Comp151

Inheritance: Introduction

Example: University Admin Info

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

- **Teacher** and **Student** are two completely separate classes. Their implementation uses *separate* code.
- However, there are some methods and members that are implemented in the same way: handling name, address, and department.
- Why do we implement the same function twice?
- That is not good software reuse!

Example: U. Admin Info — student.h

```
enum Department { accounting, business, engineering, mathematics, unknown };
class Student
{
  private:
    string name;
    string address;
    Department dept;
    Course *enrolled;
    int num_courses;
  public:
    Student(string n, string a, Department d) :
        name(n), address(a), dept(d), enrolled(NULL), num_courses(0) { };
    void set_name(const char *name);
    void set_address(const char *adr);
    void set_department(Department dept);
    string get_name() const;
    string get_address() const;
    Department get_department() const;
    bool enroll_course(const string &);
    bool drop_course(const Course &);
};
```

Example: U. Admin Info — teacher.h

```
enum Rank { instructor, assistant_prof, associate_prof, professor, dean };
class Teacher
{
  private:
    string name;
    string address;
    Department dept;
    Rank rank;
  public:
    Teacher(string n, string a, Department d, Rank r) :
        name(n), address(a), dept(d), rank(r) { };
    void set_name(const char *name);
    void set_address(const char *adr);
    void set_department(Department dept);
    void set_rank(Rank rank);
    string get_name() const;
    string get_address() const;
    Department get_department() const;
    Rank get_rank() const;
};
```

Things to Consider

- We want a way to say that **Student** and **Teacher** both have the same members: name, address, dept, but yet require them to keep a *separate* copy of these members.
- We want to *share* the code for **set_name** etc. between **Student** and **Teacher** as well.
- However, objects have state, and it needs to remain *consistent* when these methods 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 class **Address**.
- "REUSE by COPYING" is a bad idea!

Solution#2: By Inheritance — person.h

```
class Person
{
  private:
    string name;
    string address;
    Department dept;

  public:
    Person(string n, string a, Department d) :
        name(n), address(a), dept(d) { };
    void set_name(const char *name);
    void set_address(const char *adr);
    void set_department(Department dept);
    string get_name() const;
    string get_address() const;
    Department get_department() const;
};
```

Solution#2: By Inheritance — student.h


```
class Student : public Person
{
  private:
    Course *enrolled;
    int num_courses;

  public:
    Student(string n, string a, Department d) :
        Person(n, a, d), enrolled(NULL), num_courses(0) { }

    bool enroll_course(const string &);
    bool drop_course(const Course &);
};
```

Solution#2: By Inheritance — teacher.h

```
class Teacher : public Person
{
  private:
    Rank rank;

  public:
    Teacher(string n, string a, Department d, Rank r) :
        Person(n, a, d), rank(r) { }

    void set_rank(Rank rank);
    Rank get_rank() const;
};
```

Inheritance

Inheritance enables code reuse.

- Inheritance is the ability to define a new class based on an existing class with a hierarchy.
- The derived class *inherits* the data members and member methods) of the base class.
- New members and methods can be added to the derived class.
- Since the new class only has to implement the behavior that is different from the base class, we can *reuse* the code for the base class.

Inheritance ...

- **Person** is the base class of **Student**.
- **Student** is a derived class of **Person**.
- The effect is that **Student** *inherits* all data members and methods from **Person**.
- The data members of **Student** are the data members of **Person** { name, address, dept }, plus the extra data members declared in the definition of **Student** { enrolled, num_courses }.

Example: Inherited Members


```
void some_func(Person& person, Student& student)
{
    cout << person.get_name() << endl;
    cout << student.get_name() << endl;

    student.set_department(engineering);
    Department dept = person.get_department();
    student.enroll_course("COMP151");
    person.enroll_course("COMP001");          // Error!
}
```

"Is" Relationship

Inheritance implements the *is*-relationship.

- Since **Student** inherits from **Person**,
  - every object of type **Student** can be used like an object of type **Person**.
  - All methods of **Person** can be called on a **Student** object.
- In other words, a **Student** object *is* definitely a **Person** object under all circumstances.
- A derived class object can be treated like a base class object under all circumstances.

