day04

一、函数指针/typedef

1.函数指针

指针也可以指向函数

```
void run(){
    cout << "跑步" << endl;
}
void sayHi(int a,int b){
    cout << "Hi" << endl;
}
int main(){
    void run();
    // 函数指针的类型由函数的返回值和函数的参数共同决定
    // 这是一个函数指针，这个函数指针没有返回值，没有参数
    // 可以指向别的函数
    void (*p)() = run;
    void (*ps)(int,int) = sayHi;
    // 调用
    p();
    ps(10,20);
```

```
    return 0;
}
// 运行结果:
跑步
Hi
```

() 小括号是调用的时候用的、叫做调用运算符

2.函数指针的应用场景

作用：用在参数上

```
void B(){

}
void A(void (*fb)()){
    fb();
}
```

3.typedef类型定义

简化类型定义

```
// typedef 原来类型名称 现在简化的名称;
typedef int* intPointer;
typedef char* charPointer;
typedef vector<int> vt_int;
// 简化函数指针，不需要在后面补简化的名称，就是x
typedef int (*x)(int);
```

二、引用 & 左值和右值

1.引用

1. 引用就是某一个变量的别名
2. 引用不是一个全新的变量|对象，所以不会有新的空间开辟出来
3. 定义引用，需要用 & 这个符号
4. 别名只能指向一个

```
int age = 10;
int & a = age;
int age2 = 18;
a = age2;
// 相当于将age2的值赋给了age
```

5. 引用声明出来必须初始化
6. 不能创建数组的引用

2.左值和右值

1. 左值可以放在等号的左侧或者由侧、右值只能放在右侧
2. 如果能取地址，你们就是左值，如果取不了地址，那么就是右值
3. 函数的返回值都是右值
4. 左值引用只能接收左值、如果左值引用加上`const`那么就可以接收右值

```
int age = 18;
int age1 = age;
// int age2 = 18; // 报错
const int age2 = 18;
void add(const int& a,const int& b){
    return a + b;
}
int main(){
    int a = 10;
    int b = 20;
    add(a,b);
}
```

5. 右值引用、使用`&&`

```
int a = 10;
int && m = 10;
//move函数将左值转换成右值
int && mm = move(a);
```

三、类和对象

1.类的定义

```
class 类名{
    // 属性

    // 行为
};
```

```
class stu{
    // 属性
    int age = 18;
    // 行为
    void read(){
        cout << "读书" << endl;
    }
};
```

2.类的对象

创建对象

```
// 栈内创建对象
stu s;
// 堆区创建对象
stu *s1 = new stu;
```

对象的访问 、访问用 `对象.成员`

```
// 需要访问对象内容就要加上public:
class stu{
    // 属性
public:
    int age = 10;
    // 行为
    void read(){
        cout << "读书" << endl;
    }
};
int main() {
    // 栈内创建对象
    stu s;
    s.age;
    s.read();
    // 堆区创建对象
    stu *s1 = new stu;
    (*s1).age;
    (*s1).read();
    // 第二种方式
    s1->age;
    s1->read();
    return 0;
}
```

访问修饰符 ： 设置访问权限

- 默认情况下，类当中的成员，在外部是无法访问的，是私有的，用 `private:` 修饰，在类里面是可以使用的
- 想要使用就要用 `public:` ，将成员公开。
- `protected:` 使用这个则只能给自己的儿子和自己用

实现类当中的成员函数

- 在类的外面实现成员函数
- 在函数名的前面加上 `类名::`

```cpp
class stu{
    // 属性
public:
    int age = 10;
    // 行为
    void read();
};
void stu::read(){
    cout << "读书" << endl;
}
```

3.类和构造函数

特殊的成员函数

> 当定义一个类后，它的对象在未来操作中，总会不可避免的碰到以下行为：创建、拷贝、赋值、移动、销毁。这些操作实际上是六种成员函数来控制的：构造函数、析构函数、拷贝构造函数、拷贝赋值函数、移动赋值函数。

