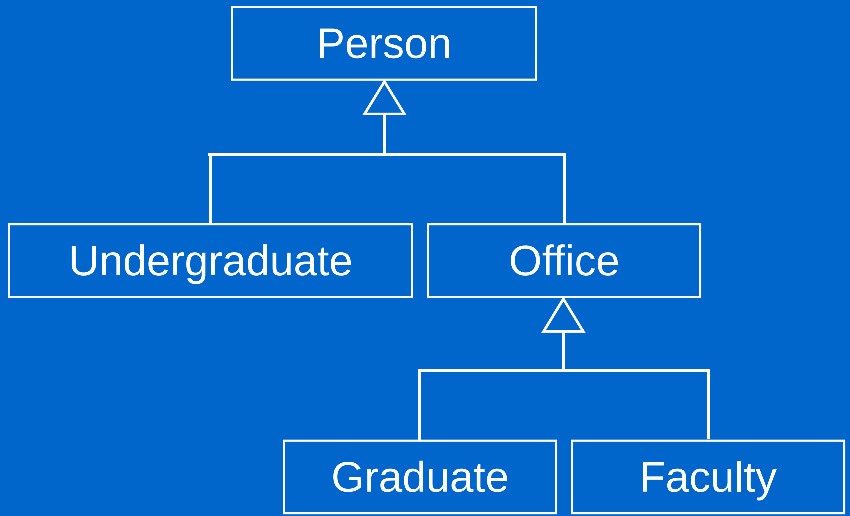
Bad design!! Problematic!!?

What's wrong?

Adding an Office Class

- Codes related to address could be merged into a single copy. How about encapsulating all data related to the address in the **Office** class?
- Anyone who needs an office can then **inherit** from Office.
- But Graduate and Faculty still need to inherit name and age categories so this design forces us to this inheritance





Bad design!! Problematic!!?

What's wrong?

- **If the Office has a clean() method, The Faculty automatically has a clean() method. What does it mean?**
- **What if a faculty has two offices?**

Code for Office Solution

```
class Office: public Person {
public:
   Office(char *name, int age, char address);
   ~Office()
   const char *getAddress() const;
private:
   char *m_address;
};
```

Code for Office Solution

```
class Office: public Person {
public:
   Office(char *name, int age, char address);
   ~Office()
   const char *getAddress() const;
private:
   char *m_address;
};
   --------------------------------------------------------------
class Graduate: public Office {
public:
   Graduate(char *name, int age, int stipend, char *address);
   int getStipend() const;
private:
   int m_stipend;
};
```

Code for Office Solution

```
class Office: public Person {
public:
   Office(char *name, int age, char address);
   ~Office()
   const char *getAddress() const;
private:
   char *m_address;
};
   ----------------------------------------------------------
class Graduate: public Office {
public:
   Graduate(char *name, int age, int stipend, char *address);
   int getStipend() const;
private:
   int m_stipend;
};
   ----------------------------------------------------------
class Faculty: public Office {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```

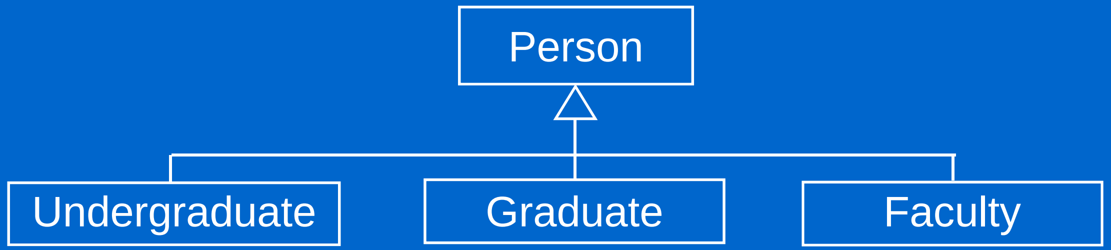
Code for Office Solution

```
class Office: public Person {
public:
   Office(char *name, int age, char address);
   ~Office()
   const char *getAddress() const;
private:
   char *m_address;
};
   -----------------------------------------------------------
class Graduate: public Office {
public:
   Graduate(char *name, int age, int stipend, char *address);
   int getStipend() const;
private:
   int m_stipend;
};
   -----------------------------------------------------------
class Faculty: public Office {
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char *getRank() const;
private:
   char *m_rank;
};
```

Poor design!!
Problematic!!?

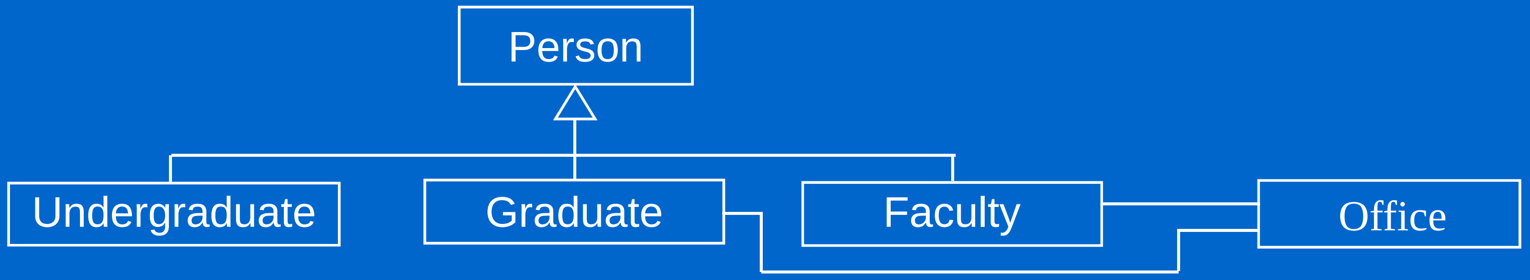
Final Solution

- Back to our original inheritance design (good design)



Final Solution

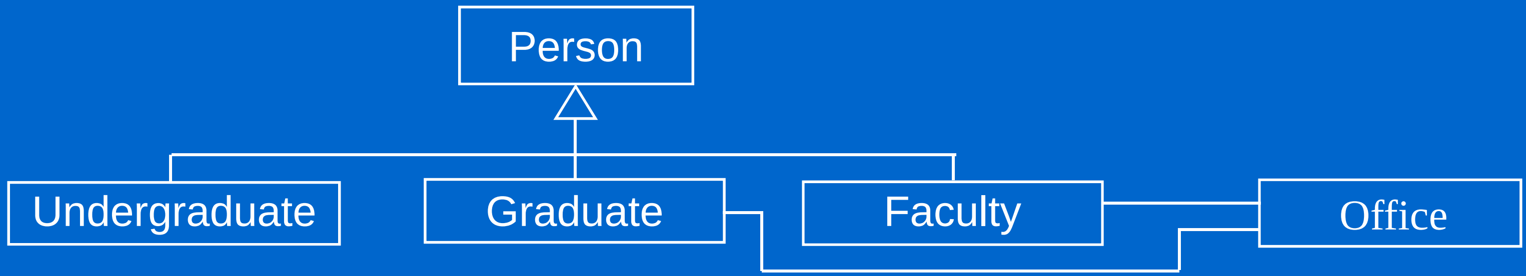
- Back to our original inheritance design (good design)



- Instead of having Graduate and Faculty inherit from Office, we store an Office object within each classes

Final Solution

- Back to our original inheritance design (good design)



- Instead of having Graduate and Faculty inherit from Office, we store an Office object within each classes
- The office class exists separately, without involving any inheritance
- Codes:

```
class Office {
public:
   Office(char *address);
   ~Office();
   const char *getAddress() const;
private:
   char *m_address;
};
```

Final Solution (cont'd)

```
class Graduate: public Person {
public:
   Graduate(char *name, int age, int stipend, char *address);
   int getStipend() const;
   const char* getAddress() const;
private:
   int m_stipend;
   Office m_office;
};
```

Final Solution (cont'd)

```
class Graduate: public Person {
public:
   Graduate(char *name, int age, int stipend, char *address);
   int getStipend() const;
   const char* getAddress() const;
private:
   int m_stipend;
   Office m_office;
};
```

```
class Faculty: public Person
{
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char* getAddress() const;
   const char *getRank() const;
private:
   char *m_rank;
   Office m_office;
};
```

Final Solution (cont'd)

```
class Graduate: public Person {
public:
   Graduate(char *name, int age, int stipend, char *address);
   int getStipend() const;
   const char* getAddress() const;
private:
   int m_stipend;
   Office m_office;
};
```

```
const char* Graduate::
        getAddress() const {
  return m_office.getAddress();
}
```

```
class Faculty: public Person
{
public:
   Faculty(char *name, int age, char *address, char *rank);
   ~Faculty();
   const char* getAddress() const;
   const char *getRank() const;
private:
   char *m_rank;
   Office m_office;
};
```

Final Solution (cont'd)

```
class Graduate: public Person {
public:
    Graduate(char *name, int age, int stipend, char *address);
    int getStipend() const;
    const char* getAddress() const;
private:
    int m_stipend;
    Office m_office;
};
```

delegation

```
const char* Graduate::
        getAddress() const {
  return m_office.getAddress();
}
```

```
class Faculty: public Person
{
public:
    Faculty(char *name, int age, char *address, char *rank);
    ~Faculty();
    const char* getAddress() const;
    const char *getRank() const;
private:
    char *m_rank;
    Office m_office;
};
```

Final Solution (cont'd)

```
class Graduate: public Person {
public:
    Graduate(char *name, int age, int stipend, char *address);
    int getStipend() const;
    const char* getAddress() const;
private:
    int m_stipend;
    Office m_office;
};
```

delegation

```
const char* Graduate::
        getAddress() const {
  return m_office.getAddress();
}
```

```
class Faculty: public Person
{
public:
    Faculty(char *name, int age, char *address, char *rank);
    ~Faculty();
    const char* getAddress() const;
    const char *getRank() const;
private:
    char *m_rank;
    Office m_office;
};
```

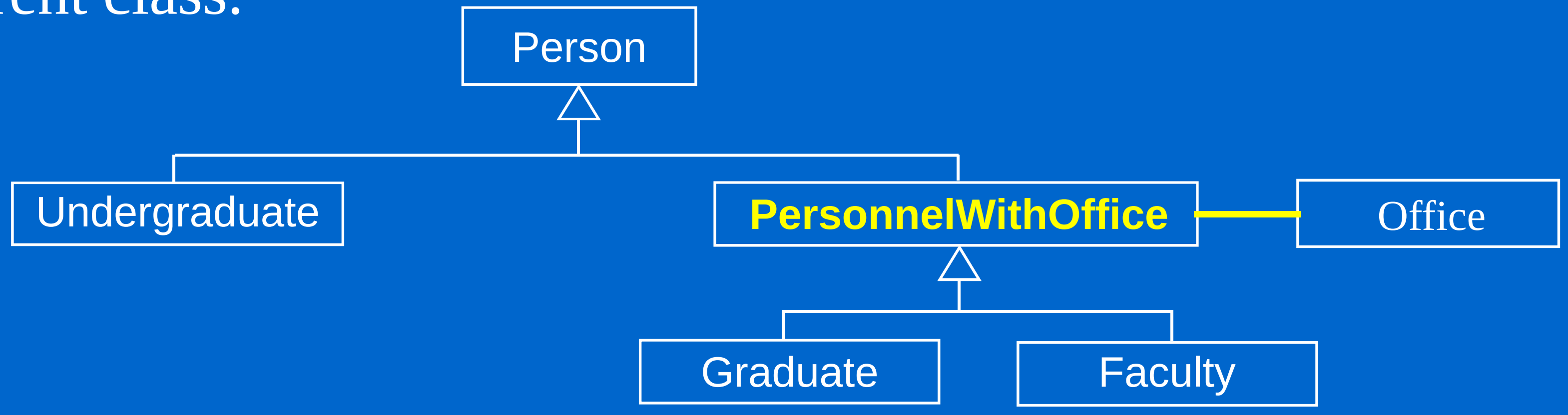
- Note: the data part m_office in Graduate and Faculty is replicated. However, the code to handle address is reduced to a single copy, i.e. Office::getAddress(). If we want to maintain a single object for the same office, we can use pointer or reference to implement m_office.

Further Abstraction

- When the relationships between Graduate or Faculty objects and other objects are common, we can model their relationships within a parent class.

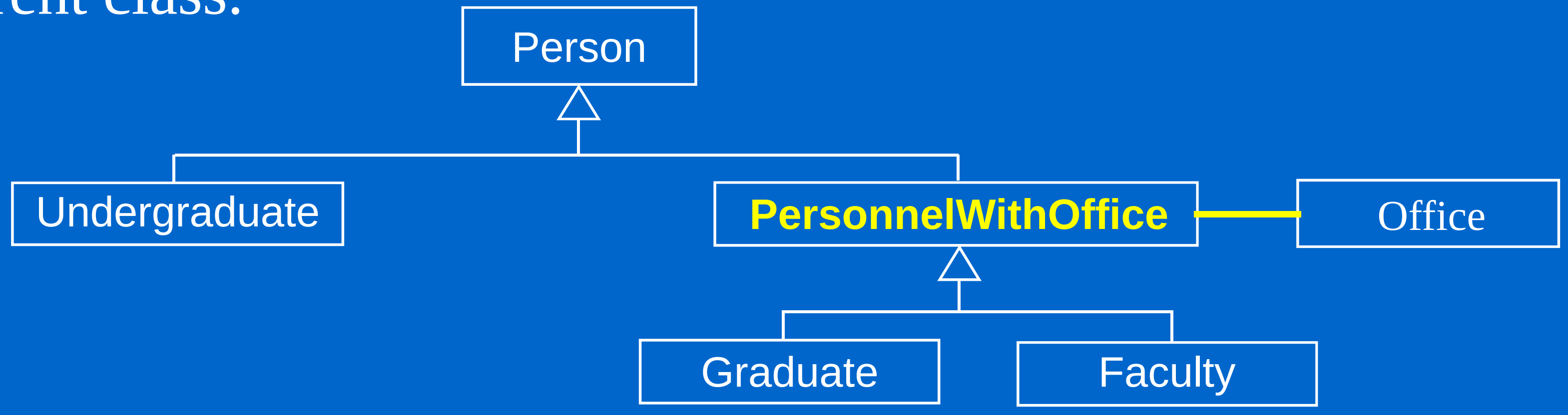
Further Abstraction

- When the relationships between Graduate or Faculty objects and other objects are common, we can model their relationships within a parent class.



Further Abstraction

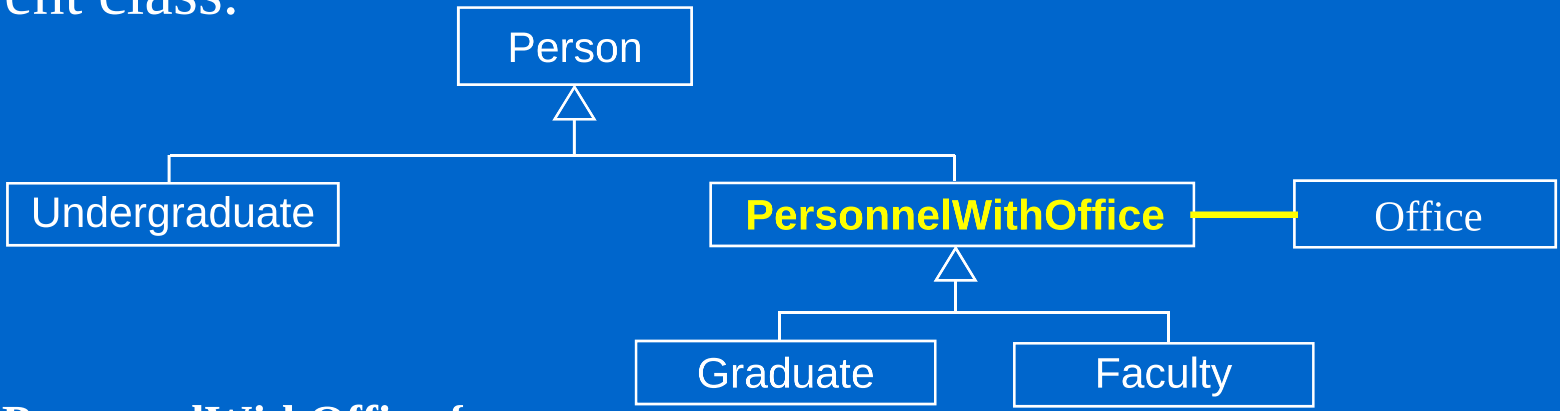
- When the relationships between Graduate or Faculty objects and other objects are common, we can model their relationships within a parent class.



Note: in the above class diagram, each Graduate object or Faculty object has an association with an Office object

Further Abstraction

- When the relationships between Graduate or Faculty objects and other objects are common, we can model their relationships within a parent class.

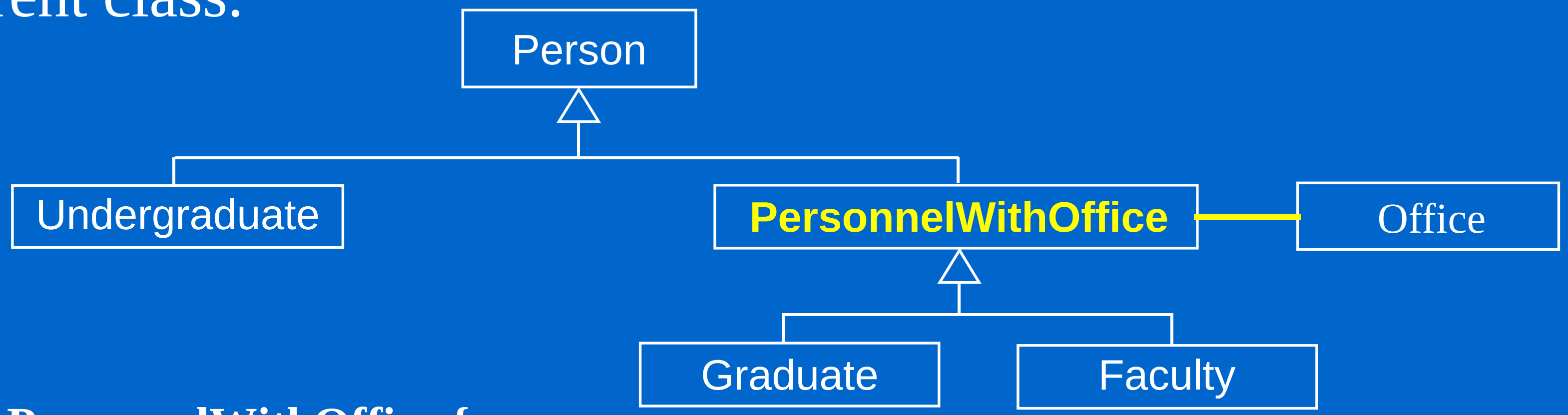



