Classes

一些术语

> *Polymorphism:多态*
>
> *instantiate:声明*
>
> *derived class:派生了*
>
> *inheritance:继承*

1.Classes

为什么要`Class类`

- `C语言`没有`Class类`
- 没有任何把数据和对数据的操作封装在一起的方式
- 没有OOP的Coding范式
- `struct`成员都是`public`的

	头文件 (.H)	源文件 (.CPP)
目的	定义接口	实现类函数
包含	函数原型、类声明、类型定义、宏、常量	函数实现、可执行代码
访问	在源文件间共享	编译成目标文件
示例	`void someFunction();`	`void someFunction() {...};`

- `constructor`构造函数
    - `constructor`是一个特殊的成员函数，用于初始化新创建的对象 它的主要作用和特点包括：

- **初始化对象**：构造函数可以设置对象的初始状态，给成员变量赋予初值。
- **自动调用**：当创建对象时，构造函数会自动被调用，无需手动调用。
- **资源管理**：在需要分配资源（如动态内存、文件句柄等）的类中，构造函数可以用于分配这些资源。

- 默认初始化和参数初始化
  - ```
    //默认初始化
    StudentID::StudentID() {
        name = "John Appleseed";
        sunet = "jappleseed";
        idNumber = 00000001;
    }
    StudentID student; // 调用默认构造函数

    //参数初始化
    StudentID::StudentID(std::string name,
    std::string sunet, int idNumber) {
        this->name = name;
        this->sunet = sunet;
        this->idNumber = idNumber;
    }
    StudentID student("Alice", "aalice", 123456);
    // 调用带参数构造函数
    ```

  - 列表初始化类似`uniform initialization`统一初始化
    - ```
      StudentID::StudentID(std::string name,
      std::string sunet, int idNumber)
          : name{name}, sunet{sunet},
      idNumber{idNumber} {}

      StudentID student{"Alice", "aalice",
      123456};
      // {} 使用带参数的构造函数进行初始化
      ```
    -

    - 构造函数中的 `this->` 用于区分成员变量与参数。`name`、`sunet` 和 `idNumber` 将被初始化为传入的参数值。
- `destructor`析构函数
    - **析构函数**主要用于在对象生命周期结束时进行清理工作，确保资源得到适当释放，防止内存泄漏
    - 析构函数不是显式调用的，它在对象超出作用域时会自动调用
    - ```
      StudentID::~StudentID()
      {
          // free/deallocate any data here
      }
      ```
    - `~`是析构函数的标识符,他不接受参数也没有返回值
- 创建同义标识符
    - ```
      class StudentID {
      private:
          // An example of type aliasing
          //同义标识符String 和string
          //using 声明具有作用域 而且会进行类型检查
          using String = std::string;
          String name;
          String sunet;
          int idNumber;
      public:
          // constructor for our student
          Student(String name, String sunet, int
      idNumber);
          // method to get name, state, and age,
      respectively
          String getName();
          String getSunet();
          int getID();
      }
      ```
    -

2.Inheritance(继承)

- 动态多态性：不同类型的对象可能需要相同的接口
- 可扩展性：继承允许你通过创建具有特定属性的子类来扩展一个类。

继承类型

> *public:公有*
>
> *private:私有*
>
> *protected:保护*

类型	公有继承	保护继承	私有继承
示例	`class B: public A {...}`	`class B: protected A {...}`	`class B: private A {...}`
公有成员	在派生类中为公有	在派生类中为保护	在派生类中为私有
保护成员	在派生类中为保护	在派生类中为保护	在派生类中为私有
私有成员	在派生类中不可访问	在派生类中不可访问	在派生类中不可访问

如果直接是`class Student : public Person`和`class Employee :public Person`继承的话,`SectionLeader`的构造会被调用两次，需要让这个两个类是虚函数继承`class Student : public virtual Person`和`class Employee : public vitural Person`

```
#include <iostream>
#include <cstring>
#include <cstdint>

class Person {
protected:
    std::string name;

public:
```

```
    Person(const std::string& name) : name(name) {}
    std::string getName() const {
        return name;
    }
};

class Student : public virtual Person {
protected:
    std::string idNumber;
    std::string major;
    std::string advisor;
    uint16_t year;

public:
    Student(const std::string& name, /* other parameters */);
    std::string getIdNumber() const;
    std::string getMajor() const;
    uint16_t getYear() const;
    void setYear(uint16_t year);
    void setMajor(const std::string& major);
    std::string getAdvisor() const;
    void setAdvisor(const std::string& advisor);
};

class Employee :public virtual Person{
protected:
    double salary;

public:
    Employee(const std::string &name){};
    virtual std::string getRole() const = 0;
    virtual double getSalary() const = 0;
    virtual void setSalary() const = 0;
    virtual ~Employee() = default;
};

class SectionLeader : public Person, public Employee {
protected:
    std::string section;
    std::string course;
    std::vector<std::string> students;
```

```
public:
    SectionLeader(const std::string& name, /* other parameters */);
    std::string getSection() const;
    std::string getCourse() const;
    void addStudent(const std::string& student);
    void removeStudent(const std::string& student);
    std::vector<std::string> getStudents() const;
    std::string getRole() const override;
    double getSalary() const override;
    void setSalary(double salary) override;
    virtual ~SectionLeader() noexcept = default;
};
```