Example: Derived Objects as Base Class Objects

```
bool print_mailing_label(const Person& person)
{
    string name = person.get_name();
    string adr = person.get_address();

    // code to print the label
}
```

- Since a **Student** *is* a **Person**, we can print a mailing label for a student like this:

```
Student student("Tom", "Sai Kung", mathematics);
print_mailing_label(student);
```

Direct and Indirect Inheritance

Let's add a new class **PG_Student**:

```
class PG_Student : public Student
{
  private:
    Topic research_topic;
  public:
    PG_Student(string n, string a, Department d) :
        Student(n, a, d), research_topic(NONE) { }
    void set_topic(const Topic& x) { research_topic = x; }
};
```

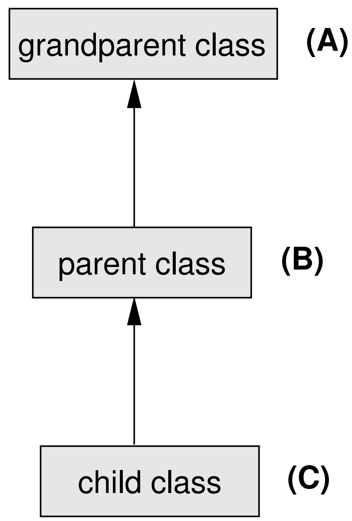
- **PG_Student** is directly derived from **Student**.
- It is *indirectly derived* from **Person**.
- So a **PG_Student** object *is a* **Person** object.
- **Person** is called an <u>indirect base class</u> for **PG_Student**.

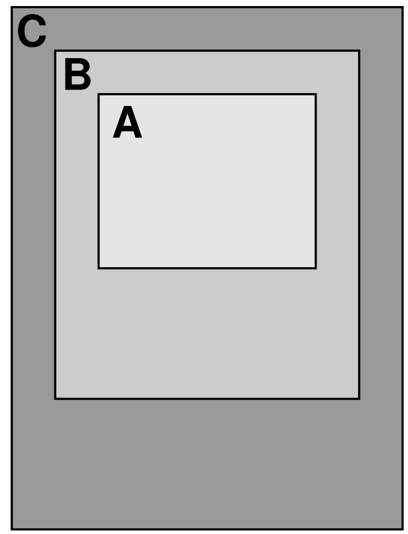
Comp151

Inheritance: Initialization & Substitution Principle

Initializing 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. i.e. it is the *responsibility* of each derived class to initialize its <u>direct base class</u> correctly.

Example: Initializing Base Class Objects

Before a **Student** object can come into existence, we have to create its **Person** part. This has to be done using one of the constructors of **Person**. We use the same "colon syntax" as for initializing data members:

```
Student::Student(string n, string a, Department d)
        : Person(n, a, d), 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 **Person** part by calling **Person**'s constructor. In fact, its **Person** part should have been created by **Student**.

```
PG_Student(string n, string a, Department d) :
    Student(n, a, d), research_topic(NONE) { }
```

Example: Order of Construction/Destruction (1)

```
#include <iostream.h>

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

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

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

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

Output: Order of Construction/Destruction


```
Person's constructor
Address's constructor
Student's constructor
Student's destructor
Address's destructor
Person's destructor
```

Example: Order of Construction/Destruction (2)

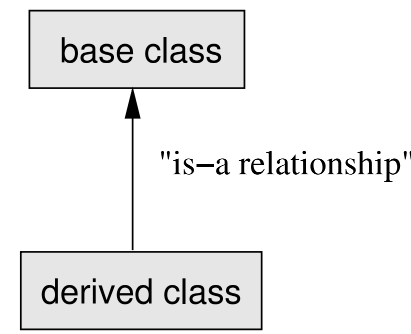
```
#include <iostream.h>

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

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

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

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

What is the output now?

Polymorphic Substitution Principle

The single most important rule in OOP with C++ is:

Inheritance means "is a."



If class $\mathbf{D}$ (the derived class) inherits from class $\mathbf{B}$ (the base class) $\Rightarrow$

- Every object of type $\mathbf{D}$ is also an object of type $\mathbf{B}$, but NOT *vice-versa*.
- $\mathbf{B}$ is a more general concept, $\mathbf{D}$ is a more special concept.
- Where an object of type $\mathbf{B}$ is needed, an object of type $\mathbf{D}$ can be used instead.

Polymorphic Substitution Principle ...

In C++, using our university administration example, where **Student** is derived from **Person**, this means:

Any Function That Expects an Argument of Type	Will Also Accepts
Person	**Student**
pointer to **Person**	pointer to **Student**
Person reference	**Student** reference

It is also known as "Liskov Substitution Principle".

Example: Substitution in Arguments