```
class PersonnelWithOffice {
public:
    const char *getAddress() const;
private:
    Office m_office;
};
```

Note: in the above class diagram, each Graduate object or Faculty object has an association with an Office object

Further Abstraction

- When the relationships between Graduate or Faculty objects and other objects are common, we can model their relationships within a parent class.



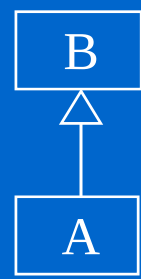

```
class PersonnelWithOffice {
public:
    const char *getAddress() const;
private:
    Office m_office;
};
```

Note: in the above class diagram, each Graduate object or Faculty object has an association with an Office object

- If there could be several offices for a certain personnel, the private member could be a container, ex. vector<Office> m_offices;

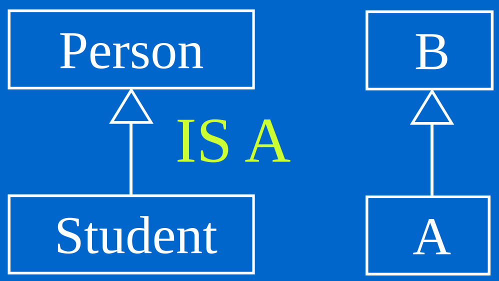
Design Rules for Inheritance

- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B



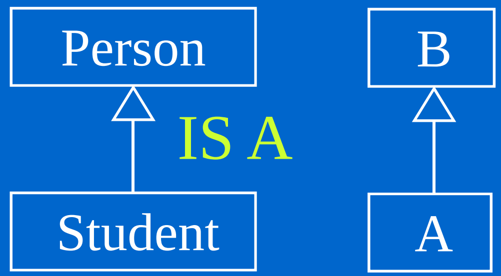
Design Rules for Inheritance

- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B
  - A student **is a** person



Design Rules for Inheritance

- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B
  - A student **is a** person

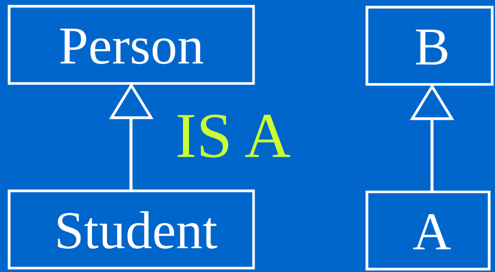


  - Inheritance is called an IS-A relationship

This def is formal
but still abstract!!
Difficult to follow!

Design Rules for Inheritance

- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B

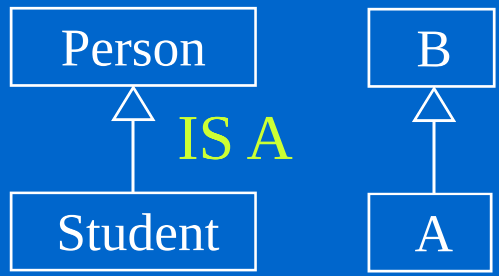


  - A student **is a** person
  - Inheritance is called an IS-A relationship
  - What we mean by "is-a" in programming is "**substitutability**".

This def is formal
but still abstract!!
Difficult to follow!

Design Rules for Inheritance

✧ **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B



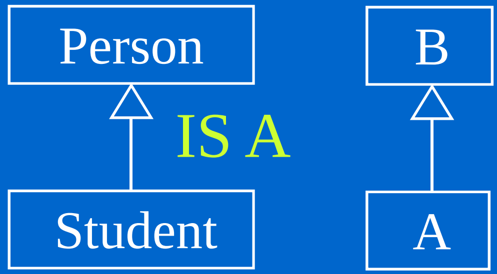
Liskov substitution Principle (LSP)

- A student **is a** person
- Inheritance is called an IS-A relationship
- What we mean by "is-a" in programming is "**substitutability**".

This def is formal but still abstract!! Difficult to follow!

Design Rules for Inheritance

- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B



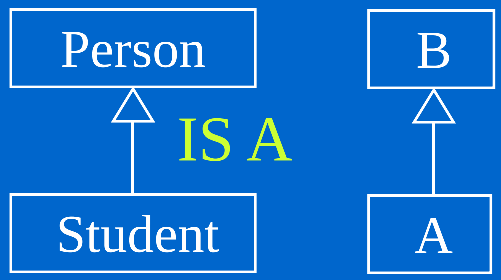
Liskov substitution Principle (LSP)

This def is formal but still abstract!! Difficult to follow!

  - A student **is a** person
  - Inheritance is called an IS-A relationship
  - What we mean by "is-a" in programming is "**substitutability**".
  - Eg. Can an object of type Student be used in whatever place of an object of type Person? This is described in terms of their interfaces (the promises and requirements), instead of their implementations. If yes, Student can inherit Person.

Design Rules for Inheritance

- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B



Liskov substitution Principle (LSP)

- A student **is a** person
- Inheritance is called an IS-A relationship

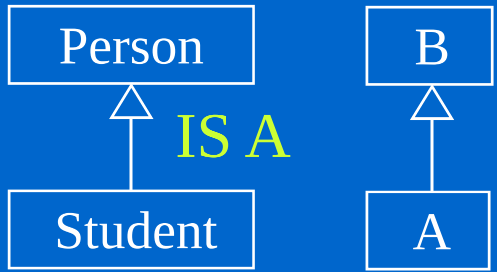
This def is formal but still abstract!! Difficult to follow!

- What we mean by "is-a" in programming is "**substitutability**".
- Eg. Can an object of type Student be used in whatever place of an object of type Person? This is described in terms of their interfaces (the promises and requirements), instead of their implementations. If yes, Student can inherit Person.

- Inheritance should be "natural"

Design Rules for Inheritance

✧ **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B



Liskov substitution Principle (LSP)

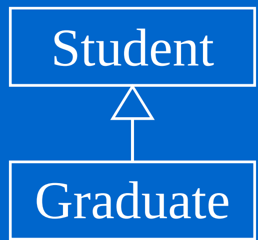
- A student **is a** person

This def is formal but still abstract!! Difficult to follow!

- Inheritance is called an IS-A relationship
- What we mean by "is-a" in programming is "**substitutability**".
- Eg. Can an object of type Student be used in whatever place of an object of type Person? This is described in terms of their interfaces (the promises and requirements), instead of their implementations. If yes, Student can inherit Person.

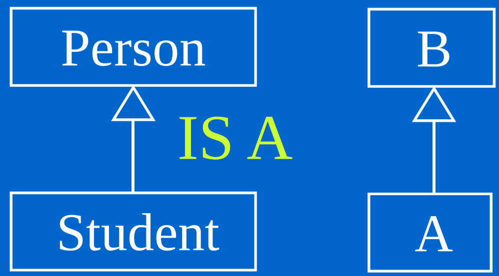
✧ Inheritance should be "natural"

