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
- 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 Designs

- 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 to the object-oriented design 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).
- Composition/aggregation and inheritance are the most important two ways to construct object hierarchies.
- In the OOD process, after objects are identified from the problem domain, and attributes and behaviors are modeled with classes in the design 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

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 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

```
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;
};
```

What is the problem with this design?

The Basic Problem: why inheritance

- In the above design
  - Student becomes a general purpose class, a set of attributes and interfaces are used for undergraduate student, while another set of attributes and interfaces are used for graduate student … a form with many <u>redundant fields</u>
  - 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.

A Solution – Separate Classes

```
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;
};
```

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

Why is this still a poor solution?

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

i.e. the common characteristics are not identified

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

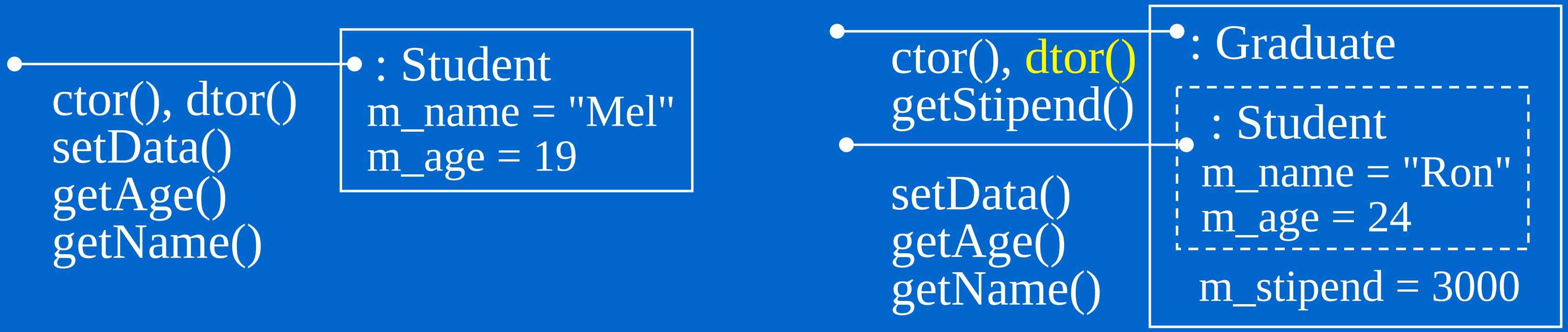
```
Graduate::Graduate(char *name, int age, int stipend) : m_stipend(stipend) {
    setData(name, age); // this is inherited from Student
}
int Graduate::getStipend() const {
    return m_stipend;
}
```

Basic Inheritance (cont’d)

- Usages:

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

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.



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;
}
```

- Private data member of the base class is implicitly declared/defined but is still private to 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;
}
```

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;
};
```

- This 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

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

Basic Inheritance (cont'd)

- Most of the member functions of the base class are implicitly inherited for 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 and dtor would chain automatically to the function defined in the base class.

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 does this happen?

```
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 Constructors (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)~~; // this 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 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) {
    …
}
```

Compiler adds **Base()** invocation automatically

Note:

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

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 Destructors (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:

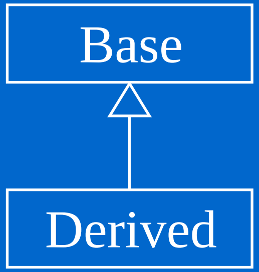
```
In Student ctor
In Graduate ctor
Michael is 24 years old and has a stipend of 6000 dollars.
His address is 8899 Storkes Rd.
In Graduate dtor
In Student dtor
```

chaining

- The compiler automatically calls each dtor when the object exits.
- 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 declare an assignment operator
- If you have a class hierarchy where a class Derived inherits from a class Base.
- There are three cases for the compiler synthesized assignment operators:



  - If both classes do not define assignment operator: both are bit-wise copy
  - If Base& Base::operator=(Base &) is defined and Derived& Derived::operator=(Derived &) is not, then compiler synthesizes

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

  - If you define Derived& Derived::operator=(Derived &rhs) yourself, you have to call Base::operator=(rhs); in Derived::operator=(), otherwise the Base part of the object would not be copied.

Multiple-derived Classes

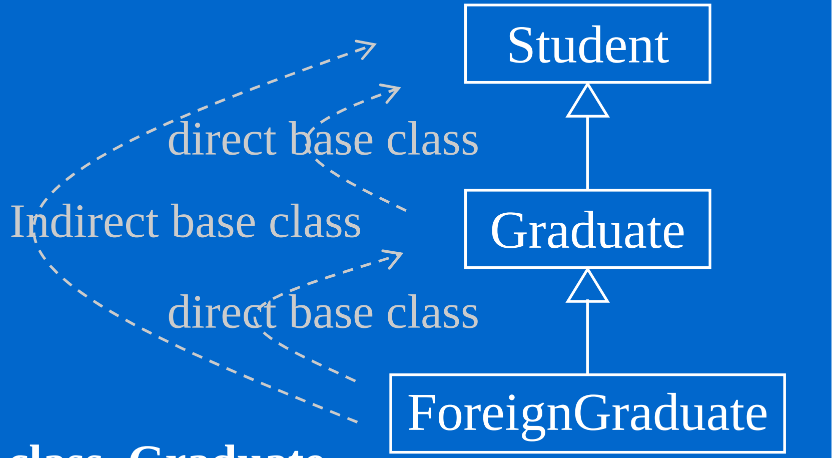
- Let us add a new type of graduate student

