静态成员与友元

主讲：陈笑沙

目录

7.1 为什么需要静态成员

什么是静态成员?

7.1 为什么需要静态成员

什么是静态成员?

- 当用关键字 `static` 说明一个类成员时，该成员为静态成员。

7.1 为什么需要静态成员

什么是静态成员?

- 当用关键字 `static` 说明一个类成员时，该成员为静态成员。
- 静态成员分为:

7.1 为什么需要静态成员

什么是静态成员?

- 当用关键字 `static` 说明一个类成员时，该成员为静态成员。
- 静态成员分为:
  - 静态成员变量

7.1 为什么需要静态成员

什么是静态成员?

- 当用关键字 `static` 说明一个类成员时，该成员为静态成员。
- 静态成员分为:
  - 静态成员变量
  - 静态成员函数

7.1 为什么需要静态成员

静态成员变量的定义和初始化

```
1 class A {
2 public:
3     static int property;
4 };
5
6 int A::property = 1;
```

7.1 为什么需要静态成员

静态成员变量的定义和初始化

```
1 class A {
2 public:
3     static int property;
4 };
5
6 int A::property = 1;
```

7.1 为什么需要静态成员

静态成员变量的定义和初始化

```
1 class A {
2 public:
3     static int property;
4 };
5
6 int A::property = 1;
```

7.1 为什么需要静态成员

静态成员变量的定义和初始化

```
1 class A {
2 public:
3     static int property;
4 };
5
6 int A::property = 1;
```

7.1 为什么需要静态成员

静态成员变量的特性

7.1 为什么需要静态成员

静态成员变量的特性

- 类的所有对象共享静态成员变量，因此无论建立多少个该类的对象，静态成员变量只有一份拷贝。

7.1 为什么需要静态成员

静态成员变量的特性

- 类的所有对象共享静态成员变量，因此无论建立多少个该类的对象，静态成员变量只有一份拷贝。
- 静态成员变量属于类，而不属于具体对象。

7.1 为什么需要静态成员

静态成员变量的特性

- 类的所有对象共享静态成员变量，因此无论建立多少个该类的对象，静态成员变量只有一份拷贝。
- 静态成员变量属于类，而不属于具体对象。
- 静态成员变量的生命周期与全局变量相同。

7.1 为什么需要静态成员

个别属性描述所有对象的共性

```
1 class EnemySlime {
2     int maxHealth;
3     float maxSpeed;
4 };
```

7.1 为什么需要静态成员

个别属性描述所有对象的共性

```
1 class EnemySlime {
2     int maxHealth;
3     float maxSpeed;
4 };
```

但是到目前为止的语法中，所有类只能描述对象的属性，而无法描述共有属性。

7.1 为什么需要静态成员

7.1 为什么需要静态成员

是否可以使用全局变量?

7.1 为什么需要静态成员

是否可以使用全局变量?

```
1 int slimeMaxHealth;
2 float slimeMaxSpeed;
3 class Slime { ... };
```

7.1 为什么需要静态成员

是否可以使用全局变量?

```
1 int slimeMaxHealth;
2 float slimeMaxSpeed;
3 class Slime { ... };
```

- 容易命名冲突

7.1 为什么需要静态成员

是否可以使用全局变量?

```
1 int slimeMaxHealth;
2 float slimeMaxSpeed;
3 class Slime { ... };
```

- 容易命名冲突
- 无法控制可访问性（代码任意位置均可见）

7.1 为什么需要静态成员

是否可以使用全局变量？

```
1 int slimeMaxHealth;
2 float slimeMaxSpeed;
3 class Slime { ... };
```

- 容易命名冲突
- 无法控制可访问性（代码任意位置均可见）
- 尽量不要使用全局变量（不好的代码风格）

7.2 如何使用静态成员

7.2 如何使用静态成员

- 静态数据成员也有 `public` 和 `private` 之分

7.2 如何使用静态成员

- 静态数据成员也有 public 和 private 之分
  - 类外只能访问 public 属性的静态数据成员

7.2 如何使用静态成员

- 静态数据成员也有 `public` 和 `private` 之分
  - 类外只能访问 `public` 属性的静态数据成员
  - 类内可以访问所有属性的静态数据成员

7.2 如何使用静态成员

- 静态数据成员也有 public 和 private 之分
  - 类外只能访问 public 属性的静态数据成员
  - 类内可以访问所有属性的静态数据成员
- 静态数据成员是属于类的

