



COMP2012 Object-Oriented Programming and Data Structures

Topic 3: Inheritance

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China



Example: University Admin Info



Let's implement a system for maintaining university administrative information.

- **Teacher** and **Student** are two completely **separate** classes.
- Their implementation uses **separate** code.
- However, **some** of their data members and member functions are implemented in the **same** way: name and department, and their handling member functions.
- Why would we implement the **same** function twice?
- That is **not** good **re-use** of software!

Example: U. Admin Info — Student Class

```
/* File: student1.h */
enum Department { CBME, CIVL, CSE, ECE, IELM, MAE };

class Student
{
    private:
        string name;
        Department dept;
        float GPA;
        Course* enrolled;
        int num_courses;

    public:
        Student(string n, Department d, float x) :
            name(n), dept(d), GPA(x), enrolled(nullptr), num_courses(0) { }
        string get_name() const;
        Department get_department() const;
        float get_GPA() const;
        bool add_course(const Course& c);
        bool drop_course(const Course& c);
};
```

Example: U. Admin Info — Teacher Class

```
/* File: teacher1.h */  
enum Department { CBME, CIVL, CSE, ECE, IELM, MAE };  
enum Rank { PROFESSOR, DEAN, PRESIDENT };  
  
class Teacher  
{  
    private:  
        string name;  
        Department dept;  
        Rank rank;  
        string research_area;  
  
    public:  
        Teacher(string n, Department d, Rank r, string a) :  
            name(n), dept(d), rank(r), research_area(a) { }  
        string get_name() const;  
        Department get_department() const;  
        Rank get_rank() const;  
        string get_research_area() const;  
};
```

Things to Consider



- We want a way to say that **Student** and **Teacher** both have the same members: **name**, **dept**, but yet require them to keep a separate copy of these members.
- We want to share the code for **get_name** etc. between **Student** and **Teacher** as well.
- However, objects have states, and their consistency should be maintained when the objects' member functions are called — so we cannot just write global functions to do it.

Solution#1: Re-use by Copying

Copy the **code** from one class to the other class, and change the class names.



- This is very **error prone**.
- It is also a **maintenance nightmare**.
 - ▶ What if we find a **bug** in the code in one class?
 - ▶ What if we want to **improve** the code? Perhaps we introduce a new member **address**.
- “**Re-use by copying**” is a bad idea!

Part I

What is Inheritance?



Solution#2: By Inheritance

Idea: Find out the **common** data members and member functions of **Student** and **Teacher** and put them into a **parent class**, called **UPerson** here, and apply the **inheritance** mechanism.

```
/* File: student1.h */
enum Department { CBME, CIVL, CSE, ECE, IELM,
MAE };

class Student
{
private:
    string name;
    Department dept;
    float GPA;
    Course* enrolled;
    int num_courses;

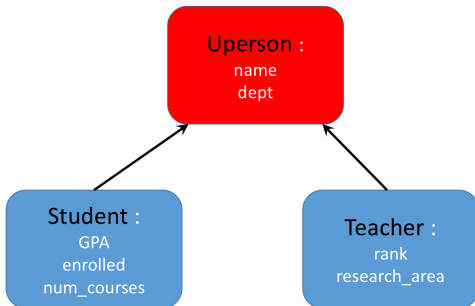
public:
    Student(string n, Department d, float x) :
        name(n), dept(d), GPA(x),
        enrolled(nullptr), num_courses(0) { }
    string get_name() const;
    Department get_department() const;
    float get_GPA() const;
    bool add_course(const Course& c);
    bool drop_course(const Course& c);
};
```

```
/* File: teacher1.h */
enum Department { CBME, CIVL, CSE, ECE, IELM,
MAE };
enum Rank { PROFESSOR, DEAN, PRESIDENT };

class Teacher
{
private:
    string name;
    Department dept;
    Rank rank;
    string research_area;

public:
    Teacher(string n, Department d, Rank r,
string a) :
        name(n), dept(d), rank(r),
        research_area(a) { }
    string get_name() const;
    Department get_department() const;
    Rank get_rank() const;
    string get_research_area() const;
};
```


Solution#2: Inheritance — Base Class + Derived Classes



(Note: only the data members are shown in each class.)

Solution#2: By Inheritance — UPerson Class

```
#ifndef UPERSON_H          /* File: uperson.h */
#define UPERSON_H

enum Department { CBME, CIVL, CSE, ECE, IELM, MAE };
class UPerson
{
private:
    string name;
    Department dept;

public:
    UPerson(string n, Department d) : name(n), dept(d) { }
    string get_name() const { return name; }
    Department get_department() const { return dept; }
};

#endif
```

Solution#2: By Inheritance — Student Class

```
#ifndef STUDENT_H          /* File: student.h */
#define STUDENT_H

#include "uperson.h"      // Don't forget your parents!!
class Course { /* incomplete */ };

class Student : public UPerson // Public inheritance
{
    private:
        float GPA;
        Course* enrolled;
        int num_courses;

    public:
        Student(string n, Department d, float x) :
            UPerson(n, d), GPA(x), enrolled(nullptr), num_courses(0) { }
        float get_GPA() const { return GPA; }
        bool enroll_course(const string& c) { /* incomplete */ };
        bool drop_course(const Course& c) { /* incomplete */ };
};
#endif
```

Solution#2: By Inheritance — Teacher Class

```
#ifndef TEACHER_H          /* File: teacher.h */
#define TEACHER_H

#include "uperson.h"       // Don't forget your parents!!
enum Rank { PROFESSOR, DEAN, PRESIDENT };

class Teacher : public UPerson // Public inheritance
{
private:
    Rank rank;
    string research_area;

public:
    Teacher(string n, Department d, Rank r, string a) :
        UPerson(n, d), rank(r), research_area(a) { }
    Rank get_rank() const { return rank; }
    string get_research_area() const { return research_area; }
};

#endif
```

Inheritance

- **Inheritance** is the ability to define a **new** class based on an **existing** class with a **hierarchy**.
- The **derived class inherits** data members and member functions of the **base class**.
- **New** members and functions are added to the **derived class**.
- The **new** class only has to implement the behavior that is **extra** to the **base class**, and **the code** of the **base class** can be **re-used** in the **derived class**.
- In this example, **UPerson** is the **base class**, and **Student** and **Teacher** are the **derived classes**.
- **Student** and **Teacher** **inherit** all data members and functions from **UPerson**.
- E.g., data members of **Student** include the data members of **UPerson** {name, dept}, **plus** the extra data members declared in **Student's** definition {GPA, enrolled, num_courses}.
- **Inheritance** enables **code re-use**.

Example: Inherited Members and Functions

```
#include <iostream>      /* File: inherited-fcn.cpp */
using namespace std;
#include "student.h"

void some_func(UPerson& uperson, Student& student) {
    cout << uperson.get_name() << endl;
    Department dept = uperson.get_department();
    // Error! Base class object can't call derived class's function
    uperson.enroll_course("COMP1001");

    // Derived class object may call base class's member function
    cout << student.get_name() << endl;
    // Derived class object calls its own member functions
    cout << student.get_GPA() << endl;
    student.enroll_course("COMP2012");
}

int main() {
    UPerson abby("Abby", CBME);
    Student bob("Bob", CIVL, 3.0);
    some_func(abby, bob);
}
```

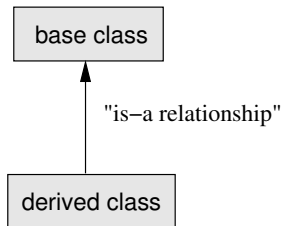
Polymorphic or Liskov Substitution Principle

Inheritance implements the **is-a relationship**.