Proper inheritance



Design Rules for Inheritance

- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B
  - A student **is a** person

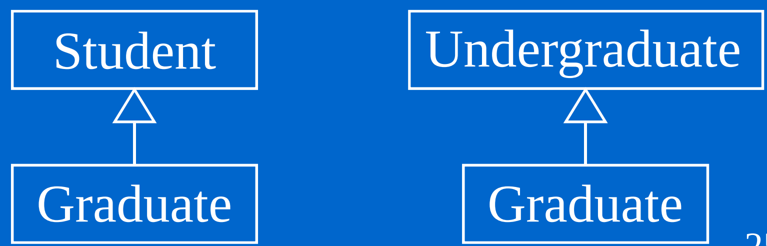


Liskov substitution Principle (LSP)

This def is formal but still abstract!! Difficult to follow!

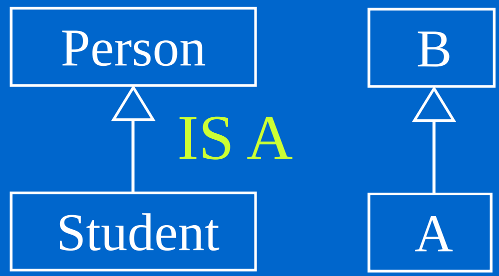
  - Inheritance is called an IS-A relationship
  - What we mean by "is-a" in programming is "**substitutability**".
  - Eg. Can an object of type Student be used in whatever place of an object of type Person? This is described in terms of their interfaces (the promises and requirements), instead of their implementations. If yes, Student can inherit Person.
- Inheritance should be "natural"

Proper inheritance **Improper inheritance**



Design Rules for Inheritance

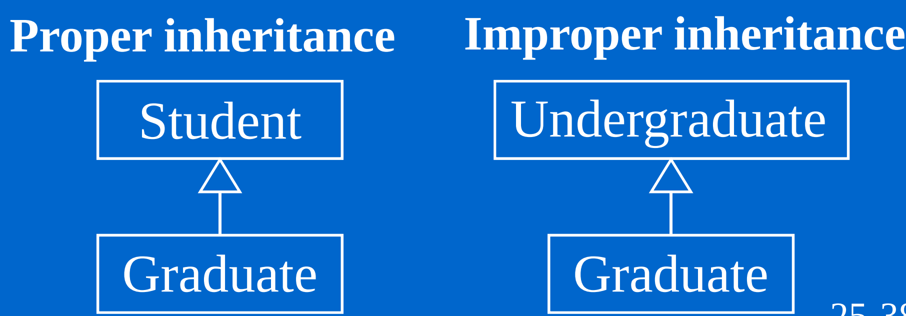
- **Primary guide**: Class A should only be derived from Class B if Class A **is a** type of Class B
  - A student **is a** person



Liskov substitution Principle (LSP)

This def is formal but still abstract!! Difficult to follow!

  - Inheritance is called an IS-A relationship
  - What we mean by "is-a" in programming is "**substitutability**".
  - Eg. Can an object of type Student be used in whatever place of an object of type Person? This is described in terms of their interfaces (the promises and requirements), instead of their implementations. If yes, Student can inherit Person.
- Inheritance should be "natural"
  - The second case is a bad inheritance even if Undergraduate is internally identical to Student.

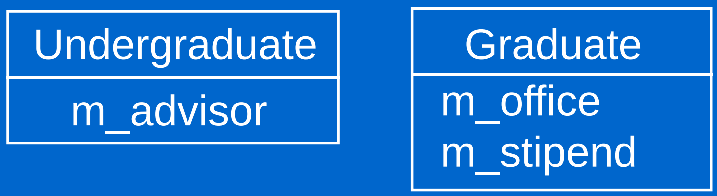


Design Rules (cont'd)

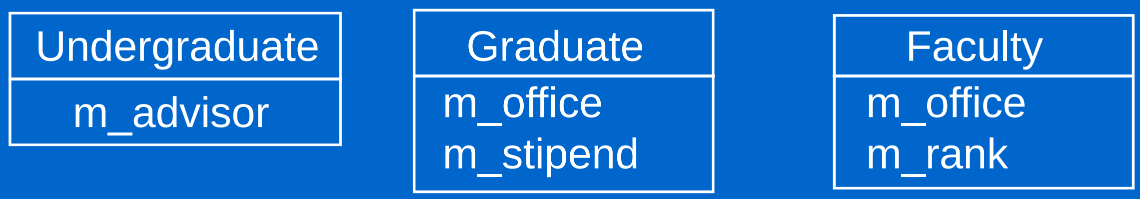
Design Rules (cont'd)

Undergraduate
m_advisor

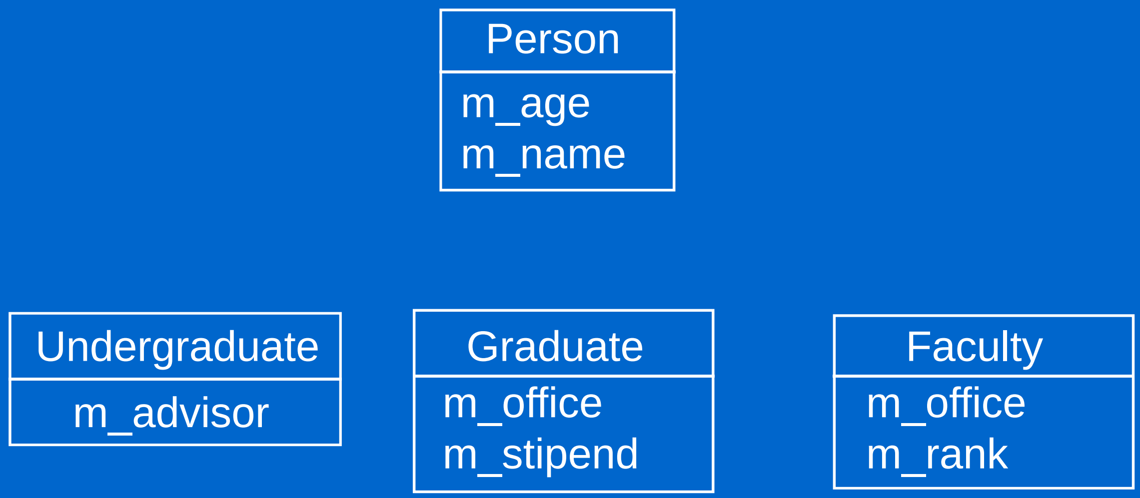
Design Rules (cont'd)



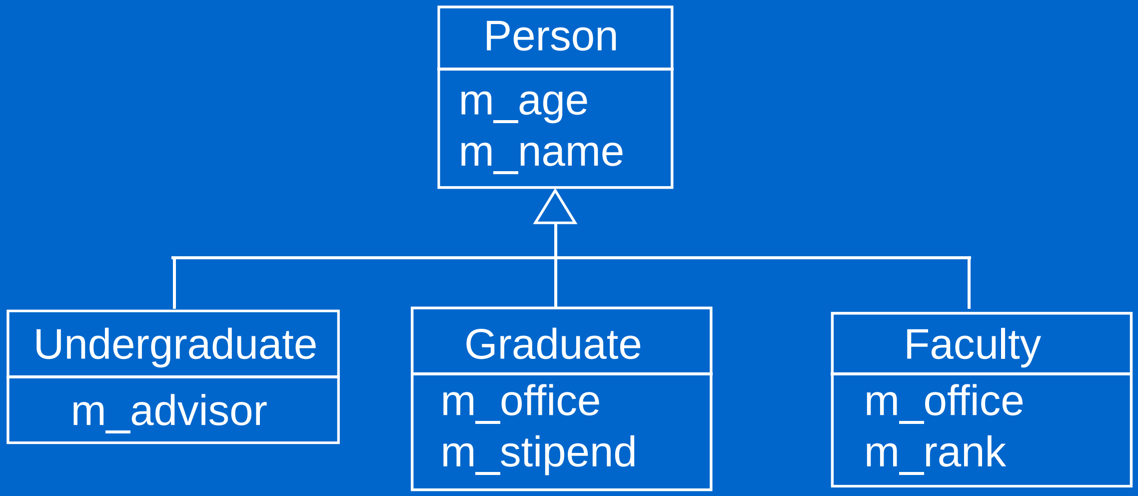
Design Rules (cont'd)



Design Rules (cont'd)

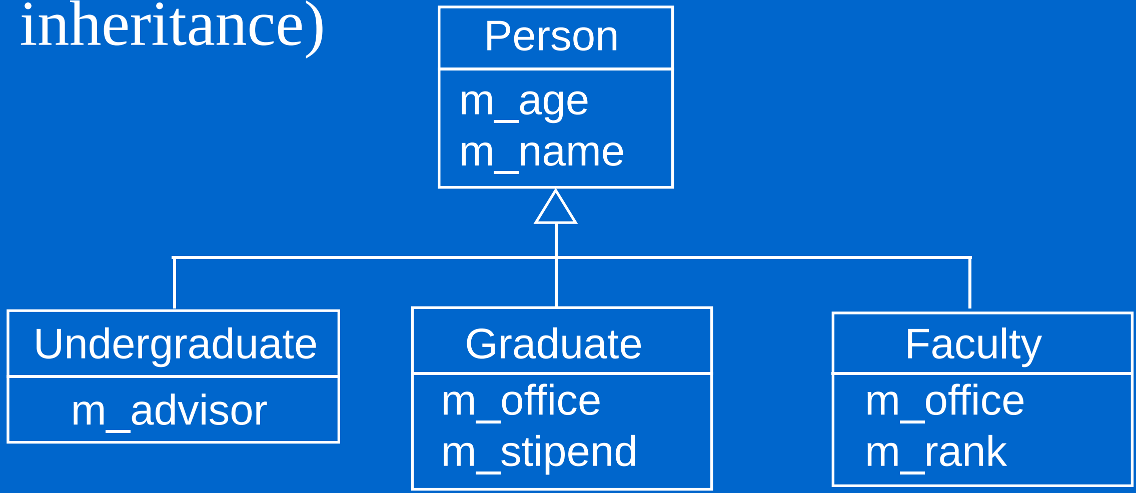


Design Rules (cont'd)



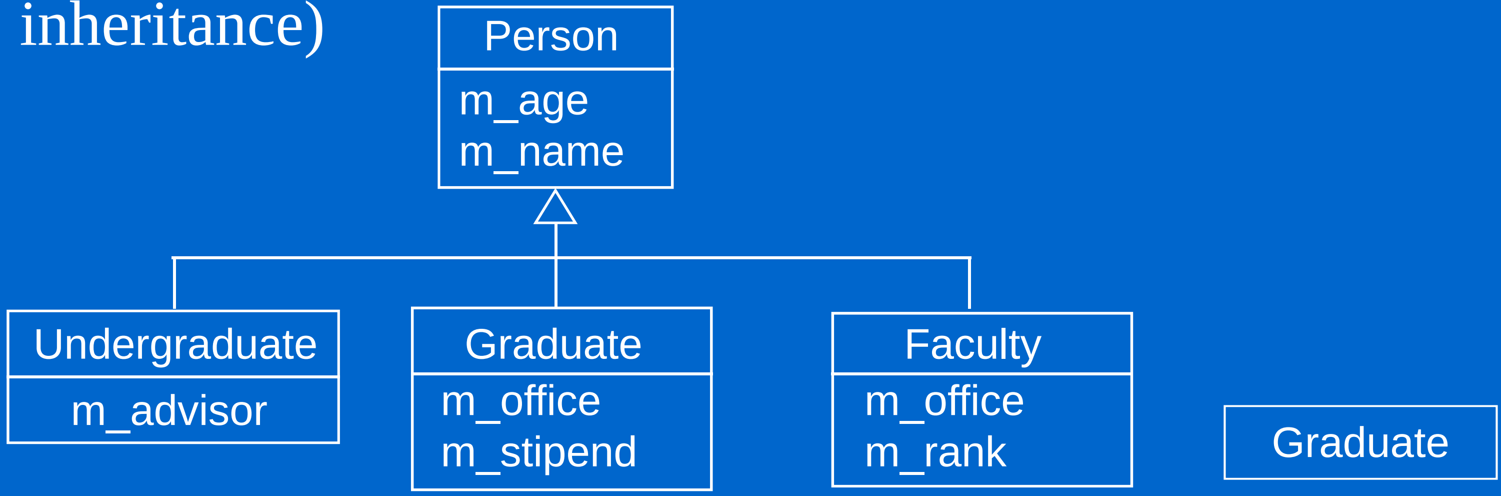
Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)



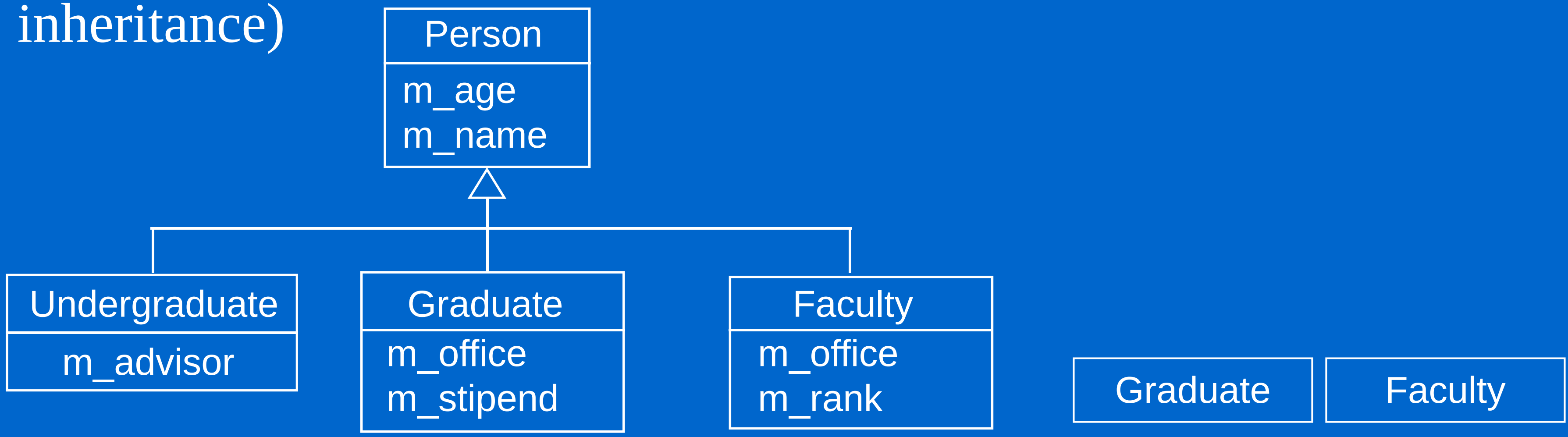
Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)



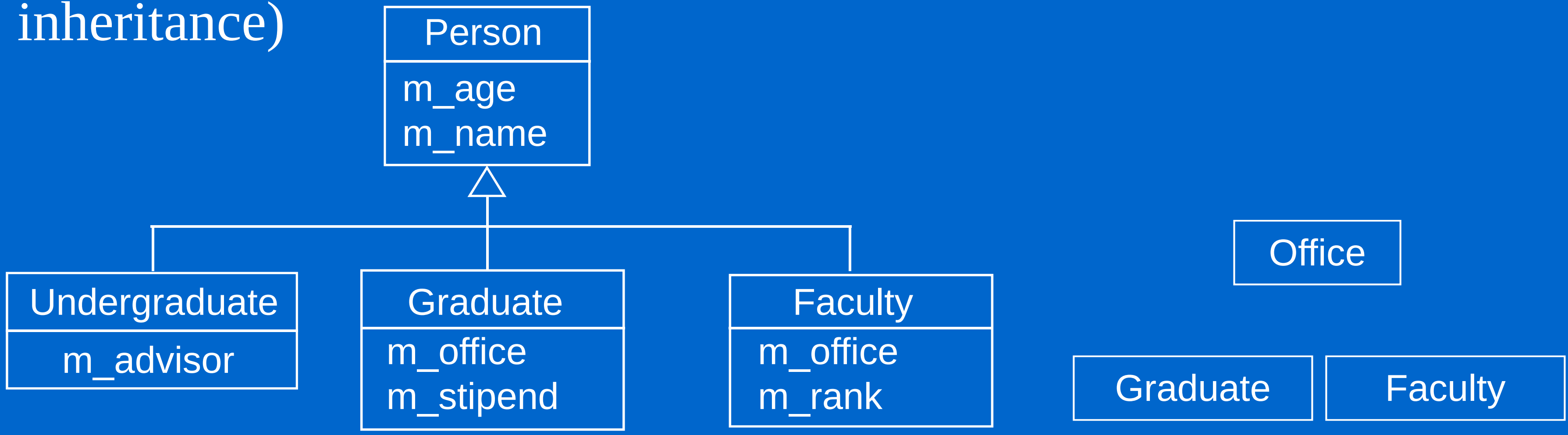
Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)



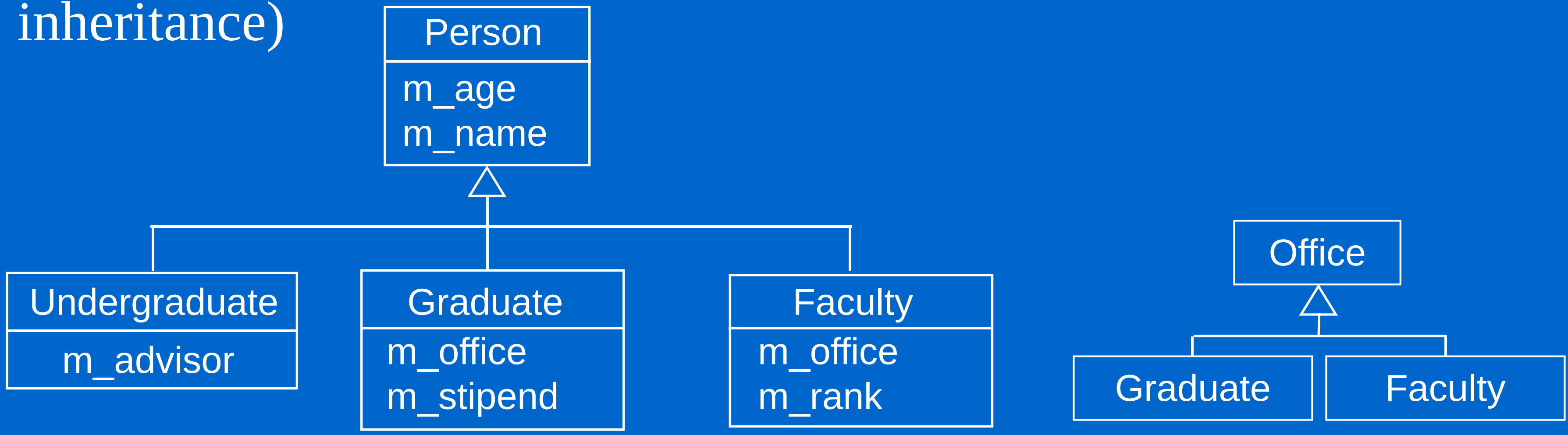
Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)



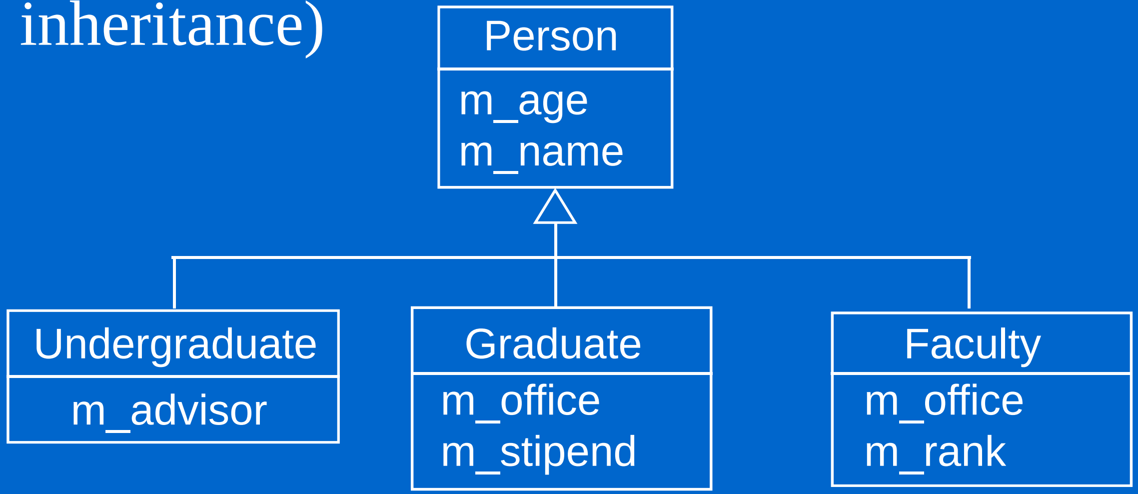
Design Rules (cont'd)

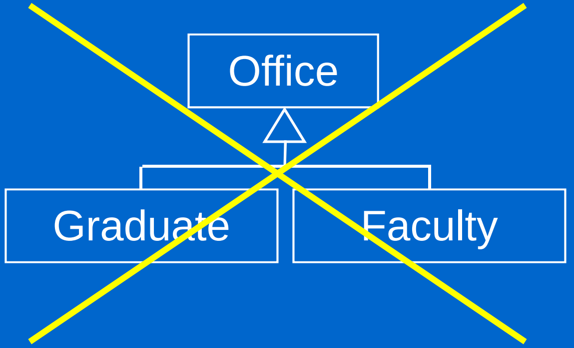
- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)



Design Rules (cont'd)

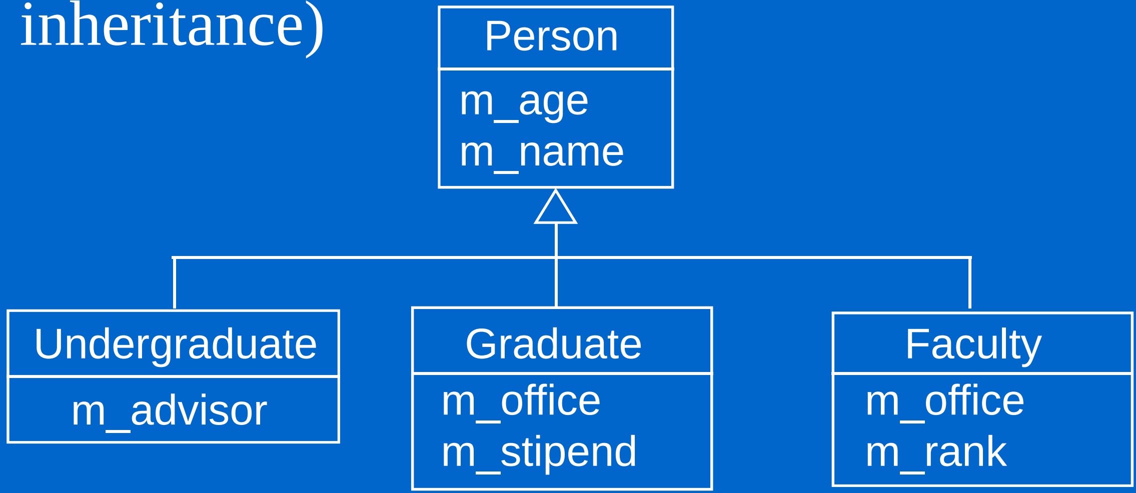
- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)

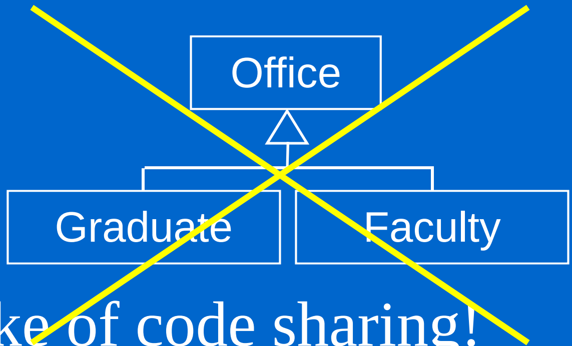




Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)

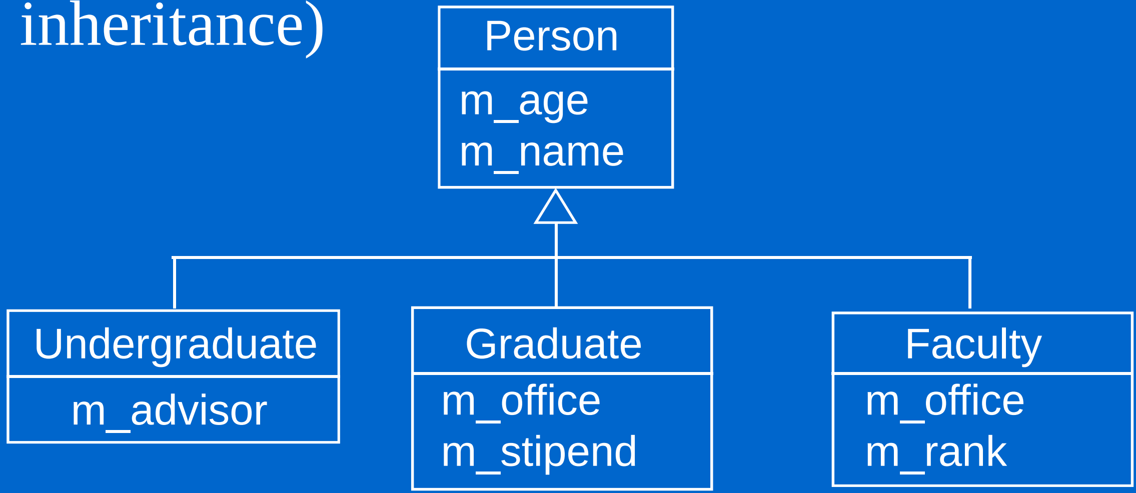


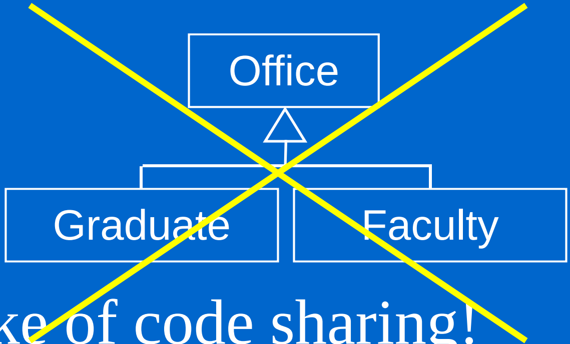


- Never violate the primary objectives for the sake of code sharing!

Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)

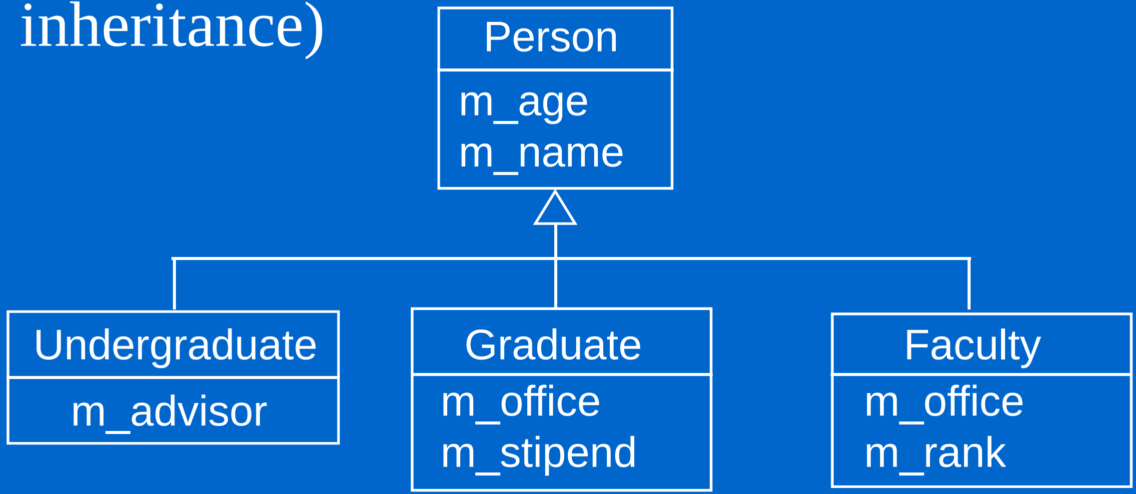


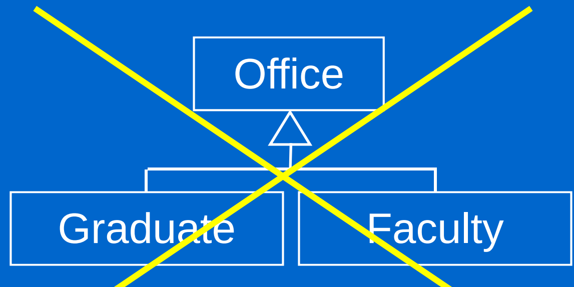


- Never violate the primary objectives for the sake of code sharing!
- Bad cases of inheritance (improper inheritances) are often cured through composition (containment / aggregation)

Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)

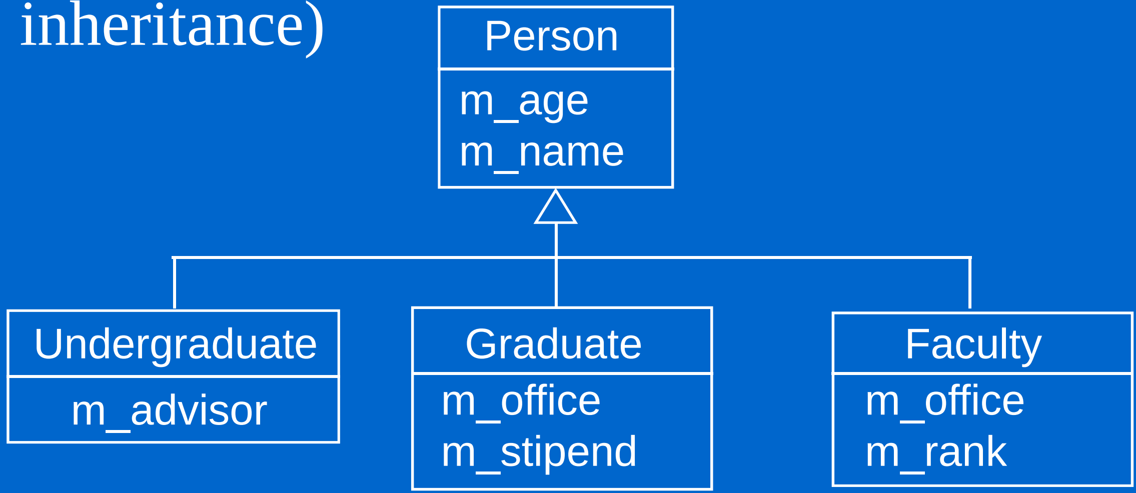


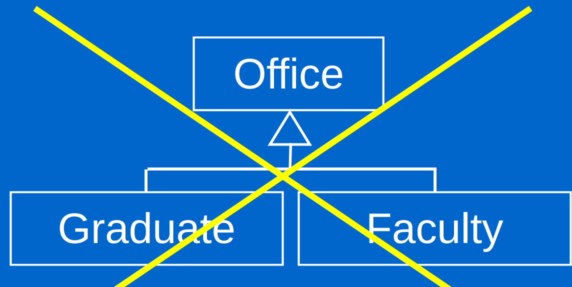


- Never violate the primary objectives for the sake of code sharing!
- Bad cases of inheritance (improper inheritances) are often cured through composition (containment / aggregation)

Design Rules (cont'd)

- **Common code and data** between classes can be shared by creating a base class (one of the two primary benefits we can get from inheritance)





- Never violate the primary objectives for the sake of code sharing!
- Bad cases of inheritance (improper inheritances) are often cured through composition (containment / aggregation)

This is referred to as the HAS-A relationship.

It operates in the form of delegation.

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