- 构造函数：只要创建对象，就会直接调用构造函数

默认情况下，编译器会给每一个类创建一个无参构造函数

构造函数其实也是一个函数，只是比较特殊，而且没有返回值，函数名就是类名

没有参数叫无参构造，有参数叫有参构造

```cpp
class stu{
public:
    stu(){
        cout << "无参构造" << endl;
    }
    // 函数的重载
    stu(int age){
        cout << "有参构造" << endl;
    }
};
stu s;
stu s1(10);
// 运行结果：
无参构造
有参构造
```

> 构造函数的作用：
>
> 只有就是为了完成数据的初始化

4.构造初始化列表

`:name{name},age{age}=:name(name),age(age)` 初始化列表

```
class stu{
public:
    string name;
    int age;
//    stu(string name,int age){
//        stu::name = name;
//        stu::age = age;
//        /*
//        this->name = name;
//        this->age = age;
//        */
//    }
//stu(string name,int age):name(name),age(age)一样
    stu(string name,int age):name{name},age{age}{

    }
};
stu s1("张三",18);
cout << s1.name << "是" << s1.age << endl;
// 运行结果:
张三是18
```

5.无参构造的问题—隐式转换的问题—委托构造的问题

1. 使用栈创建对象的时候，调用无参构造有一个小细节，不加 `()`
2. 当一个类中存在有参构造函数，且只有一个参数时，要特别小心，C++在创造对象的时候，存在隐式转换，要避免这个问题，需要在有参构造函数前加上 `explicit` ：显示的

```
class stu{
public:
    explicit stu(int age){
        cout << "有参构造" << endl;
    }
};
stu s1(10);
// 运行结果:
有参构造
class stu{
public:
    stu(int age){
        cout << "有参构造" << endl;
    }
};
stu s1 = 10;// 王者代码
// 运行结果:
有参构造
```

6.委托构造函数

```
class stu{
public:
    string name;
    int age;
    stu():stu("无名氏",18){
        cout << "无参构造" << endl;
    }
    stu(string name):stu(name,18){
        cout << "有一个参数" << endl;
    }
    stu(string name,int age):name(name),age(age){
        cout << "有两个参数" << endl;
    }
};
stu s;
stu s1("小飞");
stu s2("阿花",18);
// 运行结果:
有两个参数
无参构造
有两个参数
有一个参数
有两个参数
```

7.析构函数

特殊的成员函数，与构造函数正好相反，它会在删除创建的对象时执行

析构函数的名称和类名完全相同，只需要在前面加~作为前缀，不会返回任何值，也没有参数，不能被重载，一般用于释放资源

```
class stu{
public:
    stu(){
        cout << "无参构造" << endl;
    }
    // 析构函数
    ~stu(){
        cout << "析构函数" << endl;
    }
};
stu* ss = new stu;
// 有delete才会走析构函数
delete ss;
// 运行结果:
无参构造
析构函数
```

8.拷贝构造

```
stu s1("张三",18);
// 走的是拷贝构造
stu s2 = s1;
```

基本实现

```
// 拷贝构造,参数必须要用引用,引用的对象
stu(stu & s){
    cout << "拷贝构造" << endl;
}
```

```
class stu{
public:
    string name;
    int age;
    stu(){
        cout << "无参构造" << endl;
    }
    stu(string name,int age):name(name),age(age){
        cout << "无参构造" << endl;
    }
    // 拷贝构造
    stu(stu& s){
        cout << "拷贝构造" << endl;
        // 给s2赋值
        name = s.name;
        age = s.age;
    }
    ~stu(){
        cout << "析构函数" << endl;
    }
};
int main() {

    stu s1("张三",18);
    // 走的是拷贝构造
    stu s2 = s1;
    cout << s1.name << "=s1=" << s1.age << endl;
    cout << s2.name << "=s2=" << s2.age << endl;
    return 0;
}
// 运行结果:
无参构造
拷贝构造
张三=s1=18
张三=s2=18
析构函数
析构函数
```