- Since **Student** **inherits** from **UPerson**,
 - ▶ A **Student** object can be treated like a **UPerson** object.
 - ▶ All member functions of **UPerson** can be called by a **Student** object.
- In other words, a **Student** object **is a** **UPerson** object.
- In general, an object of the **derived class** can be treated **like** an object of the **base class** under all circumstances.

If class **D** (a **derived class**) **inherits** from class **B** (the **base class**):

- Every **D** object is also a **B** object, but **not vice-versa**.
- **B** is a more **general** concept; **D** is a more **specific** concept.
- Wherever a **B** object is needed, a **D** object can be used instead.



Polymorphic or Liskov Substitution Principle ..

By **deriving** **Student** and **Teacher** classes are from **UPerson** class, we have:

- Since a **Student/Teacher** object **is** also a **UPerson** object, any functions defined on **UPerson** objects may also be called by **any Student and Teacher** objects.
- Obviously, functions defined on **UPerson** objects can only make use of **UPerson's** data/functions; they can't use data/functions of **UPerson's derived classes** which are **not** known yet at the time of their (**UPerson**) creation!

| Function Expecting an Argument of Type | Will Also Accept |
|---|----------------------------------|
| UPerson | Student |
| pointer to UPerson | pointer to Student |
| UPerson reference | Student reference |

Example: Derived Objects Treated as Base Class Objects

```
#include <iostream>      /* File: print-label.cpp */
using namespace std;
#include "student.h"
#include "teacher.h"

void print_label(const UPerson& uperson)
{
    cout << "Name: " << uperson.get_name() << endl;
    cout << "Dept: " << uperson.get_department() << endl;
}

int main()
{
    Student tom("Tom", CIVL, 3.9);
    print_label(tom);    // Tom is also a UPerson

    Teacher alan("Alan Turing", CSE, PROFESSOR, "AI");
    print_label(alan);   // Alan is also a UPerson
    return 0;
}
```

Example: Derived Objects Treated as Base Class Objects ..

```
#include <iostream>      /* File: print-label2.cpp */
using namespace std;
#include "student.h"

void print_label(const UPerson* uperson) {
    cout << "Name: " << uperson->get_name() << endl;
    cout << "Dept: " << uperson->get_department() << endl;
}

void print_label(const UPerson& uperson) {
    cout << "Name: " << uperson.get_name() << endl;
    cout << "Dept: " << uperson.get_department() << endl;
}

void print_label(const Student& student) {
    cout << "Name: " << student.get_name() << endl;
    cout << "Dept: " << student.get_department() << endl;
    cout << "GPA: " << student.get_GPA() << endl;
}

int main() { // Which print_label()?
    Student tom("Tom", CIVL, 3.9); print_label(tom);
    UPerson& tom2 = tom; print_label(tom2);
    UPerson* p = &tom; print_label(p);
}
```

Quiz: Derived Objects Treated as Base Class Objects ..

```
#include <iostream>      /* File: substitute.cpp */
using namespace std;
#include "student.h"

int main() {
    void dance(const UPerson& p); // Anyone can dance
    void dance(const UPerson* p); // Anyone can dance
    void study(const Student& s); // Only students study
    void study(const Student* s); // Only students study
    UPerson p("P", IELM); Student s("S", MAE, 3.3);

    // Which of the following statements can compile?
    dance(p);
    dance(s);
    dance(&p);
    dance(&s);
    study(s);
    study(p);
    study(&s);
    study(&p);
}
```

Extending Class Hierarchy

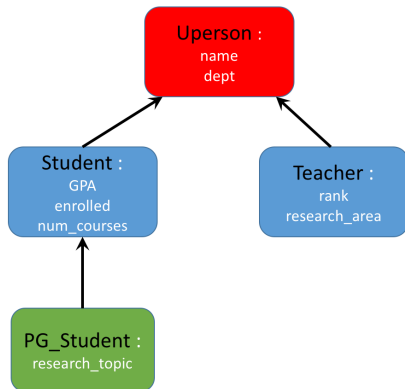
We can easily **add** classes to our **existing** class hierarchy of **UPerson**, **Student**, and **Teacher**.

- **New** classes can immediately benefit from all functions that are available to their **base classes**.
- e.g., `void print_label(const UPerson& person)` will work immediately for a new class called **PG_Student**, even though this type of objects was **unknown** when `print_label()` was designed and written.
- In fact, it is **not** even necessary to recompile the existing code: It is enough to **link** the new class with the object codes of **UPerson** and `print_label()`.
- Advanced use: **Link** in new objects while the code is running!

Direct and Indirect Inheritance

Let's add a new class **PG_Student** to the hierarchy.

- **PG_Student** is **directly derived** from **Student**.
- It is **indirectly derived** from **UPerson**.
- So a **PG_Student** object **is** also a **UPerson** object.
- **UPerson** is called an **indirect base class** of **PG_Student**.



Direct and Indirect Inheritance — PG_Student Class

```
#ifndef PG_STUDENT_H      /* File: pg-student.h */
#define PG_STUDENT_H

#include "student.h"

class PG_Student : public Student
{
private:
    string research_topic;

public:
    PG_Student(string n, Department d, float x) :
        Student(n, d, x), research_topic("") { }

    string get_topic() const { return research_topic; }
    void set_topic(const string& x) { research_topic = x; }
};

#endif
```

Example: Indirect Inheritance

- Let's promote Tom to **PG_Student**.
- Can Tom still use the **print_label()** function?

```
#include <iostream>      /* File: pg-print-label.cpp */
using namespace std;
#include "pg-student.h" // Change student.h to pg-student.h

void print_label(const UPerson& uperson)
{
    cout << "Name: " << uperson.get_name() << endl;
    cout << "Dept: " << uperson.get_department() << endl;
}

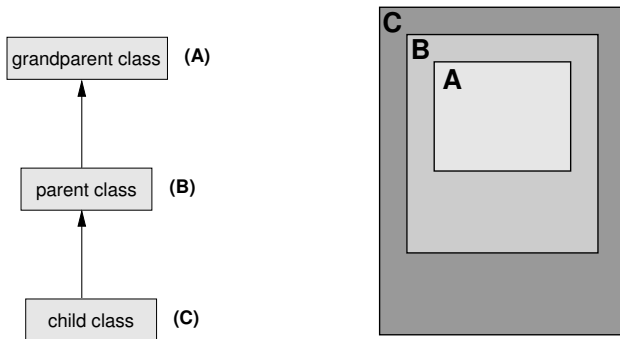
int main()
{
    PG_Student tom("Tom", CIVL, 3.9); // Tom is now a PG Student
    print_label(tom);                 // Tom is also a UPerson
    return 0;
}
```

Part II

Initialization of Classes in an Inheritance Hierarchy



Initialization of Base Class Objects



- If class C is **derived** from class B which is in turn **derived** from class A, then C will **contain** data members of both B and A.
- Class C's constructor can only call class B's constructor, and class B's constructor can only call class A's constructor.
- It is the **responsibility** of each **derived class** to **initialize** its **direct base class** correctly.

