C++核心编程

本阶段主要针对C++面向对象编程技术做详细讲解, 探讨C++中的核心和精髓。

1 内存分区模型

C++程序在执行时, 将内存大方向划分为**4个区域**

- 代码区: 存放函数体的二进制代码, 由操作系统进行管理的
- 全局区: 存放全局变量、静态变量、常量(全局常量、字符常量)
- 栈区: 由编译器自动分配释放,存放函数的参数值,局部变量,局部常量等
- 堆区: 由程序员分配(new)和释放(delete),若程序员不释放,程序结束时由操作系统回收

内存四区意义:

不同区域存放的数据, 赋予不同的生命周期, 给我们更大的灵活编程

1.1 程序运行前

在程序编译后, 生成了exe可执行程序, **未执行该程序前** 分为两个区域

代码区:

存放 CPU 执行的机器指令

代码区是 **共享** 的, 共享的目的是对于频繁被执行的程序, 只需要在内存中有一份代码即可

代码区是 **只读** 的, 使其只读的原因是防止程序意外地修改了它的指令

全局区:

全局变量和静态变量存放在此.

全局区还包含了常量, 字符串常量和全局常量也存放在此.

该区域的数据在程序结束后由操作系统释放.

示例:

```
// 全局变量—全局区
int global_a = 10;
// 全局常量—全局区
const int c_global_a = 10;

int main(){

    // 静态变量—全局区
    static int s_a = 10;
    // 局部变量—栈区
    int a = 10;
    // 局部常量—栈区

    const int c_a = 10;
```

```
    cout<<"全局变量global_a的地址为:"<<(int)&global_a<<endl;
    cout<<"全局常量c_global_a的地址为:"<<(int)&c_global_a<<endl;
    cout<<"静态变量s_a的地址为:"<<(int)&s_a<<endl;
    cout<<"局部变量a的地址为:"<<(int)&a<<endl;
    cout<<"局部常量c_a的地址为:"<<(int)&c_a<<endl;
    // 字符常量—全局区
    cout<<"字符常量hello world的地址为:"<<(int)&"hello world"<<endl;

    system("pause");
    return 0;
}

// 运行结果如下(存放在全局区的地址相邻，存放在栈区的地址相邻):
// 全局变量global_a的地址为:4214788
// 全局常量c_global_a的地址为:4218952
// 静态变量s_a的地址为:4214792
// 局部变量a的地址为:6422284
// 局部常量c_a的地址为:6422280
// 局部常量c_b的地址为:6422276
// 字符常量hello world的地址为:4219121
```

总结:

- C++中在程序运行前分为全局区和代码区
- 代码区特点是共享和只读
- 全局区中存放全局变量、静态变量、常量
- 常量包含 const修饰的全局常量 和 字符串常量

1.2 程序运行后

栈区:

由编译器自动分配释放, 存放函数的参数值,局部变量,局部常量等

注意事项: 不要返回局部变量的地址, 栈区开辟的数据由编译器自动释放

示例:

```
// 栈区的数据在函数执行完后自动释放
int *func(){
    int a = 10; // 局部变量—栈区
    return &a; // 返回局部变量的地址
}

int main(){

    // 定义指针p接受函数返回的地址
    int *p1 = func();
    cout<<*p1<<endl; // 输出p1指向的内容，报错或者输出随机数


    system("pause");
```

```
    return 0;
}
// 报错: warning: address of local variable 'a' returned [-Wreturn-local-addr]
```

堆区:

由程序员分配释放, 若程序员不释放, 程序结束时由操作系统回收

在C++中主要利用 new 在堆区开辟内存

堆区开辟的数据, 由程序员手动开辟, 手动释放, 释放利用操作符 delete

语法: `new 数据类型`

利用new创建的数据, 会返回该数据对应的类型的指针

示例:

```
// 栈区的数据在函数执行完后自动释放
int *func2(){
    int *a = new int(10); // 在堆区分配一个内存, 大小为int的大小
    return a; // 返回堆区的地址
}

int main(){

    int *p2 = func2();
    cout<<*p2<<endl; // 输出p2指向的内容: 10

    // 使用delete释放堆区数据
    delete p2;

    cout<<*p2<<endl; // 报错, 释放的空间不可访问

    // 堆区开辟数组
    int *arr = new int[10];
    // 释放数组
    delete[] arr;

    system("pause");

    return 0;
}
```

总结:

堆区数据由程序员管理开辟和释放

堆区数据利用new关键字进行开辟内存

2 引用

2.1 引用的基本使用

作用: 给变量起别名

语法: `数据类型 &别名 = 原名`

示例:

```
int main(){

    int a = 10;
    int &b = a;

    cout<<"a = "<<a<<"\tb = "<<b<<endl;

    b = 100;

    cout<<"a = "<<a<<"\tb = "<<b<<endl;

    system("pause");

    return 0;
}
// a = 10  b = 10
// a = 100 b = 100
```