```
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;
};
```

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

Multiple-derived Classes (cont’d)



```
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);
}
```

ForeignGraduate invokes the ctor of its direct base class, Graduate

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

Graduate, in turn, invokes the ctor of its direct base class, Student

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

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

Behavior Changing (cont’d)

- Example for the previous definition

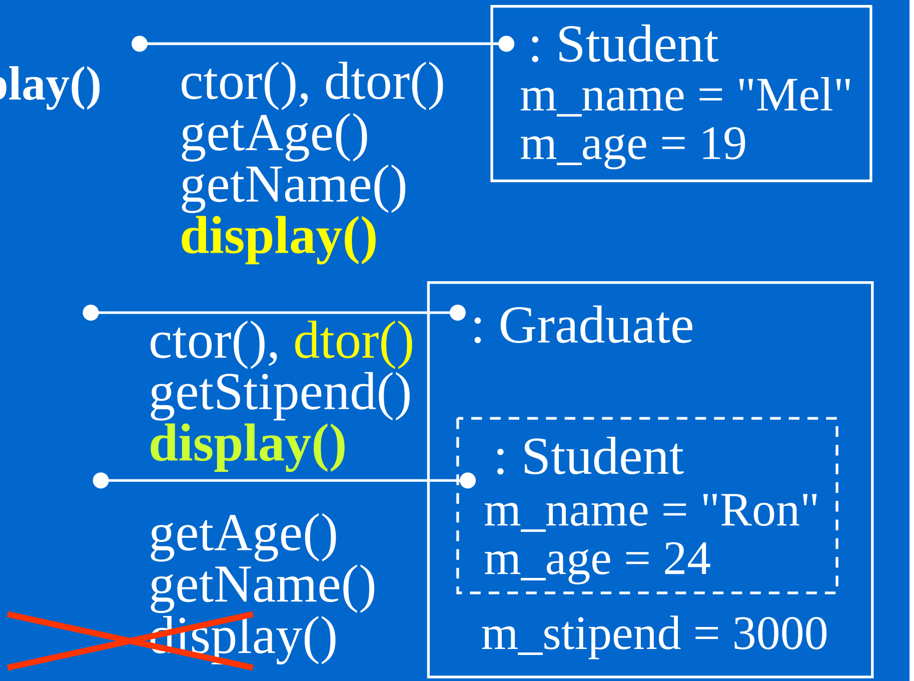
```
Student student1("Alice", 20);
Graduate student2("Michael", 24, 6000, "8899 Storkes Rd.");

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.
```



ctor(), dtor()
getAge()
getName()
display()

: Student
m_name = "Mel"
m_age = 19

ctor(), dtor()
getStipend()
display()

getAge()
getName()
~~display()~~

: Graduate

: Student
m_name = "Ron"
m_age = 24

m_stipend = 3000

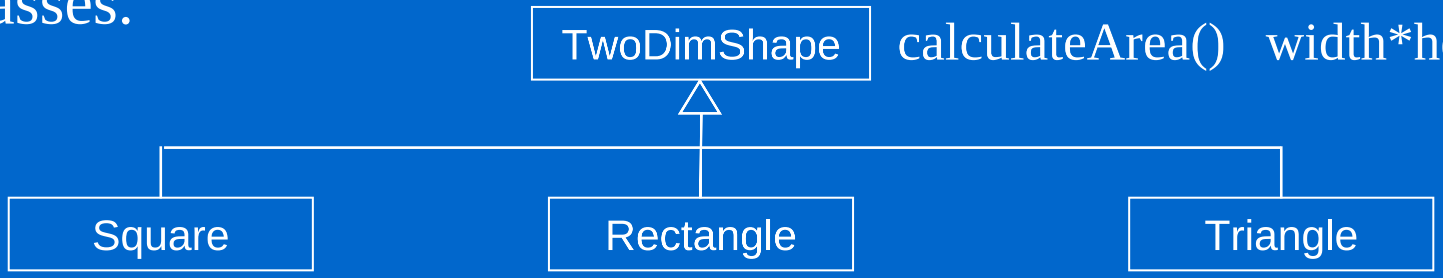
- Note: display() interface usually can enhance the encapsulation, can replace the functionality of accessor functions

Behavior Changing (cont’d)

- You can avoid the redundancy of the common code in the inherited version of display() (to be exactly Student::display()) and Graduate::display() by the following

```
void Graduate::display() const // masks the inherited version of display()
{
   Student::display(); // invoke the inherited function
   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.