Initialization of Base Class Objects by Initializers

- Before a **Student** object can come into existence, we have to create its **UPerson** parent first.
- **Student**'s **constructors** have to call a **UPerson**'s **constructor** through the **member initializer list**.

```
Student::Student(string n, Department d, float x) :  
    UPerson(n,d), GPA(x), enrolled(NULL), num_courses(0) { }
```

- Similarly, **PG_Student** has to create its **Student** part **before** it can be created.
- But, it does **not** need to create its **UPerson** part **directly** by calling **UPerson**'s **constructor**.
- In fact, its **UPerson** part should have been created by **Student**.

```
PG_Student::PG_Student(string n, Department d, float x) :  
    Student(n, d, x), research_topic("") { }
```

Order of Cons/Destruction: Student w/ an Address

```
#include <iostream>      /* File: init-order.cpp */
using namespace std;

class Address {
public:
    Address() { cout << "Address's constructor" << endl; }
    ~Address() { cout << "Address's destructor" << endl; }
};

class UPerson {
public:
    UPerson() { cout << "UPerson's constructor" << endl; }
    ~UPerson() { cout << "UPerson's destructor" << endl; }
};

class Student : public UPerson {
public:
    Student() { cout << "Student's constructor" << endl; }
    ~Student() { cout << "Student's destructor" << endl; }

private: Address address;
};

int main() { Student x; return 0; }
```

Order of Cons/Destruction: Student w/ an Address ..

UPerson's constructor
Address's constructor
Student's constructor
Student's destructor
Address's destructor
UPerson's destructor

That is, the order is construction of a class object

1. its parent
2. its data members
(in the order of their appearance in the class definition)
3. itself

Order of Cons/Destruction: Move Address to UPerson

```
#include <iostream>      /* File: init-order2.cpp */
using namespace std;

class Address {
public:
    Address() { cout << "Address's constructor" << endl; }
    ~Address() { cout << "Address's destructor" << endl; }
};

class UPerson {
public:
    UPerson() { cout << "UPerson's constructor" << endl; }
    ~UPerson() { cout << "UPerson's destructor" << endl; }
private: Address address;
};

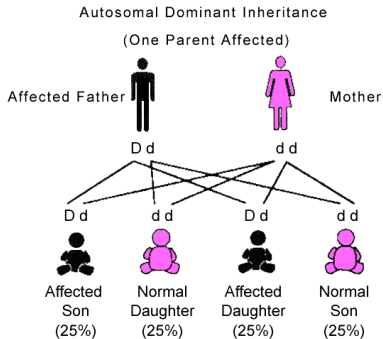
class Student : public UPerson {
public:
    Student() { cout << "Student's constructor" << endl; }
    ~Student() { cout << "Student's destructor" << endl; }
};

int main() { Student x; return 0; }
```

Question: What is the output now?

Part III

Some Problems of Inheritance



Problem #1: Slicing

- An **assignment** from a **derived class** object to a **base class** object results in “**slicing**”.
- This is rarely desirable.
- Once **slicing** has happened, there is no trace of the fact that we started with a **derived class**.

```
#include <iostream>          /* File: slice.cpp */
#include <string>
using namespace std;
#include "../basics/uperson.h"
#include "../basics/student.h"

int main()
{
    Student student("Snoopy", CSE, 3.5);
    UPerson* pp = &student;
    UPerson* pp2 = new Student("Mickey", ECE, 3.4);

    UPerson uperson("Unknown", CIVL);
    uperson = student; // What does "uperson" have?
    return 0;
}
```

Problem #2: Name Conflicts I

```
#include <iostream> /* File: name-conflict.cpp */
using namespace std;

void print(int x, int y) { cout << x << " , " << y << '\n'; }

class B
{
    private:
        int x, y;
    public:
        B(int p = 1, int q = 2) : x(p), y(q)
            { cout << "Base class constructor: "; print(x, y); }
        void f() const { cout << "Base class: "; print(x, y); }
};

class D : public B
{
    private:
        float x, y;
    public:
```


Problem #2: Name Conflicts II

```
D() : x(10.0), y(20.0) { cout << "Derived class constructor\n"; }  
void f() const { cout << "Derived class: "; print(x, y); B::f(); }  
};  
  
void smart(const B* z) { cout << "Inside smart(): "; z->f(); }  
  
int main()  
{  
    B base(5, 6); cout << endl;  
    D derive; cout << endl;  
  
    B* b = &base; b->f(); cout << endl;  
    D* d = &derive; d->f(); cout << endl;  
  
    b = &derive; b->f(); cout << endl;  
  
    smart(b); cout << endl;  
    smart(d); cout << endl;  
    return 0;  
}
```

Problem #2: Name Conflicts Output

Base class constructor: 5 , 6

Base class constructor: 1 , 2

Derived class constructor

Base class: 5 , 6

Derived class: 10 , 20

Base class: 1 , 2

Base class: 1 , 2

Inside smart(): Base class: 1 , 2

Inside smart(): Base class: 1 , 2

Problem #3: Bad Design

- Let's design a **Bird** class.

```
class Bird                                /* File: bird.h */
{
    ...
public:
    void hatch_eggs();    // Birds lay eggs
    void lay_egg(int n);
    void spread_wings(); // Birds have wings
    void fly();           // Birds can fly
    int altitude() const; // Return current altitude
};
```

- We can re-use **Bird** to implement some special cases:

```
class Swallow : public Bird { ... };
class Eagle : public Bird { public: void hunt(Bird *prey); };
```

Example: Derive a Penguin from a Bird

Now we need a penguin object, and we would like to re-use all the code we have for hatching and laying eggs, spreading wings, etc.

```
class Penguin : public Bird /* File: penguin1.h */
{
    ...
public:
    ...
    void swim();
    void catch_fish();
};
```

Oops! Penguins cannot fly!
What can we do?



Example: Derive a Penguin from a Bird ..

```
void Penguin::fly()      /* File: penguin1.cpp */
{
    cerr << "Penguins cannot fly!" << endl;
    exit(999);
}
```

- Some people try to solve the problem like above.
- But this doesn't **really** say “Penguins cannot fly”.
It says: “Penguins can fly, but they are forbidden!”

Example: Derive a Penguin from a Bird ...

- Some people try to solve the problem like this:
Penguins can fly, but the altitude is zero.

```
class Penguin : public Bird /* File: penguin2.h */
{
    ...
public:
    ...
    void swim();
    void catch_fish();
    void fly() { } // Penguin can't fly
    int altitude() const { return 0; } // Always zero
};
```

Penguin Example: What's Wrong?

```
void find_food(Bird *b) /* File: penguin-wrong.cpp */
{
    b->fly();           // Visibility decreases with altitude
    double visibility = 10.0 / b->altitude();
    ...
}
```

- Declaring **Penguin** as a **derived class** of **Bird** **violates** the **substitution principle**.
- It is not possible to use a **Penguin** in **some** functions that are supposed to work for all **Birds**.
- The only solution is: **RE-DESIGN!**

Summary

- Behavior and structure of the **base class** is **inherited** by the **derived class**.
- However, **constructors** and **destructor** are an **exception**. They are **never** inherited.
- There is a kind of contract between a **base class** and a **derived class**:
 - ▶ The **base class** provides functionality and structure (member functions and data members).
 - ▶ The **derived class** guarantees that the **base class** is initialized in a **consistent state** by calling an appropriate constructor.
- A **base class** is **constructed before** the **derived class**.
- A **base class** is **deconstructed after** the **derived class**.

Part IV

Access Control: public, protected, private



Example: Add `print()` to UPerson/Student Class

```
#include "uperson.h"    /* File: print1.cpp */
#include "student.h"

class UPerson { public: void print() const; ... };

class Student: public UPerson { public: void print() const; ... };

void UPerson::print() const
{
    cout << "--- UPerson details ---" << endl;
    cout << "Name: " << name << endl << "\nDept: " << dept << endl;
}

void Student::print() const
{
    cout << "--- Student details ---" << endl
        << "Name: " << name << endl
        << "\nDept: " << dept << endl << "Enrolled in:" << endl;
    for (int i = 0; i < num_courses; i++)
        enrolled[i].print(); // Assume a Course print function
}
```

Example: `Student::print()` Doesn't Compile!