2.2 引用注意事项

- 引用必须初始化
- 引用在初始化后, 不可以更改

示例:

```
int main(){

    int a = 10;
    int b = 20;
    // int &c; // 错误，引用必须初始化
    int &c = a; // 一旦初始化后，就不可以更改
    c = b; // 这是赋值操作，不是更改引用

    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    cout<<"c = "<<c<<endl;

    system("pause");

    return 0;
}
```

2.3 引用做函数参数

作用: 函数传参时, 可以利用引用的技术让形参修饰实参

优点: 可以简化指针修改实参

示例:

```
// 引用传递
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){

    int x = 1, y = 2;
    swap(x, y);
    cout<<"x = "<<x<<"\ty = "<<y<<endl;
    // x = 2   y = 1

    system("pause");

    return 0;
}
```

> 总结: 通过引用参数产生的效果同按地址传递是一样的。引用的语法更清楚简单

2.4 引用做函数返回值

作用: 引用是可以作为函数的返回值存在的

注意:** 不要返回局部变量引用**

用法: 函数调用作为左值

示例:

```
// 返回静态变量引用
int &ref(){
    static int a = 10;
    return a;  // 返回静态变量
}

int main(){

    int &reff = ref();
    cout<<"reff = "<<reff<<endl;  // reff = 10
    ref() = 100;
    cout<<"reff = "<<reff<<endl;  // reff = 100

    system("pause");


    return 0;
```

```
}
```

2.5 引用的本质

本质:** 引用的本质在c++内部实现是一个指针常量.**

示例:

```
int main(){
    int a = 10;

    // 引用的本质: 指针常量,编译器发现引用将自动转换
    int &ref = a;  // 自动转换为: int * const ref = &a;
    ref = 362425;  // 自动转换为: *ref = 362425;

    return 0;
}
```

> 结论: C++推荐用引用技术, 因为语法方便, 引用本质是指针常量, 但是所有的指针操作编译器都帮我们做了

2.6 常量引用

作用: 常量引用主要用来修饰形参, 防止误操作

在函数形参列表中, 可以加const修饰形参, 防止形参改变实参

示例:

```
// 引用使用的场景, 通常用来修饰形参
void showValue(const int &a, int &b){
    a += 10;  // 报错, a 不能修改
    b += 10;
}

int main(){

    // int &ref2 = 10; // 报错: 引用本身需要一个合法的内存空间
    // 加const后,编译器自动优化代码为: int temp = 10; int &ref2 = 10;
    const int & ref2  = 10;
    ref2 = 100; // 报错: 加const后不可修改变量

    // 函数中利用常量引用防止误操作修改实参
    int x1 =100, x2 = 100;
    showValue(x1, x2);

    system("pause");

    return 0;
}
```

3 函数提高

3.1 函数默认参数

在C++中, 函数的形参列表中的形参是可以有默认值的。

语法: `返回值类型 函数名 （参数 = 默认值）{}`

示例:

```
// 默认参数放在参数最后面
int func1(int a, int b = 20);
// 如果函数申明设置了默认值,则函数定义时不能设置,反之亦是如此
int func1(int a, int b){
    return a + b;
}

int main(){

    cout<<func1(10)<<endl;  // 30

    system("pause");

    return 0;
}
```

3.2 函数占位参数

C++中函数的形参列表里可以有占位参数, 用来做占位, 调用函数时必须填补该位置

语法: `返回值类型 函数名 （数据类型）{}`

在现阶段函数的占位参数存在意义不大, 但是后面的课程中会用到该技术

示例:

```
// 占位参数：调用函数时必须填补该位置
void func2(int a, int){
    cout<<a<<endl;
}
// 展位参数还可以有默认参数
void func3(int a, int = 100){
    cout<<a<<endl;
}
int main(){

    func2(10, 20);  // 10
    func3(10);  // 10

    system("pause");

    return 0;
}
```

3.3 函数重载

3.3.1 函数重载概述

作用: 函数名可以相同, 提高复用性

函数重载满足条件:

- **同一个作用域下**
- 函数名称相同
- 函数参数 **类型不同** 或者 **个数不同** 或者 **顺序不同**

注意: 函数的返回值类型不可以作为函数重载的条件

示例:

```
// 函数重载需要函数都在同一个作用域下
void func4(){
    cout<<"1"<<endl;
}
// int func4(){  // 报错, 返回值类型不能重载
//     cout<<"1.5"<<endl;
// }
void func4(int a){
    cout<<"2"<<endl;
}
void func4(double a){
    cout<<"2.5"<<endl;
}
void func4(int a, double b){
    cout<<"3"<<"3.5"<<endl;
}
void func4(double a, int b){
    cout<<"3.5"<<"3"<<endl;
}

int main(){

    func4();  // 1
    func4(2);  // 2
    func4(2.5);  // 2.5
    func4(3.5, 3);  // 3.53
    func4(3, 3.5);  // 33.5

    system("pause");

    return 0;
}
```

3.3.2 函数重载注意事项

- 引用作为重载条件
- 函数重载碰到函数默认参数

示例:

```
// 函数重载注意事项
// 1、引用作为重载条件
// 引用作重载
void func5(int &a){
    cout<<"int &a 调用"<<endl;
}
void func5(const int &a){
    cout<<"const int &a 调用"<<endl;
}

// 2、函数重载碰到函数默认参数
// 出现二义性,语法没问题,调用时不知道是哪个
void func6(int a){
    cout<<a<<endl;
}
void func6(int a, int b = 10){
    cout<<a<<b<<endl;
}

int main(){

    int num = 10;
    // 引用必须指向合法空间, 所以是 int &a = num;
    // 常引用 const int &a = 10;
    func5(num);  // int &a 调用
    func5(10);  // const int &a 调用

    // 出现二义性,语法没问题,调用时不知道是哪个
    func6(10);  // 报错

    system("pause");

    return 0;
}
```

4 类和对象

C++面向对象的三大特性为: 封装、继承、多态

C++认为万事万物都皆为对象, 对象上有其属性和行为

4.1 封装

4.1.1 封装的意义

封装是C++面向对象三大特性之一

封装的意义:

- 将属性和行为作为一个整体, 表现生活中的事物

- 将属性和行为加以权限控制

语法: `class 类名{ 访问权限: 属性 / 行为 };`

类在设计时, 可以把属性和行为放在不同的权限下, 加以控制

访问权限有三种:

1. **公共权限public** : 类内可以访问, 类外可以访问
2. **保护权限protected** : 类内可以访问, 类外不可以访问
3. **私有权限private** : 类内可以访问, 类外不可以访问

示例:

```
class Person{
    public:
        string m_Name;  // 姓名  公共权限
    protected:
        string m_Car;  // 汽车  保护权限
    private:
        int m_Password;  // 银行卡密码  私有权限
};
```

4.1.2 struct和class区别

在C++中 struct和class唯一的**区别**就在于 **默认的访问权限不同**

区别:

- struct 默认权限为公共
- class 默认权限为私有

示例:

```
#include<iostream>
#include<stdio.h>
// struct 默认权限为public
// class 默认权限为private
using namespace std;
class c1{
    int a;  // 默认权限为private
};
struct c2{
    int a; // 默认权限为public
};

int main(){

    c1 m1;
    c2 m2;
    // m1.a = 10;  // 错误, 访问权限为私有
    m2.a = 10;  // 正确, 访问权限为公共

    return 0;
```

```
}
```

4.1.3 灵活设置成员属性

优点1: 将成员属性设置为私有, 可以自己控制读写权限

优点2: 对于写权限, 我们可以检测数据的有效性

示例:

```
class Person {
    public:
        void setName(string name) {
            m_Name = name;
        }
        string getName(){
            return m_Name;
        }
        void setAge(int age) {
            if (age < 0 || age > 150) {
                cout<<"你个老妖精!"<<endl;
                return;
            }
            m_Age = age;
        }
        int getAge(){
            return m_Age;
        }
        // 设置情人
        void setLover(string lover) {
            m_Lover = lover;
        }

    private:
        string m_Name; // 姓名
        int m_Age; // 年龄
        string m_Lover; // 情人
};


int main(){

    Person p;
    // 姓名设置
    p.setName("张三");
    cout<<"姓名:  "<<p.getName()<<endl;

    // 年龄设置
    p.setAge(50);
    cout<<"年龄:  "<<p.getAge()<<endl;

    // 情人设置

    p.setLover("苍井");
```

```
    system("pause");

    return 0;
}
```

4.2 对象的初始化和清理

4.2.1 构造函数和析构函数

对象的 **初始化和清理** 也是两个非常重要的安全问题

一个对象或者变量没有初始状态, 对其使用后果是未知

同样的使用完一个对象或变量, 没有及时清理, 也会造成一定的安全问题

c++利用了 **构造函数** 和 **析构函数** 解决上述问题, 这两个函数将会被编译器自动调用, 完成对象初始化和清理工作。

对象的初始化和清理工作是编译器强制要我们做的事情, 因此 **如果我们不提供构造和析构, 编译器会提供**

编译器提供的构造函数和析构函数是空实现。

- 构造函数: 主要作用在于创建对象时为对象的成员属性赋值, 构造函数由编译器自动调用, 无须手动调用。
- 析构函数: 主要作用在于对象 **销毁前** 系统自动调用, 执行一些清理工作。

