# 一、类和对象

## 1. 类

> 1. 类是构成对象的一个蓝图
> 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
>
> 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

### 1. 在栈中创建对象

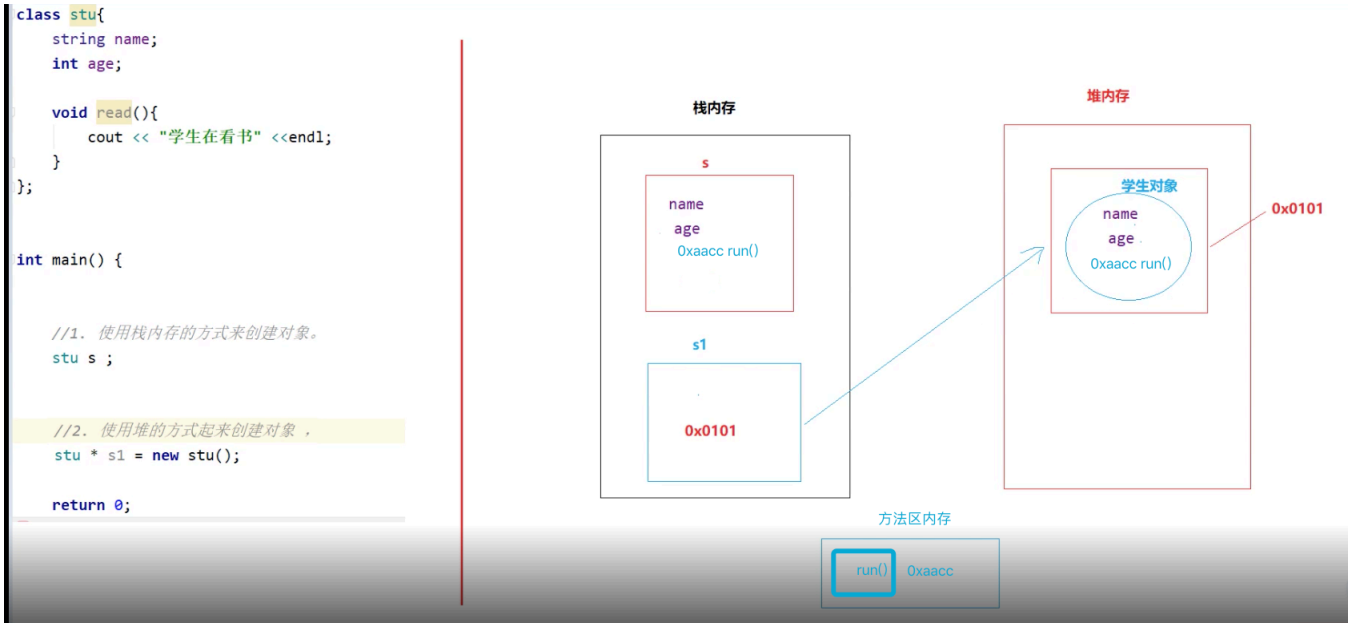
> 用这种方法创建的对象，内存分配到栈里（Stack）。 直接使用 . 来访问成员。当程序对象所在的代码块结束运行（方法运行结束），对象会自动删除，内存自动释放。

```
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，

```cpp
#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

> python中修饰成员访问权限，采用的是下划线的形态来修饰。
>
> c++ 对于成员的访问有三种访问操作符 `public` | `prive` | `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函数
> 3. `protected` ：类内、子类及友元函数可访问

```
#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你函数， 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

```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

```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)
```

## 5 .练习

> 定义一个计算器类 calc , 包含有 加减乘除 4个函数，要求在calc.h中声明类， 在calc.cpp中实现该类的函数。

# 二、特殊成员函数【重点和难点】

> 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。 在 python 中，有类似的 `____init___` 函数用于初始化工作

```
#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，并且初始化列表有一个优点是：

防止类型收窄， 换句话说就是 精度丢失

int a= 3.5

int b {3.5};

有三种情况需要使用到构造函数初始化列表

- 情况一、需要初始化的数据成员是对象的情况(这里包含了继承情况下，通过显示调用父类的构造函数对父类数据成员进行初始化)；
- 情况二、需要初始化const修饰的类成员或初始化引用成员数据；
- 情况三、子类初始化父类的私有成员；

