Inheritance & Polymorphism

Inheritance

Significant Principles

Inheritance: the language feature that makes it possible to pass features from one class to another is called inheritance——`eg.` `ostream` is termed a **base class**, `ofstream` is termed a **derived class**.

- 当 Derived class is inheriting from base class，我们可以把**子类的对象都看成是父类的对象**；
- Base class 中的**所有东西 Derived class 都有**，但是 **Base class 的 private 成员在 Derived class 中不可见**；
  - Derived class 继承了所有 base class 的方法，意味着派生类的对象可以使用基类的方法；
- Base class is always constructed first, **Derived class 在构造的时候总是先构造一个 Base class**:
  - Derived class 可以看作是**拥有 Base class 整个完整的结构**，内存结构也是相同的；
  - 如果 Base class 没有默认构造函数，我们在直接构造一个 Derived class 的时候必须要 **explicitly 传递构造 Base class 的参数**，才能通过编译；
- A base class reference can refer to a derived class object without requiring a type cast **(基类引用可以指向派生类对象)**:
  - 所以可以定义一个接受基类引用作为参数的函数，而**在调用函数时，既可以将基类对象作为参数，也可以将派生类对象作为参数**——`eg.` 参数类型为 `ostream &` 的函数可以接受 `ofstream` 对象.

Examples

`eg.` base class: `Employee` class, derived class: `Manager` class

```
class Employee {
    public:
        Employee(const std::string& name, const std::string & ssn);
        const std::string& get_name() const;
        void print(std::ostream& out) const;
        void print(std::ostream& out, const std::string & msg) const;
    private:
        std::string m_name;
        std::string m_ssn;
}
Employee::Emplyee(const string& name, const string& ssn)
    : m_name(name), m_ssn(ssn) {
        // initializer list sets up the values!
    }
inline const std::string& Employee::get_name() const {
    return m_name;
}
inline void Employee::print(std::ostream& out) const {
    out << m_name << endl;
    out << m_ssn << endl;
```

```
}
inline void Employee::print(std::ostream& out,
                            const std::string& msg) const
{
    out << msg << endl;
    print(out);
}
```

- Now add Manager class(derived class) `class Manager : public Employ:`

```
class Manager : public Employ {
    public:
        Manager( const std::string& name,
                 const std::string& ssn,
                 const std::string& title );
        const std::string title_name() const;
        const std::string& get_title() const;
        void print( std::ostream& out ) const;
    private:
        std::string m_title;
}
```

  - Think of inherited traits as an embedded object.
  - Base class is mentioned by class name `Employee::...`

```
Manager::Manager( const string& name, const string& ssn, const
string& title="" ) : Employee(name, ssn), m_title( title ) {}

inline void Manager::print( std::ostream& out ) const {
    Employee::print( out );     //call the base class print
    out << m_title << endl;
}

inline const std::string& Manager::get_title() const {
    return m_title;
}

inline const std::string Manager::title_name() const {
    return string(m_title + ": " + m_name);//access base m_name
}
```

    - Uses:

```
int main () {
    Employee bob( "Bob Jones", "555-44-0000" );
    Manager bill( "Bill Smith", "666-55-1234",
                  "ImportantPerson" );
    string name = bill.get_name(); // okay Manager
    inherits Employee
    //string title = bob.get_title(); // Error --
                                      // bob is an Employee!
    cout << bill.title_name() << '\n' << endl;
    bill.print(cout);
    bob.print(cout);
    bob.print(cout, "Employee:");
```

```
    //bill.print(cout, "Employee:"); // Error hidden!
}
```

- If you **redefine** a member function in the **derived class**, **all other overloaded functions** in the base class are **inaccessible**.