- The implementation of **`Student::print()`** given before doesn't work. It will raise an **error** during compilation:

`Student::print()`: **`name`** and **`dept`** are declared **private**.

- **`name`** is a **private** data member of the **base class `UPerson`**.
- **Public inheritance** does not change the **access control** of the data members of the **base class**.
- **Private members** are still only available to **base class'** own member functions (methods), and **not** to any **other** classes including **derived classes** (except **friends**) or **global functions**.

One Solution: Protected Data Members

```
class UPerson                                /* File: protected-uperson.h */
{
    protected:
        string name;
        Department dept;

    public:
        UPerson(string n, Department d) : name(n), dept(d) { };
        void print() const;
        ...
};
```

- By making **name** and **dept** **protected**, they are accessible to member functions in the **base class** as well as member functions in the **derived classes**.
- They should not be **public** though!
(Principle of information hiding.)

Member Access Control: public, protected, private

There are 3 levels of member (data or functions) access control:

1. **public**: accessible to

- ▶ member functions of the class (from class developer)
- ▶ any member functions of other classes (application programmers)
- ▶ any global functions (application programmers)

2. **protected**: accessible to

- ▶ member functions and **friends** of the class
- ▶ member functions and **friends** of its **derived classes** (**subclasses**)

⇒ class developer **restricts** what subclasses may directly use

3. **private**: accessible only to

- ▶ member functions and **friends** of the class

⇒ class developer **enforces information hiding**

Without inheritance, **private** and **protected** control are the same.

protected vs. private

So why not always use **protected** instead of **private**?

- Because **protected** means that we have less data **encapsulation**: Remember that all **derived classes** can access **protected** data members of the base class.
- Assume that later you decided to change the implementation of the **base class** having the **protected** data members.
- For example, we might want to represent dept of **UPerson** by a new class called **class Department** instead of **enum Department**. If the **dept** data member is **private**, we can easily make this change. The update on the **UPerson** class documentation is small.
- However, if it is **protected**, we have to go through not only the **UPerson** class, but also **all** its **derived classes** and change them. We also need to update the documentation of many classes.

protected vs. private ..

- In general, it is **preferable** to have **private** members instead of **protected** members.
- Use **protected** only where it is really necessary. **private** is the only category ensuring full data **encapsulation**.
- This is particularly true for data members, but it is less harmful to have **protected** member functions. Why?
- When a class has **protected** members, it is a **hint** that it expects others to **derive sub-classes** from it.

In our example, there is no reason at all to make **name**, and **dept** **protected**, as we can access the name and address through appropriate **public** member functions.

Write Student::print(), Teacher::print() with UPerson's Public Member Functions Only

```
void Student::print() const /* correct-student-print.cpp */
{
    cout << "--- Student details ---" << endl
         << "Name: " << get_name() << endl  // Use UPerson's public fcn
         << "Dept: " << get_dept() << endl  // Use UPerson's public fcn
         << "Enrolled in:" << endl;

    for (int i = 0; i < num_courses; i++)
        enrolled[i].print();                // Use Course's public fcn
}

void Teacher::print() const /* correct-teacher-print.cpp */
{
    cout << "--- Teacher details ---" << endl
         << "Name: " << get_name() << endl  // Use UPerson's public fcn
         << "Dept: " << get_dept() << endl  // Use UPerson's public fcn
         << "Rank: " << get_rank() << endl; // Use its own fcn
}
```


Write Student::print(), Teacher::print() with UPerson's Public Member Functions Only ..

Let's use the new **print()** functions now.

```
/* File: print-example.cpp (incomplete) */
UPerson newton("Isaac Newton", MAE);
Teacher turing("Alan Turing", CSE, DEAN);
Student edison("Thomas Edison", ECE, 2.5);
edison.enroll_course("COMP2012");

newton.print();
turing.print();
edison.print();
```

Write Student::print(), Teacher::print() with UPerson's Public Member Functions Only — Expected Output

```
--- UPerson details ---
```

```
Name: Isaac Newton
```

```
Dept: 5
```

```
--- Teacher details ---
```

```
Name: Alan Turing
```

```
Dept: 2
```

```
Rank: 1
```

```
--- Student details ---
```

```
Name: Thomas Edison
```

```
Dept: 3
```

```
Enrolled in:
```

```
COMP2012H
```

Part V

Polymorphism: Dynamic Binding & Virtual Function

Sending virtual hug



loading...



Global `print()` for `UPerson` and its Derived Objects

```
#include <iostream>      /* File: print-label.cpp */
using namespace std;
#include "student.h"
#include "teacher.h"

void print_label_v(UPerson uperson) { uperson.print(); }
void print_label_r(const UPerson& uperson) { uperson.print(); }
void print_label_p(const UPerson* uperson) { uperson->print(); }

int main() {
    UPerson uperson("Charlie Brown", CBME);
    Student student("Edison", ECE, 3.5);
    Teacher teacher("Alan Turing", CSE, PROFESSOR, "CS Theory");
    student.add_course("COMP2012"); student.add_course("MATH1003");

    cout << "\n##### PASS BY VALUE #####\n";
    print_label_v(uperson); print_label_v(student); print_label_v(teacher);

    cout << "\n##### PASS BY REFERENCE #####\n";
    print_label_r(uperson); print_label_r(student); print_label_r(teacher);

    cout << "\n##### PASS BY POINTER #####\n";
    print_label_p(&uperson); print_label_p(&student); print_label_p(&teacher);
}
```

Are These Outputs What You Want?

PASS BY VALUE

--- UPerson Details ---

Name: Charlie Brown

Dept: 0

--- UPerson Details ---

Name: Edison

Dept: 3

--- UPerson Details ---

Name: Alan Turing

Dept: 2

PASS BY POINTER

--- UPerson Details ---

Name: Charlie Brown

Dept: 0

--- UPerson Details ---

Name: Edison

Dept: 3

--- UPerson Details ---

Name: Alan Turing

Dept: 2

PASS BY REFERENCE

--- UPerson Details ---

Name: Charlie Brown

Dept: 0

--- UPerson Details ---

Name: Edison

Dept: 3

--- UPerson Details ---

Name: Alan Turing

Dept: 2

You Probably Want This

PASS BY VALUE

--- UPerson Details ---

Name: Charlie Brown

Dept: 0

--- UPerson Details ---

Name: Edison

Dept: 3

--- UPerson Details ---

Name: Alan Turing

Dept: 2

PASS BY REFERENCE

--- UPerson Details ---

Name: Charlie Brown

Dept: 0

--- Student Details ---

Name: Edison

Dept: 3

2 Enrolled courses: COMP2012 MATH1003

--- Teacher Details ---

Name: Alan Turing

Dept: 2

Rank: 0

Research area: CS Theory

PASS BY POINTER

--- UPerson Details ---

Name: Charlie Brown

Dept: 0

--- Student Details ---

Name: Edison

Dept: 3

2 Enrolled courses: COMP2012 MATH1003

--- Teacher Details ---

Name: Alan Turing

Dept: 2

Rank: 0

Research area: CS Theory

Static (or Early) Binding

- Because of the **polymorphic substitution principle**, a function accepting a **base class** object also accepts its **derived** objects.
- In our current case, the following 3 **global** print functions:

```
void print_label_v(UPerson uperson) { uperson.print(); }  
void print_label_r(const UPerson& uperson) { uperson.print(); }  
void print_label_p(const UPerson* uperson) { uperson->print(); }
```

will accept objects of **UPerson/Student/Teacher** classes, and objects derived from them **directly** or **indirectly**.