```cpp
#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 ;
}
```

# 3. 委托构造函数

> 一般来说，如果给类提供了多个构造函数，那么代码的重复性会比较高，有些构造函数可能需要包含其他构造函数中已有的代码，为了让编码工作更简单，C++11 允许程序员在一个构造函数的定义中使用另一个构造函数。这种现象被称为 委托 .

## 1. 早前的构造函数写法

> 早期的做法是，每个构造函数完成变量的赋值工作。

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

using namespace std;

class Student{

    string name;
    int age ;

    public:
            //无参构造
            Student():name{"张三"},age{19}{

        };

            //一个参数构造
            Student(string name_val):name{name_val},age{19}{

        };

            //两个参数
            Student(string name_val , int age_val ):name{name_val},age{age_val}{

        };
};

int main(){
        Student s1;
        Student s2("李四");
        Student s3("李四" , 20 );
        return 0 ;
}
```

## 2. 委托构造函数写法：

> 委托构造函数实际上就是已经有一个构造函数定义好了所有初始化工作的逻辑，那么剩下的构造函数就不用做这个活了，全部交给它来做即可。有点类似，A调用C， B调用C ， 而C 把所有的初始化代码都写完了。那么A和B只是负责委托C来完成即可

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

using namespace std;

class Student{

    string name;
    int age ;

    public:

                    //无参构造 委托给两个参数的构造函数
            Student():Student{"张三" ,  19}{

        };


             //一个参数构造 委托给两个参数的构造函数
            Student(string name_val):Student{name_val , 19}{

        };

            //由该函数完成最终的成员赋值工作
            Student(string name_val , int age_val):name{name_val},age{age_val}{

        };
};

int main(){
        Student s1;
        Student s2("李四");
        Student s3("李四" , 20 );

        return 0;
}
```

## 3.构造函数的类型转化:

类型转化只适用于单参的构造函数

上例中，系统先b的值创建一个临时对象，然后将临时对象的值赋给p1；

要防止隐式转换可以通过在构造函数前添加`explicit`关键字实现，此时实现类型转换需要用强制类型转换（也称显式转换），如`p1 = (Int) b`

构造函数使用时–
1.对象进行构造时时默认调用的函数，在对象生存周期内（由系统）只调用一次
2.函数名与类名一致
3.构造函数无函数返回类型说明；但实际上构造函数有返回值，为其创建的对象
4.可重载
5.未定义时，系统默认生成一个默认构造函数（除 this 指针外，没有参数的构造函数）

# 2. 析构函数

> 和python一样，c++也有析构函数。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。

```cpp
#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 ;
}
```

# 练习

> 定义一个学生类，包含 属性： 姓名、年龄， 行为：读书、跑步。
>
> 1. 使用初始化列表的方式完成对 姓名和 年龄的赋值工作。
> 2. 在main.cpp中分别调用跑步和读书的函数，打印内容形如： 18岁的张三在读书...
> 3. 在析构函数当中释放姓名的空间

> 综合演练题目: 友情提示:请阅读完所有要求后，再开始写代码
>
> 1.定义学生类stu，包含
>
> 1. 属性: 姓名(string*)， 学号(string)， 6个学科的分数(vector)
>
> 2.函数:
>
> 无参构造、
>
> 有参构造
>
> show()用于输出学生姓名、总分、平均分、各个学科分数。
>
> 2.学生类应该具有stu.h和stu.cpp 两个文件
>
> 1.stu.h用于声明属性以及以上三个函数的声明
>
> 2.stu.cpp负责实现构造函数与show函数。
>
> 3.在main函数里面，定义局部变量vector<stu *> *stu_vector
>
> 注意:所谓的全局函数指的是在main.cpp里面定义，不属于任何类。
>
> 4.定义一个全局函数initStu负责从键盘录入成绩获取3个人，6个学科成绩
>
> 5.定义一个全局函数updateStu, 用于更新分数，把每个人的不及格的成绩全部修改成99

分，

6.定义一个全局函数printStu, 遍历打印三个人的每个学科成绩。

1.定义两种全局打印函数

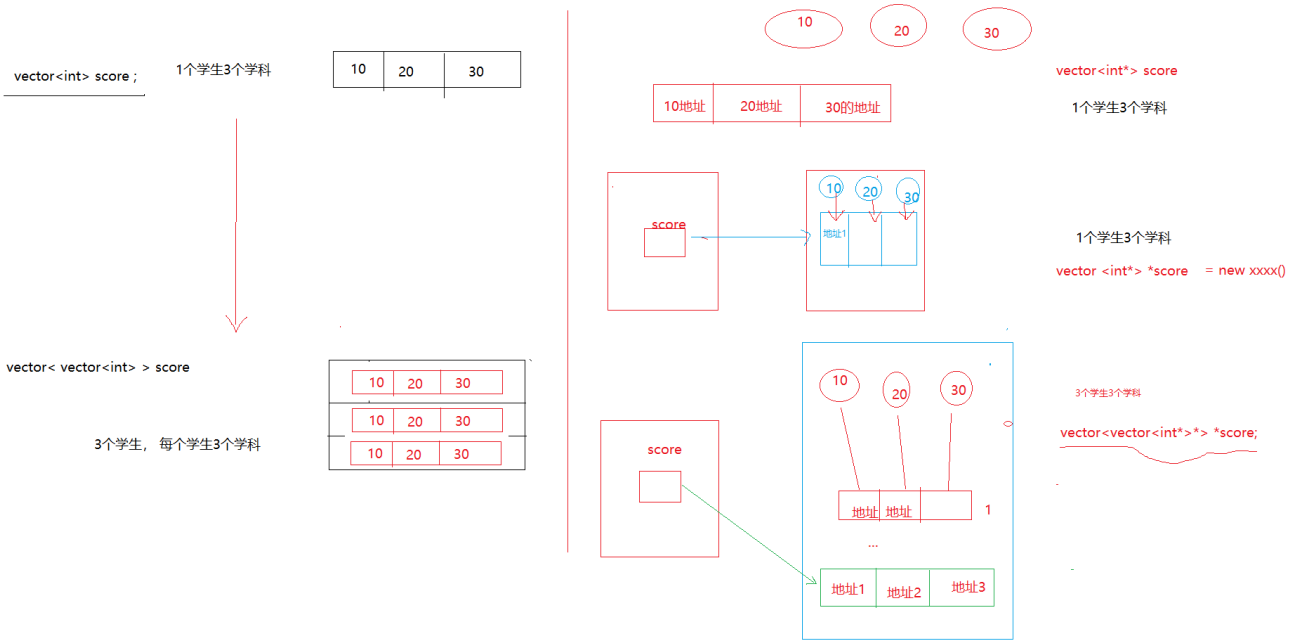
a.基于范围的for循环方式: printForRange,

b.使用fori的方式:printForI .

2.在执行printStu，应该先询问用户，要使用何种方式打印

(1:基于范围for方式，2:使用forI方式)

3.根据对应的选择，来调用printStu分数，并且传入上面打印方式的函数。




# 3. 拷贝构造函数[重点中的重点 VIP中P]

## 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;

    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 ):name(name),age(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 ;
}
```