```
void dance(const Person &p);          // Anyone can dance
void study(const Student &s);         // Only students study

void dance(Person *p);                // Anyone can dance
void study(Student *s);               // Only students study

int main()
{
    Person p; Student s;

    dance(p); dance(s);

    study(s); study(p);

    dance(&p); dance(&s);

    study(&s); study(&p);
}
```

Extending class hierarchies

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

- New classes can immediately benefit from all functions that are available for their base classes.
- e.g. **bool** print_mailing_label(**const** Person& person) will work immediately for a new class type **Research_Scholar**, even though this type of object was unknown when **print_mailing_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 code for **Person** and **print_mailing_label()**.
- Advanced use: Link in new objects while the code is running!

Slicing

An assignment from derived class to base class does "slicing". This is rarely desirable. Once slicing has happened, there is no trace of the fact that we started with a student.

```
Student student("Snoopy", "HKUST", math);
Person *pp = &student;
Person *pp2 = new Student("Mickey", "HKUST", math);

Person person;
person = student;                // What does "person" have?
```

Example: Name Conflicts?

```
class B {
      int x, y;
   public:
      B() : x(1), y(2) { cout << "Base class constructor" << endl; }
      void f() { cout << "Base class:  " << x << " , " << y << endl; }
};

class D : public B {
      float x, y;
   public:
      D() : x(10.0), y(20.0) { cout << "Derived class constructor" << endl; }
      void f() { cout << "Derived class:  " << x << " , " << y << "\t"; B::f(); }
};

void smart(B* z) { cout << "Inside smart():  "; z->f(); }

int main()
{
      B base; B* b = &base;
      D derive; D* d = &derive;

      base.f(); derive.f();
      b = &derive; b->f();
      smart(b); smart(d);
}
```

Example Output: Name Conflicts?


```
Base class constructor
Base class constructor
Derived class constructor
Base class: 1 , 2
Derived class: 10 , 20  Base class: 1 , 2
Base class: 1 , 2
Inside smart():  Base class: 1 , 2
Inside smart():  Base class: 1 , 2
```

Example: Design of Bird Class

```
class Bird
{
    ...
  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 reuse **Bird** to implement some special cases:

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

Example: Design of Penguin Class (1)

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

```
class Penguin : public Bird
{
    ...
  public:
    ...
    void swim();
    void catch_fish();
};
```

Oops! Penguins cannot fly!
What can we do?

Example: Design of Penguin Class (2)

Some people try to solve the problem like this:

```
void Penguin::fly()
{
    cerr << "Penguins cannot fly!" << endl;
    exit(999);
}
```

But this doesn't *really* say "Penguins cannot fly".

It says: "Penguins can fly, but they are forbidden!"

Example: Design of Penguin Class (3)

Some people try to solve the problem like this:

Penguins can fly, but the altitude is zero:

```
class Penguin : public Bird
{
      ...
   public:
      ...
      void swim();
      void catch_fish();
      void fly() { }
      int altitude() const { return 0; }
};
```

Penguin Example: What's Wrong?

Declaring **Penguin** as a derived class of **Bird** violates the substitution principle.

It is not possible to use a **Penguin** in *some* functions that work for **Bird** objects:

```
void find_food(Bird *b)
{
    b→fly();                          // visibility decreases with altitude
    double visibility = 10.0 / b→altitude();
    ...
}
```

The only solution is: REDESIGN!

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 base class and derived class:
  - The base class provides functionality and structure (methods 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 destructed *after* the derived class.

Comp151

Access Control:
public, protected, private

Example: print

Let's add a **print()** method to our U. Admin. classes.

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

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

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

Example: Doesn't Compile!

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

Student::print cannot access **Student::name**, **Student::address**, **Student::dept**.

- Since **name** is a private data member of the base class, the derived class cannot access it.
- Public inheritance does not change the access control of the data members of the base class: private members are still only available to its own methods, and not to any *other* classes including derived classes (except friends).

One Solution: Protected Data Members


```
class Person
{
  protected:
    string name;
    string address;
    Department dept;
  public:
    void print() const;
    ...
};
```

- By making **name, address, dept** protected, they are accessible to methods in the base class as well as methods in the derived classes.
- They should not be **public** though!