- However, when these function codes are compiled, the compiler only looks at the **static type** of **uperson** which is **UPerson**, **const UPerson&**, or **const UPerson***, and the member function **UPerson::print()** is called.
- **Static binding**: the binding (association) of a function name (here **print()**) to the appropriate member function is done by a **static** analysis of the code at **compile time** based on the **static** (or **declared**) type of the object (here, **UPerson**) making the call.

Static Binding: Who May Call Whose `print()`?

```
#include <iostream>      /* File: static-example.cpp */
using namespace std;
#include "teacher.h"

int main()
{
    UPerson uperson("Charlie Brown", CBME);
    Teacher teacher("Alan Turing", CSE, PROFESSOR, "CS Theory");
    UPerson* u; Teacher* t;

    cout << "\nUPerson object pointed by UPerson pointer:\n";
    u = &uperson; u->print();

    cout << "\nTeacher object pointed by Teacher pointer:\n";
    t = &teacher; t->print();

    cout << "\nTeacher object pointed by UPerson pointer:\n";
    u = &teacher; u->print();

    cout << "\nUPerson object pointed by Teacher pointer:\n";
    t = &uperson; t->print(); // Error: convert base-class ptr
                               //           to derived-class ptr
    t = static_cast<Teacher*>(&uperson); t->print(); // Ok, but ...
}
```


Dynamic (or Late) Binding

- By default, C++ uses **static binding**. (Same as C, Pascal, and FORTRAN.)
- In **static binding**, what a pointer really points to, or what a reference actually refers to is not considered; only the pointer **type** is.
- But C++ also allows **dynamic binding** which is supported through **virtual functions**.
- When **dynamic binding** is used, the **actual member function** to be called is selected using the **actual type** of the object in the call, but only if the object is passed by **reference** or **pointer**. i.e.,
print_label_r(a UPerson object) calls **UPerson::print()**;
print_label_r(a Teacher object) calls **Teacher::print()**;
print_label_r(a Student object) calls **Student::print()**.
- **Magic**: the possible object types **don't** need to be known at the time when the function definition is being compiled!!!

Virtual Functions

- A **virtual function** is declared using the keyword **virtual** in the **class definition**, and **not** in the member function implementation, if it is defined outside the class.
- Once a member function is declared **virtual** in the **base class**, it is automatically **virtual** in **all** directly or indirectly **derived classes**.
- Even though it is not necessary to use the **virtual** keyword in the **derived classes**, it is a good style to do so because it improves the readability of header files.
- Calls to **virtual functions** are a little bit slower than normal function calls. The difference is extremely small and it is not worth worrying about, unless you write very speed-critical code.

Virtual Function: UPerson Class

```
#ifndef V_UPERSON_H          /* File: v-uperson.h */
#define V_UPERSON_H

enum Department { CBME, CIVL, CSE, ECE, IELM, MAE };

class UPerson
{
private:
    string name;
    Department dept;

public:
    UPerson(string n, Department d) : name(n), dept(d) { };
    string get_name() const { return name; }
    Department get_department() const { return dept; }

    virtual void print() const
    {
        cout << "--- UPerson Details --- \n"
              << "Name: " << name << "\nDept: " << dept << "\n";
    }
};
```

Virtual Function: Course Class

```
#ifndef COURSE_H                /* File: course.h */
#define COURSE_H

class Course
{
    private:
        string code;

    public:
        Course(const string& s) : code(s) { }
        ~Course() { cout << "destruct course: " << code << endl; }
        void print() const { cout << code; }
};

#endif
```

Virtual Function: Student Class

```
#ifndef V_STUDENT_H      /* File: v-student.h */
#define V_STUDENT_H
#include "course.h"
#include "v-uperson.h"

class Student : public UPerson { // Public inheritance
private:
    float GPA; Course* enrolled[50]; int num_courses;

public:
    Student(string n, Department d, float x) :
        UPerson(n, d), GPA(x), num_courses(0) { }
    ~Student() { for (int j = 0; j < num_courses; ++j) delete enrolled[j]; }
    float get_GPA() const { return GPA; }
    bool add_course(const string& s)
        { enrolled[num_courses++] = new Course(s); return true; };
    virtual void print() const {
        cout << "---- Student Details ---- \n"
            << "Name: " << get_name() << "\nDept: " << get_department()
            << "\n" << num_courses << " Enrolled courses: ";
        for (int j = 0; j < num_courses; ++j)
            { enrolled[j]->print(); cout << ' '; } cout << "\n";
    }
};

#endif
```

Virtual Function: Teacher Class

```
#ifndef V_TEACHER_H      /* File: v-teacher.h */
#define V_TEACHER_H
#include "v-uperson.h"

enum Rank { PROFESSOR, DEAN, PRESIDENT };
class Teacher : public UPerson // Public inheritance
{
private:
    Rank rank;
    string research_area;

public:
    Teacher(string n, Department d, Rank r, string a) :
        UPerson(n, d), rank(r), research_area(a) { };
    Rank get_rank() const { return rank; }
    string get_research_area() const { return research_area; }
    virtual void print() const {
        cout << "---- Teacher Details ---- \n"
              << "Name: " << get_name()
              << "\nDept: " << get_department()
              << "\nRank: " << rank
              << "\nResearch area: " << research_area << endl;
    }
};
#endif
```

Polymorphism

Polymorphism

poly = multiple

morphos = shape

- **Polymorphism** in C++ means that we can work with objects **without** knowing their precise type at **compile time**.
- `void print_label_p(const UPerson* uperson) { uperson->print(); }`
The type of the object pointed to by **uperson** is **not** known to the programmer writing this code, nor to the compiler.
- We say that **uperson** exhibits **polymorphism**, because the object can take on multiple “shapes” (Student, Teacher, PG_Student, etc.).
- **Polymorphism** allows us to write programs that behave correctly even when used with objects of **derived classes**.
- Again a **pointer** or **reference** **must** be used to take advantage of **polymorphism**.

Question: Why won't polymorphism work if pass-by-value is used?

Example: Polymorphism using Virtual Function

```
#include <iostream>      /* File: v-example.cpp */
using namespace std;
#include "v-student.h"
#include "v-teacher.h"

int main()
{
    char person_type; string name; UPerson* uperson[3] = {};

    for (int j = 0; j < sizeof(uperson)/sizeof(UPerson*); ++j)
    {
        cout << "Input the uperson type (u/s/t) and his name : ";
        cin >> person_type >> name;
        switch (person_type)
        {
            case 'u': uperson[j] = new UPerson(name, MAE); break;
            case 's': uperson[j] = new Student(name, CIVL, 4.0); break;
            case 't': uperson[j] = new Teacher(name, CSE, DEAN, "AI"); break;
        }
    }

    for (int j = 0; j < sizeof(uperson)/sizeof(UPerson*); ++j)
        uperson[j]->print();
    return 0;
} // The example doesn't destruct the dynamically allocated objects
```


Run-Time Type Information (RTTI)

- **RTTI** is a **run-time** facility that keeps track of **dynamic types** \Rightarrow program can determine an object's **type** at **run-time**.
- The function **typeid(<expression>)** returns an **object** of the type **type_info**. It has a member function

const char* name() const

that returns the **type name** of the expression.

- **static_cast()** may be used to perform **type conversions**,
 - ▶ including conversions between pointers to classes in an inheritance hierarchy;
 - ▶ it **doesn't** consult **RTTI** to ensure the conversion is safe;
 - ▶ thus, it runs faster.
- **dynamic_cast()**, on the other hand,
 - ▶ **only** works on **pointers** and **references** of **polymorphic** class (with **virtual functions**) types;
 - ▶ consults **RTTI** to make sure the conversion result is a pointer to a **valid complete object** of the target type; otherwise, it returns a **null pointer**.