7.2 如何使用静态成员

- 静态数据成员也有 `public` 和 `private` 之分
  - 类外只能访问 `public` 属性的静态数据成员
  - 类内可以访问所有属性的静态数据成员
- 静态数据成员是属于类的
  - 访问方法：`ClassName::StaticMember`

7.2 如何使用静态成员

- 静态数据成员也有 `public` 和 `private` 之分
  - 类外只能访问 `public` 属性的静态数据成员
  - 类内可以访问所有属性的静态数据成员
- 静态数据成员是属于类的
  - 访问方法：`ClassName::StaticMember`
- 当对象不存在时，也可以访问类的静态数据成员

7.3 静态成员变量

```
1 class Slime {
2     static int maxHealth;
3     static int maxSpeed;
4     int health;
5     int speed;
6 };
```

7.3 静态成员变量

```
1 class Slime {
2     static int maxHealth;
3     static int maxSpeed;
4     int health;
5     int speed;
6 };
```

7.3 静态成员变量

```
1 class Slime {
2     static int maxHealth;
3     static int maxSpeed;
4     int health;
5     int speed;
6 };
```

7.3 静态成员变量

```
1 class Slime {
2     static int maxHealth;
3     static int maxSpeed;
4     int health;
5     int speed;
6 };
```

7.3 静态成员变量

```
 1 class Slime {
 2 public:
 3     void heal(int amount) {
 4         health = std::min(health + amount, maxHealth);
 5     }
 6
 7     void acc(int amount) {
 8         speed += amount;
 9         if (speed > maxSpeed)
10             speed = maxSpeed;
11         else if (speed < -maxSpeed)
12             speed = -maxSpeed;
13     }
14 };
```

7.3 静态成员变量

```cpp
class Slime {
public:
    void heal(int amount) {
        health = std::min(health + amount, maxHealth);
    }

    void acc(int amount) {
        speed += amount;
        if (speed > maxSpeed)
            speed = maxSpeed;
        else if (speed < -maxSpeed)
            speed = -maxSpeed;
    }
};
```

7.3 静态成员变量

```
class Slime {
public:
    void heal(int amount) {
        health = std::min(health + amount, maxHealth);
    }

    void acc(int amount) {
        speed += amount;
        if (speed > maxSpeed)
            speed = maxSpeed;
        else if (speed < -maxSpeed)
            speed = -maxSpeed;
    }
};
```

7.3 静态成员变量

```
 1 class Slime {
 2 public:
 3     void heal(int amount) {
 4         health = std::min(health + amount, maxHealth);
 5     }
 6
 7     void acc(int amount) {
 8         speed += amount;
 9         if (speed > maxSpeed)
10             speed = maxSpeed;
11         else if (speed < -maxSpeed)
12             speed = -maxSpeed;
13     }
14 };
```

7.3 静态成员变量

```
 1 class Slime {
 2 public:
 3     void heal(int amount) {
 4         health = std::min(health + amount, maxHealth);
 5     }
 6
 7     void acc(int amount) {
 8         speed += amount;
 9         if (speed > maxSpeed)
10             speed = maxSpeed;
11         else if (speed < -maxSpeed)
12             speed = -maxSpeed;
13     }
14 };
```

7.4 静态成员函数

静态成员函数的定义：

```
1 class A {
2   static void func() {  ... }
3 };
```

7.4 静态成员函数

看以下例子：

```
 1 class Vector2D {
 2 private:
 3     double x, y;
 4 public:
 5     Vector2D(x = 0.0, y = 0.0) : x(x), y(y) {}
 6     Vector2D& add(const Vector2D& other) {
 7         x += other.x;
 8         y += other.y;
 9         return *this; // why this?
10     }
11 
12     std::string to_string() const {
13         std::ostringstream oss;
14         oss << "{" << x << ", " << y << "}";
15         return oss.str();
```

7.4 静态成员函数

看以下例子：

```
class Vector2D {
private:
    double x, y;
public:
    Vector2D(x = 0.0, y = 0.0) : x(x), y(y) {}
    Vector2D& add(const Vector2D& other) {
        x += other.x;
        y += other.y;
        return *this; // why this?
    }

    std::string to_string() const {
        std::ostringstream oss;
        oss << "{" << x << ", " << y << "}";
        return oss.str();
```