Member Access Control: public, protected, private

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

- **public**: members can be used by itself and the whole world; any function can access them.
- **protected**: methods (and **friends**) of itself and any derived class can use it.
- **private**: members can only be used by its own methods (and its **friends**).

- Without inheritance, **private** and **protected** have exactly same meaning.
- The only difference is that methods of a derived class can access **protected** members of a base class, but cannot access **private** members of a base class.

protected vs. private

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

- Because **protected** means that we have less 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 address by a new class called **Address** instead of string. If the **address** data member is private, we can easily make this change. The class documentation does not need to be changed.
- If it is **protected**, we have to go through all derived classes and change them. We also need to update the class documentation.

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 encapsulation.
- This is particularly true for data members, but it is less harmful to have **protected** member functions. Why?

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

Student::print Using Public Functions Only

```
void Student::print() const
{
    cout << "--- Student details ---" << endl
         << "Name:   " << get_name() << endl
         << "Addr:   " << get_address() << endl
         << "Department:   " << get_dept() << endl
         << "Enrolled in:" << endl;

    for (int i = 0; i < num_courses; i++)
        enrolled[i].print();
}
```

Teacher::print Using Public Functions Only

```
void Teacher::print() const
{
    cout << "--- Teacher details ---" << endl
         << "Name:   " << get_name() << endl
         << "Addr:   " << get_address() << endl
         << "Department:   " << get_dept() << endl
         << "Enrolled in:" << endl
         << "Rank:   " << get_rank_in_string() << endl;
    // assume get_rank_in_string returns rank in string format
}
```

Example Again

Let's use the **print** method now:

Person mouse(`"Mickey"`, `"Disney World"`, arts);
Teacher einstein(`"Albert Einstein"`, `"USA"`, physics, professor);
Student plato(`"Plato"`, `"Greece"`, philosophy);
plato.enroll_course(`"COMP151"`);

mouse.print();
einstein.print();
plato.print();

Example Again: Output

(assume: enum Department { arts, physics, philosophy, ...})

```
--- Person details ---
Name: Mickey
Addr: Disney World
Dept: 0

--- Teacher details ---
Name: Albert Einstein
Addr: USA
Dept: 1
Rank: Full Professor

--- Student details ---
Name: Plato
Addr: Greece
Dept: 2
Enrolled in:
COMP151
```

Comp151

Inheritance:

public, protected, private

Introduction

So far, we have been dealing with only *public* inheritance.

class Student : **public** Person

- 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* Person's derived classes).

Example: person.h

```
class Person {
  private:
    string name;
    string address;
    Department dept;

  protected:
    void set_name(const char *name);
    void set_address(const char *adr);
    void set_department(Department dept);

  public:
    Person(string n, string a, Department d) :
        name(n), address(a), dept(d) { };
    string get_name() const;
    string get_address() const;
    Department get_department() const;
};
```

Example: student.h


```
class Student : ???? Person
{
  private:
    Course *enrolled;
    int num_courses;

  public:
    Student(string n, string a, Department d) :
        Person(n, a, d), enrolled(NULL), num_courses(0) { }

    bool enroll_course(const string &);
    bool drop_course(const Course &);
};
```

Example: Public Inheritance

class Student: **public** Person { . . . }

public	protected	private
get_name()	set_name()	name
get_address()	set_address()	address
get_department()	set_department()	dept
enroll_course()		enrolled
drop_course()		num_courses

Example: Protected Inheritance

class Student: **protected** Person { ... }

public	protected	private
enroll_course()	set_name()	name
drop_course()	set_address()	address
	set_department()	dept
	get_name()	enrolled
	get_address()	num_courses
	get_department()	

Example: Private Inheritance

class Student: **private** Person { . . . }

public	protected	private
enroll_course()		name
drop_course()		address
		dept
		enrolled
		num_courses
		set_name()
		set_address()
		set_department()
		get_name()
		get_name()
		get_address()
		get_department()

Summary

- Public inheritance *preserves* the original accessibility of inherited members:

$$\begin{array}{rcl} \text{public} & \Rightarrow & \text{public} \\ \text{protected} & \Rightarrow & \text{protected} \\ \text{private} & \Rightarrow & \text{private} \end{array}$$

- Protected inheritance affects only public members and renders them protected.

$$\begin{array}{rcl} \text{public} & \Rightarrow & \text{protected} \\ \text{protected} & \Rightarrow & \text{protected} \\ \text{private} & \Rightarrow & \text{private} \end{array}$$

Summary ..