Example: RTTI typeid()

```
#include <iostream>      /* File: rtti.cpp */
using namespace std;
#include "v-student.h"
#include "v-teacher.h"

int main()
{
    UPerson* uperson[3] = { };
    char person_type; string name;

    for (int j = 0; j < sizeof(uperson)/sizeof(UPerson*); ++j)
    {
        cout << "Input the uperson type (s/t) and his name : ";
        cin >> person_type >> name;

        if (person_type == 's') // No error checking
            uperson[j] = new Student(name, CIVL, 4.0);
        else if (person_type == 't')
            uperson[j] = new Teacher(name, CSE, DEAN, "AI");
    }

    for (int j = 0; j < sizeof(uperson)/sizeof(UPerson*); ++j)
        cout << "The uperson #" << j << " is a "
            << typeid(*uperson[j]).name() << endl; // RTTI
    }
```

Example: RTTI typeid() Output

```
Input the uperson type (s/t) and his name : s Abby  
Input the uperson type (s/t) and his name : t Brian  
Input the uperson type (s/t) and his name : s Chris  
The uperson #0 is a 7Student  
The uperson #1 is a 7Teacher  
The uperson #2 is a 7Student
```

- The returned type name is implementation dependent.
- i.e., different compilers may give different printout.
- In this course, we assume the above printout from the g++ compiler.

Overriding and Virtual Functions

- When a **derived class** defines a member function with the same name as a **base class** member function, it **overrides** the **base class** member function. e.g.

Student::print() overrides **UPerson::print()**

- This is necessary if the behaviour of the **base class** member function is not good enough for **derived classes**.
- All **derived classes** should respond to the same request (**print!**), but their response varies depending on the object.
- The designer of a **base class** (**UPerson**) must realize that this is necessary, and declare its **print()** a **virtual function**.
- Overriding** is **not** possible if the member function is not **virtual**.
- For **overriding** to work, the **prototype** of the **virtual function** in the **derived class** must be **identical** to that of the **base class**. To safeguard this, C++11 recommend a new keyword **override** in the function declaration in the **derived classes**.

```
/* in derived classes: Student or Teacher, etc. */  
virtual void print() const override;
```

C++11 Keyword: **override**

```
#include <iostream>      /* File: override.cpp */
using namespace std;

class Base
{
    public:
        virtual void f(int a) const { cout << a << endl; }
};

class Derived: public Base
{
    int x {25};
    public:
        void f(int) const override;
};

// Don't repeat the keyword override here
void Derived::f(int b) const { cout << x+b << endl; }

int main() { Derived d; Base& b = d; b.f(5); return 0; }
```

Virtual Functions vs. Non-Virtual Functions

- The designer of the **base class** must distinguish carefully between two kinds of member function:
 - ▶ If the member function works exactly the **same** for all **derived classes**, it should **not** be a **virtual function**.
 - ▶ If the precise behaviour of the member function depends on the object, it should be a **virtual function**.
- However, **derived classes** have to be careful in implementing such member functions because of the **substitution principle**. The “**effect**” (meaning) of calling the **derived class** member function must be the “same” as that for the **base class** member function.

Virtual Functions vs. Non-Virtual Functions ..

- **Overriding** is for specializing a behaviour, not changing the **semantics**.
- For example, **print()** should not be a member function that does something completely different.
- **fly()** must do what it promises, and therefore we could not implement **Penguin** as a **derived class** of **Bird**.
- The compiler can only check that **overriding** is done **syntactically** correct, **not** whether the **semantics** of the member function are preserved.

Overloading vs. Overriding

Overloading

Allows programmers to use functions with the **same** name, but **different** arguments for similar purposes.

- The decision on which function to use — **overload resolution** — is done by the compiler when the program is compiled.
- There is no **dynamic binding**.

Overloading vs. Overriding ..

Overriding

Allows a **derived class** to provide a **different** implementation for a function declared in the **base class**.

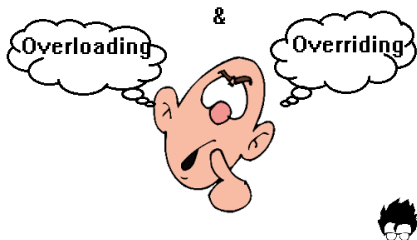
- **Overriding** is only possible with **inheritance** and **dynamic binding** — without **inheritance** there is **no overriding**.
- The decision of which member function to use is done at the **moment** that the member function is called.
- It only applies to member functions, not **global functions**.

Question: Can a **virtual function** be declared as **protected** or **private** in the **derived** classes (while it is declared as **public** in the **base** class)?

E.g., for the **UPerson** example, try to declare the print function as **protected** or **private** in **Student** or **Teacher**.

Part VI

Virtual Functions and Destructors & Constructors



Example: Destruction with No Substitution

```
#include <iostream>      /* File: concrete-destructors.cpp */
using namespace std;
#include "v-student.h"

int main()
{
    UPerson *p = new UPerson("Adam", ECE);
    delete p;

    Student *s = new Student("Simpson", CSE, 3.8);
    s->add_course("COMP1021");
    s->add_course("COMP2012");
    delete s;
}
```

- **delete p** will call **UPerson's destructor**, and **delete s** will call **Student's destructor** respectively. Everything works fine.

Example: Destruction with Substitution

```
#include <iostream>      /* File: require-v-destructors.cpp */
using namespace std;
#include "v-student.h"

int main()
{
    Student *s = new Student("Simpson", CSE, 3.8);
    s->add_course("COMP1021");
    s->add_course("COMP2012");

    UPerson *p = s;
    delete p; // Can we call UPerson's destructor on a Student?
}
```

- Here **p** actually points to a **Student** object.
- **delete p** calls the **UPerson's destructor**, and not **Student's destructor**.
- The behavior of **destructing** a **derived class** object by its **base class destructor** is **undefined**!

Virtual Destructor

- The solution is again using **dynamic binding**, and making the destructors **virtual**.

```
class UPerson /* File: v-uperson2.h */
{
    public: virtual ~UPerson() { };
    ...
};

class Student : public UPerson /* File: v-student2.h */
{
    public:
        virtual ~Student()
        {
            for (int j = 0; j < num_courses; ++j)
                delete enrolled[j];
        }
    ...
};
```

Virtual Destructor ..

```
#include <iostream>      /* File: v-destructors.cpp */
using namespace std;
#include "v-student2.h" // With virtual destructor

int main()
{
    Student *s = new Student("Simpson", CSE, 3.8);
    s->add_course("COMP1021");
    s->add_course("COMP2012");

    UPerson *p = s;
    delete p;           // Actually call Student's destructor
}
```

- Now, **delete p** correctly calls the **Student's destructor** if **p** points to a **Student** object.
- When a class does **not** have a **virtual destructor**, it is a strong hint that the class is **not** designed to be used as a **base class**.

Example: Order of Constructions and Destruction

```
#include <iostream>      /* File: construction-destruction-order.cpp */
using namespace std;

class Base
{
public:
    Base() { cout << "Base's constructor\n"; }
    ~Base() { cout << "Base's destructor\n"; }
};

class Derived : public Base
{
public:
    Derived() { cout << "Derived's constructor\n"; }
    ~Derived() { cout << "Derived's destructor\n"; }
};

int main()
{
    Base *p = new Derived;
    delete p;
}
```

Question: What is the output?

Example: Order of Constructions and Destruction ..

