14 Derived Classes and Inheritance

```
class Vehicle {
private:
    int num_wheels;
    string color;
    int year;
    void check_invariant();
public:
    Vehicle();
    Vehicle(int nw_in, string color__in, int year_in);
    double get_insurance_amount();
};
```

现在我们想要创建一个 Special type of Vehicles 叫做 `Car`，

对于 `Car`，我们需要更多的只属于 `Car` 自己的 Member functions，比如 `replace_tires()`，其他的 Vehicles 是没有的。

并且，我们需要更换一些 member functions，比如 `get_insurance_amount()`，因为 Car 的 insurance 和其他 Vehicles 是不一样的。

于是我们需要一个 **Derived Class**.

14.1 Derived Classes

我们的 `Vehicle` class 是一个 **base class**，我们可以基于它建立一些 **derived Classes**.

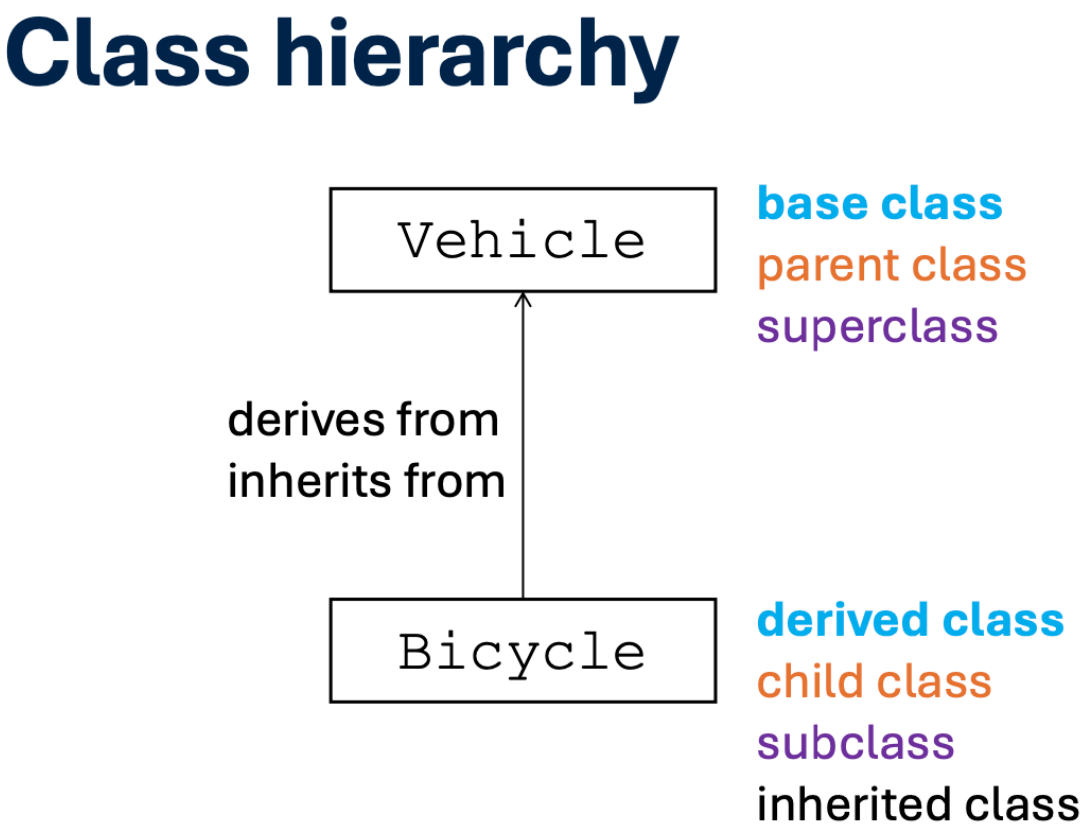
一个 Derived class 可以 **inherit** base class 所有的 member variables 和 member functions.

