15 Polymorphism

题外话1: `auto` type

`auto`的意思是让 compiler 自己根据语境判断这是个什么 type.

```
auto i = 5;
auto d = 3.5;
```

`auto` 在比较新的 C++ version 里面可以用作 parameter type.

题外话2: `vectors`

`std::vector` 支持一种类似于 python 一样的遍历方式：

```
std::vector<int> v = {10, 11, 12, 100};
// 正常
for(int i = 0; i < int(v.size()); i++)
  std::cout << v[i] << endl;

// 简短
for(int num: v)
  cout <<num << endl;

// 甚至 auto
for (auto num: v)
  cout << num << endl;
```

复习一下 vector 添加元素的方式：

```
std::vector<int> v = {10, 11, 12, 100};
v.push_back(101);
```

Vector 去除一个元素的方式：

```
std::vector<int> vec = {10, 20, 30, 40, 50};
    int index = 2; // 假设我们想要移除索引为 2 的元素（即 30）

    // 检查索引有效性
    if (index >= 0 && index < vec.size()) {
        vec.erase(vec.begin() + index);
    }
```

15.1 Static type 和 Dynamic type

"Dynamic type" 是因为 derived class 的出现而导致的现象。

只有 1. pointer 类型以及 2. 作为引用的 variables 才会有 dynamic type！！

综上也就是说，只有 class 类型的 variables 的指针或者引用变量才有 dynamic type 这个说法.

15.1.1 Static type

一个 variable 的 static type 指的是它在 compile time 的 type，也就是我们 declare 的 type.

```
Vehicle v;
cout << v.get_insurance_amount();

Vehicle *v_ptr = &v;
cout << v_ptr -> get_insurance_amount();
// 1, v_ptr 的 static type 为 Vehicle *, 调用 Vehicle 的该函数

Car c;
cout << c.get_insurance_amount();

Car *c_ptr = &c;
cout << c_ptr -> get_insurance_amount();
// 2, c_ptr 的 static type 为 Car *, 调用 Car 的成员函数
```

15.1.2 Dynamic type: 由 derived classes 导致的 runtime type

因为 Derived classes 的存在，一些 variables (主要是 parameters) 有潜在的 dynamic type.

一个 variable 的 dynamic type 是它在 runtime 时的 type.

比如：

```
Animal* animal = new Dog();
```

这里 `animal` 的 static type 是 `Animal *`，这是因为我们 declare 它的类型是 `Animal *`；而它的 dynamic type 是 `Dog *`，因为它 runtime 时通过 `new Dog()` 成为了 `Dog *`.

作为函数的参数的变量的 dynamic type 取决于 what is passed.

比如下面这个函数的 parameter variable `vp`.

```
void fp(Vehicle *vp) { //vp 的 static type 是 Vehicle *, 因为我们这样 declare
  cout << vp -> get_insurance_amount() << endl;
}

int main() {
  Vehicle mv;
  fp(&mv);  //这里 parameter vp 的 dynamic type 是 Vehicle *, 因为 compile 时传进去的类型是个
Vehicle *.
  // 结果是 1

  Car mc;
  fp(&mc);  //这里 parameter vp 的 dynamic type 是 Car *, 因为 compile 时传进去的类型是个 Car *.
  // 但是结果还是 1
}
```

这里尽管 `vp` 的 dynamic type 是 `Car *`，我们对它使用的仍然是 `Vehicle *` 的函数，这是因为下面讲的 Static Binding。

15.1.3 Static Binding

C++ 中，by default，当我们使用一个 variable 的成员函数（以及成员）时，使用的是它的 static type 对应的 class 的成员和方法。这叫做 *Static Binding*.

（注意成员函数也只能接触它 static type 的成员，比如 `Bicycle` 如果有个成员叫 `a` 而 `Vehicle` 没有，一个变量的 dynamic type 是 `Bicycle *`，而 static type 是 `Vehicle *`，那么 by default，它也没法 access `a` 这个成员）。

比如

```
void fp(Vehicle *vp) {
  cout << vp -> get_insurance_amount() << endl;
}
```

中，如果我们：

```
Car mc;
fp(&mc);
```

我们传给 `fp` 的参数 `vp` 的是 `&mc`，因而 `vp` 这个指针变量的 Static type 是 `Vehicle *`，Dynamic type 是 `Car *`；

而由于 Static Binding，`get_insurance_amount()` 是 based on `vp` 的 Static type 的，也就是 `Vehicle` 的对应函数。因而我们用的是 `Vehicle` 下的 `get_insurance_amount()` 成员函数，所以获得结果是 `1` 而不是 `2`.