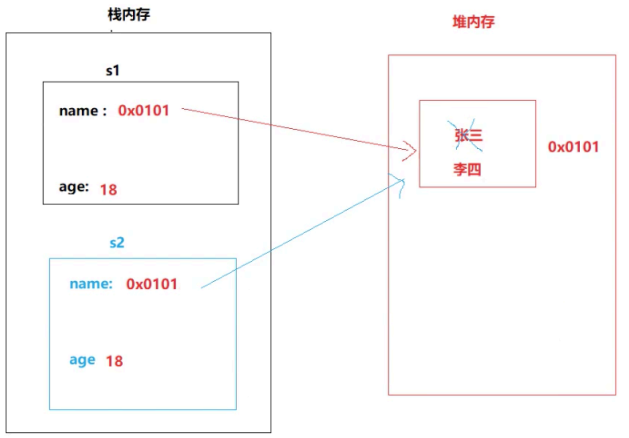
## 3. 浅拷贝引发的问题

> 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。

```
class stu{
public:
    string *name;
    int age;
    stu(string name , int age ) : name(new string (name)) , age (age){
        cout << "有参构造" <<endl;
    }
    stu ( const stu & s){
        cout <<"拷贝构造" << endl;
        name = s.name;
        age = s.age;
    }
};

int main() {
    stu s1( name: "张三" , age: 18);
    stu s2 = s1;
    *s1.name = "李四";
    return 0;
}
```