构造函数语法: `类名(){}`

1. 构造函数, 没有返回值也不写void
2. 函数名称与类名相同
3. 构造函数可以有参数, 因此可以发生重载
4. 程序在调用对象时候会自动调用构造, 无须手动调用,而且只会调用一次

析构函数语法: `~类名(){}`

1. 析构函数, 没有返回值也不写void
2. 函数名称与类名相同,在名称前加上符号 ~
3. 析构函数不可以有参数, 因此不可以发生重载
4. 程序在对象销毁前会自动调用析构, 无须手动调用,而且只会调用一次

4.2.2 构造函数的分类及调用

两种分类方式:

按参数分为: 有参构造和无参构造

按类型分为: 普通构造和拷贝构造

三种调用方式:

括号法

显示法

隐式转换法

4.2.3 拷贝构造三种调用

C++中拷贝构造函数调用时通常有三种情况

- 使用已经创建的对象来 **初始化** 新对象会调用拷贝构造
- 函数传参时, **值传递** 的方式会调用拷贝构造
- 以 **值方式** 返回局部对象时会调用拷贝构造

示例:

```
#include<iostream>
#include<stdio.h>
using namespace std;
// 构造函数分类
// 按照参数分类分为有参和无参构造，无参又称为默认构造函数
// 按照类型分类分为普通构造和拷贝构造

// 拷贝构造的三种调用情况
// 已经创建的对象初始化新对象
// 函数传参时，值传递的方式
// 函数返回值为对象时(编译器此处使用返回值优化的技术，从而不需用调用拷贝构造)
class Person{
    public:
        // 无参(默认)构造
        Person(){
            cout<<"无参构造"<<endl;
        }
        // 有参构造
        Person(int a){
            cout<<"有参构造"<<endl;
            age = a;
        }
        // 拷贝构造
        // const Person &p: 常引用
        Person(const Person &p){
            cout<<"拷贝构造"<<endl;
            name = p.name;
            age = p.age;
        }
        // 析构函数
        ~Person(){}
    private:
        string name;
        int age;
};

void copyPerson(Person p){}
Person retuenPerson(){
    Person p;
    cout<<&p<<endl;
    // 返回的不是栈区的p，而是将p拷贝到全局区
    // 但编译器使用了一项名为返回值优化的技术，使得调用函数时不需要调用复制构造函数
    return p;
}
```

```
int main(){

    // 三种调用构造方法
    // 括号法
    Person p1_1;  // 默认构造
    Person p1_2(10);  // 有参构造
    Person p1_3(p1_1);  // 拷贝构造
    // 显示法
    Person p2 = Person(10);
    Person p3 = Person(p2);
    // 隐式法
    Person p4 = 10;
    Person p5 = p3;

    // 使用拷贝构造的三种情况
    // 初始化新对象
    Person p6(p2);
    // 值传递时，将全局区实参拷贝一份传入栈区函数中
    copyPerson(p6);
    // 值方式返回局部对象
    Person p = retuenPerson();
    cout<<&p<<endl;
    // 此处输出的地址相同
    // 编译器使用了一项名为返回值优化的技术，使得调用函数时不需要调用复制构造函数

    return 0;
}
```

4.2.4 构造函数调用规则

默认情况下, c++编译器至少给一个类添加3个函数

1．默认构造函数(无参, 函数体为空)

2．默认析构函数(无参, 函数体为空)

3．默认拷贝构造函数, 对属性进行 **值** 拷贝

构造函数调用规则如下:

- 如果用户定义有参构造函数, c++不在提供默认无参构造, 但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数, c++不会再提供其他构造函数

