Object-Oriented Programming: Advanced Inheritance

Lectured by Ming-Te Chi 紀明德

First Semester, 2022

Computer Science Department
National Chengchi University

Slides credited from 李蔡彥 and 廖峻鋒

Advanced Inheritance

- Polymorphism
  - Virtual Function
  - Abstract classes
- Private inheritance
- Multiple inheritance
- Virtual base class

Polymorphism

- Assignment between static base and derived classes
- Assignment between dynamic base and derived classes
- Virtual functions
- Compile-time vs. Run-time binding
- Virtual functions vs. Overloading
- Polymorphism
- Virtual destructors

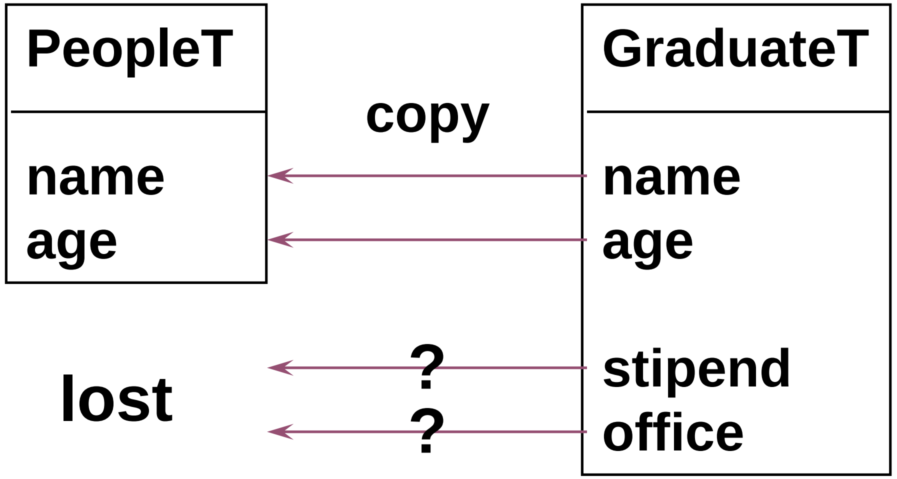
Assignment to Static Base Class From Derived Class

- Though it is unusual to do so, you can assign a static derived object to an object of its (direct/indirect) base class.

```
PeopleT person("Joe", 20);
GraduateT graduateStudent("Tim", 25, 5000, "C250");
person.display();
person = graduateStudent;
person.display();
```

Output:

```
Joe is 20 years old.
Tim is 25 years old.
```

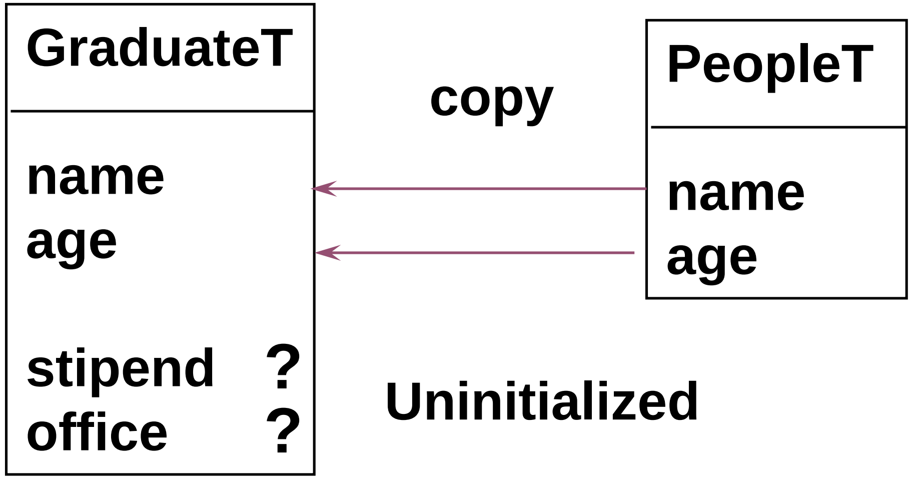


Assignment to Static Derived Class From Base Class

- You can NOT assign a base class object to a derived class object even with explicit type casting.

```
graduateStudent = person;
```