- Private inheritance renders all inherited members private.

$$\begin{aligned} \text{public} &\Rightarrow \text{private} \\ \text{protected} &\Rightarrow \text{private} \\ \text{private} &\Rightarrow \text{private} \end{aligned}$$

- The various types of inheritance control the highest accessibility of the <u>inherited member</u> 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.

Comp151

Polymorphism: Virtual Functions

Example: Global print Function By Reference

Because of the substitution principle, you may want to write a global print function for `Person` and its derived classes as follows:

```
void print_by_ref(const Person &person) { person.print(); }

int main()
{
    Person mouse("Mickey", "Disney World", arts);
    Teacher einstein("Albert Einstein", "USA", physics, professor);
    Student plato("Plato", "Greece", philosophy);
    plato.enroll_course(COMP151);

    print_by_ref(mouse);
    print_by_ref(einstein);
    print_by_ref(plato);
}
```

Example: Global print Function By Pointer

Or by a pointer argument:

```
void print_by_ptr(const Person *person) { person->print(); }

int main()
{
    Person mouse("Mickey", "Disney World", arts);
    Teacher einstein("Albert Einstein", "USA", physics, professor);
    Student plato("Plato", "Greece", philosophy);
    plato.enroll_course(COMP151);

    print_by_ptr(&mouse);
    print_by_ptr(&einstein);
    print_by_ptr(&plato);
}
```

Example: Output


```
--- Person details ---
Name: Mickey
Addr: Disney World
Dept: 0

--- Person details ---
Name: Albert Einstein
Addr: USA
Dept: 1

--- Person details ---
Name: Plato
Addr: Greece
Dept: 2
```

Oops!

Example: This Is What We Want

(assume: enum Department { arts, physics, philosophy, ...})

```
--- Person details ---
Name: Mickey
Addr: Disney World
Dept: 0

--- Teacher details ---
Name: Albert Einstein
Addr: USA
Dept: 1
Rank: Full Professor

--- Student details ---
Name: Plato
Addr: Greece
Dept: 2
Enrolled in:
COMP151
```

Static or Early Binding

We want

$$\textbf{void } \text{print\_by\_ref}(\textbf{const Person } \&\text{person}) \ \{ \ \text{person.print}(); \ \}$$

to call `Student::print()` when a `Student` object is passed as argument, to call `Teacher::print()` when a `Teacher` object is passed as argument, and so forth.

- However, when the compiler generates the function call in the line, it looks at the *static type* of `person` which is `const Person &`, so the method `Person::print()` is called.
- Static binding: the binding (association) of a function name to the appropriate method is done by a static analysis of the code at *compile* time based on the *static* (or *declared*)type of the object used in the call.
- The fact that the pointer can point to, (or the reference is actually a reference of) an object of a derived class is not considered.

Dynamic or Late Binding

- By default, C++ uses static binding. (Same as C, Pascal, and FORTRAN.)
- In C++, another type of binding called dynamic binding is supported through virtual functions.
- When dynamic binding is used, the method to be called is selected using the actual type of the object in the call. i.e. `print_by_ref(mouse)` (a `Person` object) would call `Person::print()`, `print_by_ref(plato)` (a `Student` object) would call `Student::print()`, and `print_by_ref(einstein)` (a `Teacher` object) would call `Teacher::print()`.

Note that the possible object types do *not* need to be known at the time that the function call is being compiled!

Virtual Functions

A virtual function is a method that is declared using the `virtual` keyword in the class definition (but *not* in the method implementation, if it is outside the class).

```
class Person {
    string name;
    string address;
    Department dept;
  public:
    virtual void print() const;
    ...
};

void Person::print() const {
    cout << "--- Person details ---" << endl;
    cout << "Name:   " << name << endl;
    cout << "Addr:   " << address << endl;
    cout << "Dept:   " << dept << endl;
}
```

Virtual Functions . . .