```
class Point {
public:
   Point(double x=0, double y=0);
protected:
   double x, y;
};
```

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

```
class Point {
public:
   Point(double x=0, double y=0);
protected:
   double x, y;
};
class Circle: public Point {
public:
   Circle(double x=0, double y=0, double radius=0);
   void display() const;
private:
   double radius;
};
```

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

```
class Point {
public:
   Point(double x=0, double y=0);
protected:
   double x, y;
};
```

```
class Circle: public Point {
public:
   Circle(double x=0, double y=0, double radius=0);
   void display() const;
private:
   double radius;
};
```

```
void Circle::display() {
   cout << "Center = " << c.x << ", " << c.y
         << "]; Radius = " << radius;
}
```

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

```
class Point {
public:
    Point(double x=0, double y=0);
protected:
    double x, y;
};
```

```
void Circle::display() {
    cout << "Center = " << c.x << ", " << c.y
         << "]; Radius = " << radius;
}
```

```
class Circle: public Point {
public:
    Circle(double x=0, double y=0, double radius=0);
    void display() const;
private:
    double radius;
};
```

- Design rationale: A circle is a type of point, with common data. The radius of a circle is zero or approaching zero. ... Purely mathematical!

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

```
class Point {
public:
   Point(double x=0, double y=0);
protected:
   double x, y;
};
```

```
void Circle::display() {
   cout << "Center = " << c.x << ", " << c.y
         << "]; Radius = " << radius;
}
```

```
class Circle: public Point {
public:
   Circle(double x=0, double y=0, double radius=0);
   void display() const;
private:
   double radius;
};
```

- Design rationale: A circle is a type of point, with common data. The radius of a circle is zero or approaching zero. ... Purely mathematical!
- Critiques: A circle is not a point. Instead, a circle has a point corresponding to its center

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

```
class Point {
public:
   Point(double x=0, double y=0);
protected:
   double x, y;
};
```

```
void Circle::display() {
   cout << "Center = " << c.x << ", " << c.y
         << "]; Radius = " << radius;
}
```

```
class Circle: public Point {
public:
   Circle(double x=0, double y=0, double radius=0);
   void display() const;
private:
   double radius;
};
```

- Design rationale: A circle is a type of point, with common data. The radius of a circle is zero or approaching zero. ... Purely mathematical!
- Critiques: A circle is not a point. Instead, a circle has a point corresponding to its center. Substitutability: Can a circle be used as a point in constructing the four corners of a rectangle?

Dubious Examples of Inheritance

- Taken from Deitel & Deitel, C: How to program, p. 736

```
class Point {
public:
   Point(double x=0, double y=0);
protected:
   double x, y;
};
```

```
void Circle::display() {
   cout << "Center = " << c.x << ", " << c.y
        << "]; Radius = " << radius;
}
```