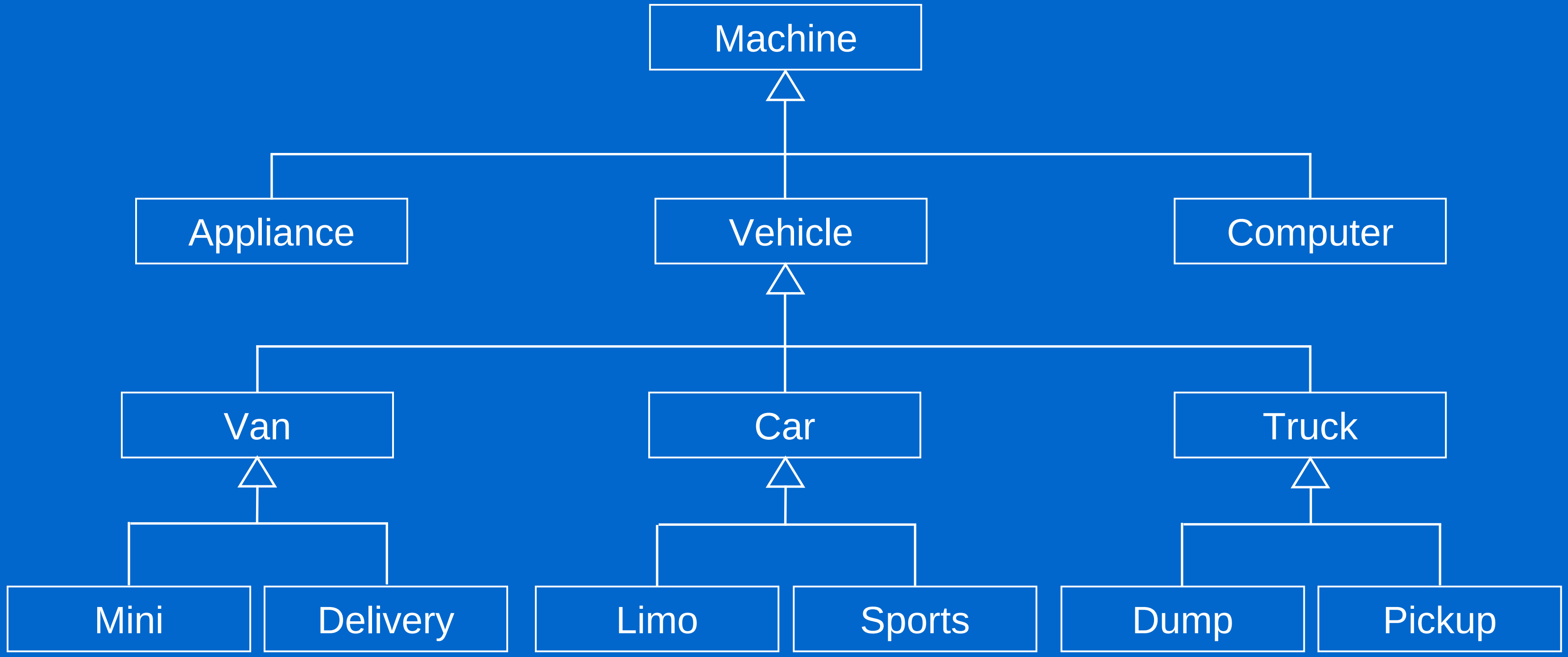
calculateArea() width*height

calculateArea()
1/2*width*height

Class Hierarchy

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

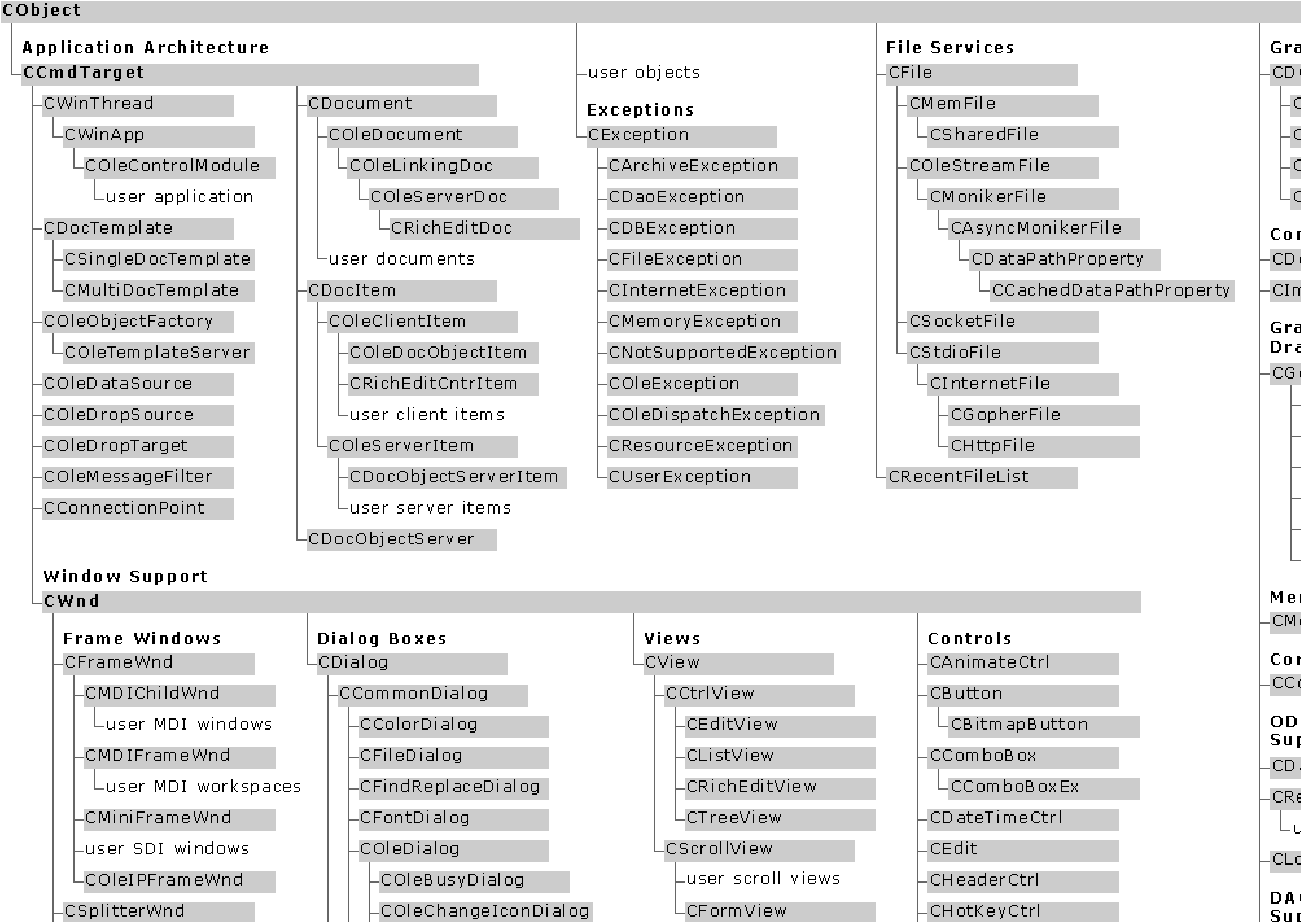
Example



A Real-World Example Of Inheritance

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

Microsoft Foundation Class Library Version 6.0

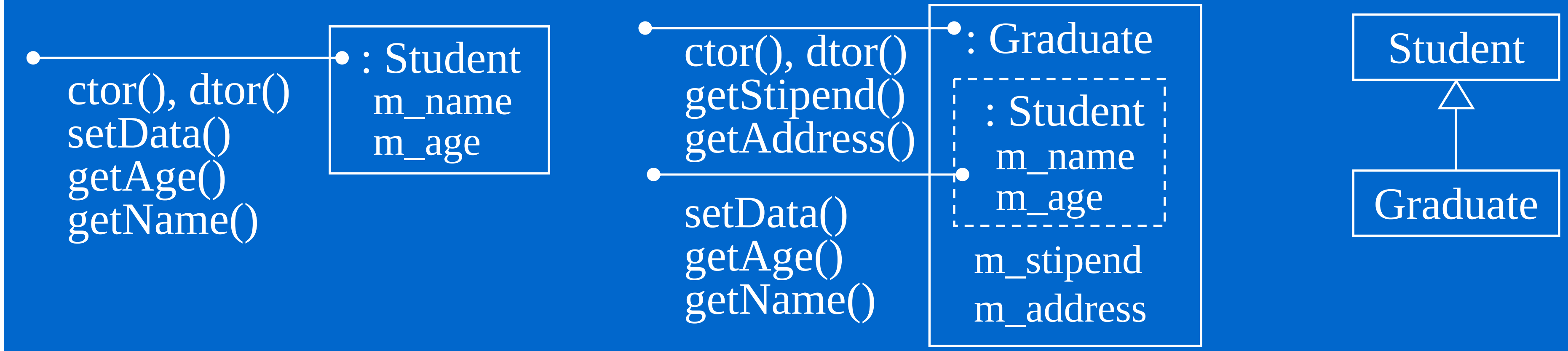


- CObject
  - **Application Architecture**
  - CCmdTarget
    - CWinThread
      - CWinApp
        - COleControlModule
          - user application
    - CDocTemplate
      - CSingleDocTemplate
      - CMultiDocTemplate
    - COleObjectFactory
      - COleTemplateServer
    - COleDataSource
    - COleDropSource
    - 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**
  - CD
    - C
    - C
    - C
    - C
  - **Co**
  - CD
  - CI
  - **Gra Dra**
  - CG