Question: What is the output when virtual destructors are used?

```
#include <iostream>      /* File: construction-v-destruction-order.cpp */
using namespace std;

class Base
{
public:
    Base() { cout << "Base's constructor\n"; }
    virtual ~Base() { cout << "Base's destructor\n"; }
};

class Derived : public Base
{
public:
    Derived() { cout << "Derived's constructor\n"; }
    virtual ~Derived() { cout << "Derived's destructor\n"; }
};

int main()
{
    Base *p = new Derived;
    delete p;
}
```


Example: Calling Virtual Functions in Constructors

```
#include <iostream>      /* File: construct-vf.cpp */
using namespace std;

class Base {
public:
    Base() { cout << "Base's constructor\n"; f(); }
    virtual void f() { cout << "Base::f()" << endl; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived's constructor\n"; }
    virtual void f() { cout << "Derived::f()" << endl; }
};

int main() {
    Base *p = new Derived;
    cout << "Derived-class object created" << endl;
    p->f();
}
```

Example: Calling Virtual Functions in Constructors ..

The output is:

```
Base's constructor  
Base::f( )  
Derived's constructor  
Derived-class object created  
Derived::f( )
```

- Do not rely on the **virtual function** mechanism during the execution of a constructor.
- This is not a bug, but necessary — how can the **derived** object provide services if it has **not** been constructed yet?
- Similarly, if a **virtual function** is called inside the **base class destructor**, it represents **base class' virtual function**: when a **derived class** is being deleted, the derived-specific portion has already been deleted before the **base class destructor** is called!

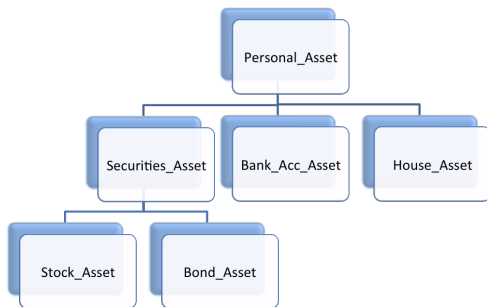
Part VII

As Simple as ABC: Abstract Base Class



ABC Example: Assets

- Let's design a system for maintaining our assets: stocks, bank accounts, real estate, cars, yachts, etc.
- Each asset has a net worth (monetary value). We would like to be able to make listings and compute the total net worth.
- There are different kinds of assets, and they are all derived from **Personal_Asset**.



ABC Example: Personal_Asset + Bank_Acc_Asset Classes

```
class Personal_Asset    /* File: personal-asset.h */
{
public:
    Personal_Asset(const string& date) : purchase_date(date) { }
    void set_purchase_date(const string& d);
    virtual double compute_net_worth() const; // Current net worth
    virtual bool is_insurable() const;       // Can this asset be insured?

private:
    string purchase_date;
};

class Bank_Acc_Asset : public Personal_Asset /* File: bank-acc-asset.h */
{
public:
    Bank_Acc_Asset(const string& d, double m, double r = 0.0)
        : Personal_Asset(d), balance(m), interest_rate(r) { }
    virtual double compute_net_worth() const { return balance; }

private:
    double balance;
    double interest_rate;
};
```

ABC Example: compute-assets.cpp

- There can be **other** classes of assets such as **Car_Asset**, **Securities_Asset**, **House_Asset**, etc.
- One may compute the total asset value for an array of **different** kinds of assets as follows:

```
/* File: compute-assets.cpp */
double compute_total_worth(const Personal_Asset* asset[], int num_assets)
{
    double total_worth = 0.0;

    for (int i = 0; i < num_assets; i++)
        total_worth += assets[i]->compute_net_worth();

    return total_worth;
}
```

ABC Example: Personal_Asset Class Implementation

- Now we have to implement the member functions of the **base class Personal_Asset**.
- How to implement **Personal_Asset::compute_net_worth()**?
- It depends completely on the **actual** type of asset. There is **no** “standard way” of doing it!

```
/* File: personal-asset.cpp */  
Personal_Asset::Personal_Asset(const string& date)  
    : purchase_date(date) { }  
  
void Personal_Asset::set_purchase_date(const string& date)  
    { purchase_date = date; }  
  
double Personal_Asset::compute_net_worth() const  
    { return /* What? */ }
```

ABC Example: How to Implement `compute_net_worth()`?

- The truth is: It makes **no** sense to have objects of type **Personal_Asset**.
- Such an object has only a purchase date, but otherwise **no** meaning. It is not a bank account, not a car, not a house — it is too general to be used.
- We **cannot** implement the **compute_net_worth()** member function in the **base class Personal_Asset** as the information needed to implement it is missing.
- However, we don't want to remove the member function because that would make a **polymorphic** function like **compute_total_worth()** impossible.

Solution: Abstract Base Class (ABC)

The solution is to make **Personal_Asset** an **abstract base class** (ABC), and **compute_net_worth()** now becomes a **pure virtual function**.

```
class Personal_Asset    /* File: personal-asset-abc.h */
{
    public:
        Personal_Asset(const string& date) : purchase_date(date) { }
        void set_purchase_date(const string& d);
        virtual bool is_insurable() const; // Can this asset be insured?

        // A pure virtual function to compute the current net worth
        virtual double compute_net_worth() const = 0;

    private:
        string purchase_date;
};
```

Abstract Base Class (ABC)

```
Personal_Asset p_asset("1997/07/01");           // Error  
Bank_Acc_Asset b_asset("2000/01/01", 100.0);    // Ok
```

- An ABC has two properties:
 1. **No** objects of ABC can be created.
 2. Its **derived classes** must implement the **pure virtual functions**, otherwise they will also be ABC's.
- If a **derived class**, e.g., **Securities_Asset**, does not implement the **pure virtual functions**, then
 - ▶ the **derived class** is also an **ABC**, and
 - ▶ there **cannot** be objects of that type,
 - ▶ but it can be used as a **base class** itself, for instance for **Stocks_Asset**, **Bonds_Asset**, etc.

Interface Re-use

ABC as an Interface

An abstract base class provides a uniform interface to deal with a number of different derived classes.

- A **base class** contains what is **common** about several classes.
- If the only thing that is **common** is the **interface**, then the **base class** is a “**pure interface**,” called ABC in C++.
- We discussed before that code re-use is an advantage of **inheritance**.
- For ABC's, we do not re-use code, but create an **interface** that can be re-used by its **derived classes**.
- **Interfaces** are the soul of **object-oriented programming**. They are the most effective way of separating the **use** and **implementation** of objects.
- The user (of **compute_total_worth()**) only knows about the **abstract interface**, objects from different **derived classes** of the ABC may implement the **interface** in different ways.

Final Remarks on ABC

- A **pure virtual function** is **inherited** as a **pure virtual function** by a **derived class** unless it implements the function.
- An **abstract base class** cannot be used
 - ▶ as an argument type that is passed by **value**
 - ▶ as a function return type that is returned by **value**
 - ▶ as the type of an **explicit conversion**
- However, **pointers** and **references** to an **ABC** can be declared.
- Calling a **pure virtual function** from the **constructor** of an ABC is **undefined** — don't do that.

ABC Example: Do and Don't

```
#include <string>          /* File: can-and-cant.cpp */
using namespace std;

#include "personal-asset-abc.h"
#include "bank-acc-asset.h"

Personal_Asset x("20010/01/01");      // Error: can't create ABC object
Personal_Asset f1(int x) { /* .. */ } // Error: can't return ABC object
int f2(Personal_Asset x) { /* .. */ } // Error: can't CBV with ABC object

Bank_Acc_Asset b("01/01/2000", 0.0); // OK!
Personal_Asset* p_asset_ptr = &b;    // OK!
Personal_Asset& p_asset_ref = b;     // OK!

Personal_Asset* f3(const Personal_Asset& x) { /* incomplete */ } // OK!
```

Part VIII

The C++11 Keyword final: No More Offspring



FINAL

A final Class

```
1  #include <iostream>      /* File: final-class-error.cpp */
2  using namespace std;
3
4  class A {};
5  class B: public A {};
6  class C final: public B {};
7  class D: public B {};
8  class E: public C {};
9
10 int main()
11 {
12     A a; B b; C c; D d; E e;
13     return 0;
14 }
```

final-class-error.cpp:8:7: error: cannot derive from 'final' base 'C'
in derived type 'E'

```
    class E: public C {};
```

^

No sub-classes can be derived from a **final** class.