- [x] What is not inherited?
    - Constructors，Base class 的构造函数在 Derived class 中是不存在的
        - Synthesized constructors use member wise initialization;
        - In explicit copy ctor, explicitly call base-class copy ctor or the default ctor will be called instead.
    - Destructors，Base class 的析构函数在 Derived class 中是不存在的
    - Assignment operation，运算符重载是不会从 Base class 传到 Derived class 的
        - synthesized operator `=` uses member wise assignment;
        - explicit operator `=` be sure to explicitly call the base class version of operator `=`
    - Private data is hidden, but still present.

Access Protection

- Members:
    - `public:` visible to all clients;
    - `protected:` visible to classes derived from self (and to friends);
    - `private:` visible **only to self** and to friends，在子类中有，但是访问不了；
- Inheritance:

```
public:     class Derived : public Base ...
protected:  class Derived : protected Base ...
private:    class Derived : private Base ...
```

Suppose class `B` is derived from `A`, then:

Inheritance Type (`B` is)	**public**	**protected**	**private**
public `A`	public in `B`	protected in `B`	hidden
private `A`	private in `B`	private in `B`	hidden
protected `A`	protected in `B`	protected in `B`	hidden

Base class 的 `private` 变量在 Derived class 中存在，但是在 Derived class 不能访问；但是如果 Base class 的 `protected` 变量在 Derived class 中存在，并且可以直接访问。

- examples:

```
class Base {
protected:
    int alpha = 1;
public:
    int blphb = 2;
    Base( int k ) : alpha(k) {
        cout << "Base()\n";
    }
```

```
    ~Base() {
        cout << "~Base()\n";
    }
    void prt() {
        cout << alpha << " " << blphb << endl;
    }
};

class Derived : public Base{
private:
    int clphc = 3;
public:
    Derived() : Base(9) {
        cout << "Derived()\n";
    }
    ~Derived() {
        cout << "~Derived()\n";
    }
    void prt() {
        Base::prt();
        cout << alpha << endl;      // Can access alpha of Base class if it is protected
        cout << blphb << endl;
        cout << clphc << endl;
    }
};

class Eerived : public Derived {
public:
    void pp() {
        cout << alpha << endl;  // Can access alpha of Base class if it is protected
    }
};

int main() {
    /* The order of constructor */
    Eerived ed;
    ed.pp();
    return 0;
}
--------------------------------------------------------------------------------
Base()
Derived()
9
~Base()
~Derived()
```

- `alpha` 在 Base class 中是 `protected` 的，那么 `alpha` 在 public Derived class of Base class 以及 public derived class of Derived class 中都是可以进行访问的。
  - 如果 `alpha` 在 Base class 中是 `private` 的，那么在 Derived classes 都不可访问；

```
class Base {
private:
    int alpha = 1;
public:
```

```
    int blphb = 2;
    Base( int k ) : alpha(k) {
        cout << "Base()\n";
    }
    ~Base() {
        cout << "~Base()\n";
    }
    void prt() {
        cout << alpha << " " << blphb << endl;
    }
};

class Derived : public Base{
private:
    int alpha = 666;
    int clphc = 3;
public:
    Derived() : Base(9) {
        cout << "Derived()\n";
    }
    ~Derived() {
        cout << "~Derived()\n";
    }
    void prt() {
        Base::prt();
        cout << alpha << endl;      // Can access alpha of Base class if it is protected
        cout << blphb << endl;
        cout << clphc << endl;
    }
};

int main() {
    /* The order of constructor */
    Derived td;
    int *r = (int*)&td;
    for ( int i = 0; i < sizeof(Derived) / sizeof(int); i++ ) {
        cout << r[i] << " ";
    }
}
--------------------------------------------------------------------------------
Base()
Derived()
9 2 666 3
~Base()
~Derived()
```

- Derived 中含有：Base class 的 `private` 的 `alpha`，`public` 的 `blphb`，以及 Derived class 自己的 `private` 的 `alpha, clphc`
  - Base class 的 `alpha` 和 Derived 自己定义的 `alpha` 是互不影响的，Base class 的成员变量 Derived class 都有，且放在前面的内存空间；

Polymorphism