  - **Me**
  - CM
  - **Co**
  - CC
  - **OD Su**
  - CD
  - CR
    - u
  - CL
  - **DA Su**

Inheritance Design

Exploring Solutions to Inheritance

- The University database program



: Student
m_name
m_age

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

: Graduate

: Student
m_name
m_age

m_stipend
m_address

ctor(), dtor()
getStipend()
getAddress()

setData()
getAge()
getName()

Student

Graduate

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

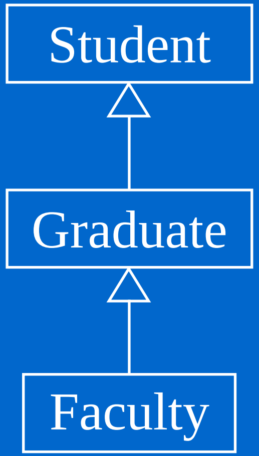
  m_name
  m_age
  m_address
  m_rank
  there is no stipend

- 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

```
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

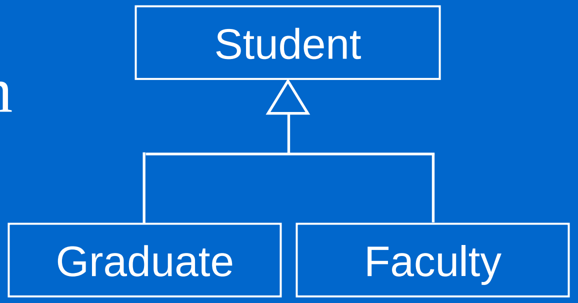
```
Faculty prof("Lin", 40, "#2 Bei-Ning", "Associate Professor");
cout << prof.getStipend();
```

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