Error: **cannot convert 'PeopleT' to 'GraduateT'**

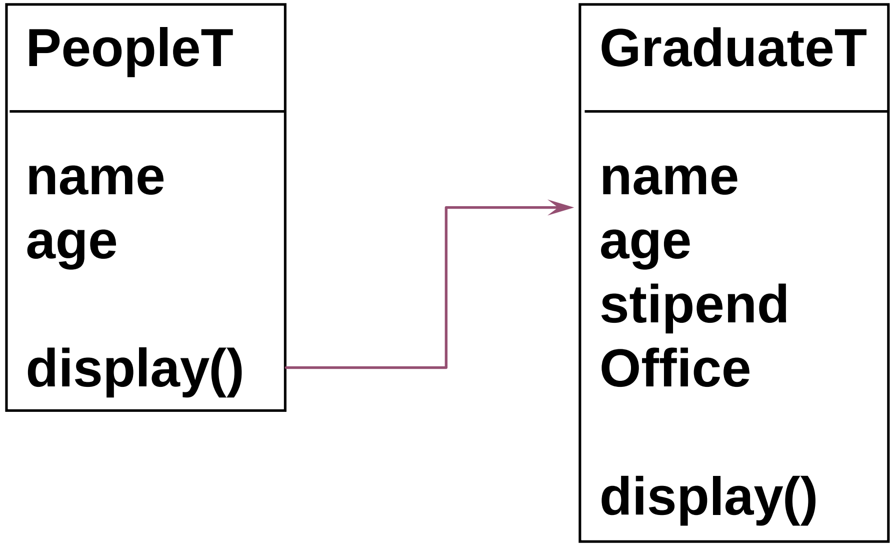


- Why does C++ allow an assignment which lose data but not an assignment that merely causes some fields to be left uninitialized?

Assignment From Derived Class to Dynamic Base Class

- A pointer object of a derive class can be assigned to a pointer of the (direct/indirect) base class.

```
PeopleT *person;
GraduateT *graduateStudent;
person = new PeopleT("Joe", 20);
person->display();
graduateStudent = new GraduateT("Tim", 24, 5000, "C250");
person = graduateStudent;
person->display();
```

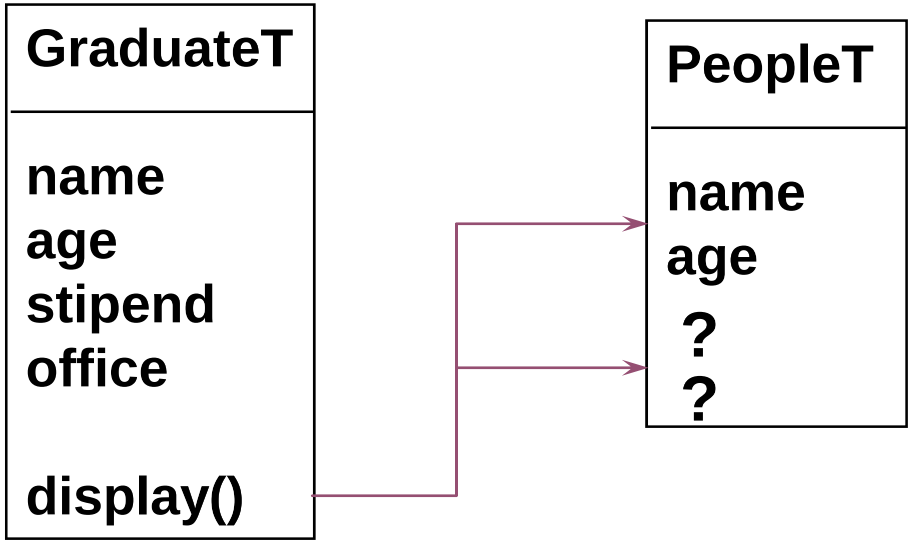


Note:
person.display() calls the function in class PersonT, but reference the data in graduateStudent.
The data are always present.

Assignment From Base Class to Dynamic Derived Class

- You can assign the pointer of a base object to the pointer of a derived object. An explicit cast is required.

```
graduateStudent = (GraduateT *) person;
```



- This will work only if the pointer dereferences data and functions also contained with the base class.

Making Use of Assignments

- You can declaring an array of base pointers to store different derived object.

```
PeopleT *database[3];
UnderGraduateT *undergradStudent;
GraduateT *graduateStudent;
FacultyT *prof;

undergradStudent = new UnderGraduateT("John", 18);
graduateStudent = new GraduateT("Tim", 24, 5000, "C124");
prof = new FacultyT("Li", 31, "C4","associate professor");

database[0] = undergradStudent;
database[1] = graduateStudent;
database[2] = prof;

database[0]->display();
database[1]->display();
database[2]->display();
```

Output:

```
John is 18 years old.
Tim is 24 years old.
Li is 31 years old.
```

Note: In all cases the display function in the base class is called.