Example: No PG_Student if Student Class is final

```
1  #include <iostream>      /* File: pg-final-error.cpp */
2  using namespace std;
3
4  class UPerson { /* incomplete */ };
5  class Student final : public UPerson { /* incomplete */ };
6  class PG_Student : public Student { /* incomplete */ };
7
8  int main()
9  {
10     UPerson abby("Abby", CBME);
11     Student bob("Bob", CIVL, 3.0);
12     PG_Student matt("Matt", CSE, 3.8);
13 }
```

pg-final-class-error.cpp:6:7: error: cannot derive from 'final' base
'Student' in derived type 'PG_Student'

```
class PG_Student : public Student { /* incomplete */ };
~~~~~
```


Example: No PG_Student::print if Student::print is final

```
1  #include <iostream>      /* File: final-vfcn-error.cpp */
2  using namespace std;
3
4  class UPerson {
5      public: /* Other data and functions */
6          virtual void print() const { /* incomplete */ }
7  };
8
9  class Student : public UPerson {
10     public: /* Other data and functions */
11         virtual void print() const override final { /* incomplete */ }
12 };
13
14 class PG_Student : public Student {
15     public: /* Other data and functions */
16         virtual void print() const override { /* incomplete */ }
17 };
18
19 int main() { PG_Student jane("Jane", CSE, 4.0); jane.print(); }
```

Example: No PG_Student::print if Student::print is final ..

```
final-vfcn-error.cpp:16:18: error: virtual function
'virtual void PG_Student::print() const'
    virtual void print() const override { /* incomplete */ }
           ~~~~~
final-vfcn-error.cpp:11:18: error: overriding final function
'virtual void Student::print() const'
    virtual void print() const override final { /* incomplete */ }
           ~~~~~
```

Can't override a final virtual function.

Further Reading



Part IX

Public / Protected / Private Inheritance



"You are such a drama queen! Heaven knows where you get that from!"

Different Types of Inheritance

- So far, we have been dealing with only **public inheritance**.
`class Student: public UPerson { ... }`
- There are two other kinds of inheritance: **protected** and **private inheritance**.
- They **control** how the **inherited members** of Student are accessed by Student's **derived classes** (*not* by the Student class itself) or **global functions**.

UPerson Class Again

```
#ifndef UPERSON_H          /* File: uperson.h */
#define UPERSON_H

enum Department { CBME, CIVL, CSE, ECE, IELM, MAE };

class UPerson
{
private:
    string name;
    Department dept;

protected:
    void set_name(string n) { name = n; }
    void set_department(Department d) { dept = d; }

public:
    UPerson(string n, Department d) : name(n), dept(d) { }
    string get_name() const { return name; }
    Department get_department() const { return dept; }
};

#endif
```

Student Class Again

```
#ifndef STUDENT_H          /* File: student.h */
#define STUDENT_H

#include "uperson.h"

class Course { /* incomplete */ };

class Student : ??? UPerson // ??? = public/protected/private
{
    private:
        float GPA;
        Course* enrolled;
        int num_courses;

    public:
        Student(string n, Department d, float x) :
            UPerson(n, d), GPA(x), enrolled(nullptr), num_courses(0) { }
        float get_GPA() const { return GPA; }
        bool enroll_course(const string& c) { /* incomplete */ };
        bool drop_course(const Course& c) { /* incomplete */ };
};

#endif
```

Example: Public Inheritance

```
class Student: public UPerson { ... }
```

| public | protected | private |
|-------------------|-------------------|-------------|
| get_name() | set_name() | name |
| get_department() | set_department() | dept |
| get_GPA() | | GPA |
| enroll_course() | | enrolled |
| drop_course() | | num_courses |

Example: Protected Inheritance

```
class Student: protected UPerson { ... }
```

| public | protected | private |
|------------------|-------------------|-------------|
| | set_name() | name |
| | set_department() | dept |
| get_GPA() | get_name() | GPA |
| enroll_course() | get_department() | enrolled |
| drop_course() | | num_courses |

Example: Private Inheritance

```
class Student: private UPerson { ... }
```

| public | protected | private |
|------------------|-----------|-------------------|
| | | name |
| | | dept |
| get_GPA() | | GPA |
| enroll_course() | | enrolled |
| drop_course() | | num_courses |
| | | set_name() |
| | | set_department() |
| | | get_name() |
| | | get_department() |

Slicing with Public Inheritance Again

Given the following definitions and **public inheritance** is used:

```
class Derived : public Base { ... }  
Base base;  
Derived derived;
```

- The following assignments are **fine**:

```
base = derived;           // Slicing  
Base* b = &derived;       // Can't use derived-class specific members  
Base& b = derived;        // Can't use derived-class specific members
```

- The following assignments give compilation **errors**:

```
derived = base;           // Unless you define such conversion  
Derived* d = &base;       // No such conversion  
Derived& d = base;        // No such conversion
```

No Slicing for Protected and Private Inheritance

If you use **protected/private** inheritance, **slicing** won't work either. That is, none of the assignments in the previous page work.

```
1  #include <string>          /* File: no-slicing.cpp */
2  using namespace std;
3
4  // class Student: protected UPerson { ... }
5  #include "protected-student.h"
6
7  int main()
8  {
9      Student ug("UG", ECE, 3.0);
10     UPerson p = ug;        // Allowed or not?
11     UPerson* q = &ug;     // Allowed or not?
12     UPerson& r = ug;      // Allowed or not?
13     return 0;
14 }
```

No Slicing for Protected and Private Inheritance ..

```
no-slicing.cpp:10:17: error: cannot cast 'const Student' to its
protected base class 'const UPerson'
```

```
    UPerson p = ug;      // Allowed or not?
                  ^
```

```
./protected-student.h:7:17: note: declared protected here
class Student : protected UPerson
    ~~~~~
```

```
no-slicing.cpp:11:18: error: cannot cast 'Student' to its
protected base class 'UPerson'
```

```
    UPerson* q = &ug;    // Allowed or not?
                  ^
```

```
no-slicing.cpp:12:18: error: cannot cast 'Student' to its
protected base class 'UPerson'
```

```
    UPerson& r = ug;     // Allowed or not?
                  ^
```

Quiz: Why the first error mentions a 'const Student' instead of a 'Student'?

Inheritance: Summary

1. **Public inheritance** **preserves** the original accessibility of inherited members:

public \Rightarrow public
protected \Rightarrow protected
private \Rightarrow private

2. **Protected inheritance** affects only public members and renders them **protected**.

public \Rightarrow protected
protected \Rightarrow protected
private \Rightarrow private

3. **Private inheritance** renders all inherited members **private**.

public \Rightarrow private
protected \Rightarrow private
private \Rightarrow private

Inheritance: Summary ..

- The various types of inheritance **control** the **highest** accessibility of the **inherited member** data and functions.
- **Public inheritance** implements the “**is-a**” relationship.
- **Private inheritance** is similar to “**has-a**” relationship.
- **Public inheritance** is the most **common** form of inheritance.
- **Private** and **protected inheritance** do not allow casting of objects of derived classes back to the base class.