```
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;
};
```

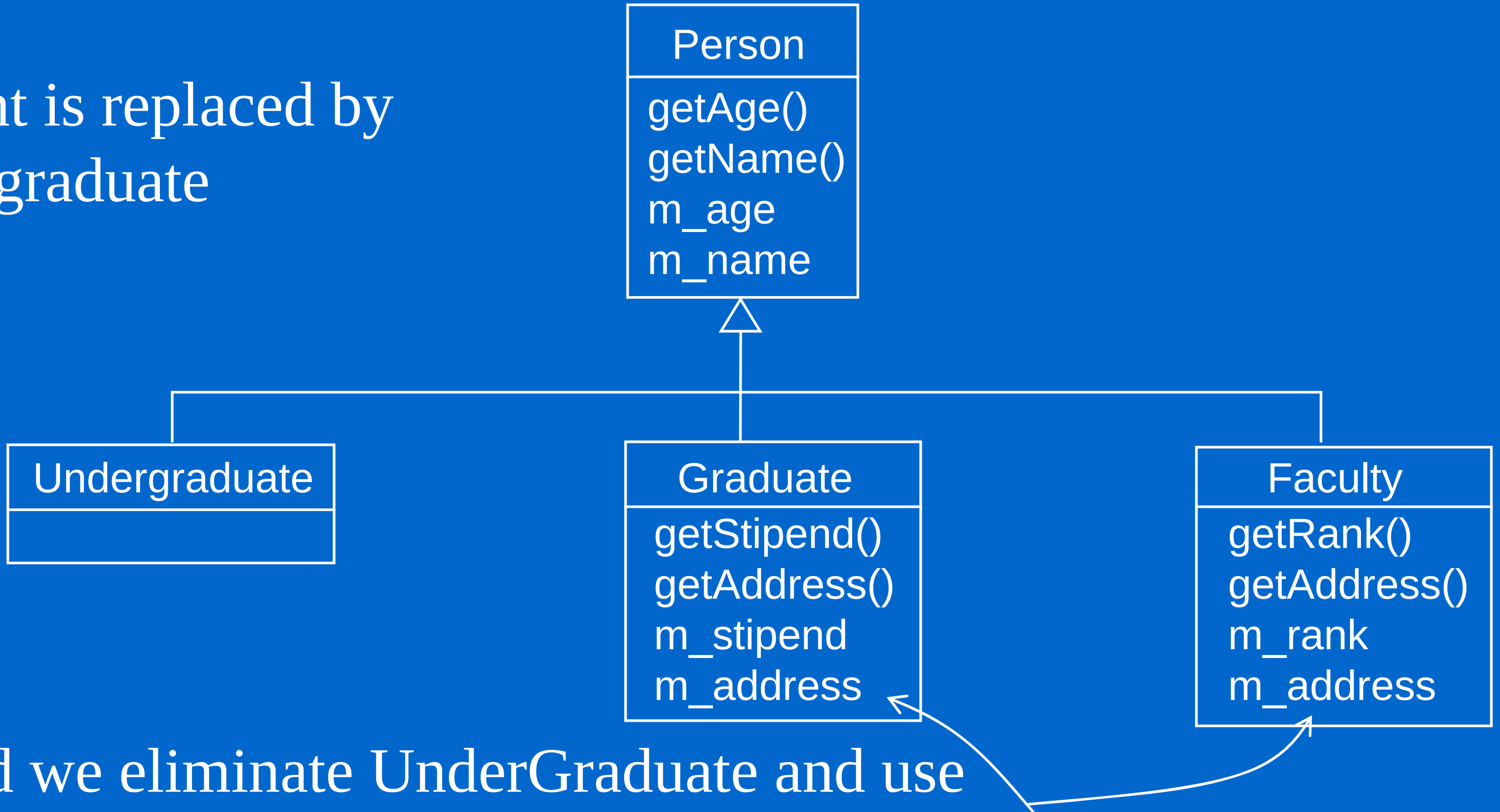
- What is the problem now?
  - Faculty duplicates some codes in Graduate: m_address related
  - What happens if Student adds a field for "undergraduate advisor"? The problem is that Faculty is intrinsically **not** a Student.

"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.

Student is replaced by Undergraduate



Person
getAge()
getName()
m_age
m_name

Undergraduate

Graduate
getStipend()
getAddress()
m_stipend
m_address

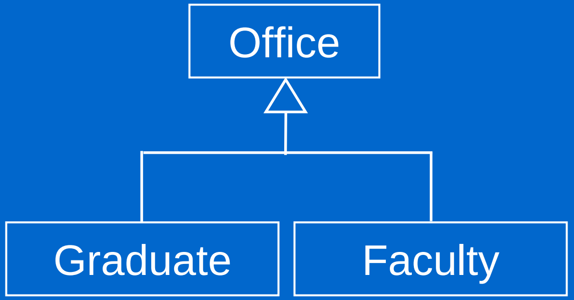
Faculty
getRank()
getAddress()
m_rank
m_address

Is there any redundancy?

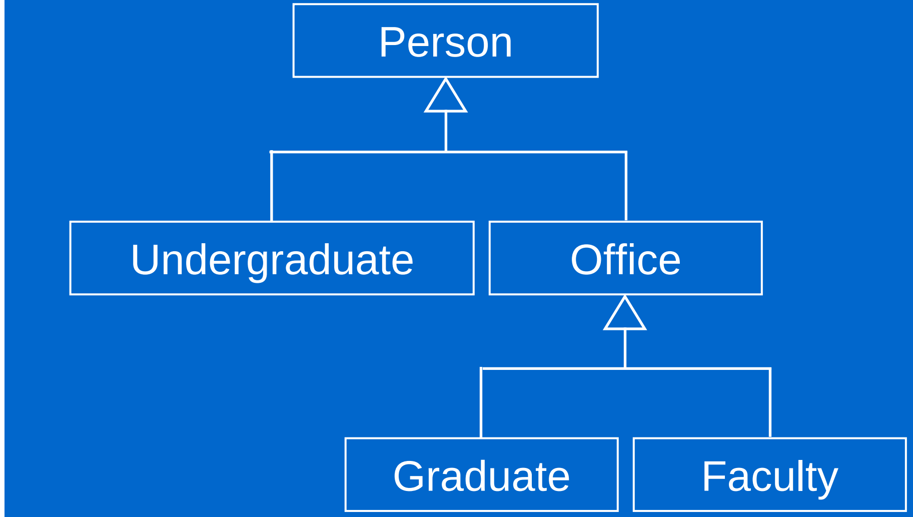
- Should we eliminate UnderGraduate and use only Person in its place?
- Should Graduate be derived from Undergraduate?

Adding an Office class

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

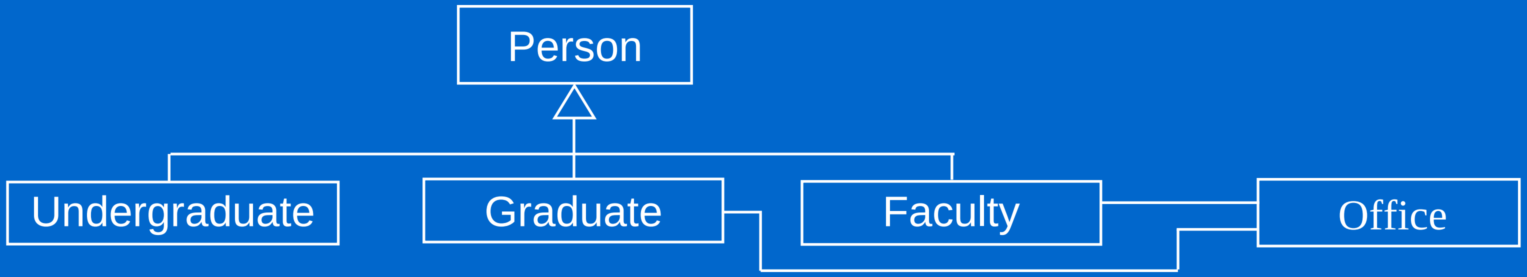
- **If the Office has a clean() method, The Faculty automatically has a clean() method. What does it mean?**
- **What if a faculty has two offices?**

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;
};
            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;
};
```