- Once a method 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 class, it is good style to do so, because it improves the readability of header files.

```
class Student: public Person {
    Course *enrolled;
    int num_courses;
  public:
    virtual void print() const;
    ...
};
```

- Calls to virtual functions are a little bit slower than normal function calls. The difference is extremely small and not worth worrying about, unless you write very speed-critical code.

Polymorphism

poly = multiple   *morphos* = shape

- Polymorphism in C++ means that we can work with objects without knowing their precise type at compile time:
- In: **void** print_by_ptr(**const Person** *person) { person→print(); }
  the type of the object pointed to by `person` is not known to the programmer writing this code, nor to the compiler.
  We say that `person` exhibits polymorphism, because the object can take on multiple shapes.
- Polymorphism allows us to write programs that behave correctly even when used with objects of derived classes.
- A pointer or reference *must* be used to have polymorphism. If call-by-value is used, no polymorphism can happen (WHY?).

void print_by_value(**Person** person) { person.print(); }   /* wrong use */

Example: Virtual Function


```
#include "people.h"
// class Person { ... virtual void print() const; }
// class Teacher: public Person { ... virtual void print() const; }
// class Student: public Person { ... virtual void print() const; }
// class PG_Student: public Student { ... virtual void print() const; }
void print_by_ref(const Person& person) { person.print(); }
void print_by_ptr(const Person* person) { person->print(); }

int main() {
    const int N = 4;
    Person p("Mickey", "Disney World", arts);
    Teacher t("Albert Einstein", "USA", physics, professor);
    Student s("Plato", "Greece", philosophy);
    s.enroll_course("COMP151");
    PG_Student g("Brian", "HK", computer_sci, "AI");
    g.enroll_course("COMP527");

    Person* x[N]; x[0] = &p; x[1] = &t; x[2] = &s; x[3] = &g;

    for (int j = 0; j < N; ++j) x[j]->print();              // by pointer to the base class

    for (int j = 0; j < N; ++j) print_by_ptr(x[j]);         // by pointer to the base class

    for (int j = 0; j < N; ++j) print_by_ref(*(x[j]));      // by reference to the base class
}
```

Comp151

Overriding vs. Overloading

Overriding and Virtual Functions

- When a derived class defines a method with the same name as a base class method, it <u>overrides</u> the base class method. e.g.

  `Student::print()` overrides `Person::print()`

- This is necessary if the behaviour of the base class method is not good enough for derived classes. The derived classes should all respond to the same message (`print()`!), but their response varies depending on the object.
- The designer of the base class (`Person`) must realize that this will be necessary, and declare `print()` to be a virtual function.
- Overriding is not possible if the method is not virtual.

Virtual Functions vs. Non-Virtual Functions

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

- For example, `print()` should not be a method that does something completely different.
- Overriding is for specializing a behaviour, not changing the *semantics*.
- `fly()` must do what it promises, 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 method are preserved.

Overriding vs. Overloading

Overloading: allows us to use functions or methods with the same name, but different arguments.

- The decision on which function to use (overload resolution) is done by the compiler when the program is compiled.
- There is NO dynamic binding.

Overriding: allows a derived class to provide a different implementation for a method declared in the base class.

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

Example 1

```
int main()
{
    Person *p = new Person("Bill Clinton", "White House", law);
    delete p;

    Student *s = new Student("Simpson", "Springfield", computer_sci);
    s→enroll_course("comp151");
    delete s;
}
```

This works fine.

Example 2


```
int main()
{
    Student *s = new Student("Simpson", "Springfield", computer_sci);
    s->enroll_course("comp151");

    Person *p = s;
    delete p;
}
```

- `delete p` calls the `Person` destructor. The `Student` destructor is not executed.
- The `Student` object itself is removed from the heap, but the resources it owns are not deleted.
- Therefore there is a memory leak in this code. The course array for Simpson is not destructed.

Virtual Destructor

Again, the solution is to switch on dynamic binding, in this case for the destructor:

```
class Person
{
  public:
    Person(const string n, const string a, Department d);
    virtual ~Person() { }
    ...
};
```

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

Order of Construction: Example


```
#include <iostream>

class B
{ public: B() { cout << "B's construction" << endl; } };

class D : public B
{ public: D() { cout << "D's construction" << endl; } };

int main() { B *p = new D; }
```

What is the output?