4.2.5 深拷贝与浅拷贝

深浅拷贝是面试经典问题, 也是常见的一个坑

浅拷贝: 简单的赋值操作

深拷贝: 在堆区重新申请空间, 进行拷贝操作

示例:

```
// 浅拷贝：简单的赋值操作，若申请了堆区内存则在析构函数中会重复delete
// 深拷贝：在堆区重新申请空间，进行拷贝操作
```

```
// 编译器提供的拷贝构造为浅拷贝，需要自己编写深拷贝的拷贝构造
class Person{
    public:
        Person(){}
        Person(int a_age, int a_height){
            age = a_age;
            // 堆区开辟一块空间存放升高
            height = new int();
            *height = a_height;
        }
        Person(const Person &p){
            // 编译器默认实现下两行
            // age = p.age;
            // height = p.height;  // 浅拷贝，只把地址拷贝过去，指向的是堆里的同一空间

            age = p.age;
            // 利用深拷贝解决浅拷贝问题
            // *p.height先解引用，获取堆空间里存放的值，在重新申请堆空间
            height = new int(*p.height);
        }
        ~Person(){
            if(height != NULL){
                cout<<"delete "<<height<<endl;
                delete height;  // 删除堆区开辟的内存空间
                height = NULL;
            }
        }
        int* getheight(){
            return height;
        }

    private:
        int age;
        int * height; // 定义一个指向身高的指针
};

void test(){
    Person p1(18, 170);
    Person p2(p1);
    // 析构时采用先定义后析构的准则
    // 先析构p2,若使用浅拷贝，则把p2指向的堆空间释放了，p1析构时报错
}
int main(){

    test();

    return 0;
}
```

> 总结: 如果属性在堆区开辟来内存, 一定要自己提供拷贝构造函数, 防止编译器浅拷贝带来的问题

4.2.6 初始化列表

作用:

C++提供了初始化列表语法, 用来初始化属性

语法: `构造函数(): 属性1(值1), 属性2(值2)...{}`

示例:

```
class Person{
    public:
        // // 传统方式初始化
        // Person(int a, int b, int c) {
        //   A = a;
        //   B = b;
        //   C = c;
        // }

        // 初始化列表方式初始化
        Person(int a, int b, int c): A(a), B(b), C(c){}
        void PrintPerson(){
            cout <<"A:"<<A<<endl;
            cout <<"B:"<<B<<endl;
            cout <<"C:"<<C<<endl;
        }
    private:
        int A;
        int B;
        int C;
};

int main(){

    Person p(1, 2, 3);
    p.PrintPerson();

    return 0;
}
```

4.2.7 类对象作为类成员

C++类中的成员可以是另一个类的对象, 我们称该成员为 对象成员

例如:

```
// 当类中成员是其他类对象时，我们称该成员为 对象成员
// B类中有对象A作为成员，A为对象成员
// 构造的顺序是：先调用对象成员的构造，再调用本类构造
// 析构顺序与构造相反
class A{};
class B{
    A a;
};
```

4.2.8 静态成员

静态成员就是在成员变量和成员函数前加上关键字static, 称为静态成员, 静态成员是一个类共享的即每个对象的都一样

静态成员分为:

- 静态成员变量
  - 所有对象共享同一份数据
  - 在编译阶段分配内存
  - 类内声明, 类外初始化
- 静态成员函数
  - 所有对象共享同一个函数
  - 静态成员函数只能访问静态成员变量

示例1: 静态成员变量

```
// 静态成员变量特点:
// 1 在编译阶段分配内存
// 2 类内声明，类外初始化
// 3 所有对象共享同一份数据
class Person{
    public:
        static int A; // 静态成员变量

    private:
        static int B; // 静态成员变量也可以是私有访问权限
};
// 类外初始化
int Person::A = 10;
int Person::B = 10;

void test01(){
    // 静态成员变量两种访问方式

    // 1、通过对象
    Person p1;
    p1.A = 100;
    cout<<"p1.A = "<<p1.A<<endl; // p1.A = 100

    Person p2;
    p2.A = 200;
    // 共享同一份数据
    cout<<"p1.A = "<<p1.A<<endl; // p1.A = 200
    cout<<"p2.A = "<<p2.A<<endl; // p2.A = 200

    // 2、通过类名
    cout<<"A = "<<Person::A<<endl; // A = 200

    // cout<<"B = "<<Person::B<<endl; // 私有权限访问不到
}
```

```
int main(){

    test01();

    return 0;
}
```

示例2: 静态成员函数

```
// 静态成员函数特点:
// 1 程序共享一个函数
// 2 静态成员函数只能访问静态成员变量
class Person{
    public:
        static void func(){
            cout<<"func调用"<<endl;
            A = 100;
            // B = 100; // 错误，不可以访问非静态成员变量
        }
        static int A; // 静态成员变量
        int B;
    private:
        // 静态成员函数也是有访问权限的
        static void func2(){
            cout<<"func2调用"<<endl;
        }
};
int Person::A = 10;

void test01(){
    // 静态成员变量两种访问方式
    // 1、通过对象
    Person p1;
    p1.func();  // func调用
    // 2、通过类名，不需要创建对象
    Person::func();  // func调用
    // Person::func2(); // 私有权限访问不到
}

int main(){

    test01();

    return 0;
}
```