Final Solution

- Instead of having Graduate and Faculty inherit from Office, we store an Office object within each classes
- Return to our original inheritance design (good design)



Person

Undergraduate

Graduate

Faculty

Office

- The office class exists separately, without regard to 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;
};
```

delegation

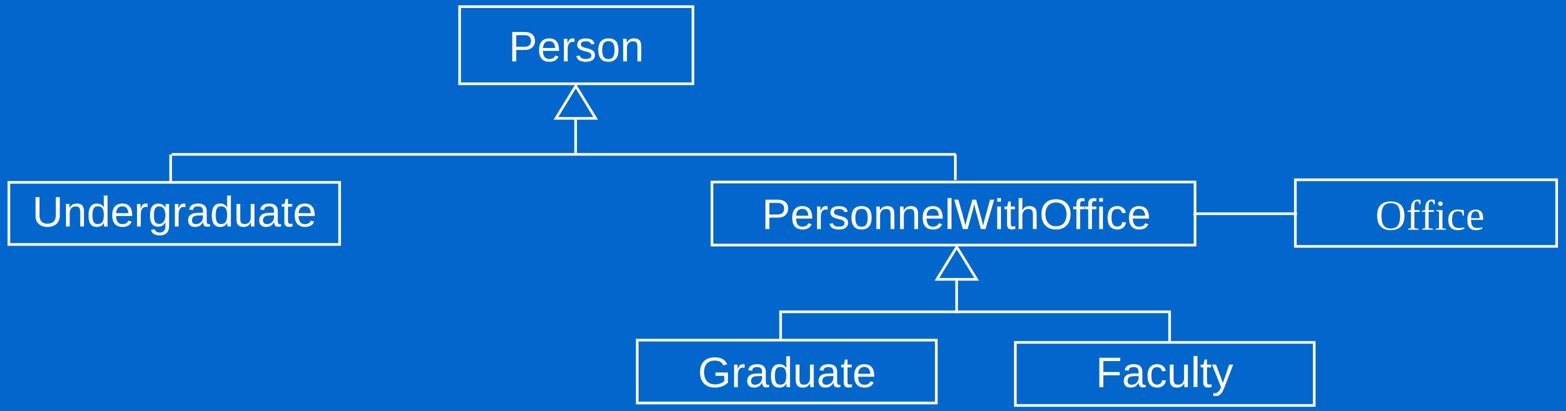
```
const char* Graduate::
        getAddress() const {
  return m_office.getAddress();
}
```

```
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_address in Graduate and Faculty is bound to replicate. However, the code to handling m_address is reduced to a single copy, i.e. Office::getAddress(). If the address has a certain format to follow, the saved codes would be more.