Virtual Functions

```
class PeopleT {
  public:
    PeopleT();
    PeopleT(char *inName, int inAge);
    ~PeopleT();
    virtual void display() const;
  private:
    char *name;
    int age;
    int getAge() const;
    char *getName() const;
};

// the code here is the same
database[0]->display();
database[1]->display();
database[2]->display();
```

Output:

```
John is 18 years old.
He is an undergraduate.

Tim is 24 years old.
He is a graduate student.
He has a stipend of 5000 dollars.
His office is c124.

Li is 31 years old.
His address is c4.
His rank is associate professor.
```

- In run-time, it examines the type of the *object* referenced by the pointer and calls the function for that object.

Virtual背後的原理

- 當類別中有任一方法被宣告為virtual時，compiler會偷偷將一個隱藏的void*放到類別(及子類別)中

```
class OneVirtual {
  int a;
  void* VPTR;    (偷放的)
public:
  virtual  void x() {}
  int        i()  { return 1; }
};
```

(see Size.cpp)

```
class TwoVirtuals {
  int a;
  void* VPTR;  ←
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};
```

(既使是有二個virtual，也只會偷放一個VPTR!)

typeinfo

```
#include <iostream>
#include <typeinfo>

struct Base { virtual ~Base() = default; };
struct Derived : Base {};

int main() {
    Base b1;
    Derived d1;

    const Base *pb = &b1;
    std::cout << typeid(*pb).name() << '\n';
    pb = &d1;
    std::cout << typeid(*pb).name() << '\n';
}
```

Output:

```
John is 18 years old.
Tim is 24 years old.
Li is 31 years old.
```

Virtual Functions

- By definition, static objects are non-virtual.
- The keyword *virtual* must *not* be used in the function definition, only in the declaration.
- The keyword virtual is not required in any derived class, direct or indirect. But it is a good style to include the keyword for clarity.
- **Compile-time binding** (static binding): non-virtual functions are determined in compile-time.
- **Run-time binding** (dynamic binding): virtual functions are called on the basis of the object the pointer references, which can be changed in run-time.
- Virtual functions prototypes must match exactly; otherwise, it reverts to compile-time binding.

Polymorphism

- Polymorphism is the ability to do different things in different contexts.
- C++ implements polymorphism in three ways
  - Overloading - one name can stand for several functions.
  - Templates - one name can stand for several types.
  - Virtual functions - one function can take on several forms depending on the underlying object referred to in a pointer.
- Drawbacks to virtual functions: less efficient

Abstract Classes

```
class ShapeT {
  public:
    virtual void draw() const = 0;
};
class CircleT : public ShapeT {
  public:
    void draw() const;
  private:
    ...
};
class LineT : public ShapeT {
  public:
    void draw() const;
  private:
    ...
};
class RectangleT: public ShapeT {
  public:
    ...
  private:
    ...
};
```

Error: **must declare draw()**

Note: draw() is called *pure virtual function* and ShapeT() is called *abstract class.*

```
ShapeT *shape[3];
shape[0] = new CircleT();
shape[1] = new LineT();
shape[2] = new ShapeT();
```

Error: illegal use of abstract class ('ShapeT::draw() const')

Destructor problem

- delete p的時候實際上會變成一個 undefined behavior。
- 有可能發生的事情包括異常終止、memory leak、只 call 了 base class 的 destructor 而沒有 call derived class 的 destructor。

```
class ShapeT {
   public:
     virtual void draw() const = 0;

};

class CircleT : public ShapeT {
   public:
     void draw() const;

   private:
     ...
};
```

```
void user(ShapeT* p)
{
     p->draw();
     // ...
     delete p;   //undefine behavior
}
```

Virtual Destructor

- Base and derived classes may each have destructors.
- To ensure that the destructors for the derived classes are called before the base class, most destructors need to be virtual.
- e.g. `virtual ~Based();`

```
class ShapeT {
  public:
    virtual void draw() const = 0;
    virtual ~ShapeT();
};

class CircleT : public ShapeT {
  public:
    void draw() const;
    ~CircleT();     //override ~ShapeT()
  private:
    ...
};
```

```
void user(ShapeT* p)
{
    p->draw();
    // ...
    delete p;  //revoke the right dctor
}
```

```
class base {
  public:
    base()
    { cout << "Constructing base\n"; }
    ~base()
    { cout<< "Destructing base\n"; }
};


class derived: public base {
  public:
    derived()
     { cout << "Constructing derived\n"; }
    ~derived()
       { cout << "Destructing derived\n"; }
};


int main()
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