并且它还可以只对于自己这个 derived class 更改和添加 member functions 和 member variables，来建立独特的 functioning 和 invariants.

```
class Bicycle: public Vehicle {
  // empty
}
```

这里 bicycle 自动继承了所有的 member variables 和 functions of `Vehicle`，**除了 constructor之外**。

14.1.1 自动 inherit



14.1.2 添加 member variables 和 member functions

```
class Bicycle : public Vehicle {
private:
  string type; // a new member variable
public:
  Bicycle();
  Bicycle(string color_in, int year_in, string type_in);
  void set_type(string type_in);
};
```

```
string Bicycle::get_type() const {
  return type;
}

string Bicycle::set_type(string type_in) {
  type = type_in;
}
```

14.1.3 添加 Derived Class 自己的 check invariant

```
class Bicycle : public Vehicle {
private:
  string type; // a new member variable
  void Bicycle::check_invariant();
public:
  Bicycle();
  Bicycle(string color_in, int year_in, string type_in);
  void set_type(string type_in);
};

void Bicycle::check_invariant() {
  if(type == "eBike") {assert(year >= 1997 && year <= 2024);}
}

Bicycle::Bicycle()
  :color("red"), year_in(1997), type_in(MOUNTAIN) {}

Bicycle::Bicycle(string color_in, int year_in, string type_in)
  :color(color_in), year_in(year_in), type_in(type_in) {
    check_invariant(); // 对于非 default 的 constructor, 需要 check invariant以防止破坏
interface
  }
```

14.1.4 Derived Class 无法 assess Base Class 中的 Private members

```
class Vehicle {
private:
  int num_wheels;
  string color;
  int year;
    ...
public:
  int get_year();
  void set_year();
  ...
}
```

```
class Bicycle : public Vehicle {
private:
  Bicycletype type;
  void check_invariant();
};
```

这个时候：

```
void Bicycle::check_invariants() {
  if(type == ELECTRIC)
  assert(year >= 1997 && year <= 2024); // error
  assert(get_year() >= 1997 && get_year() <= 2024); //可以
}
```

`Bicycle` 的确继承了 `year` 这个 Base Class 的 member，但是是无法 access 的。对于父类的 private mamber，子类必须通过父类的 public function 来 access.

14.1.5 keyword: `protected` 表示子类能够 access 的父类 private member

对于刚才这个问题，我们还有一个办法，就是把我们想要让子类也能够 access 的 private member 设置成 `protected` 而不是 `private`.

```
class Vehicle {
protected:
  int year;
  int num_wheels;
  ...
private:
  ...
public:
  ...
}
```

总结一下：

member 有三种安全性级别，从低到高：

1. `public`: 所有 code 都能 access.
2. `protected`: 这个 class 和它所有的 derived classes 里的 code 都能 access.
3. `private`: 只有这个 class 里面的 code 能够 access.

14.1.6 Scope resoluton operator `::`

我们可以在 derived class 中 call base class 的 functions，但是必须要加上 `::`

```
void Bicycle::check_invariants(){
  Vehicle::check_invariants(); // 调用 base class 里的 functions
  if(type == ELECTRIC) {
    assert(get_year() >= 1997 && get_year() <= 2024);
  }
}
```

14.2 Enums

一个 `enum` 即 enumerated type，是一种 定义 **named int constants** 的方法。

一个 `enum` 类的数据类型，它的一个 object 就是一个 `int`，

我们在这个 `enum` 中给出了一些 constants，这些 constants，如果没有特别指定每一个分配的 `int` 值是多少，会被自动分配 0, 1, 2, ... 的值.

而我们可以定义一个该 `enum` 类变量类型的变量，给他赋一个 constant。比如 `Bicycletype a = MOUNTAIN`，就是把 `MOUNTAIN` 这个 constant 也就是 `0` 赋给了 `a`.

`enum` 的目的是让代码结构和意图更加清晰。

```
enum Bicycletype {
  MOUNTAIN,
  ROAD,
  ELECTRIC,
  RACING
};

class Bicycle : public Vehicle {
private:
    Bicycletype type; // type 是一个 Bicycletype
};

void Bicycle::check_invariants() {
  if(type == ELECTRIC) {assert(get_year() >= 1997 && get_year() <= 2024);}
  // 如果 type 是 ELECTRIC 也就是值为 2, 那么执行这个函数
}
```

这里 `Bicycletype` 是一个我们定义的变量类型，这个变量类型里有 4 个不同的可选值：`MOUNTAIN`, `ROAD`, `ELECTRIC` 和 `RACING`. 分别对应了0，1，2，3；这就像是 `bool` 类型只有 2 个不同的可选值 `True` 和 `False` 分别对应了 1 和 0 一样。

14.3 Derived class constructor

Constructors 并不会被 inherited.

因而 Derived class 也需要自己的 Constructor.