Further Abstraction

- Sometimes, if the relationships between Graduate or Faculty objects and objects of some other classes are uniform, we can model their relationships in the following way



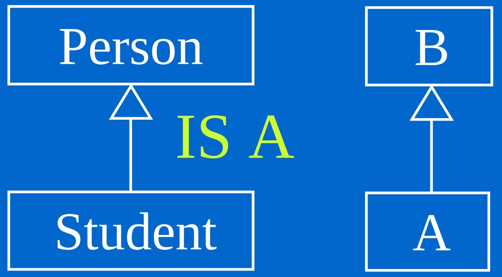
```
class PersonnelWithOffice {
public:
    const char *getAddress() const;
private:
    Office m_office;
};
```

Note: in the above class diagram each Graduate 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;

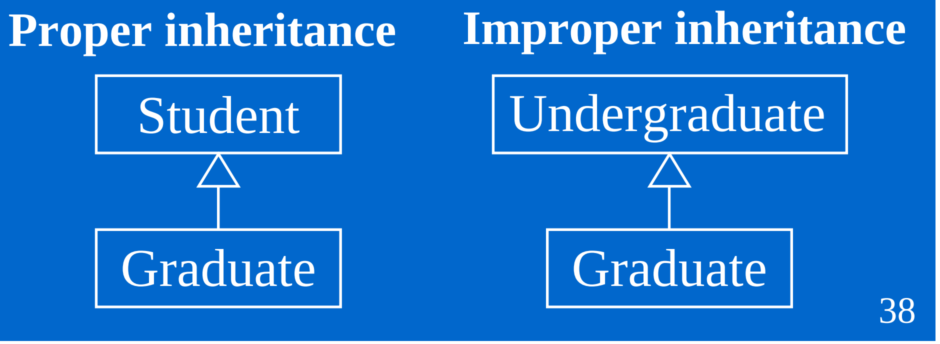
Design Rules for Inheritance

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



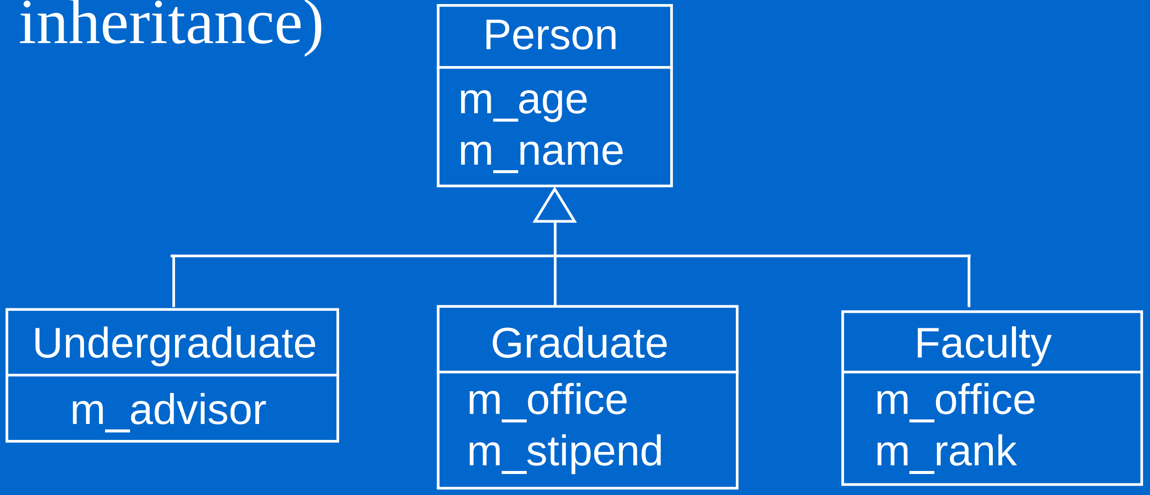
I bet this def is formal but still abstract!!

  - ✶ 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)

- 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)



Person

m_age
m_name

Undergraduate

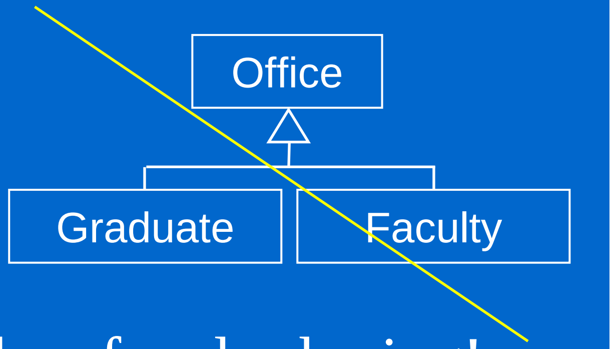
m_advisor

Graduate

m_office
m_stipend

Faculty

m_office
m_rank

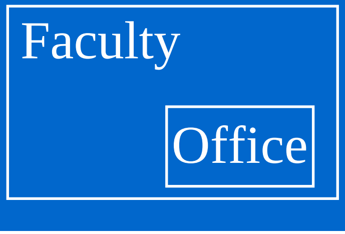


Office

Graduate

Faculty

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



Faculty

Office

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

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

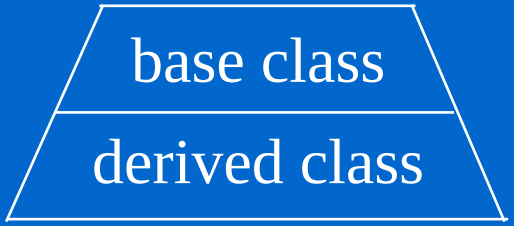
```
void Circle::display() {
   cout << "Center = " << c.x << ", " << c.y
         << "]; Radius = " << radius;
}
```