（题外话：Java 中则是默认 dynamic binding 的，这就是语言的不同。）

15.1.4 Virtual function and Dynamic Binding

"polymorph" 的意思是多个 forms.

Dynamic type 允许了 polymorphism 行为的发生，即同一个 interface 可以用来调用不同的 implementation；而 polymorphism 的使用方式就是 virtual function：给成员函数加上 `virtual` 这个关键词。

当给 member function 加上 `virtual` 之后，这个函数就变成了一个 dynamic binding 的函数。 这就是 polymorphism 的函数，因为**调用它的子类型是哪一个，它就表示哪一个子类型的 override 函数。**

```
class Vehicle {
public:
  virtual double get_insurance_amount () const;
  ...
}

class Car : public Vehicle {
public:
```

```
  double get_insurance_amount () const;
  ...
}

void fp(Vehicle *vp) {
  cout << vp -> get_insurance_amount() << endl;
}

// 由于 get_insurance_amount 现在是一个 virtual function 支持 dynamic binding, 它现在是 called on
variable 的 dynamic type 的
int main() {
  Vehicle mv;
  fp(&mv);  // 1

  Car mc;
  fp(&mc);  // 2
}
```

这里有一些点需要声明，professor 根本没有说但是笔者查阅之后觉得声明这几点对于理解这个东西是很重要的：

1. **`virtual` 必须从 base class 开始**，不存在 derived class 的 override 中这个 function 是 `virtual` 的但是 base class 中同一个 function 却不是，这种可能性。
2. **一旦 base class 中一个 member function 是 `virtual` 的，那么它的 derived class 中所有这个函数的 override 函数自动变成 `virtual` 的**，不需要加上 `virtual` 关键词。

就是说：一个成员函数要么是 virtual 的要么不是 virtual 的，不论怎么 override 这个 virtuality 都不变！这个 virtuality 也是会被继承的。

（所以说，如果从 A 的一个子类 B 开始多出了一个新的函数，也可以把它定义为 `virtual` 的，这样这个 B 作为 B 的子类的基类，它的子类使用这个函数就是 dynamic binding 的.）

（题外话：上一节我们说了虽然 `override` 关键词是可以忽略不写的，但是还是建议写，这是因为相对于 `override` 还有一个对应的关键词 `final`.

- 使用 `override` 时，编译器会检查该函数是否真的覆盖了基类中的一个虚函数。如果没有，编译器会报错。
- 使用 `final` 可以阻止进一步的继承。

）

> Exercise:
>
> ```
> class Fruit {
> public:
> int f1() {return 1;}
> virtual int f2() {return 2;}
> };
>
> class Citrus : public Fruit {
> public:
> ```

```
int f1() {return 3;}
int f2() override {return 4;}
};

class Lemon : public Citrus {
public:
int f1() {return 5;}
int f2() override {return 6;}
};

int main() {
Fruit fruit;
Cirtrus citrus;
Lemon lemon;
Fruit *fptr = &lemon;
Citrus *cptr = &citrus;

cout << fruit.f2(); //2
cout << citrus.f1(); //3
cout << fptr -> f1(); //1, 因为 fptr static 为 Fruit*, dynamic 为 Lemon*, f1 不是一个 virtual function 因而 static binding, 使用 Fruit 的 f1();
cout << fptr -> f2(); //6, 因为 f2 是一个 virtual function 因而 dynamic binding, 使用 Lemon 的 f2();
cout << cptr -> f2(); //4, cptr static 和 dynamic 都为 Citrus*

cPtr = &lemon; //更改 Citrus* 类型的变量 cPtr 的值为指向 Citrus 的子类 Lemon 的一个 object 的值, 注意这是可行的!!!

cout << cptr -> f1; //3, 因为现在 cptr static 为 Citrus* 而 dynamic 为 Lemon*, f1 不是一个 virtual function 因而 static binding, 使用 Cirtrus 的 f1();
cout << cptr -> f2; //6, f2 是一个 virtual function 因而 dynamic binding, 使用 Lemon 的 f2();
}
```

一个 **polymorphic object（多态对象）** 是至少有一个 virtual function 的 Class 的 一个 instance.