```
Constructing base
Constructing derived
Destructing base
```

[code]

```
class base {
  public:
    base()
    { cout << "Constructing base\n"; }
    virtual ~base()
    { cout << "Destructing base\n"; }
};


class derived : public base {
  public:
    derived()
    { cout << "Constructing derived\n"; }
    virtual ~derived()
    { cout << "Destructing derived\n"; }
};


int main()
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

Private Inheritance (1)

- Private inheritance is sometimes called a REUSE-A relationship and is equivalent to a HAS-A relationship.
- Advantages of private inheritance over HAS-A relationship:
  - The derived class can override functions in the base class.
  - The functions from the base class can be used directly.
- The HAS-A relationship is considered a better style.

Private Inheritance (2)

- You can adjust the status of members in derived classes.

```
class PeopleT {
  public:
   void Foo() const;
};
class GraduateT: private PeopleT {
  public:
   PeopleT::Foo(); // make it public
  private:
   ...
};
class ForeignGraduateT: public GraduateT { ... }
```

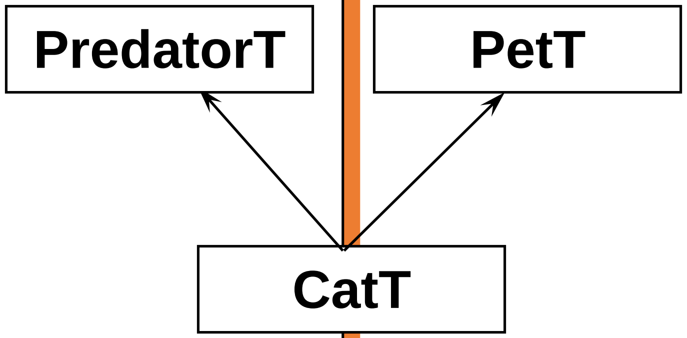
Note: ForeignGraduateT can access Foo() now.

Multiple Inheritance

- Objects may have an IS-A relationship to *more than one* class.

```
class PredatorT {
  public:
    PredatorT(char *inPrey, char *inHabitat);
    ~PredatorT();
    char *getPrey() const;
    char *getHabitat() const;
  private:
    char *prey;
    char *habitat;
};
class PetT {
  public:
    PetT(char *inName, char *inHabitat);
    ~PetT();
    char *getName() const;
    char *getHabitat() const;
  private:
    char *name;
    char *habitat;
};
```

A cat is a predator and is a pet as well.



Multiple Inheritance Continued

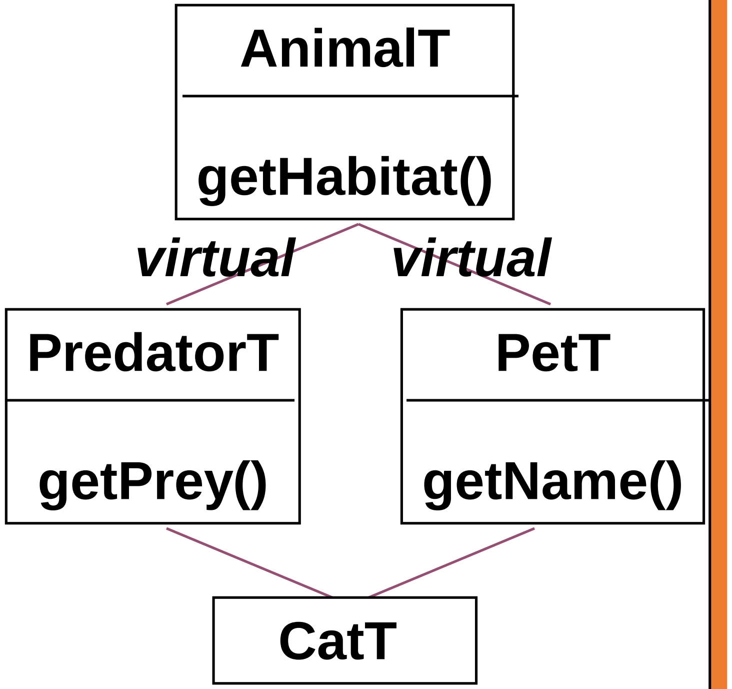
```
class CatT: public PredatorT, public PetT {
  public:
    CatT(char *inName, char *inPrey, char *inHabitat);
    void reduceLives();
    int getLives() const;
  private:
    int lives;
};
CatT::CatT(char *inName, char *inPrey, char *inHabitat)
    : PredatorT(inPrey, inHabitat), PetT(inName, inHabitat),
      lives(9) {
};
int main() {
    CatT cat("Shiba", "mice", "indoors");
    cout << cat.getHabitat();
    cout << cat.PetT::getHabitat(); // OK
}
```

Error: ambiguous access to class member

- Both PredatorT and PetT have a `habitat` field. We can create a AnimalT where they can derive from to avoid the duplication.