Calling Virtual Functions in Constructors: Example

```
class B {
  public:
    B() { this→f(); }
    virtual void f() { cout ≪ "B::f()" ≪ endl; }
};

class D : public B {
  public:
    D() { }
    virtual void f() { cout ≪ "D::f()" ≪ endl; }
};

int main() {
    B *p = new D;
    cout ≪ "Object created" ≪ endl;
    p→f();
}
```

Calling Virtual Functions in Constructors: Output

The output is:

```
B::f()
Object created
D::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?
- In effect, the derived class object acts like a base class object inside the base class constructor.
- 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!

Comp151

Inheritance: Abstract Base Class

ABC Example: personal_asset.h

- Let's design a system for maintaining our assets: stocks, bank accounts, real estate, horses, cars, yachts, etc.
- Each asset has a net worth (value), we would like to be able to make listings and compute total net worth.

```
class Personal_Asset {
  public:
    Personal_Asset(const Date& purchase_date);

    // What is the current net worth?
    virtual double compute_net_worth() const;
    virtual bool is_insurable() const;        // Can this asset be insured?
    void set_purchase_date(const Date& d);

  private:
    Date purchase_date;
};
```

ABC Example: bank_asset.h

There are different kinds of assets, and they are all derived from `Personal_Asset`. e.g.

```
class Bank_Account_Asset : public Personal_Asset
{
  public:
    // ...
    virtual double compute_net_worth() const { return balance; }
  private:
    double balance;
    double interest_rate;
};
```

ABC Example: asset_fcn.cc

There can be other classes of assets such as `Car_Asset`, `Stock_Asset`, `House_Asset`, etc.

To compute the total asset value for an array of assets:

```
double compute_total_worth(const Personal_Asset* assets[], int size)
{
    double total_worth = 0.0;
    for (int i = 0; i < size; i++)
        total_worth += assets[i]->compute_net_worth();

    return total_worth;
}
```

It works for any kind of asset.

ABC Example: asset_base.cc

But now we have to implement the methods of the base class `Personal_Asset`:

```
Personal_Asset::Personal_Asset(const Date& date)
        : purchase_date(date) { }

void Personal_Asset::set_purchase_date(const Date& date)
        { purchase_date = date; }

double Personal_Asset::compute_net_worth() const
{
    return /* What??? */
}
```

How should we implement `compute_net_worth( )`? It depends completely on the type of the asset. There is no "standard way" of doing it!

ABC Example: 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( )` method in the base class `Personal_Asset` as the information needed to implement it is missing.
- However, we do not want to remove the method, 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 ABC:

```
class Personal_Asset
{
  public:
    Personal_Asset(const Date& purchase_date);

    // What is the current net worth?
    virtual double compute_net_worth() const = 0;

    virtual bool is_insurable() const;        // Can this asset be insured?
    void set_purchase_date(const Date& d);

  private:
    Date purchase_date;
};
```

`compute_net_worth( )` has become a pure virtual method.

Abstract Base Class (ABC)

- An ABC has two properties:
  1. There cannot be objects of that type.
  2. Derived classes have to implement the pure virtual methods.

```
Personal_Asset ass("01/07/1997");  // error
Bank_Account_Asset acc("01/01/2000", 0.0); // ok
```

- If a derived class (for instance, `Securities_Asset`) does not implement the pure virtual methods, then the derived class is also abstract, 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 reuse

"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 reuse is an advantage of inheritance. For ABC's we do not reuse code, but create an interface that can be reused by derived classes.
- Interfaces are the soul of object-oriented programming. They are the most effective way of separating use and implementation of objects. The user (`compute_total_worth( )`) only knows about the abstract interface, while we can have many objects that implement this interface in different ways.

Final Remark

- Pure virtual functions are inherited as pure virtual functions unless the derived class implements the function.
- An abstract base class cannot be used
  - as an argument type (called by value)
  - as a function return type (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.

Example: Do and Don't

```
Personal_Asset x("01/01/2000");     // Error: can't create objects of ABC

Personal_Asset f1() { ... }   // Error: Can't return ABC objects
void f2(Personal_Asset x) { ... }   // Error: Can't CBV with ABC objects


Bank_Account_Asset y("01/01/2000", 0.0);                        // Ok!
Personal_Asset* passet = &y;                                    // Ok!
Personal_Asset& rasset = y;                                     // Ok!

Personal_Asset* f3(const Personal_Asset& x) { ... }             // Ok!
```