4.3 C++对象模型和this指针

4.3.1 成员变量和成员函数分开存储

在C++中, 类内的成员变量和成员函数分开存储

只有非静态成员变量才属于类的对象上

示例:

```
class Person1{

};
class Person2{
    public:
        Person(){
            A = 0;
        }
        // 非静态成员变量占对象空间
        int A;
        // 静态成员变量不占对象空间
        static int B;
        // 函数也不占对象空间，所有函数共享一个// /静态成员函数
        void func(){
            cout<<"A:"<<this->A<<endl;
        }
        // 静态成员函数也不占对象空间
        static void sfunc(){}
};

int main(){

    Person1 p1;  // 创建一个空类的对象
    // 空对象占用内存为 1
    // C++编译器给每个空对象分配一个字节从而确定空对象地址
    cout<<sizeof(p1)<<endl;  // 1

    Person p2;
    // 只有非静态成员变量占对象空间
    cout<<sizeof(p2)<<endl;  // 4

    return 0;
}
```

4.3.2 this指针概念

通过4.3.1我们知道在C++中成员变量和成员函数是分开存储的

静态成员变量和静态成员函数存放在全局区

非静态成员函数

- 每一个非静态成员函数只会诞生一份函数实例, 也就是说多个同类型的对象会共用一块函数代码
- 这一块代码是如何区分那个对象调用自己的呢？c++通过特殊的对象指针, this指针。**this指针指向被调用的成员函数所属的对象**

this指针是隐含每一个非静态成员函数内的一种指针

this指针不需要定义, 直接使用即可

this指针的用途:

- 当形参和成员变量同名时, 可用this指针来区分
- 在类的非静态成员函数中返回对象本身, 可使用return *this

```
// this指针指向被调用的成员函数所属的对象
// this指针的用途:
// 1 当形参和成员变量同名时，可用this指针来区分
// 2 在类的非静态成员函数中返回对象本身，可使用return *this
// 3 类中调用成员变量时，都默认前面省略了 this->
class Person{
    public:
        int age;
        Person(int age){
            // 1 当形参和成员变量同名时，可用this指针来区分
            this->age = age;
        }
        Person& PersonAddPerson(Person p){
            this->age = p.age;
            // 2 在类的非静态成员函数中返回对象本身，可使用return *this
            return *this;
        }
};
int main(){

    Person p1(18);
    cout<<p1.age<<endl; // 18

    // 隐式拷贝构造
    Person p2 = p1.PersonAddPerson(p1);
    cout<<p2.age<<endl; // 18

    return 0;
```

4.3.3 空指针访问成员函数情况

C++中空指针也是可以调用成员函数的, 但是不能使用到this指针(类中调用成员变量时, 都默认前面省略了this->)

```
// 如果用到this指针，需要加以判断保证代码的健壮性
if(this == NULL)
    return;
```

示例:

```
// 空指针访问成员函数
class Person{
    public:
        void ShowClassName(){
            cout<<"我是Person类!"<<endl;
        }
```

```
        void ShowPerson(){
            // 类中调用成员变量时，都默认前面省略了 this->
            // 此处实际为 this->age
            cout<<age<<endl;
        }
    private:
        int age;
};

int main(){

    Person * p = NULL;
    p->ShowClassName(); // 空指针，可以调用成员函数
    // p->ShowPerson();  // 报错: this is nullptr

    return 0;
}
```

4.3.4 const修饰成员函数

常函数:

- 成员函数后加const后我们称为这个函数为 **常函数**
- 常函数内不可以修改成员属性
- 若成员属性声明时加关键字mutable后, 在常函数中依然可以修改

常对象:

- 声明对象前加const称该对象为 **常对象**
- 常对象只能调用常函数
- 常对象不能修改成员变量的值,但是可以访问
- 若成员变量声明时加关键字mutable后,则可以修改成员变量的值

示例:

```
// 常函数:
// 成员函数()后加const后我们称为这个函数为常函数
// 常函数内不可以修改成员属性
// 若成员属性声明时加关键字mutable后，在常函数中依然可以修改

// 常对象:
// 声明对象前加const称该对象为常对象
// 常对象只能调用常函数
// 常对象不能修改成员变量的值,但是可以访问
// 若成员变量声明时加关键字mutable后,则可以修改成员变量的值
class Person{
    public:
        int A;
        mutable int B; // 可修改可变的
        Person(){
            A = 0;
            B = 0;

        }
```

```
        void ShowPerson1(){
            // this指针的本质是一个指针常量(指针的指向不可修改)
            // this = NULL; // 报错:不能修改指针的指向 Person* const this;
            this->A = 100; // 但是this指针指向的值是可以修改的
        }
        // 如果想让指针指向的值也不可以修改
        // 即 const Type* const this;
        // 所以声明常函数来指代const Type* const this;
        void ShowPerson2() const{
            // const修饰成员函数, 表示指针指向的内存空间的数据不能修改
            // this->A = 100; 报错

            // mutable修饰的变量可以修改
            this->B = 100;
        }
};

int main(){

    // 常对象
    const Person p;
    cout<<p.A<<endl;  // 10
    // p.A = 100; // 报错: 常对象不能修改成员变量的值,但是可以访问
    p.B = 100; // 常对象可以修改mutable修饰的成员变量的值

    // 常对象只能调用函数
    // 普通函数可以修改成员变量的值,但是常对象不允许修改,所以只能调用修改不了的常函数
    // p.ShowPerson1(); // 报错
    p.ShowPerson2();

    return 0;
}
```

