一、类和对象

1. 类

> 1. 类是构成对象的一个蓝图|基石， 可以把类当成是汽车的图纸（抽象）， 现实生活中实实在在的汽车就是对象（真实存在的对象）
> 2. 可以拥于属性（用于表示数据） | 变量 name ，属性name , 数据 name
> 3. 可以拥有方法 (用于表示行为动作) | 方法read , 函数 sleep , 行为 | 动作 run ， sleep
> 4. 可以隐藏数据和方法
> 5. 可以对外提供公开的接口（方法）
>
>   请定义一个教师类，它具备name 和 subject 属性， 还具备教书的行为。

```
#include<string>

using namespace std;

class Student{
private:
    string name;   // 姓名
    int age;  //年龄

public:
    void read(){
        std::cout << "学生在看书" << std::endl;
    }
};
```

2. 对象

类就好比是汽车的图纸， 对象就好比是实实在在的汽车（真实的汽车）

1. 在栈中创建对象

> 用这种方法创建的对象，内存分配到栈里（Stack）。 直接使用 `.` 来访问成员。当程序对象所在的代码块结束运行（方法运行结束），对象会自动删除，内存自动释放。

```
class student{
    ...

}

int main(){
    //对象存储于栈内存中
    student stu1 ;
    student stu2 ;

    return 0 ;
}
```

2. 在堆中创建对象

> 这种方法创建的对象，内存分配到堆里（Heap）。一般使用 `*` 或者 `->` 调用对象的方法。箭头操作符"->"将解引用（dereferencing*）和成员使用（member access.）结合起来，

```
#include<iostream>
#include<string>

using namespace std;

class Student{
public:
    void run(){
        cout << "学生在跑步" << endl;
    }

}

int main(){

```

```
    //对象存储于栈内存中 new 关键字是在堆中申请动态内存，所以这里不能写成 s3 而应该是指针。

    Student *s3 = new Student;
    s3->run();

    *s3.run();
    return 0 ;
}
```

3. 访问修饰符

> c++ 对于成员的访问有三种访问操作符 `public` | `private` | `protected`， 默认情况下是private
>
> 1. `public`：公开权限，任何位置都可以访问
> 2. `private`：私有权限，只能自己内部访问及其友元类和函数
> 3. `protected`：类内、子类及友元类和函数

```
#include<iostream>
#include<string>

using namespace std;

class Student{
    private: //表示下面的成员为私有
      string name;   // 姓名
      int age;  //年龄

    public: //表示下面的函数为公开
        void run(){}
};

ing main(){
    Student stu ;

    stu.name = "张三" ; // 禁止访问
    stu.run(); //允许访问
    return 0 ;
}
```

4. 实现类的成员函数

1. 类中实现 或 外部实现

> 1. 成员函数可以在类中声明时直接实现，也可以在类的外部。
> 2. 可以在类的外部实现成员函数， 需要使用 类名::函数名

```
#include <iostream>
#include <string>

using namespace std;

class Student{

    private :
        int age ;
        string name;

    public :
        void read(string bookname){
            cout<< bookname << endl;
        }

       void speak(string something);
}


void Student::speak(string something){
    cout << "说说说---" << something << endl;
}

int main(){

    Student stu;

    stu.read("哈利波特");
    stu.speak("我喜欢看哈利波特");

    return 0 ;
```

```
}
```

2. 分离声明和实现

> 声明放到 头文件中，实现放到cpp文件中 。 头文件的声明，需要在前后包裹一段特殊的样式代码。这段代码确保了一旦该头文件多次包含执行时，出现重复定义的错误。

> 如下所示： 当第一次包含`Student.h`时，由于没有定义`_STUDENT_H_`，条件为真，这样就会包含（执行）`#ifndef _STUDENT_H_`和`#endif`之间的代码，当第二次包含`Student.h`时前面一次已经定义了`_STUDENT_H_`，条件为假，`#ifndef_STUDENT_H_`和`#endif`之间的代码也就不会再次被包含，这样就避免了重定义了。

- Student.h

```
//后面的大写表示标识，可以随意命名.
#ifndef HELLOWORLD_STUDENT_H
#define HELLOWORLD_STUDENT_H

#include <string>
using namespace std;

class Student{

private :
    int age ;
    string name;

public :
    void read(string bookname);

    void speak(string something);
};


#endif //HELLOWORLD2_STUDENT_H
```