Improving the Multiple Inheritance

```
class AnimalT {
  public:
    AnimalT(char *inHabitat);
    ~AnimalT();
    char *getHabitat() const;
  private:
    char *habitat;
};
class PredatorT: public virtual AnimalT{
    ...
};
class PetT: public virtual AnimalT {
    ...
};
class CatT: public PredatorT, public PetT{
    ...
};
CatT::CatT(char *inName, char *inPrey, char *inHabitat)
    :AnimalT(inHabitat), PredatorT(inPrey, inHabitat),
     PetT(inName, inHabitat), lives(9) {
}
cout << cat.getHabitat(); // OK now
```



a must

Why is inheritance in OOP considered bad?

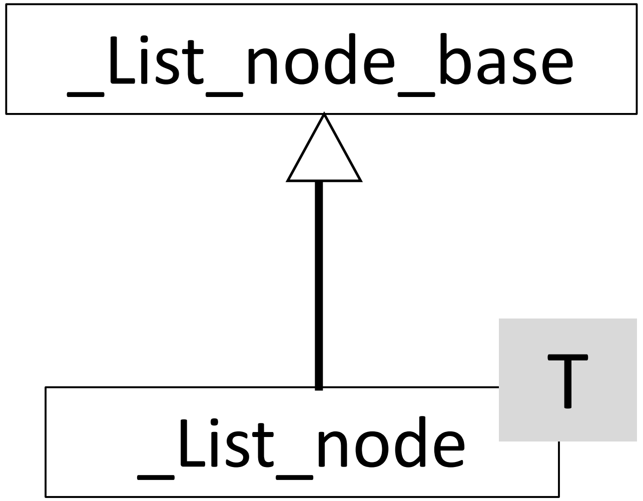
Recap OOP

- Inheritance
- Composition
- Delegation

Inheritance, is-a

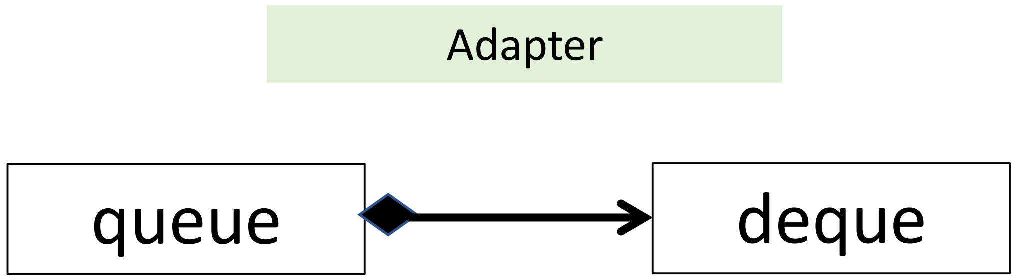
```
struct _List_node_base {
   _List_node_base* _M_next;
   _List_node_base* _M_prev;
 };

Template<typename _Tp>
Struct _List_node
 : public _List_node_base
{
  _Tp _M_data;
};
```



Composition, has-a

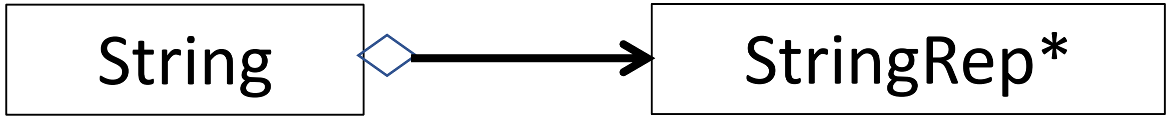
```
template<typename T>
class queue {
  …
protected:
  deque<T> c;
public:
   bool empty() const;
   void push(const value_type& x) { c.push_back(x); }
   void pop() { c.pop_front(); }
};
```



Delegation(委託). Composition by reference

```
class String {
 public:
   String();
   ~String();
   …
private:
   StringRep* rep;  //pimpl
};
```

Handle/Body



```
class StringRep {
friend class string;
   StringRep();
   ~StringRep();
   int count;
   char* rep;
};
```