4.4 友元

生活中你的家有客厅(Public), 有你的卧室(Private)

客厅所有来的客人都可以进去, 但是你的卧室是私有的, 也就是说只有你能进去

但是呢, 你也可以允许你的好闺蜜好基友进去。

在程序里, 有些私有属性也想让类外特殊的一些函数或者类进行访问, 就需要用到友元的技术

友元的目的就是让一个函数或者类访问另一个类中私有成员

友元的关键字为 friend

友元的三种实现

- 全局函数做友元
- 类做友元
- 成员函数做友元

示例:

```
// 友元的三种方式
// 1 全局函数做友元，全局函数的申明和定义随便在什么位置
// 2 类做友元，类的定义随便在什么位置
// 3 类A中的成员函数做类B友元，注意：此处的函数声明必须在类B前面
class GoodGay1{
    public:
        void visit();
    private:
        Building building;
};
class GoodGay2{
    public:
        void visit1();
        void visit2();
    private:
        Building building;
};
class Building{
    // 全局函数做友元，全局函数的申明和定义随便在什么位置
    friend void goodGay2(Building &building);
    // 类做友元，类的定义随便在什么位置
    friend class GoodGay1;
    // 类中的成员函数做友元，注意：此处的函数声明必须在这个类前面
    friend void GoodGay2::visit2();

    public:
        string living_room; // 客厅
        Building();

    private:
        string bedroom; // 卧室
};

Building::Building(){
            living_room = "客厅";
            bedroom = "卧室";
        }

void GoodGay1::visit(){
    cout<<"private权限"<<building.bedroom<<endl;
}

void GoodGay2::visit1(){
    cout<<"public权限"<<building.living_room<<endl;
    // cout<<"private权限"<<building.bedroom<<endl; // 报错：私有权限不可访问
}
void GoodGay2::visit2(){
    cout<<"public权限"<<building.living_room<<endl;
    cout<<"private权限"<<building.bedroom<<endl;
}

void goodGay1(Building &building){
    cout<<"public权限"<<building.living_room<<endl;
```

```
    // cout<<"private权限"<<building.bedroom<<endl; // 报错: 私有权限不可访问
}
void goodGay2(Building &building){
    cout<<"public权限"<<building.living_room<<endl;
    cout<<"private权限"<<building.bedroom<<endl; // 可以访问,类中申明了友元
}

int main(){

    Building b;
    goodGay1(b); // 客厅
    goodGay2(b); // 客厅  卧室

    GoodGay1 gg;
    gg.visit();  // 卧室

    GoodGay2 gg2;
    gg2.visit1(); // 客厅
    gg2.visit2(); // 客厅  卧室

    return 0;
}
```

4.5 运算符重载

运算符重载概念: 对已有的运算符重新进行定义, 赋予其另一种功能, 以适应不同的数据类型

4.5.1 加号运算符重载

作用: 实现两个自定义数据类型相加的运算

示例: 通过 **成员函数** 运算符重载

```
// 通过成员函数运算符重载
class Person {
    public:
        Person(){};
        Person(int a, int b){
            A = a;
            B = b;
        }
        // 成员函数实现 + 号运算符重载
        Person operator+(const Person& p) {
            Person temp;
            temp.A = this->A + p.A;
            temp.B = this->B + p.B;
            return temp;
        }
        void printPrivate(){
            cout<<"A = "<<A<<" B = "<<B<<endl;
        }
    private:
        int A;
```

```
        int B;
    };

int main(){

    Person p1(10, 10);
    Person p2(20, 20);

    // 成员函数方式
    Person p3 = p2 + p1;  // 相当于 p2.operaor+(p1)
    p3.printPrivate();

    return 0;
}
```