浅拷贝：只会拷贝数据，一股脑的拷贝数据，源对象是什么，我就拷贝什么。

引发问题：如果源对象里面有指针成员，浅拷贝拷贝的是指针的值：这就会让源对象和现在的新对象指向同一个地方。

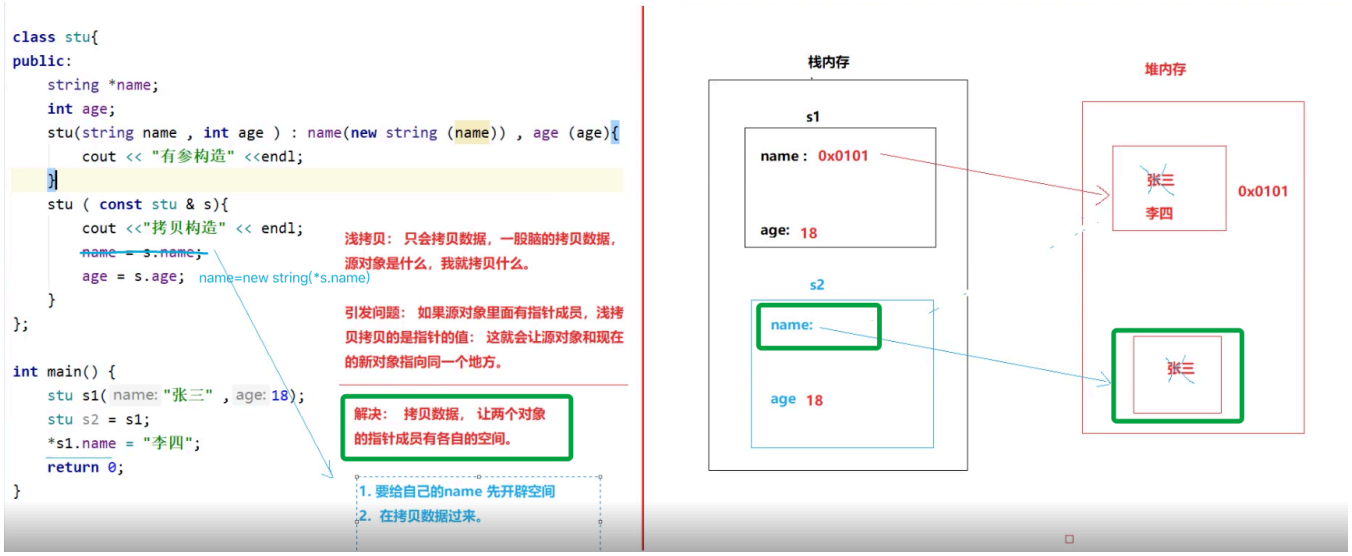


```cpp
#include <iostream>
#include <string>

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 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。



```cpp
#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 ;
}
```

## 5. 触发拷贝的场景

> 如果是生成临时性对象或者是使用原有对象来初始化现在的对象，那么都会执行拷贝构造。一般调用拷贝构造函数的场景有以下几个：
>
> - 对象的创建依赖于其他对象。
> - 函数参数（传递对象）
> - 函数返回值 （返回对象）
> - Student.cpp

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

using namespace std;

class Student {
public:
    int age ;
    string name;

public :
    //构造函数
    Student(){
        age = 19 ;
        name = "张三";
        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;
    }
};
```

## 1. 对象创建依赖于其他对象

```
#include <iostream>
#include <string>
#include "Student.cpp"

using namespace std;



int main(){

    Student stu1("张三",18); //执行构造函数
        Student stu2 = stu1; //执行拷贝构造函数

    return 0 ;
}
```

## 2. 函数参数

> 编译器不会优化这个方向，因为一旦优化掉了，那么就不会执行拷贝的工作，那就代表传递进来的对象实际上就是外部的原对象，有可能在函数内部对对象进行了修改，会导致外部对象跟着修改。所以默认情况下，只要是函数参数传递，都会发生拷贝。 如果不想发生拷贝，请使用 引用 或者 指针。

```
#include <iostream>
#include <string>
#include "Student.cpp"

using namespace std;


void printStun(Student s){

    cout << s.name << " : "<< s.age << endl;
}

int main(){
    Student stu1("张三",18); //执行构造函数
    printStudent(stu1); //执行拷贝构造函数

    return 0;
}
```

## 3. 函数返回值

> 为了避免过多的临时性对象创建，消耗内存，编译器内部做了优化，函数的返回值不会再产生临时对象，直接把生成的对象赋值给外面接收的变量

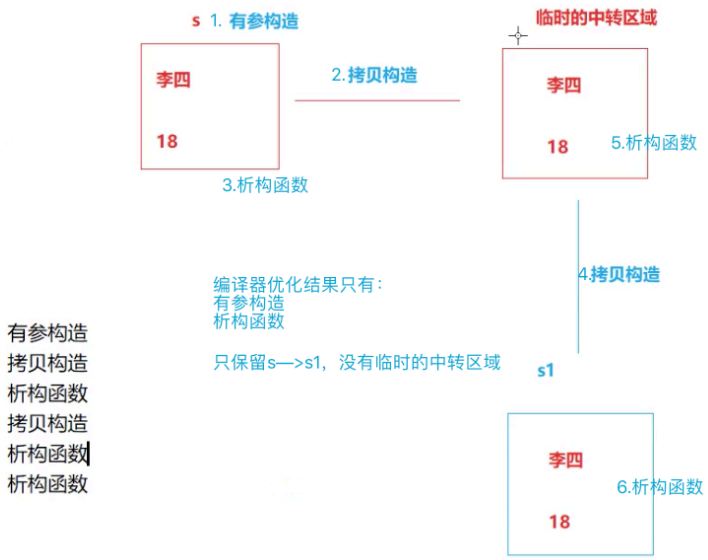
```cpp
#include <iostream>
#include <string>
#include "Student.cpp"

using namespace std;


//构造函数 由于函数执行完毕会有个临时对象来接收值，所以这里还会执行拷贝构造
Student createStu(){
    return Student("张三",18);
}

int main(){
    Student stu = createStu(); //这里还会执行构造函数
    return 0;
}
```

