

13 ADT in C++

13.1 Class Declaration

13.1.1 前情提要: class 型的 object 可以使用 `=` 赋值

13.2 Calling a member function

13.2.1 自动创建的 `this` 作为指针变量

13.2.2 `const` function

13.2.3 `this` 不仅可以 `->` member variable 还可以 `->` member function

13.2.4 `const` 配对: `const` 的 object 不能使用非 `const` 的 member function

13.2.5 参数包含其他同 class object 的 member function

13.2.6 不使用 `this` implement member function: compiler 自动补全

13.3 Member Accessibility (或称 visibility)

13.4 Initialization

13.4.1 使用 member initializer list 的 Constructors

13.4.2 多个 Constructors

13.4.3 Classes 作为 members

13.4.4 Check Invariants

13 ADT in C++

`struct` 是 C-style heterogeneous aggregate data type, 而 `class` 是 C++-style heterogeneous aggregate data type.

`struct` 和 `class` 的区别在于:

1. `struct` 的 member variables 必须 call init function 来 initialize, 而 `class` 可以写 default constructor 自动 initialize
2. `struct` 的所有 data 都是 `public` by default 的, `class` 的所有 data 都是 `private` by default 的.
3. `struct` 只包含 member variables, functions 是 technically separate 的, 而 `class` 包含了自己的 member variables 和 member functions.

因而用 class 可以:

- Enforce the interface
- Maintain representation invariants

13.1 Class Declaration

格式:

```
class Triangle {  
private: // private(一般是variables/data)  
    double a;  
    double b;  
    double c;  
};
```

```

public: // public(大部分functions)
    Triangle(double a_in, double b_in, double c_in) {
        ...
    } // Constructor(s)

    double perimeter() const { // 注意不改变 member variables 的 function 要加 const
        ...
    }

    void scale(double s) {
        ...
    }

}; // 注意和struct一样要加semicolon!

```

initialize 一个 instance object:

```

int main() {
    Triangle t1{3, 4, 5};
}

```

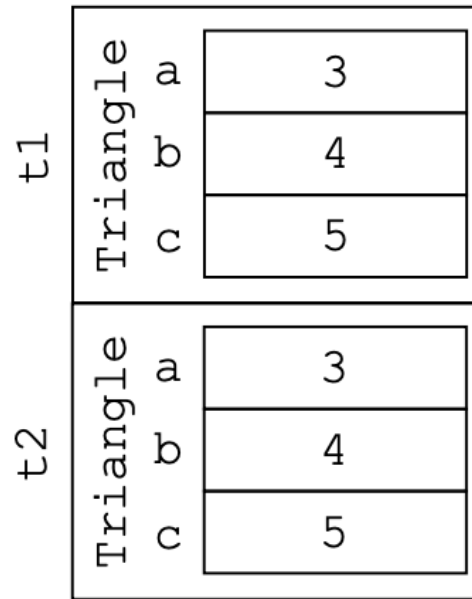
13.1.1 前情提要: class 型的 object 可以使用 = 赋值

```

int main() {
    Triangle t1(3, 4, 5);
    Triangle t2 = t1;
    ...
}

```

能这么做。



13.2 Calling a member function

一个 Class 的每个 object 都有它自己的 copy of each member variable.

调用 Class 的 member function 总是在某个 object 上, 使用 `.` 调用。比如 `Triangle` 有一个 member function 叫做 `scale()`, 那么我们无法直接调用 `scale()`, 而是必须在某个 instance object 上调用, 比如 `t1` 是一个 object, `t1.scale()`.

13.2.1 自动创建的 `this` 作为指针变量

记得在 `struct` 中, 我们是先自己创建某个 object 的一个指针变量或者直接使用 `&t1` 把地址传进去作为一个指针型变量, 然后把它传给一个以指针 `Triangle *ptr` 作为参数的 function, 这个 function 通过 `ptr -> a` 这样使用 `->` operator 获取传进来的指针作为参数指向对象的 member variable.

而 `class` 中就不同。我们可以在 member function 中使用 `this` 这个关键词, 使得当一个 object 调用这个 function 时自动创建一个指向这个 object 的指针 `this`, 然后通过 `this -> a` 这样使用 `->` operator 来 access 原对象参数的 member variable.