15.2 Abstract Classes

15.2.1 Pure virtual function 纯虚函数

一个 **pure virtual function** 指的是一个以 `=0` 为结尾 declare 的 function。这个结尾声明表示这个 function 是没有定义的，这就是“pure virtual"的意思。

这样的一个 function 在 Base class 中是**不能有 implementation** 的，并且**要求所有的 non-abstract derived Class 提供一个 implementation. Derived classes 的 implementation 中，override 的函数就不是一个 pure virtual function 了，结尾也没有 `=0`.**

pure virtual function 的主要目的是定义一个 interface，强制 derived classes 遵循某种 si nature （设计协议）；通过这种方式，base class 定义了一个有待派生类 implemnent 的函数框架，从而实现 polymorphism。

```
class Shape {
public:
    virtual double area() const = 0; // pure virtual function
    ...
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() const override { // pure virtual function override
        return 3.14159 * radius * radius;
    }
};
```

15.2.2 Abstract class 抽象类

一个 **abstract class** 指包含至少一个 pure virtual function 的 class.

一个 abstract class 是无法 instantiate (实例化) 的，因而也无法作为函数参数。

```
class Vehicle {
  ...
  virtual double get_insurance_amount() const = 0;
};

// 无法 instantiate
Vehicle v; //error

void f(Vehicle v) { //error: object slicing
  ...
}
```

(在C++中，"object slicing"（对象切片）是指当一个派生类对象被赋值给一个基类对象时，派生类对象中超出基类部分的信息会被切除的现象。这通常发生在通过值传递对象给函数，或者从函数返回对象时，以及在赋值操作中。)

15.2.3 将 Abstract Class 的 ptr 或 ref 变量作为函数参数

我们刚才说到，一个 abstract class 无法 instantiate，也无法作为函数参数；

但是**一个 ptr or ref to an abstract class type 则可以，并且也可以作为参数。**

```
int main() {
```

```
  // Vehicle 是一个 abstract class, Car 是它的一个 derived class
  Car c;
  Vehicle * vp = &c; // ptr
  Vehicle & vr = c; // ref
  fp(&c);
  fp(vp);
  fp(&vr);
  fr(c);
  fr(*vp);
  fr(vr);
}

void fp(Vehicle * v) {
  cout << v->get_insurance_amount() << endl;
}
void fr(Vehicle & v) {
  cout << v.get_insurance_amount() << endl;
}
```

我们发现，其实我们是需要建立一个 abstract class 的 non-abstract derived class 的 instance，然后用一个指向 abstract base class 的 ref 或 ptr 型变量来指向它，从而实现 polymorphism。

可以看到这就是 polymorphism 的工作方式。我们的 virtual member function 是基于抽象基类的，但是当我们写一些参数是抽象基类的指针或引用的函数时，我们可以把任何它的非抽象子类传递给这个函数，并且会使用这些子类的 member，从而只用了一个函数就涵盖了所有子类的不同情况，这就是 polymorphism.

15.2.4 Pure Abstract Class

需要注意的是：**abstract class 虽然不能 instantiate，但如果有 data member，则仍然需要一个 constructor。这是因为所有 instances of a derived class 都是从 construct base class 开始的。**

```
class Bicycle : public Vehicle {
//...
  Bicycle(){ cout << "Bicycle default ctor\n";}

  Bicycle(int year_in)
    : year(year_in) {cout << " Bicycle int ctor\n";}
};

// 设我们对 base class 和其他所有 derived classes 也这样

int main() {
  ElectricCar e1;
  // 输出:
  // Vehicle default ctor
  // Car default ctor
  // ElectricCar default ctor

  ElectricCar e2(1)
```

```
  ElectricCar e2(1);
  // 输出:
  // Vehicle default ctor
  // Car default ctor
  // ElectricCar int ctor
}
```

但是我们有一个新的定义：**pure abstract class，指所有函数都是 abstract functions，并且没有 data members 的 class.**

pure abstract class 就是一个纯 interface，不需要写 constructor；并且 **compiler 会自动 implicitly 为 pure abstract class 提供一个 default constructor。（普通的 abstract class 不会这样）**

15.3 容纳 different derived classes 的 ptr/ref 的 container

Polymorphism 的另一个好处就是我们可以把指向一个 base class 的不同 derived classes 的 objects 的 ptr/ref 放在同一个基类的 ptr/ref 的 container 里面。

```
Car c(2);
Bicycle b(1);
ElectricCar e(5);
vector<Vehicle*> fleet = {&c, &b, &e};

double monthly_cost = 0;
for (auto veh:fleet)
  monthly_cost += veh->get_insurance_amount();

cout << "total insurance cost = " << monthly_cost << "\n"; //根据dynamic type 对应的 member function 产生不同结果
```