## 4. 编译器自动优化

> 编译器有时候为了避免拷贝生成临时对象而消耗内存空间，所以默认会有优化、避免发生过多的拷贝动作所以打印的日志可能不是我们所期望的，这时候，如果手动编译的话，可以添加参数，

```cpp
#如果手动编译 可以添加以下参数  -fno-elide-constructors
g++ -std=c++11  main.cpp -fno-elide-constructors

# 如果使用cmake编译，可以添加配置    clion写代码
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-elide-constructors")
```

```
stu createStu(){
    stu s( name: "李四", age: 18);
    return s;
}

int main() {
    //3. 函数的返回值也会发生拷贝
    stu s1 = createStu();

    return 0;
}
```




# 练习

定义一个学生类，包含 属性： 姓名、年龄， 行为：读书、跑步。 其中姓名字段是 指针类型

1. 使用stu. h 声明学生类
2. 使用stu.cpp实现读书和跑步的行为。
3. 使用初始化列表的方式完成对 姓名和 年龄的赋值工作。
4. 张三和李四年龄一样，唯独姓名不同，请先创建张三，然后使用拷贝构造的方式创建李四
5. 由于是拷贝创建李四，所以需要手动修改李四的姓名为 '李四' , 并且注意解决浅拷贝的问题。

# 6. 再说函数返回值

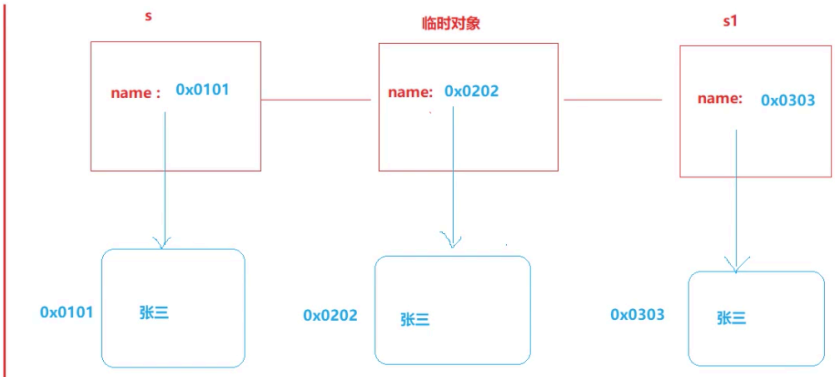
c++ 严禁函数返回内部的存放于栈内存的局部变量引用或者指针，因为函数一旦执行完毕，那么内部的所有空间都将会被释放，这时如果在外面操作返回值，将出现错误。所以有时候临时拷贝的工作就变得不可避免。

```
int* getNum(){
    int a = 9;
    return &a ;
}

int* getNum2(){
    int a = new int(9);
    return &a ;
}

int main(){
    int *p = getNum(); //错误！ 一旦getNum()执行完毕，内部栈空间都将释放掉。

    int *p = getNum2(); // 正确，因为函数返回的指针指向的地址位于堆内存中。
    return 0 ;
}
```