The object on which a function is called is pointed to by the pointer named `this`.

就是说如果我们在 member function 中用 `this` 这个关键词, 意思就是 `this` 是指向调用这个函数的 object 的一个指针。

```
void Triangle::scale(double s) {
    this -> a *= s;
    this -> b *= s;
    this -> c *= s;
}
```

```
int main() {
    Triangle t1(3, 4, 5);
    t1.scale(2);
    // 6 8 10
}
```

对比一下 C-style (`struct`):

```
void Triangle_scale(Triangle *tri, double s) {
    tri->a *= s;
    tri->b *= s;
    tri->c *= s;
}
```

```
int main() {
    Triangle t1;
    Triangle_init(&t1, 3, 4, 5);
    Triangle_scale(&t1, 2);
    // 6 8 10
}
```

13.2.2 `const` function

在使用 `struct` 的函数中我们如果需要不改变原 object 就需要使用 **pointer to const**.

```
double Triangle_perimeter(Triangle const *tri) { // pointer to const
    return tri->a + tri->b + tri->c;
}

int main() {
    Triangle t1;
    Triangle_init(&t1, 3, 4, 5);
    cout << Triangle_perimeter(&t1); // 使用 & 传地址作为指针变量
}
```

而在 `class` 中我们可以直接给 member function 加上 `const` 表示它不改变调用它的原 object 的值。

```
double Triangle::perimeter() const{ // const function
    return this->a + this->b + this->c;
}

int main() {
    Triangle t1(3, 4, 5);
    cout << t1.perimeter(); // 不需要传入地址
}
```

13.2.3 `this` 不仅可以 `->` member variable 还可以 `->` member function

```
int operation() {
    this -> scale(2.0);
    cout << this -> perimeter();
}
```

`this -> scale()` 就是 `*(this).scale()`.

表示在 member function 里调用其他 member function.

13.2.4 `const` 配对: `const` 的 object 不能使用非 `const` 的 member function

```
class Triangle {
private:
    ...
public:
    double scale(double s);
    double perimeter() const;
    ...
}

int main() {
    const Triangle t1(3, 4, 5);
    t1.scale(); // error, 因为scale不是const function. 它在尝试strip off t1's const
    cout << t1.perimeter(); //ok
}
```

13.2.5 参数包含其他同 class object 的 member function

```
int double(const Triangle &anothertriangle) const {
    this->scale(2.0); //不行,
    cout << this->perimeter(); //可以

    anothertriangle.scale(2.0); //也不行, 因为 anothertriangle 是一个 const reference
    cout << anothertriangle.perimeter(); //可以
}
```

13.2.6 不使用 `this` implement member function: compiler 自动补全

```
class Triangle {
private:
    double a, b, c;
public:
    void scale(double s) {
        this->a *= s;
        this->b *= s;
        this->c *= s;
    }
    void shrink(double s) {
        this->scale(1.0 / s);
    }
};
```

我们可以直接不写 `this ->`. compiler 会自动帮我们补全 `this ->`.

等于:

```
class Triangle {
private:
    double a, b, c;
public:
    void scale(double s) {
        a *= s;
        b *= s;
        c *= s;
    }
    void shrink(double s) {
        scale(1.0 / s);
    }
};
```

13.3 Member Accessibility (或称 visibility)

我们知道 private 的 member 只能在 class scope 里 access，也就是通过 public member function 来 access.

但是我们要补充一点是：只要是同一个 class，那么这个 class 其他 object 的 member 也可以由 member function access. 也就是说一个 member function 不只可以用 `this ->` access 自己的 private 属性，还可以随使用 `.` access 其他同类 object 的 private 属性.

比如：

```
class Triangle {
private:
    double a;
    double b;
    double c;
public:
    bool isSame(const Triangle &someOtherTriangle) {
        return a == someOtherTriangle.a &&
               b == someOtherTriangle.b &&
               c == someOtherTriangle.c;    // private的，但是可以
    }
};

int main() {
    Triangle t1(3, 4, 5);
    Triangle t2(3, 4, 7);
    cout << t1.isSame(t2); // 不会有 error
}
```

这就凸显了 class 的灵活性。

13.4 Initialization

Every object in C++ is initialized upon creation.