7.4 静态成员函数

看以下例子：

```
class Vector2D {
private:
    double x, y;
public:
    Vector2D(x = 0.0, y = 0.0) : x(x), y(y) {}
    Vector2D& add(const Vector2D& other) {
        x += other.x;
        y += other.y;
        return *this; // why this?
    }

    std::string to_string() const {
        std::ostringstream oss;
        oss << "{" << x << ", " << y << "}";
        return oss.str();
```

7.4 静态成员函数

看以下例子：

```
class Vector2D {
private:
    double x, y;
public:
    Vector2D(x = 0.0, y = 0.0) : x(x), y(y) {}
    Vector2D& add(const Vector2D& other) {
        x += other.x;
        y += other.y;
        return *this; // why this?
    }

    std::string to_string() const {
        std::ostringstream oss;
        oss << "{" << x << ", " << y << "}";
        return oss.str();
```

7.4 静态成员函数

看以下例子：

```
private:
    double x, y;
public:
    Vector2D(x = 0.0, y = 0.0) : x(x), y(y) {}
    Vector2D& add(const Vector2D& other) {
        x += other.x;
        y += other.y;
        return *this; // why this?
    }

    std::string to_string() const {
        std::ostringstream oss;
        oss << "{" << x << ", " << y << "}";
        return oss.str();
    }
        return oss.str();
```

7.4 静态成员函数

看以下例子：

```
    double x, y;
public:
    Vector2D(x = 0.0, y = 0.0) : x(x), y(y) {}
    Vector2D& add(const Vector2D& other) {
        x += other.x;
        y += other.y;
        return *this; // why this?
    }

    std::string to_string() const {
        std::ostringstream oss;
        oss << "{" << x << ", " << y << "}";
        return oss.str();
    }
};
```

7.4 静态成员函数

返回自身可以做到链式调用：

```
 1 #include "vector2d.hpp"
 2 #include <iostream>
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 int main(int argc, char **argv) {
 8   Vector2D v;
 9   v.add({1, 2}).add({3, 4});
10   cout << v.to_string() << endl;
11   return 0;
12 }
```

7.4 静态成员函数

返回自身可以做到链式调用：

```
#include "vector2d.hpp"
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char **argv) {
  Vector2D v;
  v.add({1, 2}).add({3, 4});
  cout << v.to_string() << endl;
  return 0;
}
```

7.4 静态成员函数

返回自身可以做到链式调用：

```cpp
#include "vector2d.hpp"
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char **argv) {
  Vector2D v;
  v.add({1, 2}).add({3, 4});
  cout << v.to_string() << endl;
  return 0;
}
```

7.4 静态成员函数

返回自身可以做到链式调用：

```
 1 #include "vector2d.hpp"
 2 #include <iostream>
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 int main(int argc, char **argv) {
 8   Vector2D v;
 9   v.add({1, 2}).add({3, 4});
10   cout << v.to_string() << endl;
11   return 0;
12 }
```

7.4 静态成员函数

返回自身可以做到链式调用：

```
#include "vector2d.hpp"
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char **argv) {
  Vector2D v;
  v.add({1, 2}).add({3, 4});
  cout << v.to_string() << endl;
  return 0;
}
```

7.4 静态成员函数

返回自身可以做到链式调用：

```
 1 #include "vector2d.hpp"
 2 #include <iostream>
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 int main(int argc, char **argv) {
 8   Vector2D v;
 9   v.add({1, 2}).add({3, 4});
10   cout << v.to_string() << endl;
11   return 0;
12 }
```

7.4 静态成员函数

返回自身可以做到链式调用：

```
 1 #include "vector2d.hpp"
 2 #include <iostream>
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 int main(int argc, char **argv) {
 8   Vector2D v;
 9   v.add({1, 2}).add({3, 4});
10   cout << v.to_string() << endl;
11   return 0;
12 }
```

7.4 静态成员函数

但是加法非得是一个向量加到另一个向量上么?

```
1 class Vector2D {
2   static Vector2D add(const Vector2D& v1, const Vector2D& v2)
3     Vector2D v(v1.x + v2.x, v1.y + v2.y);
4     return v;
5   }
6 };
```