为什么参数是引用

> `stu s2 = s1;`中的`s1`被当做参数来传递，要是没有`&`的话，那么就会进入无休止的拷贝中。
>
> 并且在拷贝构造函数中，需要加入`const`关键字，修饰，防止原数据被修改

```
// 拷贝构造 ,默认提供的拷贝构造就是这个样子
stu(const stu& s){
    cout << "拷贝构造" << endl;
    // 给s2赋值
    name = s.name;
    age = s.age;
}
```

浅拷贝 :默认的拷贝

> 浅拷贝存在拷贝值时，当原对象存在指针类型的数据时，拷贝过来的成员属性，实则也是所对应的成员属性的地址。怎么解决这个问题，就要用到深拷贝

```
class stu{
public:
    string* name;
    stu(string* name):name(name){
        cout << "有参构造" << endl;
    }
    // 拷贝构造
    stu(stu& s){
        cout << "拷贝构造" << endl;
        // 给s2赋值
        name = s.name;
    }
};
int main() {
    string* n = new string("张三");
    stu s1(n);
    stu s2 = s1;
    *s2.name = "李四";
    cout << *s1.name << "=" << *s2.name << endl;
    return 0;
}
// 运行结果:
有参构造
拷贝构造
李四=李四
```

深拷贝

```
// 深拷贝
stu(stu& s){
    cout << "深拷贝" << endl;
    // 给s2赋值
    // name = new string(*s.name);
    name = new string;
    *name = *s.name;
}
// 运行结果:
有参构造
深拷贝
张三!=李四
```

拷贝构造的触发场景

- `stu s2 = s1;`
- 将对象作为函数参数传进函数中时，也会存在拷贝，可以用引用来解决拷贝的问题
- 函数的返回值也会发生拷贝

四、其他

1. `this`指针、常函数、常对象

this指针

- 类成员中的所有成员函数都含有`this`指针，会指向当前对象
- 在初始化赋值的时候，解决的参数与属性同名的问题
- 在链式调用的时候比较多，充当函数的返回值

```
class stu{
public:
    string name;
    stu(string name){
        cout << "有参构造" << endl;
        this->name = name;
    }
};
int main() {
    stu s("张三");
    cout << s.name << endl;
    return 0;
}
```

- 链式调用

```
class stu{
public:
    string name;
    stu(string name){
        cout << "有参构造" << endl;
        this->name = name;
    }
```

```
    stu run(){
        cout << "跑步" << endl;
        return *this;
    }
    stu sleep(){
        cout << "睡觉" << endl;
        return *this;
    }
};
int main() {
    s.run().sleep().run().sleep();
    return 0;
}
// 运行结果:
有参构造
跑步
睡觉
跑步
睡觉
```

常函数

- const可以修饰变量，指针，函数，对象
- 常函数修饰的是this指针，将this变成const stu *
- 作用，防止修改数据

```
class stu{
public:
    string name = "张三";
    void sleep() const {
//        name = "李四";
        //不可在修改值,但是可以调用，如果非得改，在变量前面加mutable
        cout << "睡觉" << endl;
    }
};
```

常对象

```
const stu s("李四");
// 那么对应的成员属性不能进行修改，想修改需要加mutable
s.sleep(); // 想要调用常方法，就必须写成常函数，普通函数不能调用
// 常对象只能调用常函数，不能调用普通函数
// 普通对象可以调用普通函数，也能调用常函数
// 所以常的东西只能用常的东西调用
```

2.静态

静态成员变量

```
class stu(){
public:
    string name;
    int age;
```

```
    static string school;
    // static创建静态成员变量，那么可以用对象名和类名调用，静态成员变量一定要在类的外面初始化
}
// 初始化
string stu::school = "啊哈学院";
cout << stu::school << endl;
stu s;
cout << s.school << endl;
// 运行结果:
啊哈学院
啊哈学院
```

静态成员函数

普通成员函数必须有对象才能调用。

```
class stu(){
public:
    static string name;
    static void run(){
        // 静态成员函数禁止调用非静态成员，也不存在this指针
    }
}
```