（其实我们也可以直接把不同子类 objects 放进同一个 container，但是这会导致 object slicing.）

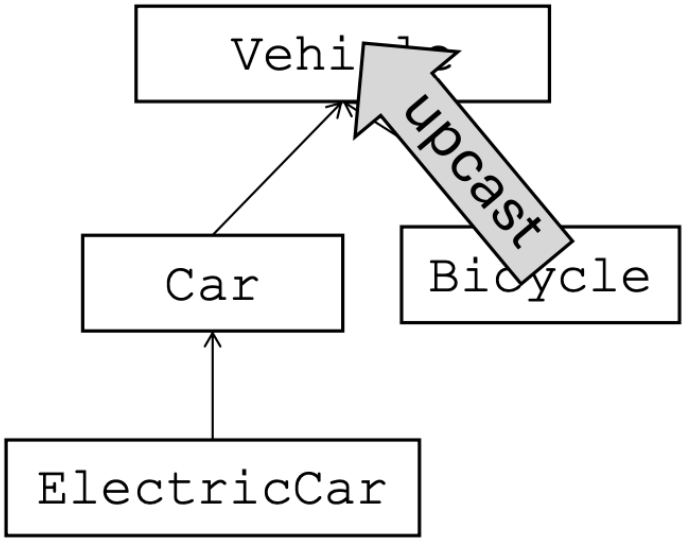
15.4 Upcasting 和 Downcasting

15.4.1 Upcasting

```
Bicycle b;
Vehicle* vp = &b; //upcasting
```

这就是一个 upcasting.

现在 `vp` 的 static type 为 `Vehicle*`，dynamic type 为 `Bicycle*`.

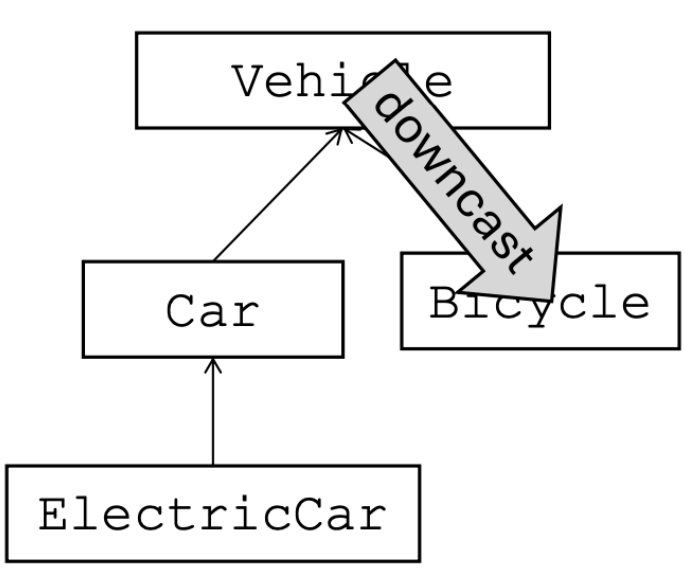


15.4.2 Downcasting

Downcasting 尝试 move down the class hierachy.

```
Bicycle b;
Vehicle* vp = &b; //upcasting

Bicycle* bp = vp; //尝试downcast, 但是error
```



compiler 报错的原因是这种做法有潜在的风险，比如：

```
Bicycle b;
Vehicle* vp = &b; //upcasting

Car c;
vp = &c;

Bicycle* bp = vp; //如果允许这样downcast那这个时候就不对了
```

因而有一个function `dynamic_cast<Derived*>()` 作为 downcasting 的途径。

```
Bicycle b;
Vehicle* vp = &b; //upcasting

Bicycle* bp = dynamic_cast<Bicycle*>(vp); //downcast
```

如果 cast fail 了，那么 `bp` 会成为一个 `nullptr`

如果我们想 `dynamic_cast` 一个 ref 而不是 ptr，那么如果 cast fail 了就会报错。

15.4.3 不要这样使用 `dynamic_cast`

```
if(it_is_ElectricCar)
  do_electric_thing();
else if(it_is_car)
  do_car_thing;
else ...
```

因为如果要给这个 hierarchy 添加一个新的 class，那么我们要找到所有这样的代码并把这个新 class 的东西加进去。