7.4 静态成员函数

但是加法非得是一个向量加到另一个向量上么?

```
1 class Vector2D {
2   static Vector2D add(const Vector2D& v1, const Vector2D& v2)
3     Vector2D v(v1.x + v2.x, v1.y + v2.y);
4     return v;
5   }
6 };
```

7.4 静态成员函数

但是加法非得是一个向量加到另一个向量上么?

```
1 class Vector2D {
2   static Vector2D add(const Vector2D& v1, const Vector2D& v2)
3     Vector2D v(v1.x + v2.x, v1.y + v2.y);
4     return v;
5   }
6 };
```

7.4 静态成员函数

但是加法非得是一个向量加到另一个向量上么?

```
 1 #include "vector2d.hpp"
 2 #include <iostream>
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 int main(int argc, char **argv) {
 8   Vector2D v;
 9   v.add({1, 2}).add({3, 4});
10   cout << v.to_string() << endl;
11
12   Vector2D v2{100, 200};
13   Vector2D v3 = Vector2D::add(v, v2);
14   cout << v3.to_string() << endl;
15   return 0;
```

7.4 静态成员函数

但是加法非得是一个向量加到另一个向量上么?

```
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char **argv) {
  Vector2D v;
  v.add({1, 2}).add({3, 4});
  cout << v.to_string() << endl;

  Vector2D v2{100, 200};
  Vector2D v3 = Vector2D::add(v, v2);
  cout << v3.to_string() << endl;
  return 0;
}
```

7.4 静态成员函数

另外一个例子：/example/lec07/itemManage

7.5 为什么需要友元

思考一个问题：

如果有一个 `List` 类，一个 `Node` 类，我们该如何实现 `list.append(Node&)`

7.5 为什么需要友元

7.5 为什么需要友元

- Node 类中的next是 private 的。

7.5 为什么需要友元

- Node 类中的next是 private 的。
- List 类中的 head 的是private的。

7.5 为什么需要友元

- Node 类中的next是 private 的。
- List 类中的 head 的是private的。
- 但是需要在 list 中对 node 的 next 进行修改。

7.5 为什么需要友元

7.5 为什么需要友元

- 封装的目的就是为了实现信息隐蔽。

7.5 为什么需要友元

- 封装的目的就是为了实现信息隐蔽。
- 一个对象的私有成员只能被自己的成员访问到。当类外的对象或函数要访问这个类的私有成员时，只能通过该类提供的公有成员间接地进行。

7.5 为什么需要友元

- 封装的目的就是为了实现信息隐蔽。
- 一个对象的私有成员只能被自己的成员访问到。当类外的对象或函数要访问这个类的私有成员时，只能通过该类提供的公有成员间接地进行。
- C++提供了友元机制来打破私有化的界限，即一个类的友元可以访问到该类的私有成员。

7.6 如何使用友元

例子：/example/lec07/list

7.6 如何使用友元

7.6 如何使用友元

- 友元具有如下的性质:

7.6 如何使用友元

- 友元具有如下的性质：
- 类的友元可以直接访问它的所有成员。

7.6 如何使用友元

- 友元具有如下的性质：
- 类的友元可以直接访问它的所有成员。
- 友元的声明必须放在类的内部，但放在哪个段没有区别。

7.6 如何使用友元

- 友元具有如下的性质：
- 类的友元可以直接访问它的所有成员。
- 友元的声明必须放在类的内部，但放在哪个段没有区别。
- 友元关系不具备对称性，即X是Y的友元，但Y不一定是X的友元。

7.6 如何使用友元

- 友元具有如下的性质：
- 类的友元可以直接访问它的所有成员。
- 友元的声明必须放在类的内部，但放在哪个段没有区别。
- 友元关系不具备对称性，即X是Y的友元，但Y不一定是X的友元。
- 友元关系不具备传递性，即X是Y的友元，Y是Z的友元，但X不一定是Z的友元。

7.6 如何使用友元

好的代码风格：尽量避免使用友元！

练习

- 实现 Vector2D 的**静态**与**非静态**版本的以下方法
  - sub
  - dot
  - cross
- 利用 vector 实现Matrix2D，实现以下方法：
  - add
  - scale （矩阵和标量的乘法）
  - mult （矩阵和矩阵的乘法）
- 实现 vector 以下方法
  - 非静态成员函数：applyTransform(const Matrix2D&)
  - 静态成员函数：mult(const Matrix2D&, const Vector2D&)