- Student.cpp

```
#include "student.h"

#include <iostream>
using namespace std;

void Student::read(string bookname){
cout << "看书: "<<bookname<<endl;
}

void Student::speak(string something){
cout << "说说说---" << something << endl;
}
```

- main.cpp

```
#include <iostream>
#include "student.h"

int main() {

    Student s;
    s.read("哈利波特");
    s.speak("hi harry");

    return 0;
}
```

- CMakeList.txt

```
cmake_minimum_required(VERSION 3.14)
project(HelloWorld2)

set(CMAKE_CXX_STANDARD 14)

# 需要在后面添加student.cpp 因为main.cpp 依赖该文件
add_executable(HelloWorld main.cpp student.cpp)
```

二、特殊成员函数

> 当定义一个类后，它的对象在未来的操作中，总会不可避免的总会碰到如下的行为：`创建`、`拷贝`、`赋值`、`移动`、`销毁`。 这些操作实际上是通过六种特殊的成员函数来控制的：`构造函数`、`析构函数` `拷贝构造函数`、`拷贝赋值函数`、`移动构造函数`、`移动赋值函数`。默认情况下，编译器会为新创建的类添加这些函数（**默认不会添加 移动构造和移动赋值**），以便它的对象在未来能够执行这些操作。

1. 构造函数

1. 一般方式构造

> 构造函数是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。与类名同名，没有返回值，可以被重载，通常用来做初始化工作。

```
#include<string>
using namespace std;

class student{

    string name;
    int age ;

    public :
        //构造函数
        student(){
            cout << "执行无参构造函数" << endl;
        }

        student(string name ){
            cout << "执行含有一个参数的构造函数" << endl;
        }

        student(string name , int age ){
            cout << "执行含有两个参数的构造函数" << endl;
        }

};
```

```
int main(){

    //创建三个对象，会执行三个对应你的构造函数
    student s1 ;
    student s1{"张三"};
    student s1{"张三", 28};

    return 0 ;
}
```

2. 初始化列表方式

> 在之前成员的初始化工作，都是在构造函数的函数体里面完成的。如果使用初始化列表，那么成员的初始化赋值是在函数体执行前完成.

```
#include <iostream>
#include <string>

using namespace std;

class student{
    string name;
    int age;

    /*
    //早期的方式
    student(string name_val , int age_val){
       name = name_val;
       age = age_val;
    }
    */

    //更好的方式
    student(string name ,int age):name{name},age{age}{
        cout << "执行有参构造函数" <<endl;
    }

};

int main(){
    //编译允许通过，输出 a1 和 a2 为 30 和20 ，小数点省略
```

```
    int a (30.22);
    int a = 20.33;

    //编译失败，不允许赋值。防止类型收窄看精度丢失。
    //int a{20.33};

    student s("张三"  ,  18);

    return 0 ;
}
```

2. 析构函数

> 类的**析构函数**是类的一种特殊的成员函数，与构造函数正好相反，它会在每次删除所创建的对象时执行。

> 析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀，它不会返回任何值，也不能带有任何参数。不能被重载，一般用于释放资源。

```
#include <iostream>
#include <string>

using namespace std;

class Student{

    string name;
    int age ;

    public :
        //构造函数
        Student(){
            cout << "执行无参构造函数" <<endl;
        }
        Student(string name ){
            cout << "执行含有一个参数的参构造函数" <<endl;
        }
        Student(string name , int age ){
            cout << "执行含有两个参数的构造函数" <<endl;
        }
```

```
        //析构函{}
        ~Student(){
            cout << "执行析构函数" <<endl;
        }
};

int main(){

    Student *s1 = new Student();
    Student *s2 = new Student();
    Student *s3 = new Student();


    //释放对象
    delete s1;
    delete s2;
    delete s3;

    return 0 ;
}
```

3. 拷贝构造函数

1. 初探拷贝

> C++中经常使用一个常量或变量初始化另一个变量， 比如：

```
int mian(){

    int a = 3;
    int  b = a;


    stu s1;
    stu s2 = s1;


    return 0 ;
}
```

> 使用类创建对象时，构造函数被自动调用以完成对象的初始化，那么能否象简单变量的初始化一样，直接用一个对象来初始化另一个对象呢?