- Design rationale: A circle is a type of point. The radiuses of some circles are zero. ... Purely mathematical idea!
- 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?

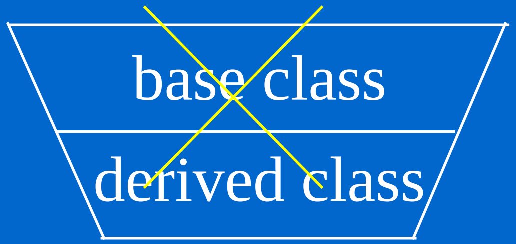
Some Other Dubious Examples

- Ex 1: A stack derived from a linked list. 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. i.e. Stack IS-NOT 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.

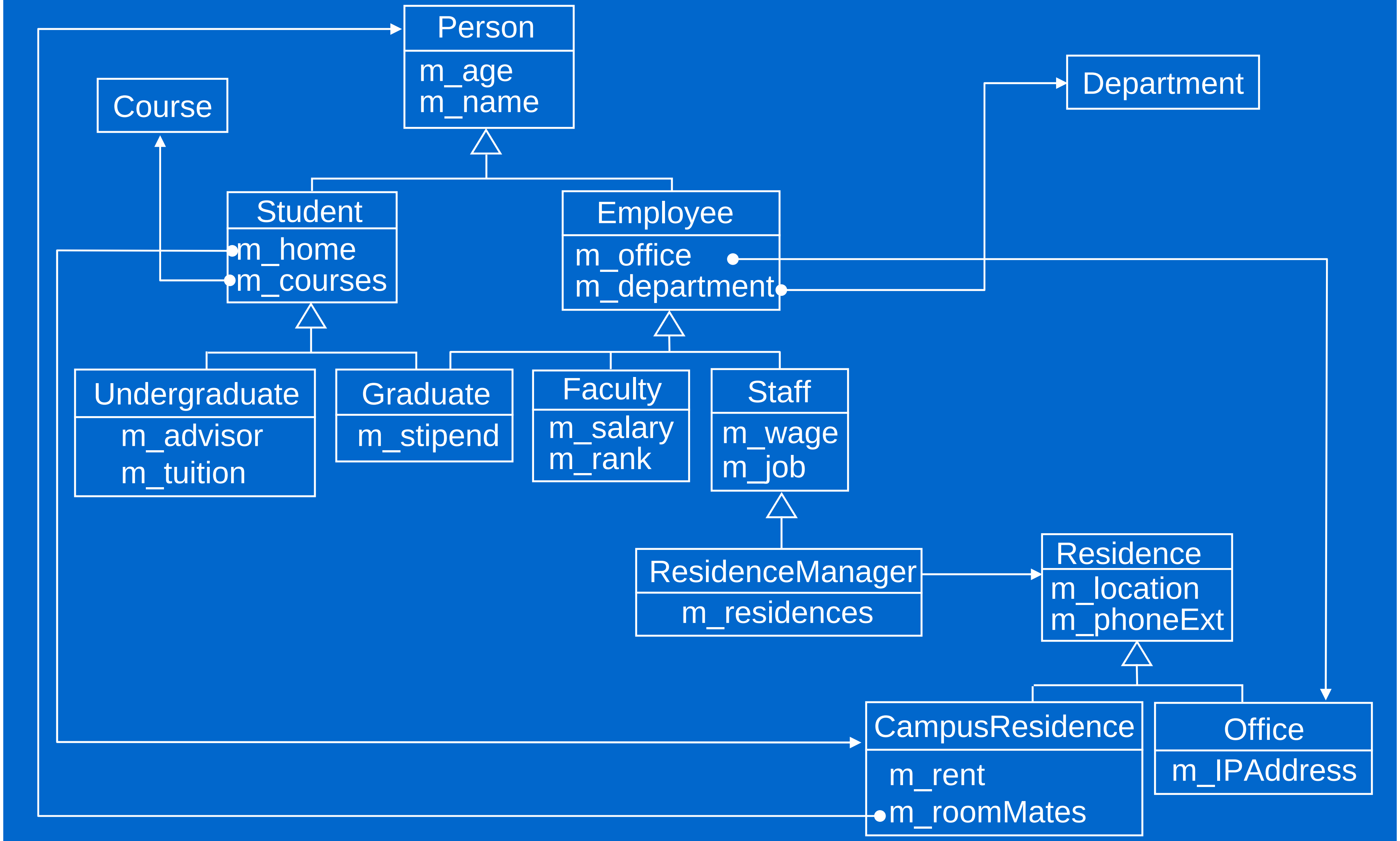

base class

derived class


base class

derived class

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