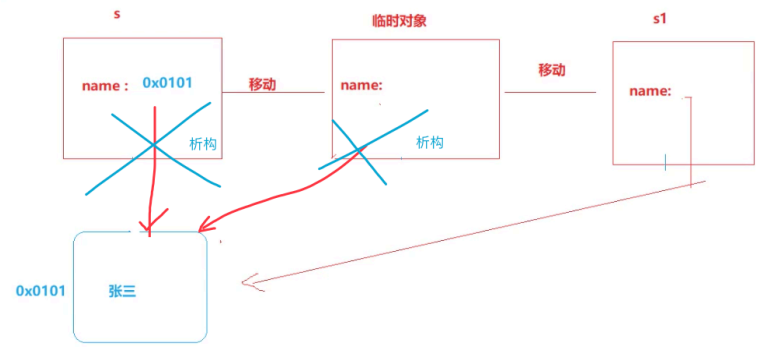
## 7. 移动构造函数#

> 有时候我们需要新创建的对象，拥有旧对象一模一样的数据，当然这可以使用拷贝操作来完成。但是假设旧对象不再使用，那么拷贝操作就显得有点弱，因为旧对象仍然占据着空间。C++11中推出了 移动构造操作，也就是完全的把旧对象的数据 "移动" 到新对象里面去，并且把旧对象被清空了。
>
> 注意： 移动构造或者是拷贝构造，针对的都是数据的拷贝或者移动，对象的该创建还是创建，他们两针对的仅仅是数据是以何种方式得来而已。

```
class stu{
public:
    string * name = nullptr ;
    stu(string name ) : name(new string(name)) {
        cout << "有参构造" <<endl;
    }
    stu ( const stu & s){
       cout <<"拷贝构造" << endl;

          name = new string (*s.name); //深拷贝

    }
};

stu getStu(){
    stu s ( name: "张三");
    return s;
}

int main() {
    stu  s1 = getStu ();
    cout << "s1.name=" << *s1.name <<endl;
    return 0;
}
```




1. 拷贝构造有时候也有会一些弊端，数据拷贝太多，浪费内存了。
2. 尤其是在面对一些即将消亡的对象的时候，这些对象销毁了，但是它的数据还是要拷贝出来，为我所用。
3. 能不能让我们的新对象去接管即将消亡对象的数据就可以了。没有必要去开辟太多的空间来存重复一样的数据。




虽然语义叫做移动：但是整个全局并没有去移动任何的东西，而是窃取了数据的所有权而已。

当某一个对象即将要消亡的时候| 出现临时 对象的时候，把临时对象赋值给一个新的对象，那么此时就应该会出现移动构造函数而不是拷贝构造函数。

- **移动语义**

> 要想完成移动构造，需要配合右值引用来实现。因为把某一份数据，移动到另一个地方去，实际上操作的是右值，而右值引用恰好指向的是右值。这常常出现在对象的搬运上，假设现在要做两份数据的交换，最初的设想是进行值的拷贝、复制，但是这样会产生一份临时拷贝值，不如直接移动过去划算。但在C++里面的 移动 实际上并不是真的在移动数据，只是窃取了原来数据的控制权而已。

## 1. 没有移动构造

> 使用getStu函数生成一个学生对象，调用返回后，使用stu来接收。 在没有移动构造函数的情况下，临时对象的产生会调用拷贝构造。此举会造成资源的浪费，并且效率也低。编译器为了避免过多的临时对象的创建工作，内部已经做了优化。

- 学生类

```
class Student{
public :
    int *age;
    Student():age(new int(18)){
        cout << "执行构造函数~! ~"<<endl;
    }

    //深拷贝
    Student(const Student &s):age(new int(*s.age)){
        cout << "执行拷贝构造函数~! ~"<<endl;
    }//移动赋值调用拷贝构造

    ~Student(){
        cout <<"执行析构函数~! " << endl;
        delete  age;
    }
};

Student getStu(){
    Student s ;
    cout <<"getTemp =" << __func__ << " : " << hex << s.age << endl;
    return s;
}
```

- main函数

```
int main(){

    Student stu = getStu();

    return 0 ;
}
```

## 2. 使用移动构造

> 使用移动构造，会使得在返回对象时，不会调用拷贝构造函数，这也就避免产生了对象的拷贝工作。

- 学生类

```
class Student{
public :
    int *age;
    Student():age(new int(18)){
        cout << "执行构造函数~! ~"<<endl;
    }

    //深拷贝
    Student(const Student &s):age(new int(*s.age)){
        cout << "执行拷贝构造函数~! ~"<<endl;
    }

    //移动构造
    Student( Student &&s):age(s.age){
         cout << "执行移动!!!构造函数~! ~"<<endl;
        //移动之后，一般即可让原有对象的指针变成空指针
        s.age = nullptr;

    }//移动赋值调用移动构造

    ~Student(){
        cout <<"执行析构函数~! " << endl;
        delete  age;
    }
};

Student getStu(){
    Student s ;
    cout <<"getTemp =" << __func__ << " : " << hex << s.age << endl;
    return s;
}
```

- main

```
int main(){

    Student stu = getStu();
    return 0 ;
}
```

## 3. std::move函数

> move 函数名字很具有迷惑性，但是它并不能移动任何东西。它唯一的作用就是把一个左值转化成一个对应的右值引用类型，继而可以通过右值引用使用该值。并且在使用move函数可以避免不必要的拷贝工作， move 是将对象的状态或者所有权从一个对象转化到另一个对象，只是转换状态或者控制权，没有内存的搬迁和拷贝 . 使用move函数即表示该对象已经不再使用，要被即将销毁了。

```
int main(){

    int a = 3;
    int &b = a ; //左值引用指向左值
    int &&c = 3; //右值引用指向右值
    int &&d = move(b); //右值引用，指向右值， move函数强制转化 左值引用b 成右值引用

    return 0 ;
}
```