```
class Circle: public Point {
public:
   Circle(double x=0, double y=0, double radius=0);
   void display() const;
private:
   double radius;
};
```

- Design rationale: A point is a type of circle, with common data, when the radius of a circle is approaching zero. ... Purely mathematical!
- Critiques: A circle is not a point. Instead, a circle has a point corresponding to its center. Substitutability: Can a circle be used as a point in constructing the four corners of a rectangle? Can a circle be used as the center of another circle?

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list What are the problems?

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list   What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.
  - If you try to turn off the insert()/delete() interface that could manipulate entries in any order, you basically make the Stack class different from the LinkList base class in terms of operations.

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list   What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.
  - If you try to turn off the insert()/delete() interface that could manipulate entries in any order, you basically make the Stack class different from the LinkList base class in terms of operations. **Client codes break!**

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list   What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.
  - If you try to turn off the insert()/delete() interface that could manipulate entries in any order, you basically make the Stack class different from the LinkList base class in terms of operations. **Client codes break!** A Stack **IS-NOT** a LinkList.

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.
  - If you try to turn off the insert()/delete() interface that could manipulate entries in any order, you basically make the Stack class different from the LinkList base class in terms of operations. **Client codes break!** A Stack **IS-NOT** a LinkList.
- Ex 2: A file pathname class derived from a string class

Some Other Dubious Examples

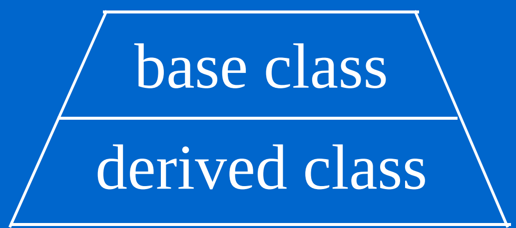
- Ex 1: A stack derived from a linked list What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.
  - If you try to turn off the insert()/delete() interface that could manipulate entries in any order, you basically make the Stack class different from the LinkList base class in terms of operations. **Client codes break!** A Stack **IS-NOT** a LinkList.
- Ex 2: A file pathname class derived from a string class

  note: a pathname IS indeed implemented by a string, but it is a special string that cannot be longer than 32 characters

Some Other Dubious Examples