但是，Constructors 会自动 strat with the base class，也就是说 `Bicycle` 的 constructor 不需要写任何 `Vehicle()` 的 default constructor 的重复内容，而是会自动添加。我们只需要在 其中添加 Bicycle 的独特属性就可。

14.3.1 Derived class 的 Constructor 会在 initializer list 中自动调用 Base Class 的 default constructor

```
Bicycle::Bicycle()
  : type(MOUNTAIN) {}
```

我们这里经历的过程是：

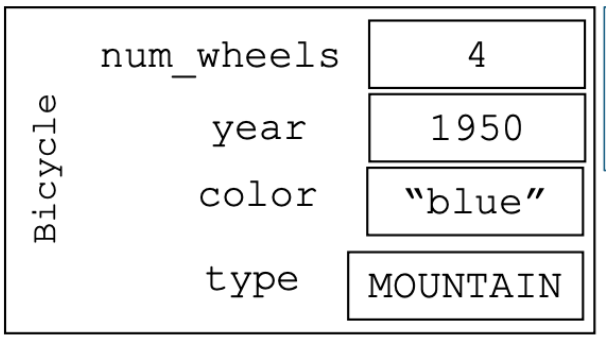
1. Implicitly calls default `Vehicle` constructor
2. Executes (rest of) `Bicycle` member initializer list
3. Executes `Bicycle` constructor body (这里是空的)

而我们也可以 explicitly call `Vehicle()` 的 constructor. 下面这个和上面这个是等价的。

```
Bicycle::Bicycle()
  : Vehicle(), type(MOUNTAIN) {}
```

我们这里经历的过程是：

1. Explicitly calls default `Vehicle` constructor
2. Executes (rest of) `Bicycle` member initializer list
3. Executes `Bicycle` constructor body (这里是空的)



于是如果我们没有专门 specify `Bicycle` 这个 derived class 与 base class 在共有属性上的不同，运行它的 constructor 一定会继承 `Vehicle()` 的 default constructor 中 specify 的属性。

14.3.2 通过 base class 非default constructor 来制作 derived class 的 default constructor 的 member initializer list

我们可以这样写 `Bicycle` 的 default constructor:

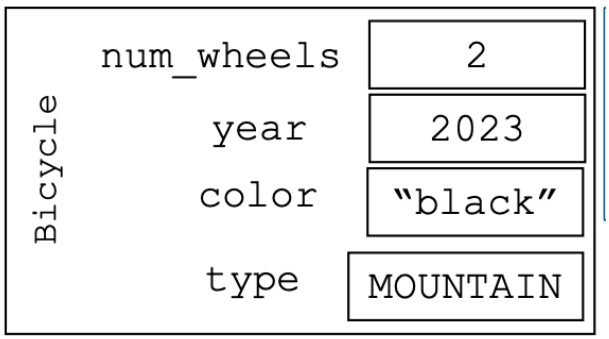
```
Bicycle::Bicycle()
  :Vehicle(2, "black", 2023),
  type(MOUNTAIN) {}
```

这里我们:

我们这里经历的过程是:

1. Explicitly calls non-default `Vehicle` constructor
2. Executes (rest of) `Bicycle` member initializer list
3. Executes `Bicycle` constructor body (这里是空的)

于是我们每个 default 的 Bicycle 都变成了这样的:



我们需要注意的是：member initializer list 是在 derived class 的 constructor 中调用 base class 的唯一方法！在框体里调用就会出问题！

比如我们:

```
Bicycle::Bicycle()
  :type(MOUNTAIN) {
    Vehicle(2, "black", 2023),
}
```

这里我们实际上是 create 了一个 新的 `Vehicle` object，并且马上把它扔掉了，因为我们没有用 variable 存它（`Bicycle` 没有类型为 `Vehicle` 的 member variable，并且即便有，这样也没存)

所以这个 initialization 并不成功。

14.4 在 Derived Class 中 Override base class 的 member functions

```
class Vehicle {
private:
    Vehicle();
    Vehicle(string color_in,int year_in);
    int num_wheels;
    string color;
    int year;
    void check_invariant();
public:
    ...
    double get_insurance_amount();
};
```

```
class Bicycle : public Vehicle {
public:
  Bicycle();
  Bicycle(string color_in,int year_in);
  double get_insurance_amount();
private:
  double Bicycle::get_insurance_amount(){
    return 5;
  }
};
```

14.4.1 Override 和 Overload 的区别

一个 Function override 指的是 derived class 的 function，其 name and prototype 和 parent 的一样.

```
Vehicle::get_insurance_amount();
Bicycle::get_insurance_amount();
```

一个 Function overload 指的是同一个 class 中两个不同的 function，name 相同但是 prototype 不一样，表示在接收不同参数时 function 不同表现。

```
Vehicle::Vehicle();
Vehicle::Vehicle(int nw_in, string color_in, int year_in);
```