- 使用move出发移动构造函数

> 默认情况下，如果直接赋值，那么执行的是拷贝构造函数，并且有时候为了避免频繁的执行拷贝工作，可以直接使用 move 函数转化左值成右值引用，进而变成直接操作数据。

```
int main(){
    Student stu1 ;
    Student stu2 = stu1; // 执行拷贝构造函数

    Student stu3 = move(stu1); // 执行移动构造函数
    Student stu3(movestu1) ; // 和上一行代码同效果
    return 0 ;
}
```

# 三、其他细节

## 1. this 指针

### 1. 引入

> 在早前的编码中，我们都会刻意的避免的函数的参数名称和成员变量名称一致，这是为了让我们更方便的赋值。其实他们的名称也可以一样，只是要多费点功夫而已。如下：

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



class Student{

    int age ;
    string name;


    Student(int age_val , string name_val){
        //默认情况下，前面的两个变量有编译器加上的this指针
        age = age_val ;
        name = name_val ;
    }


    Student(int age , string name){
        //默认情况下，下面两行代码并没有完成赋值工作。
        age = age ;
        name = name ;

        //============================

        //前面加上this -> 即可完成操作。
        this-> age = age ;
        this-> name = name ;
    }

};
```

## 2. 含义

> this 只能在成员函数中使用。全局函数，静态函数都不能使用 this 。全局函数不属于任何一个类，静态函数也不依赖任何一个类，所以他们都不会有 this指针 。 在外部看来，每一个对象都拥有自己的成员函数，一般情况下不写this，而是让系统进行默认配置。**this指针永远指向当前对象**， 所以成员函数中的，通过this即可知道操作的是哪个对象的数据。

> this指针的作用有两个
>
> - 处理同名变量问题
> - 返回对象本身

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

class Student{

    int age ;
    string name;

    Student(int age , string name){
        //默认情况下，下面两行代码并没有完成赋值工作。
        age = age ;
        name = name ;

        //============================

        //前面加上this -> 即可完成操作。
        this-> age = age ;
        this-> name = name ;
    }

    //获取对象本身。注意：  这里的返回值不要写成Student，否则返回的是一个临时的拷贝对象
    Student& getStu(){
        return *this;  //this表示指针，   在指针前面加上*  表示解引用。根据指针获取到指向的对象
    }

};

int main(){

    Student s1 (18 , "张三");
        Student &s2 = s1.getStu(); //使用引用来接收，表示指向同一个引用，

    return 0 ;
}
```

## 2. 常函数

> 在编写代码的时候，如果确定了某个成员函数不会修改成员变量，那么可以把该函数声明为常函数。常函数其实也是函数，它浓缩了函数和 常 这个概念在里面，函数体内不允许修改成员变量，除非该变量使用 mutable 修饰.

> 常函数的const修饰的是this指针，在常函数中的this形同： const 类名 ，表示指向常量的指针。 所以也就解释了为什么在常函数中无法修改成员变量。

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

using namespace std;

class  Student{

public:
   string name  = "无名氏";
   int age  = 18;

public :

     //在此添加 const 即可变成常函数,
    void printStudent() const{

       // age = 10 ;  错误，不允许修改
       cout << name  << "  "  << age << endl;
    }

};
```

## 3. 常对象

> 常对象其实也就是一个变量，和以前的常量没有相差多少。不同的是如今对象里面可以能包含很多的变量，以前的常量指的只是一个变量而已。 若类中有变量使用 mutable 修饰，那么该变量在常对象状态下，可以被修改

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

using namespace std;

class  Student{

public:
   string name  = "无名氏";
   int age  = 18;

public :

     //在此添加 const 即可变成常函数,
    void printStudent() const{
       cout << name  << "  "  << age << endl;
    }

    void run(){

        cout<< age << "的" <<name <<"在跑步" <<endl;
    }

};

int main(){

    const Student s1 ("zhangsan " ,18);

    cout << s1.name << endl; //访问常对象中的成员 , OK
    s1.name = "李四" ; // 试图修改常对象中的成员 ,  error

    return 0 ;
}
```

- 注意：
  - a. 无法修改常对象中的成员(变量)
  - b. 除非某个成员使用`mutable`修饰
  - c. 常函数函不能访问普通的成员函数，但是可以访问常函数

d. 普通的对象可以访问成原函数，也可以访问常函数

# 4. 静态成员

> 有时候需要不管创建多少次对象，某些变量的值都是一样的，那么此时可以把该变量变成静态修饰(static).