- Ex 1: A stack derived from a linked list   What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.
  - If you try to turn off the insert()/delete() interface that could manipulate entries in any order, you basically make the Stack class different from the LinkList base class in terms of operations. **Client codes break!** A Stack **IS-NOT** a LinkList.
- Ex 2: A file pathname class derived from a string class

  note: a pathname IS indeed implemented by a string, but it is a special string that cannot be longer than 32 characters
- **Design rule**: The derived class **extends** the base class

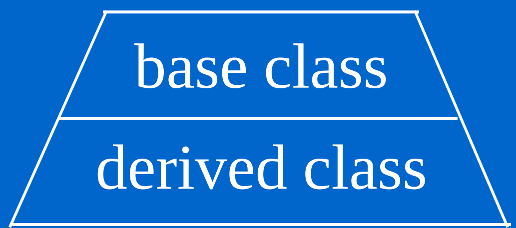


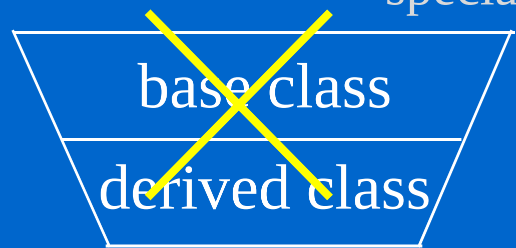
Some Other Dubious Examples

- Ex 1: A stack derived from a linked list   What are the problems?
  - This stack can then be operated as a linked list, the mechanism of a stack would be completely broken.
  - If you try to turn off the insert()/delete() interface that could manipulate entries in any order, you basically make the Stack class different from the LinkList base class in terms of operations. **Client codes break!** A Stack **IS-NOT** a LinkList.
- Ex 2: A file pathname class derived from a string class

  note: a pathname IS indeed implemented by a string, but it is a special string that cannot be longer than 32 characters
- **Design rule**: The derived class **extends** the base class, not the other way around.

specialization





Points to Consider

To design a **Shape** inheritance hierarchy

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes
- What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …)   Circle-Ellipse   Square-Rectangle

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes
- What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …)   Circle-Ellipse   Square-Rectangle
- Why do you need a Rectangle class as the base class of a Square?

Points to Consider

To design a **Shape** inheritance hierarchy

- ✧ What are the **common operations** you want to perform on all Shapes
- ✧ What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …)   Circle-Ellipse   Square-Rectangle
- ✧ Why do you need a Rectangle class as the base class of a Square?
- ✧ Can a Square substitute for a Rectangle?

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes
- What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …)   Circle-Ellipse   Square-Rectangle
- Why do you need a Rectangle class as the base class of a Square?
- Can a Square substitute for a Rectangle?
- A Rhombus is four-sided, like a Rectangle, so should Rectangle derive from Rhombus?

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes
- What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …)   Circle-Ellipse   Square-Rectangle
- Why do you need a Rectangle class as the base class of a Square?
- Can a Square substitute for a Rectangle?
- A Rhombus is four-sided, like a Rectangle, so should Rectangle derive from Rhombus?
- Should you have a base class for all four-sided objects?

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes
- What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …) Circle-Ellipse Square-Rectangle
- Why do you need a Rectangle class as the base class of a Square?
- Can a Square substitute for a Rectangle?
- A Rhombus is four-sided, like a Rectangle, so should Rectangle derive from Rhombus?
- Should you have a base class for all four-sided objects?
- Should you have another base class for all five-sided objects?

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes
- What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …)   Circle-Ellipse   Square-Rectangle
- Why do you need a Rectangle class as the base class of a Square?
- Can a Square substitute for a Rectangle?
- A Rhombus is four-sided, like a Rectangle, so should Rectangle derive from Rhombus?
- Should you have a base class for all four-sided objects?
- Should you have another base class for all five-sided objects?
- Should you have a general base class for polygons with the number of sides as an attribute?

Points to Consider

To design a **Shape** inheritance hierarchy

- What are the **common operations** you want to perform on all Shapes
- What other kinds of Shapes might you use in your application? (Triangle, Circle, Polygon, Ellipse, Square, Rectangle Rhombus, Pentagon, …)   Circle-Ellipse   Square-Rectangle
- Why do you need a Rectangle class as the base class of a Square?
- Can a Square substitute for a Rectangle?
- A Rhombus is four-sided, like a Rectangle, so should Rectangle derive from Rhombus?
- Should you have a base class for all four-sided objects?
- Should you have another base class for all five-sided objects?
- Should you have a general base class for polygons with the number of sides as an attribute?
- Will your program perform geometric searches to identify objects?

Summary

Summary

Undergraduate
m_advisor m_tuition

Summary

Undergraduate
m_advisor m_tuition

Graduate
m_stipend

Summary

Student

m_home
m_courses

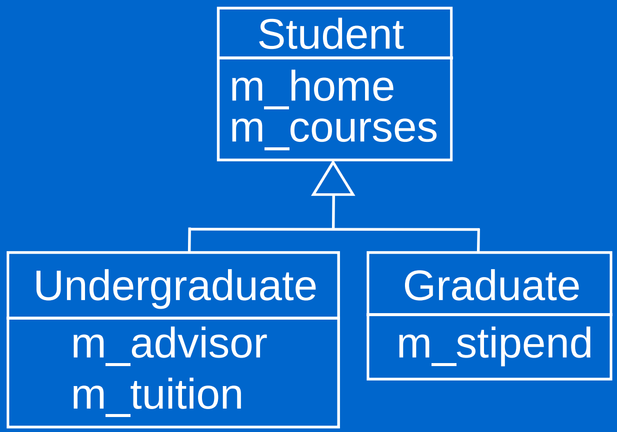
Undergraduate

m_advisor
m_tuition

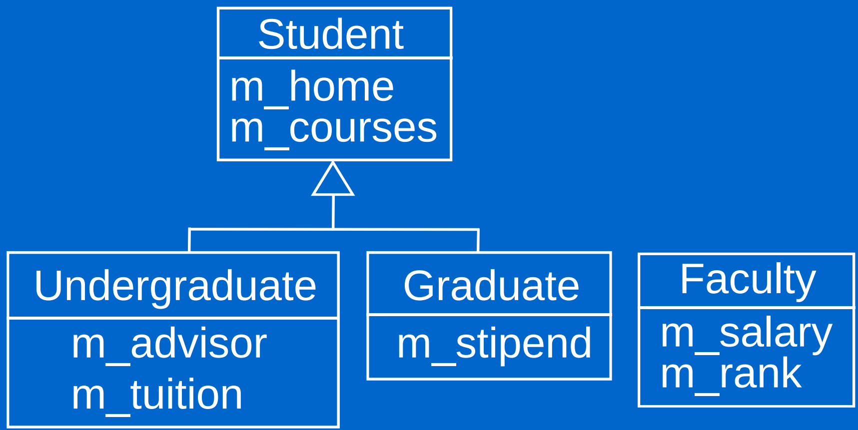
Graduate

m_stipend

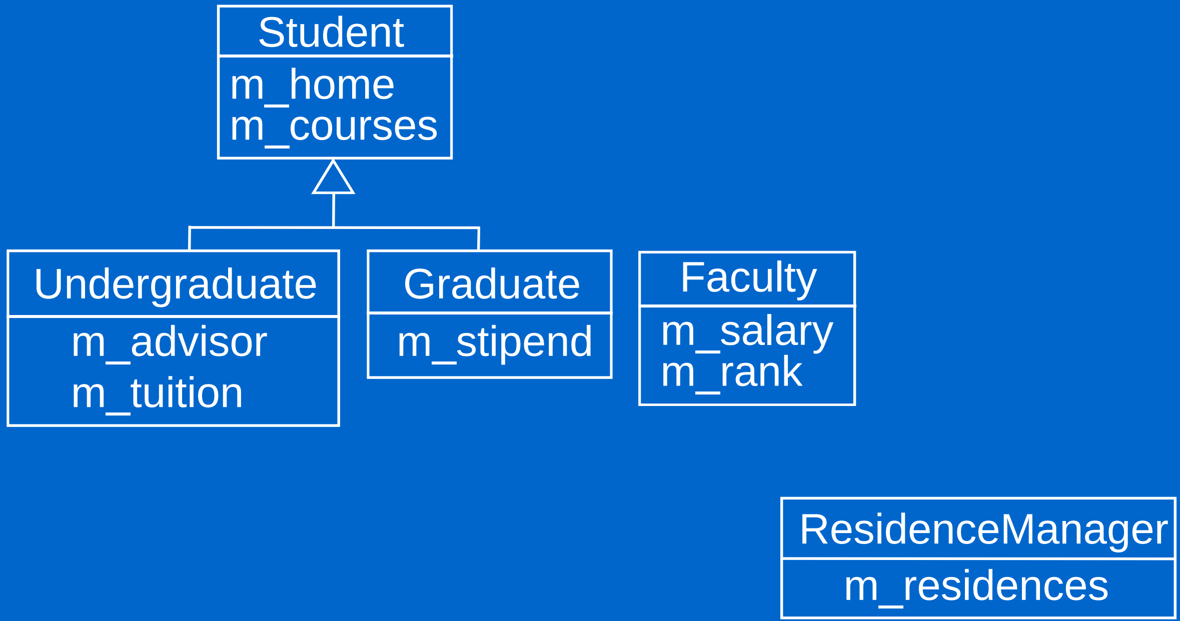
Summary


Student
m_home
m_courses
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend

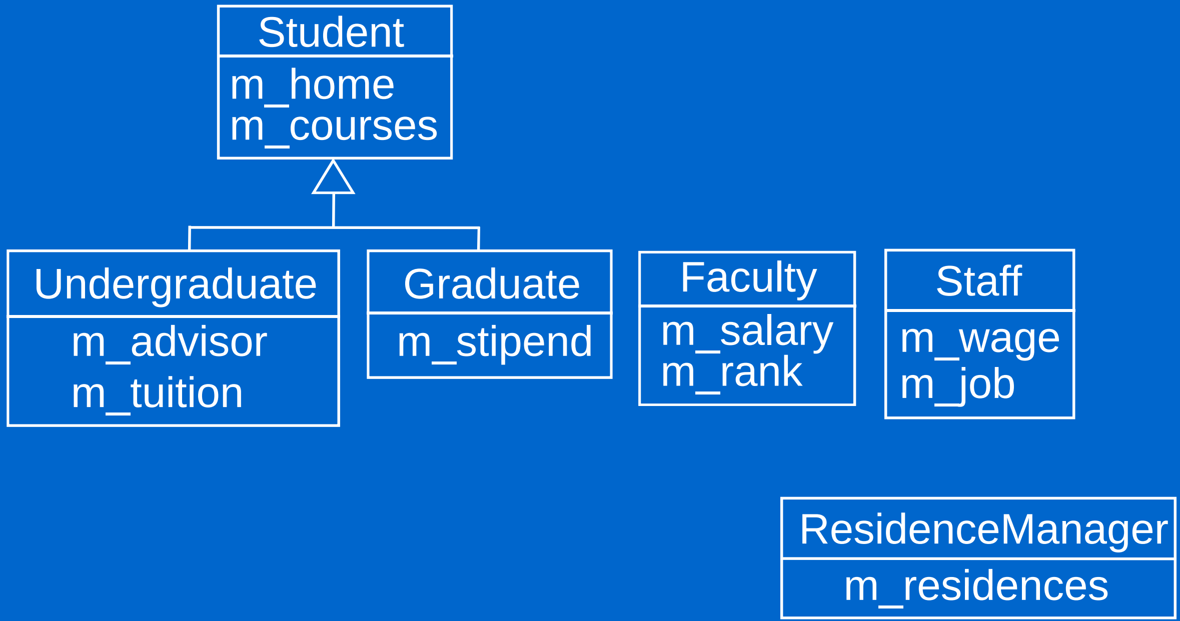
Summary


Student
m_home
m_courses
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank

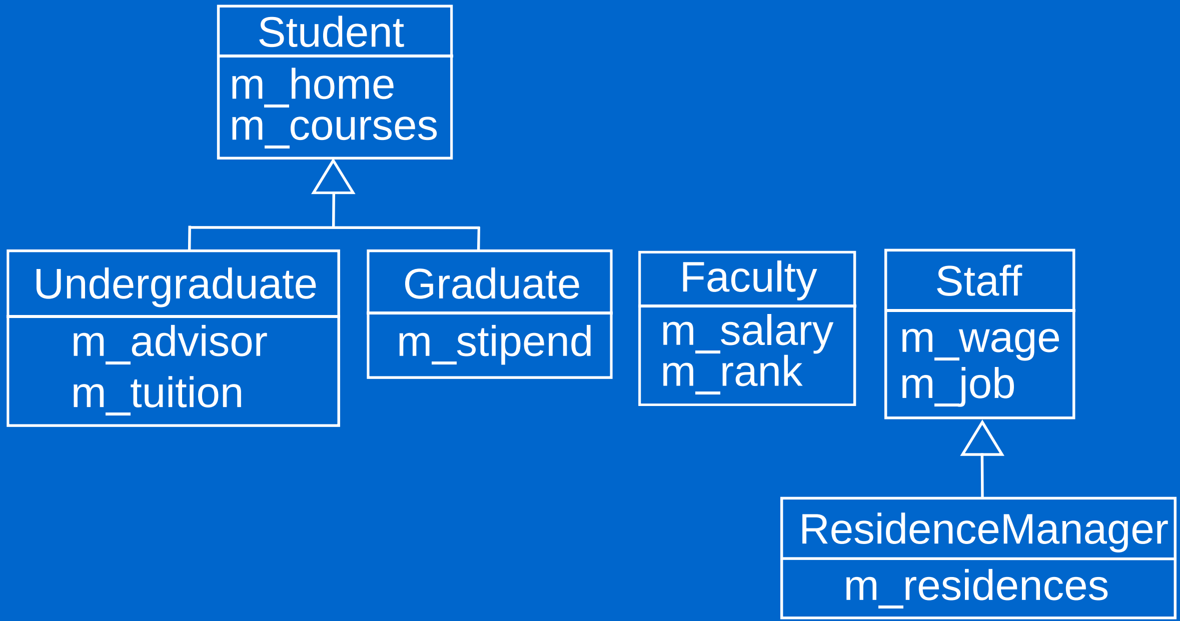
Summary


Student
m_home
m_courses
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
ResidenceManager
m_residences

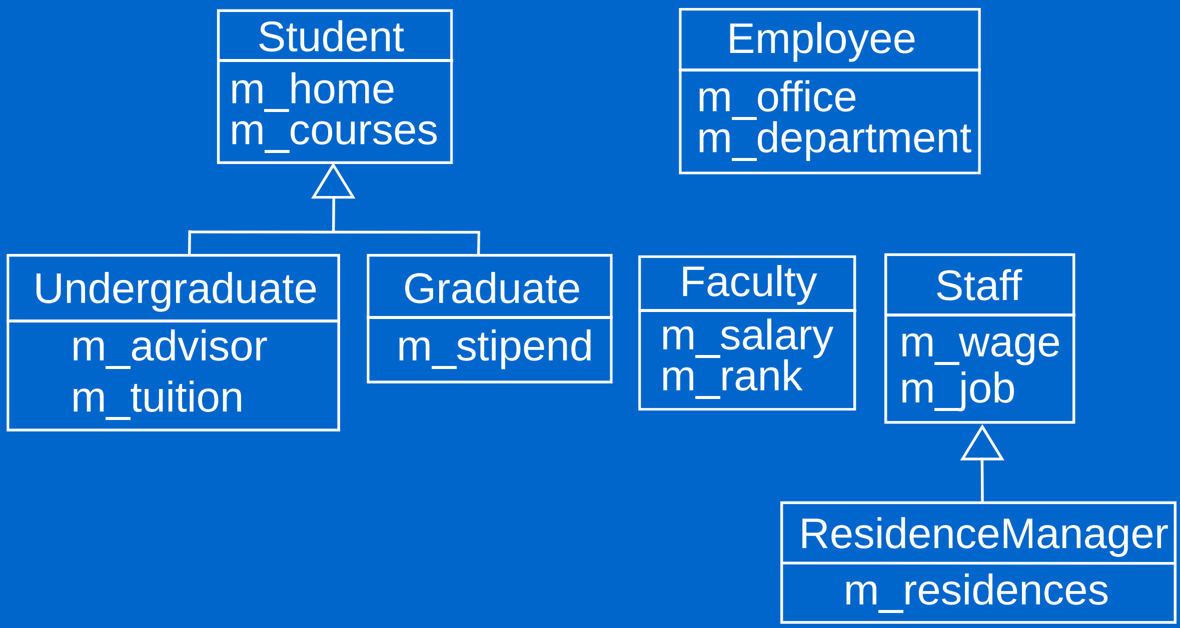
Summary


Student
m_home
m_courses
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

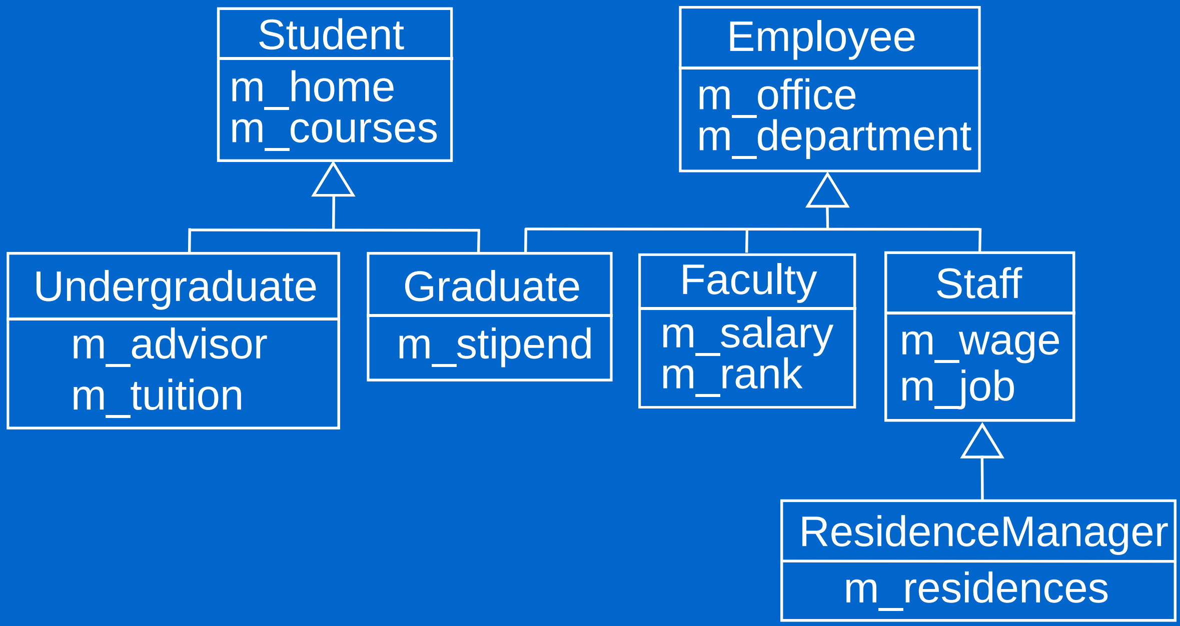
Summary


Student
m_home
m_courses
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

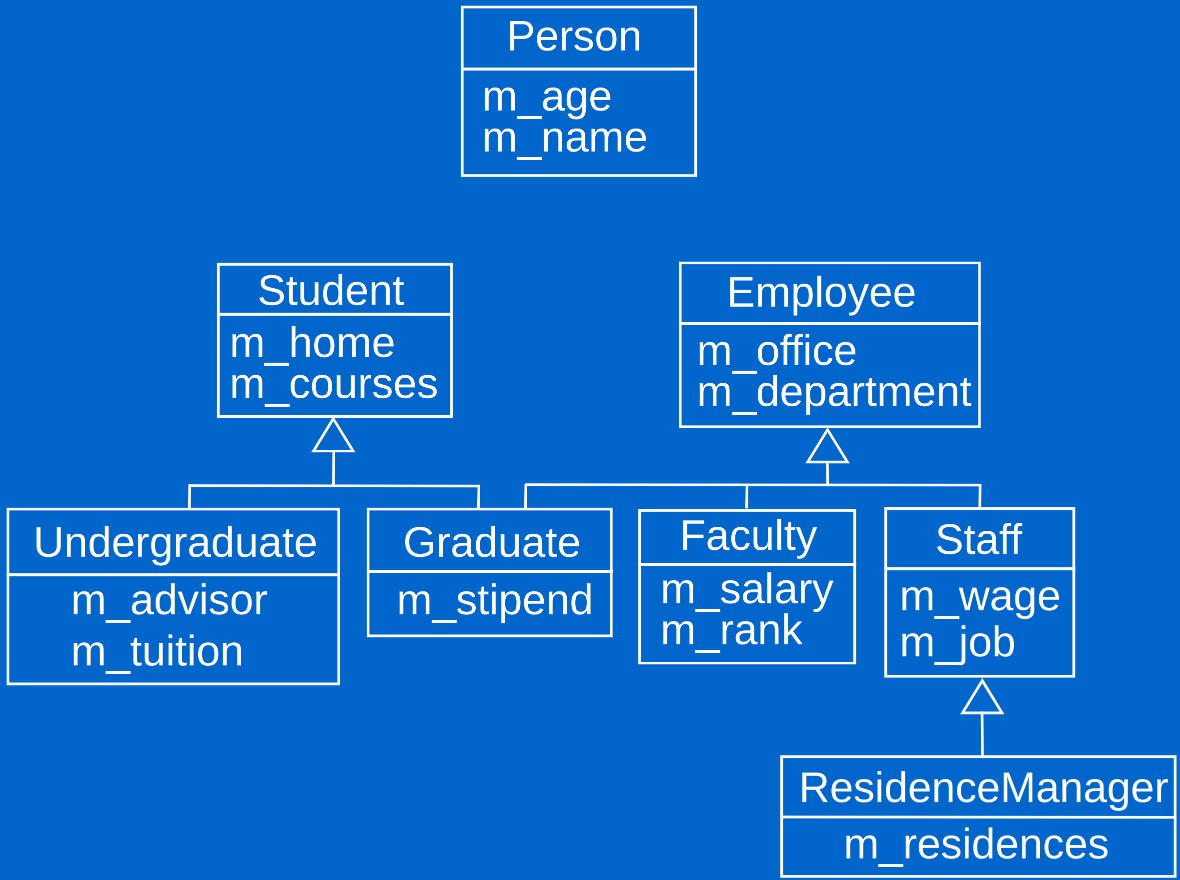
Summary


Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

Summary

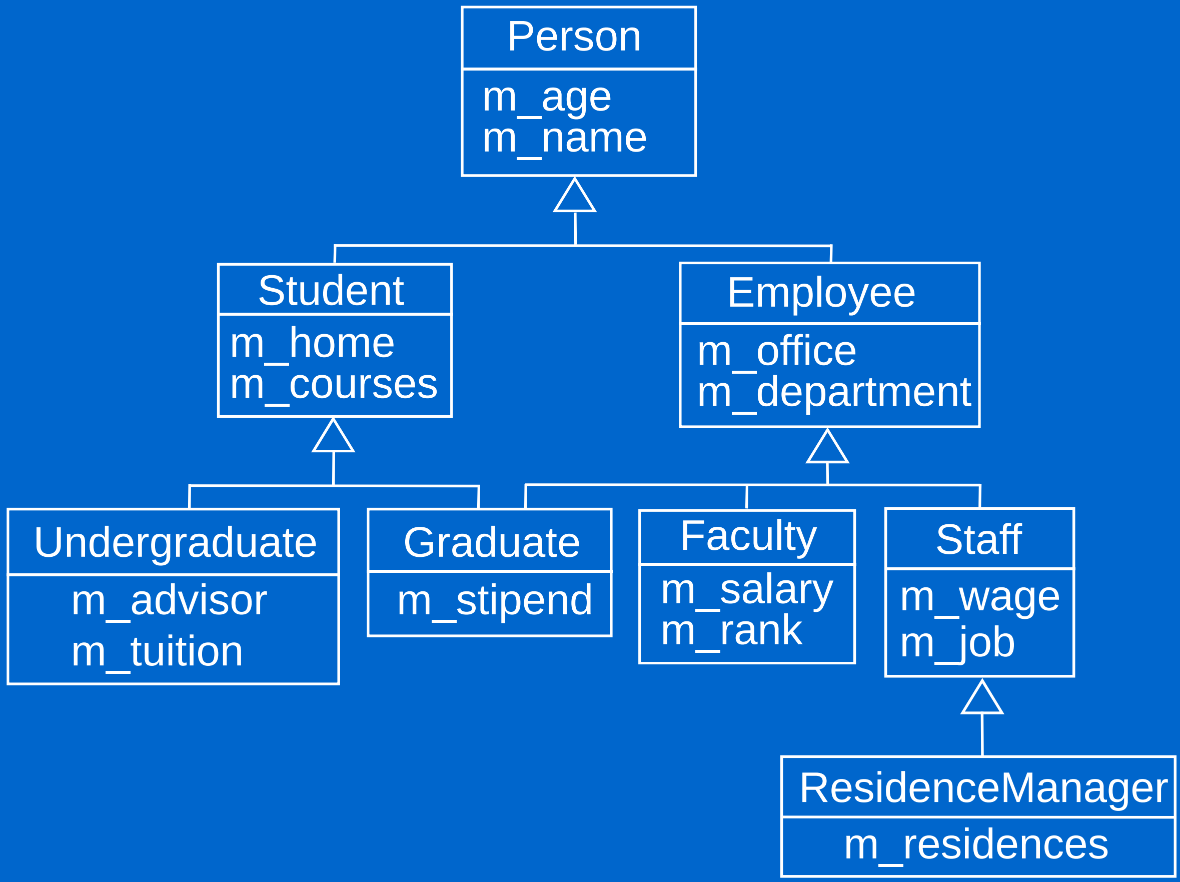

Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

Summary

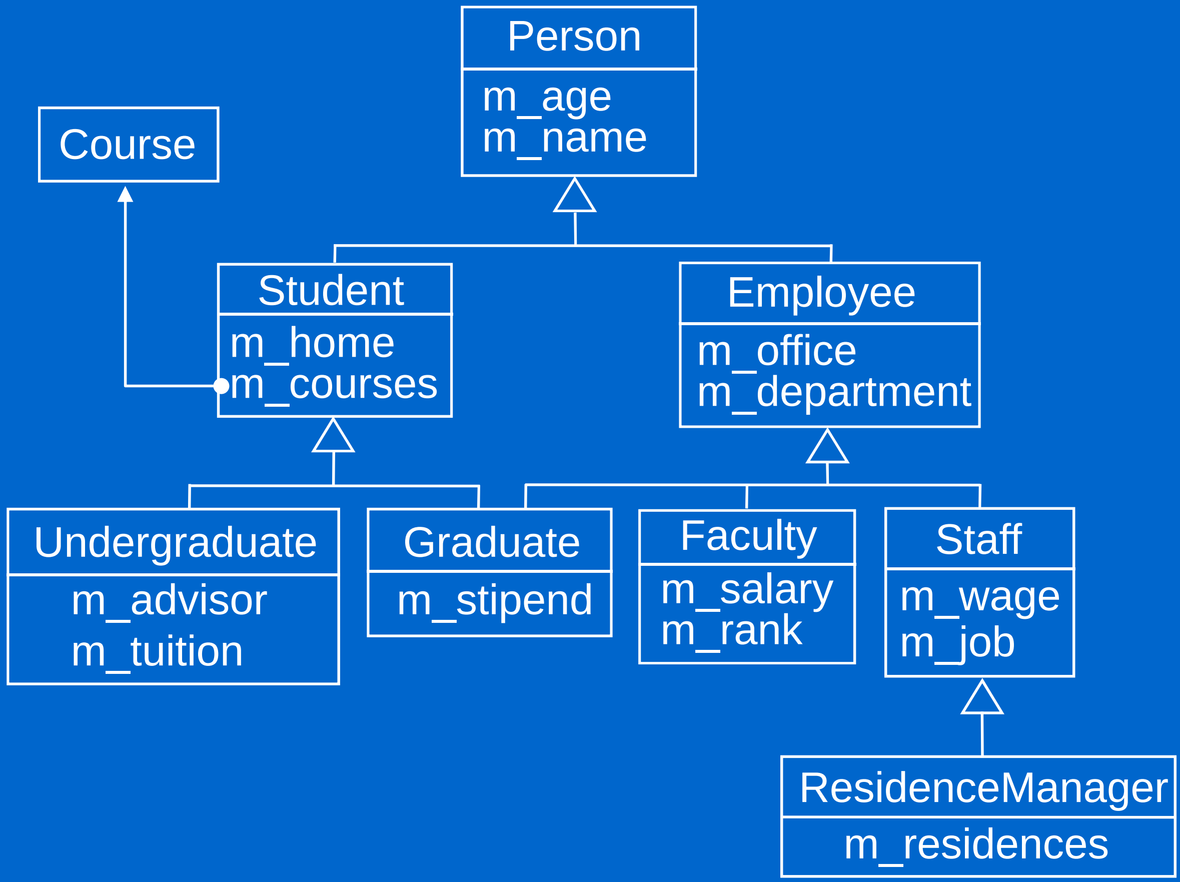


Person
m_age
m_name
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

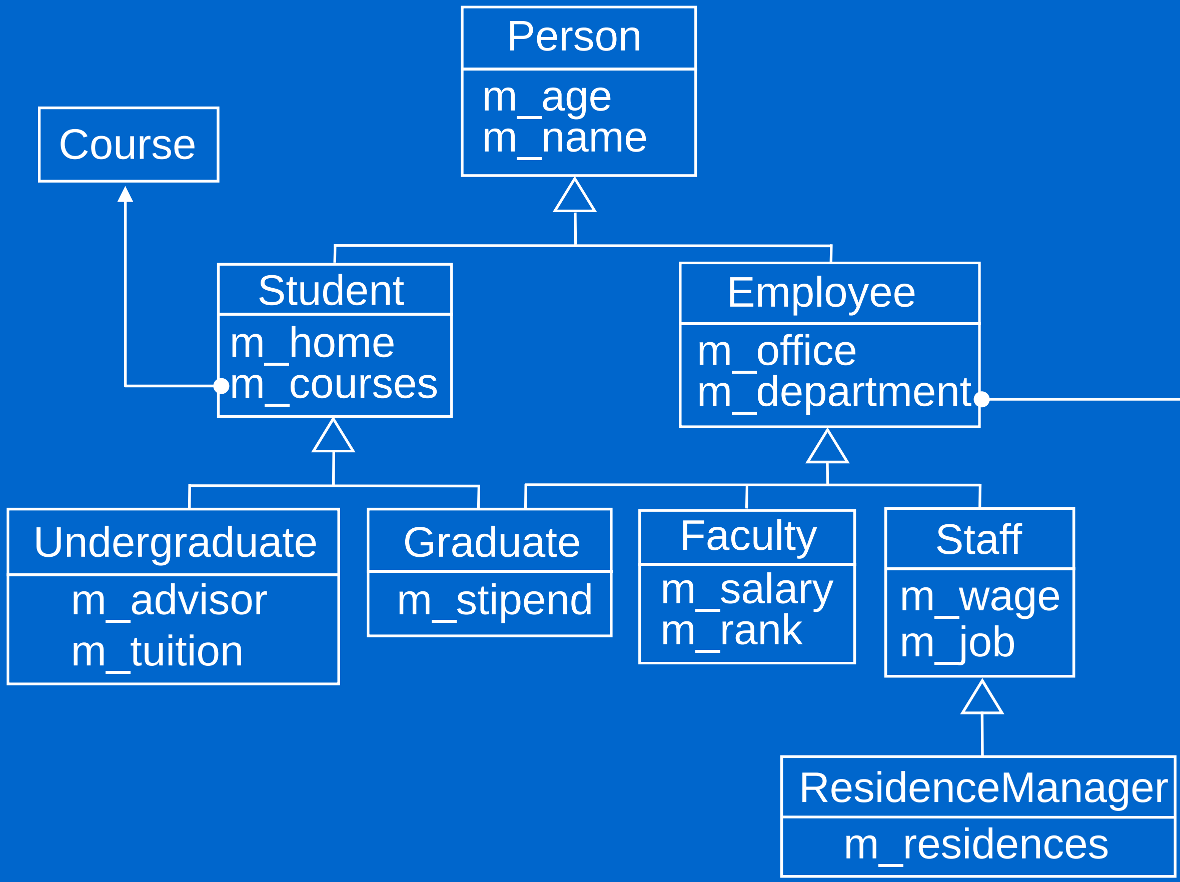
Summary


Person
m_age
m_name
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

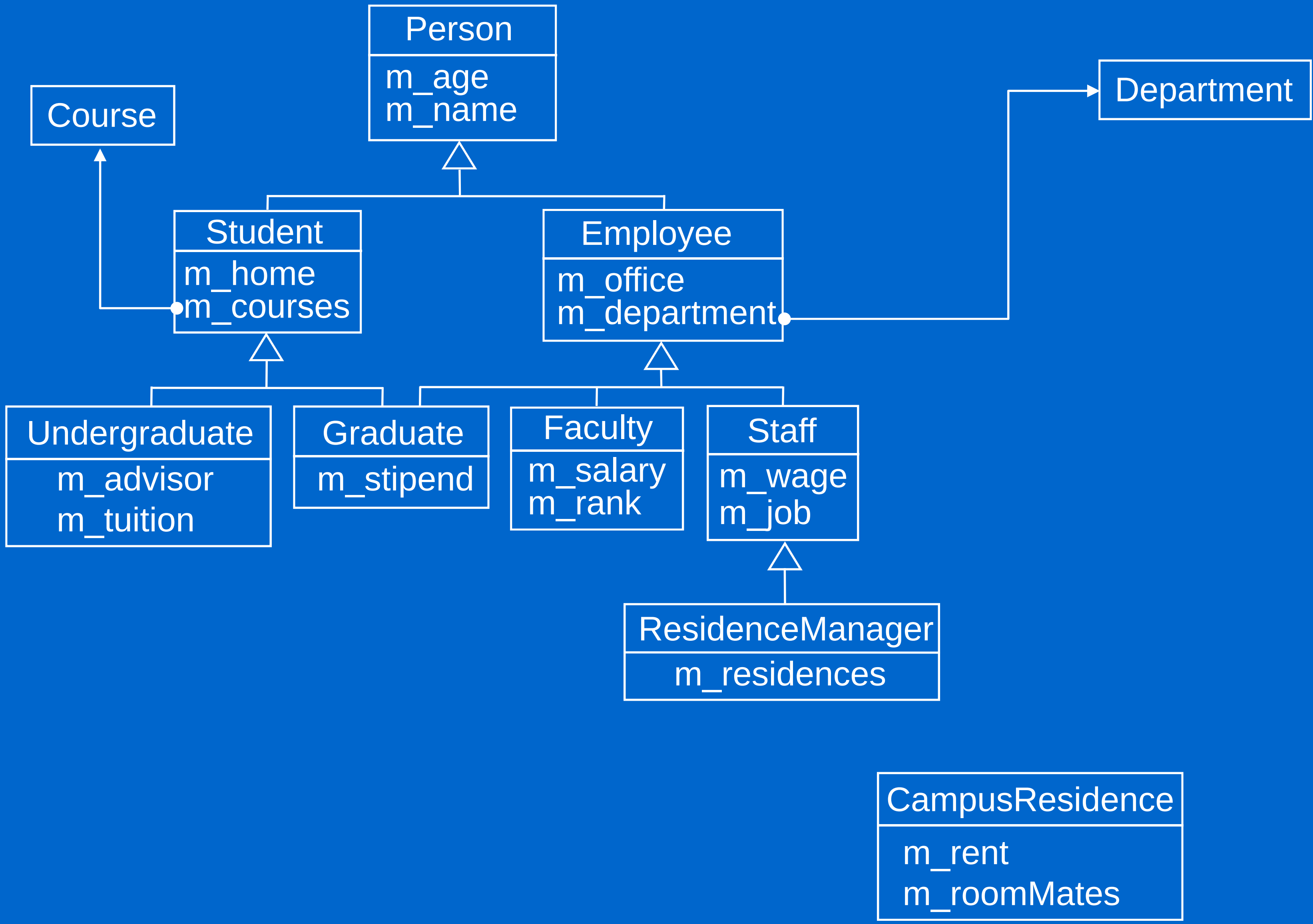
Summary


Person
m_age
m_name
Course
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

Summary

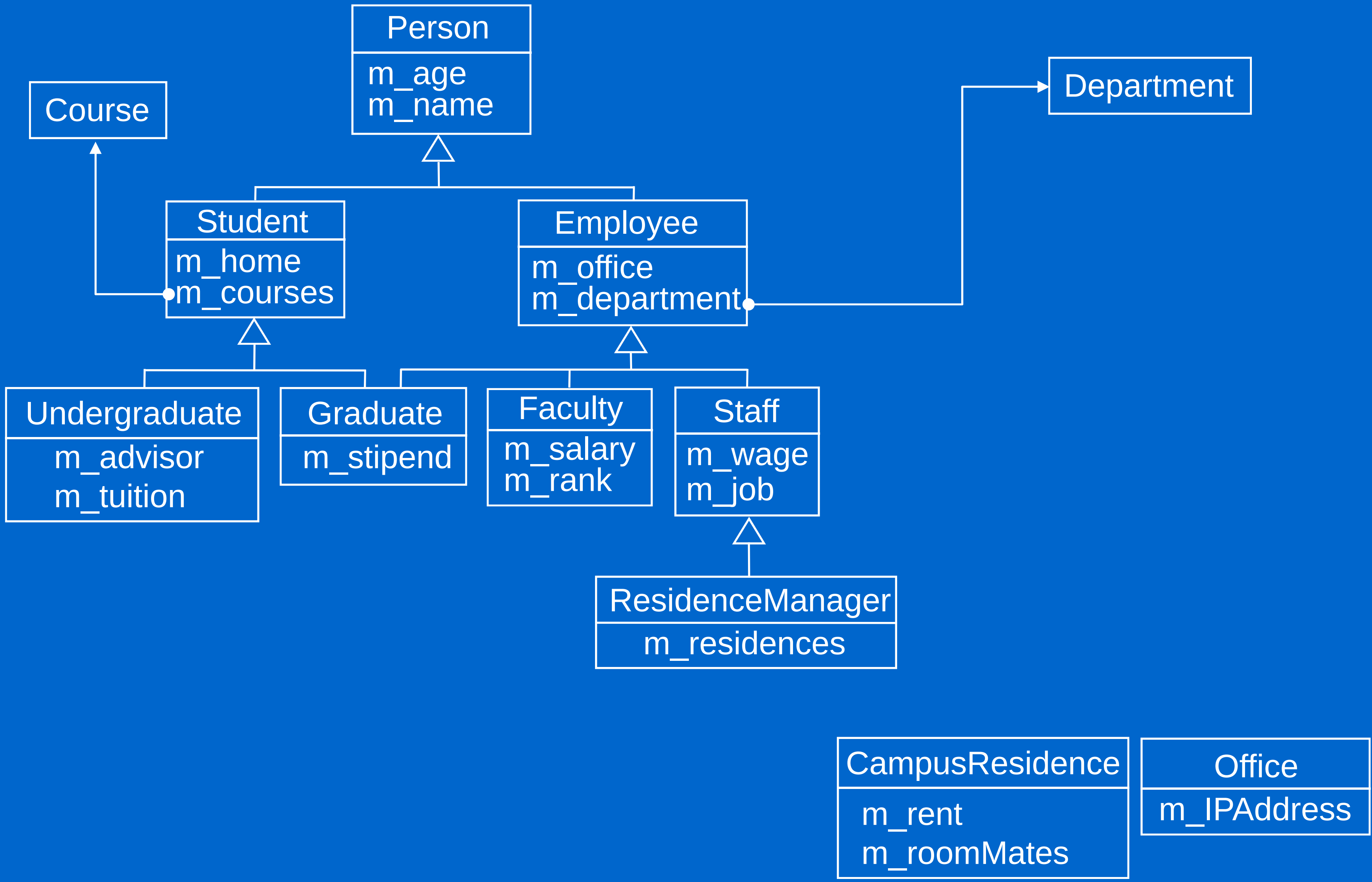

Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences

Summary

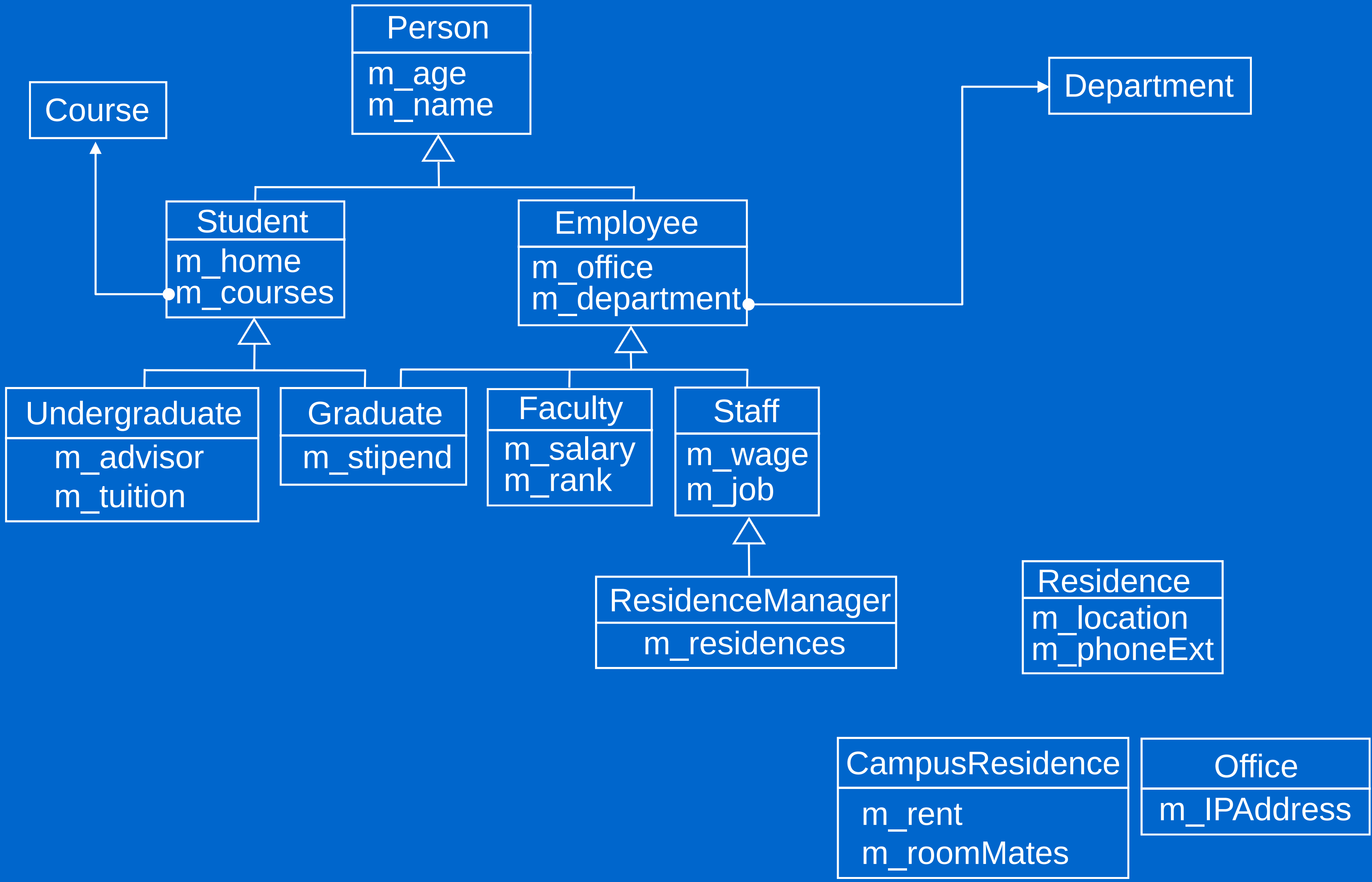


Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
CampusResidence
m_rent
m_roomMates

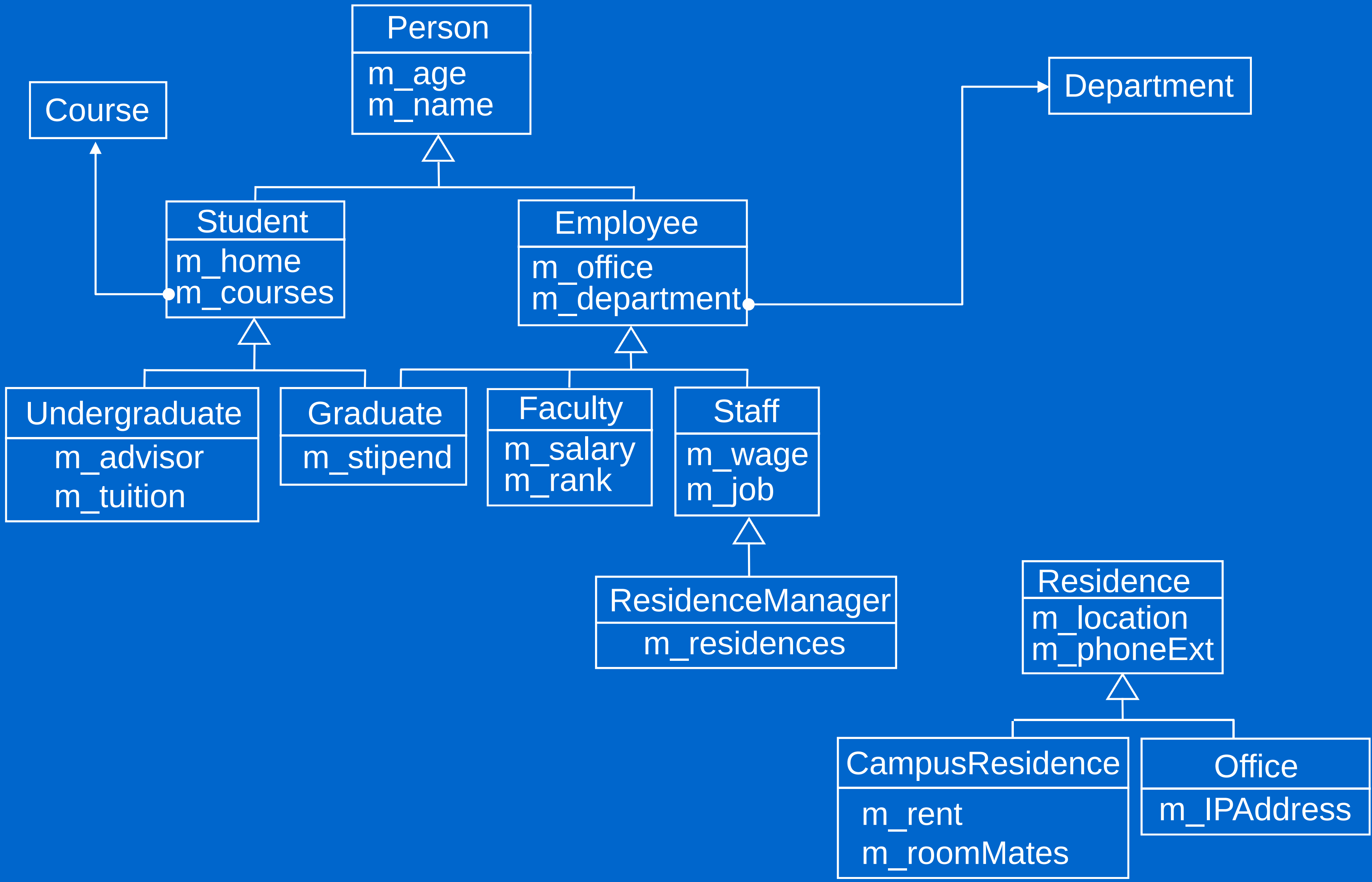
Summary


Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
CampusResidence
m_rent
m_roomMates
Office
m_IPAddress

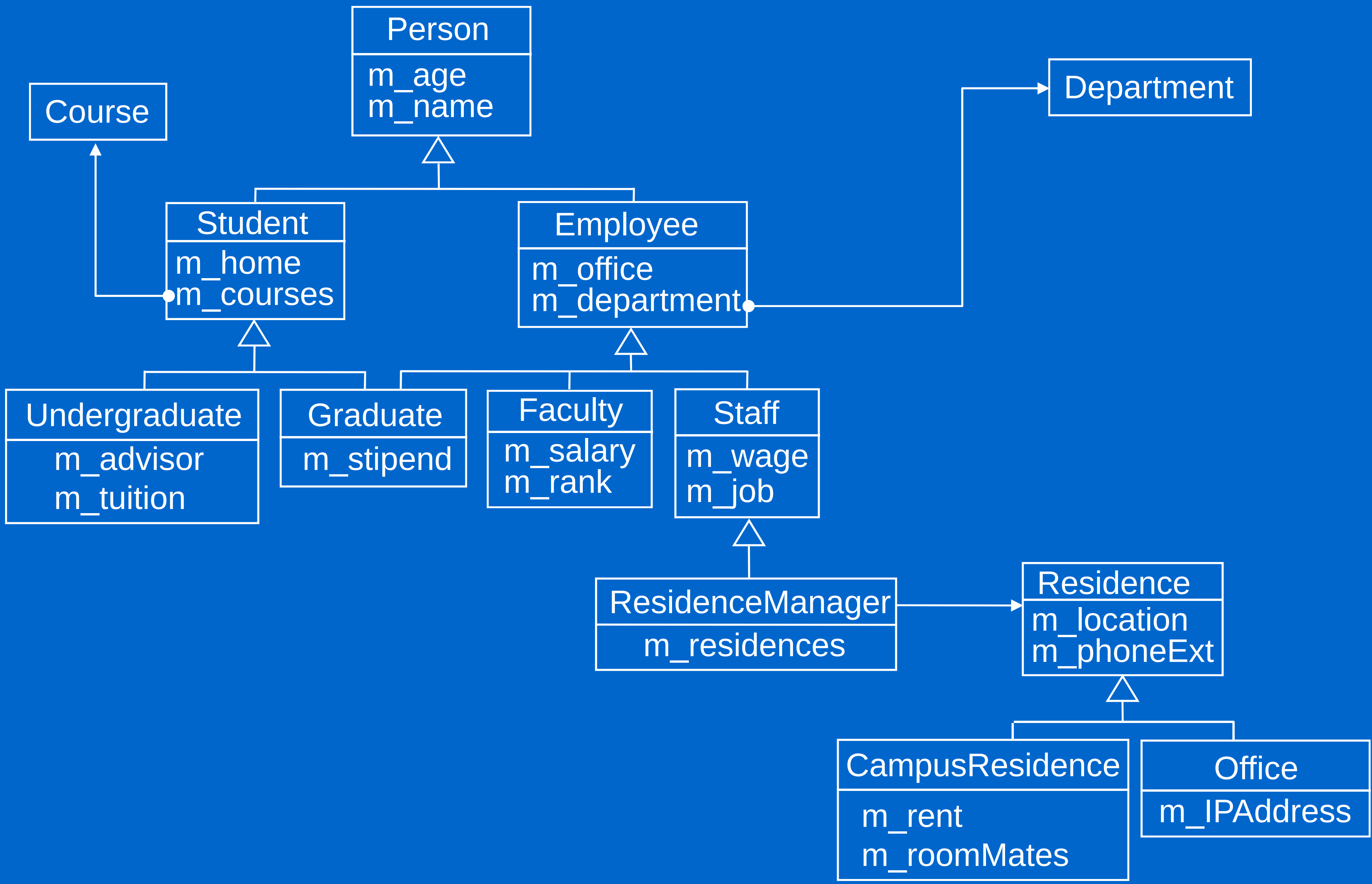
Summary


Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
Residence
m_location
m_phoneExt
CampusResidence
m_rent
m_roomMates
Office
m_IPAddress

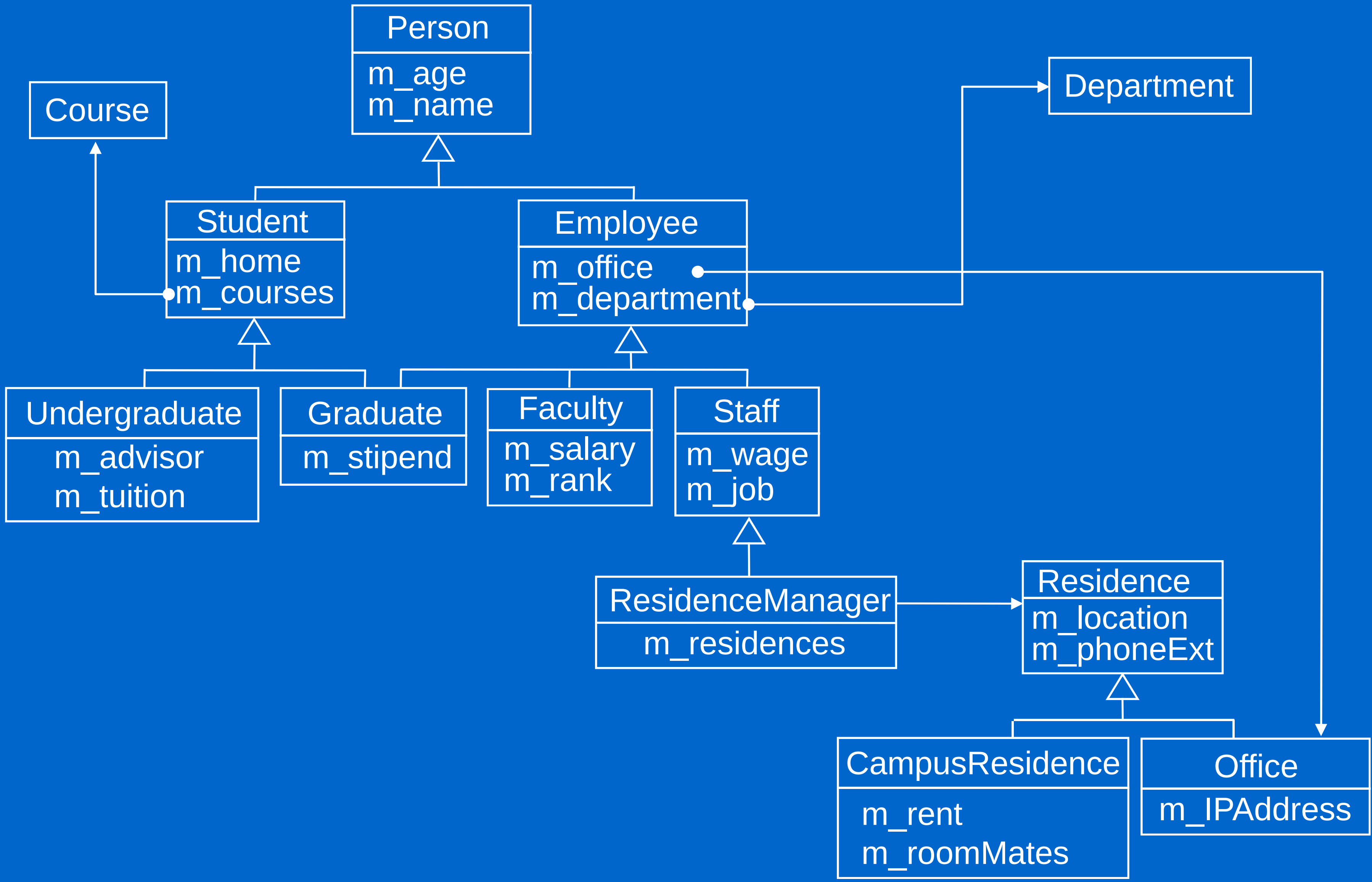
Summary


Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
Residence
m_location
m_phoneExt
CampusResidence
m_rent
m_roomMates
Office
m_IPAddress

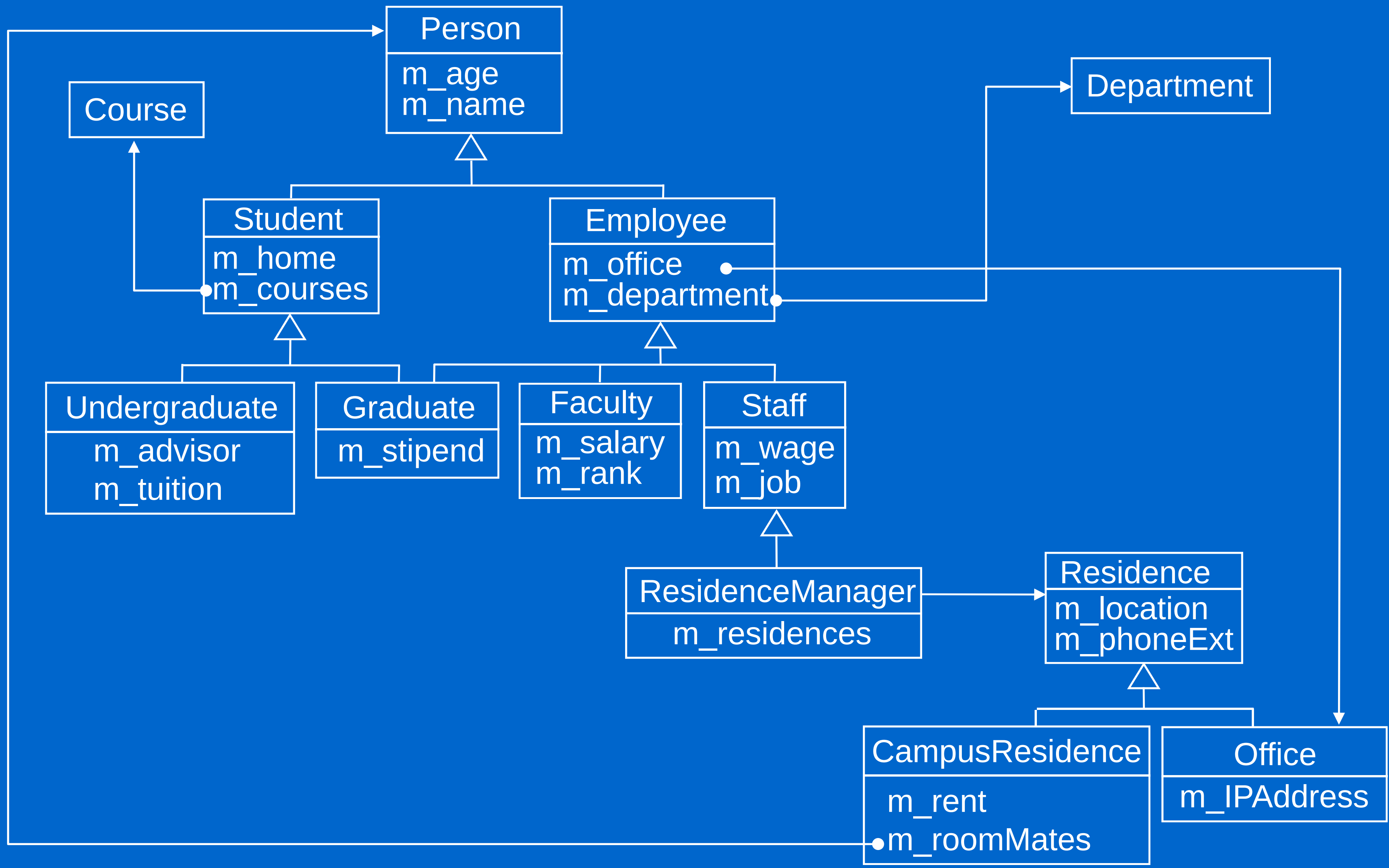
Summary


Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
Residence
m_location
m_phoneExt
CampusResidence
m_rent
m_roomMates
Office
m_IPAddress

Summary

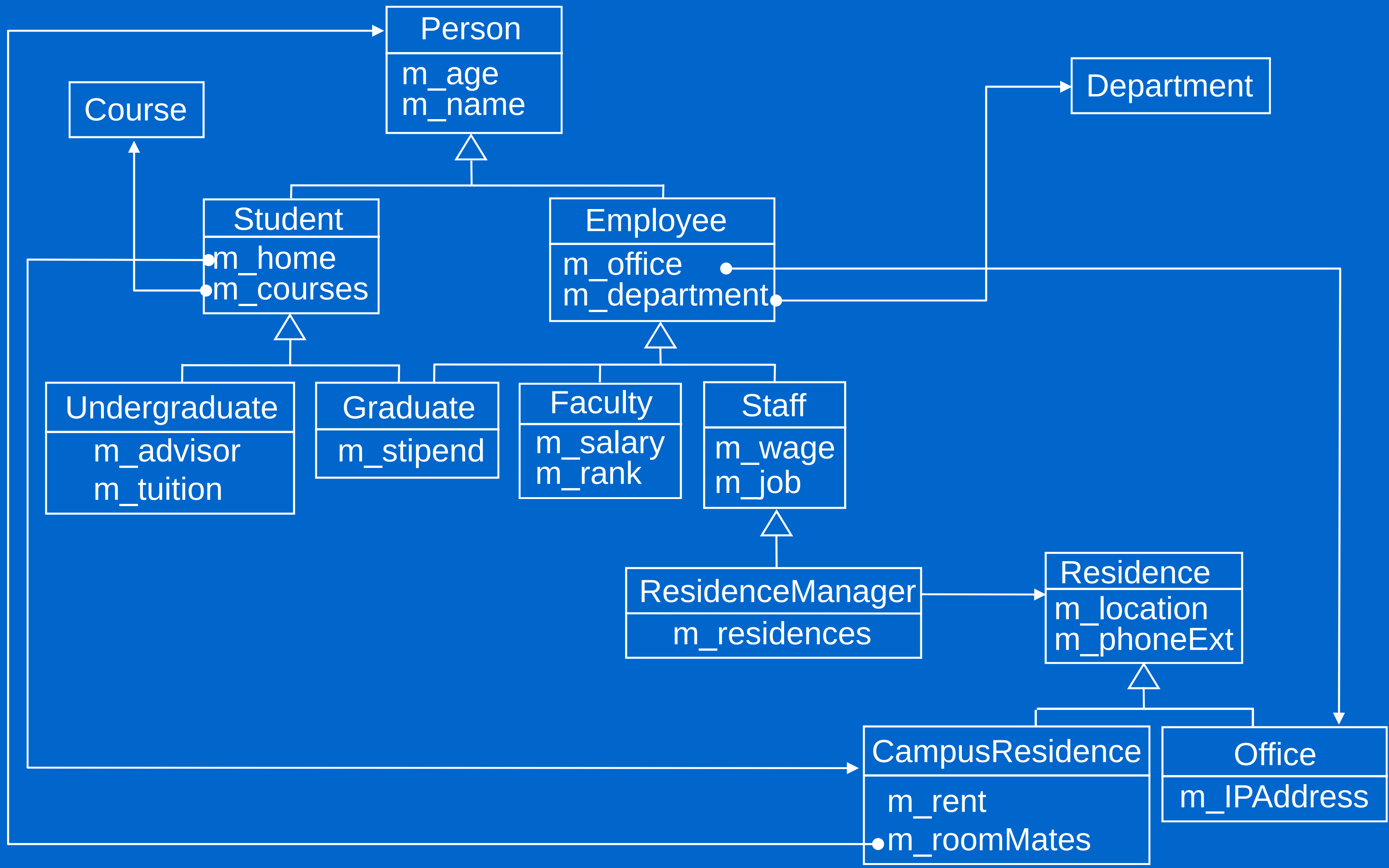

Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
Residence
m_location
m_phoneExt
CampusResidence
m_rent
m_roomMates
Office
m_IPAddress

Summary



Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
Residence
m_location
m_phoneExt
CampusResidence
m_rent
m_roomMates
Office
m_IPAddress

Summary


Person
m_age
m_name
Course
Department
Student
m_home
m_courses
Employee
m_office
m_department
Undergraduate
m_advisor
m_tuition
Graduate
m_stipend
Faculty
m_salary
m_rank
Staff
m_wage
m_job
ResidenceManager
m_residences
Residence
m_location
m_phoneExt
CampusResidence
m_rent
m_roomMates
Office
m_IPAddress