1. explicitly initialized

```
Triangle t1(3, 4, 5);
Triangle t2 = Triangle(3, 4, 5);
```

2. default initialized

```
Triangle t3;
```

Atomic objects (int, double, bool, char, pointers) default initialization 自动赋一个 junk 值.

Array objects default initialization 给每个元素自动赋一个 junk 值.

Compound objects 通过 default constructor 来 default initialization. 这个 constructor 为 `Triangle(){}` , 是 compiler 自动加的, 给每个 member variable 赋了 junk。一旦你自己写写了其他 **constructor**, 这个 **default** 就失效了! 这个时候你再 `Triangle t` 就会报 **error!**

```
int main() {
    int y; // contains junk
    int array2[3]; //each element contains junk
    Triangle t3; // 根据 default constructor
}
```

```
Triangle::Triangle(double a_in, double b_in, double c_in) {
    a = a_in;
    b = b_in;
    c = c_in;
}

int main() {
    Triangle t; // error!
}
```

13.4.1 使用 member initializer list 的 Constructors

实际上, 刚才这样的 initializer 是一个很低效而且有时候不会起作用的方式 (之后会讲)。每个 member variable 都被 default initialized 了一次之后又被重新设置了一个值。

我们有一个更加高效的方式叫做 **member initializer list**:

```
class Triangle {
private:
    double a;
    double b;
    double c;
public:
    Triangle(double a_in, double b_in, double c_in)
        : a(a_in), b(b_in), c(c_in) {} //这里的顺序不重要。{}里面不需要写东西
};
```

13.4.2 多个 Constructors


```

class Triangle {
private:
    double a;
    double b;
    double c;
public:
    Triangle(double a_in, double b_in, double c_in)
        : a(a_in), b(b_in), c(c_in) {}
    Triangle(double side)
        : a(side), b(side), c(side) {}
    Triangle()
        : a(1), b(1), c(1) {}
};

```

我们可以创建多个 constructor，针对不同的 arguments 数量。

注意：使用 **0-argument construtor** 的写法比较特殊，是必须写 `Triangle t;` 而不能写 `Triangle t();`，后者会被 **compiler** 认为是 **declare** 了一个 **return Triangle** 的 **function** 叫做 `t`，没有参数。

注意2：如果我们没有写某个 argument 数量的 constructor，那就不能用。比如这里我们没有写两个参数的 constructor，所以 `Triangle t(3, 4);` 会报错。

• What does line B do?

```

class Triangle {
private:
    double a, b, c;

public:

    Triangle()
        : a(1), b(1), c(1) { }

};

```

```

A    Triangle t1;
B    Triangle t2();
    }

```

Question

- A) It does the same thing as line A.
- B) The () syntax can't be used in declarations, so this doesn't compile.
- C) t2 is created as a triangle, but initialized with memory junk.
- D) t2 is declared as a function that returns a Triangle.**
- E) It calls the constructor as a function, but doesn't create a Triangle object.

13.4.3 Classes 作为 members

```
class Professor {
private:
    string name;
    vector<string> students;
    Coffee favCoffee;
    Triangle favTriangle;

public:
    Professor(const string &name)
        : name(name), favCoffee(0, 0, false);
};
```

有一点注意是，我们不要这样写：

```
Professor(const string &name)
    : name(name), favCoffee = Coffee(0, 0, false);
```

因为这样的话 compiler 就会先 default construct `favCoffee`，然后再用这个 constructor 构建，which causes error because 这个时候已经没有 default constructor 了。

13.4.4 Check Invariants

应该写一个 respect interface 的函数，并且塞在 constructor 里。

```
class Triangle {
private:
    double a;
    double b;
    double c;

    void check_invariants() {
        assert(0 < a && 0 < b && 0 < c);
        assert(a + b < c && a + c > b && b + c > a)
    }

public:
    Triangle(double a_in, double b_in, double c_in)
        : a(a_in), b(b_in), c(c_in) {
        check_invariants();
    }
};
```