> 不难看出，s2对象中的成员数据和 s1 是一样的。相当于将s1中每个数据成员的值复制到s2中，这是表面现象。实际上，系统调用了一个拷贝构造函数。

```
#include <iostream>
#include <string>

using namespace std;

class student{

    public :
       string name;
        int age ;

        student(string name , int age ){
            cout << "执行含有两个参数的构造函数" << endl;
        }

        ~student(){
            cout << "执行析构函数" <<endl;
        }
};

int main(){

    Student s1{"张三" , 19 };
    cout << s1.name << " : " << s1.age <<ednl;

    Student s2 = s1;
    cout << s2.name << " :: " << s2.age <<ednl;

    return 0 ;
}
```

2. 浅拷贝

> 指的是在对象复制时，只对对象中的数据成员进行简单的赋值，默认拷贝构造函数执行的也是浅拷贝。如果数据中有属于`动态成员`（在堆内存存放），那么浅拷贝只是做指向而已，不会开辟新的空间。默认情况下，编译器提供的拷贝操作即是浅拷贝。

```
include <iostream>
include <string>

using namespace std;

class Student {
public:
    int age ;
    string name;

public :
    //构造函数
    Student(string name , int age ):name(name),age(age){
        cout<< " 调用了 构造函数" << endl;
    }

    //拷贝构造函数
    Student(const Student & s){
        cout << "调用了拷贝构造函数" << endl;
        age = s.age;
        name = s.name;

    }

    //析构函数
    ~Student(){
        cout << "调用了析构函数" << endl;
    }
};

int main(){

    Student s1("张三" , 18);
    cout << s1.name << " : " << s1.age <<endl;
```

```
    Student s2 = s1;
    cout << s2.name << " :: " << s2.age <<endl;

    return 0 ;
}
```

3. 浅拷贝引发的问题

> 默认情况下，浅拷贝已经足以应付日常的需求了，但是当类中的成员存在动态成员（指针）时，浅拷贝往往会出现一些奇怪的问题。

```
#include <iostream>
#include <string>

using namespace std;

class Student {
public:
    string name;
    string *address;

    Student(string name , string *
address):name(name),address(address){
        cout << "执行构造函数" << endl;
    }


   // 这里还是默认的浅拷贝。 由于address是指针类型，如果是浅拷贝，那么两个指针会指向同一个位置。
 Student(const Student & s){
        cout << "调用了拷贝构造函数" << endl;

        name = s.name;
       address = s.address;

    }

     //析构函数
    ~Student(){
        cout << "调用了析构函数" << endl;
```

```
        //这里将会删除两次内存空间
        delete address;
        address = nullptr;
    }

};

int main(){
    string address="深圳";
    Student s1("张三" , &address);

    //此处会执行拷贝。
    Student s2 = s1;
    cout << s2.name << " : " << s2.address << endl;


    //修改第一个学生的地址为：北京
    *s1.address = "北京"

    //第二个学生的地址也会变成北京
    cout << s2.name << " : " << s2.address << endl;


    return  0 ;
}
```

4. 深拷贝

> `深拷贝` 也是执行拷贝，只是在面对对象含有`动态成员| 指针`时，会执行新内存的开辟，而不是作简单的指向。在发生对象拷贝的情况下，如果对象中动态成员，就不能仅仅简单地赋值了，而应该重新动态分配空间如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配。

```
#include <iostream>
#include <string>

using namespace std;

class Student {
public:
```

```
    string name;
    string *address;

    Student(string name , string *
address):name(name),address(address){
        cout << "执行构造函数" << endl;
    }

    //深拷贝
    Student(const Student & s){
            cout << "调用了拷贝构造函数" << endl;
            age = s.age;
            name = s.name;

         if(address == nullptr){
             //开辟新的空间
             address = new string();
             *address = s.address;
         }
    }

      //析构函数
     ~Student(){
         cout << "调用了析构函数" << endl;

         if(address != nullptr){
             delete address;
             address = nullptr;
         }
     }

};

int main(){
    string address="深圳";
    Student s1("张三" , &address);

    //此处会执行拷贝。
    Student s2 = s1;
    cout << s2.name << " : " << s2.address << endl;
```

```
    //修改第一个学生的地址为：北京
    *s1.address = "北京"

    //第二个学生的地址也会变成北京
    cout << s2.name << " : " << s2.address << endl;


    return  0 ;
}
```

###