> 静态成员被类的所有对象所共享，成员不能在类中定义的时候初始化，但是可以在类的外部通过使用范围解析运算符 **::** 来重新声明静态变量从而对它进行初始化

## 1. 静态成员变量

> 假设现在要记录来参加当年毕业的学生信息，无论他们的身份迥异，都有一个共有的属性，就是同一个学校。如果不对该属性进行修饰，那么100个学生对象就要开辟100个位置来存储同一个school数据，但是其实可以使用static修饰，只需要一份空间即可。

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

using namespace std;

class Student{

    public:
            int age ;
            string name;
            static  string school;  //静态成员

};

Student::school = "北京大学";

int main(){
        Student s;
        s.name = "张三";
        s.age = 18 ;

        cout <<s.school << " " << s.name  <<" " << s.age << endl;

        return 0 ;
}
```

## 2. 静态成员函数

> 如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。静态成员函数即使在类对象不存在的情况下也能被调用，**静态函数**只要使用类名加范围解析运算符 **::** 就可以访问。
>
> 静态成员函数只能访问静态成员数据、其他静态成员函数和类外部的其他函数。
>
> 静态成员函数有一个类范围，他们不能访问类的 this 指针。您可以使用静态成员函数来判断类的某些对象是否已被创建。

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

using namespace std;


class Student{

public:
    int age ;
    string name;
    static  string school; //静态成员


public :
    static string getSchool(){
        //不能访问 name 和 age
        return school;
    }
};

Student::school = "北京大学";

int main(){

    Student s ;
    cout << s.getSchool() << endl; //可以访问

    //也可以使用类名的方式访问
    cout << Student::school <<endl;

    return 0 ;
}
```

- 注意：
    - 静态成员属于类，不属于对象
    - 静态成员变量必须在类中声明，在类的外部初始化
    - 静态成员变量的声明周期是整个程序，作用域在类的内部
    - 静态成员函数只能访问静态成员变量

    - 静态成员可以使用类来访问，也可以使用对象来访问。

# 5. 结构体和类

> 结构体是一个由程序员定义的数据类型，可以容纳许多不同的数据值。在过去，面向对象编程的应用尚未普及之前，程序员通常使用这些从逻辑上连接在一起的数据组合到一个单元中。一旦结构体类型被声明并且其数据成员被标识，即可创建该类型的多个变量，就像可以为同一个类创建多个对象一样。 结构体是由C语言发明出来的，它和类其实没有差多少，只是它的所有成员默认都是public公开。

> 声明结构体的方式和声明类的方式大致相同，其区别如下：
>
> - 使用关键字 struct 而不是关键字 class。
> - 尽管结构体可以包含成员函数，但它们很少这样做。所以，通常情况下结构体声明只会声明成员变量。
> - 结构体声明通常不包括 public 或 private 的访问修饰符， 因为它的所有成员默认都是public
> - 类成员默认情况是私有的，而结构体的成员则默认为 public。程序员通常希望它们保持公开，只需使用默认值即可。

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

struct Student{

    string name;
    int age;

    void run(){

    }

}

int main(){

    Student s1 ;
    s1.name = "zhangsan";
    s1.age = 18 ;

    return 0 ;
}
```

# 6. 友元

- 概念理解

> 私有成员只能在类的成员函数内部访问，如果想在别处访问对象的私有成员，只能通过类提供的接口（成员函数）间接地进行。这固然能够带来数据隐藏的好处，利于将来程序的扩充，但也会增加程序书写的麻烦。
>
> 在类的外部无法访问类的私有成员，但是有时候需要它，那么可以借助**友元函数** 来实现。 友元函数 是一种特权函数，C++允许它访问私有成员。程序员可以把一个全局函数、成员函数、甚至整个类声明为 友元
>
> 友元可以看成是现实生活中的 好闺蜜 或者是 好基友

# 1. 友元函数

> 友元函数可以直接访问类中的私有成员

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

using namespace std;

class Car{
private:
    string color {"红色"};
    friend void showColor(Car c);
};

//实现友元函数，函数内部可以直接访问私有成员
void showColor(Car c) {
    cout << c.color << endl;
}

int main(){

    Car c ;

    showColor(c);

    return 0 ;
}
```

# 2. 友元类

> 一个类 A 可以将另一个类 B 声明为自己的友元，类 B 的所有成员函数就都可以访问类 A 对象的私有成员

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

using namespace std;

class Car{
private:
    string color {"红色"};
    friend class SSSS;  //声明 4S 为友元类

public:

    string getColor(){
        return color;
    }
};

class SSSS{
public:
    void modifyCar( Car &myCar){  //改装汽车

        myCar.color = "黑色";
    }
};


int main(){

   SSSS s;
   Car c;
   s.modifyCar(c);
   cout << c.getColor() << endl;

    return 0 ;
}
```