示例: 通过 **全局函数** 运算符重载

```
// 通过全局函数运算符重载
class Person{
    friend Person operator+(Person &p1, Person &p2);
    friend Person operator+(Person &p, int val);
    public:
        Person(int a, int b){
            A = a;
            B = b;
        }
        void printPrivate(){
            cout<<"A = "<<A<<" B = "<<B<<endl;
        }
    private:
        int A;
        int B;
};
// 若使用 const Person &p1, 则无法p1.A
// C++常引用不能访问private权限?
Person operator+(Person &p1, Person &p2){
    Person temp(0, 0);
    temp.A = p1.A + p2.A;
    temp.B = p1.B + p2.B;
    return temp;
}
// 函数重载
Person operator+(Person &p, int val){
    Person temp(0, 0);
    temp.A = p.A + val;
    temp.B = p.B + val;
    return temp;
}

int main(){


    Person p1(10, 20), p2(15, 15);
```

```
    Person p3 = p1 + p2;
    p3.printPrivate();  // A = 25 B = 35
    Person p4 = p1 + 10;
    p4.printPrivate(); // A = 20 B = 30

    return 0;
}
```

> 总结 : 内置的数据类型运算符不能改变, 全局函数进行运算符重载时可以伴随着函数重载

> 疑惑 : C++常引用不能访问private权限?

4.5.2 左移运算符(<<)、i++、++i 重载

示例

```
class Person{

    friend ostream& operator<<(ostream &out, Person p);

    public:
        Person(int a, int b){
            A = a;
            B = b;
        }
        // 前置++
        // 引用本质: 指针常量,
        Person& operator++(){
            A++;
            B++;
            return *this;
        }
        // 后置++
        Person operator++(int){
            Person temp = *this; // 记录当前本身的值，然后让本身的值加1，但是返回的是以前的值，达到先返回后++;
            A++;
            B++;
            return temp;
        }

    private:
        int A;
        int B;
};
// iostreasm流实现<<重载
ostream& operator<<(ostream &out, Person p){
    out<<"A = "<<p.A<<" B = "<<p.B<<endl;
    return out;
}

int main(){
```

```
    Person p1(10, 20);
    cout<<p1++; // A = 10 B = 20
    cout<<p1; // A = 11 B = 21
    cout<<++p1; // A = 12 B = 22
    cout<<p1; // A = 12 B = 22

    return 0;
}
```

4.5.4 赋值运算符重载

c++编译器至少给一个类添加4个函数

1. 默认构造函数(无参, 函数体为空)
2. 默认析构函数(无参, 函数体为空)
3. 默认拷贝构造函数, 对属性进行值拷贝
4. 赋值运算符 operator=, 对属性进行值拷贝

如果类中有属性指向堆区, 做赋值操作时也会出现深浅拷贝问题

示例:

```
class Person{
    public:
        // 年龄的指针
        int *m_Age;

        Person(int age){
            // 将年龄数据开辟到堆区
            m_Age = new int(age);
        }
        // 重载赋值运算符
        Person& operator=(Person &p){
            if(m_Age != NULL){
                delete m_Age;
                m_Age = NULL;
            }
            // 编译器提供的代码是浅拷贝
            // m_Age = p.m_Age;

            // 提供深拷贝 解决浅拷贝的问题
            m_Age = new int(*p.m_Age);

            // 返回自身
            return *this;
        }
        ~Person(){
            if (m_Age != NULL){
                delete m_Age;
                m_Age = NULL;
            }
        }
};
```

```
int main(){

    Person p1(18);
    Person p2(20);

    p2 = p1; // 赋值操作

    cout<<"p1的年龄为: "<<*p1.m_Age<<endl;
    cout<<"p2的年龄为: "<<*p2.m_Age<<endl;

    return 0;
}
```

4.5.5 关系运算符重载

作用: 重载关系运算符, 可以让两个自定义类型对象进行对比操作

示例:

```
class Person{
public:
    Person(string name, int age){
        this->m_Name = name;
        this->m_Age = age;
    };
    bool operator==(Person & p){
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age){
            return true;
        }
        else{
            return false;
        }
    }
    bool operator!=(Person & p){
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age){
            return false;
        }
        else{
            return true;
        }
    }

    string m_Name;
    int m_Age;
};

int main(){

    Person a("孙悟空", 18);
    Person b("孙悟空", 18);

    if (a == b){
```

```
        cout<<"a和b相等"<<endl;
    }
    else{
        cout<<"a和b不相等"<<endl;
    }

    if (a != b){
        cout<<"a和b不相等"<<endl;
    }
    else{
        cout<<"a和b相等"<<endl;
    }

    return 0;
}
```

4.5.6 函数调用运算符重载

- 函数调用运算符 () 也可以重载
- 由于重载后使用的方式非常像函数的调用, 因此称为仿函数
- 仿函数没有固定写法, 非常灵活

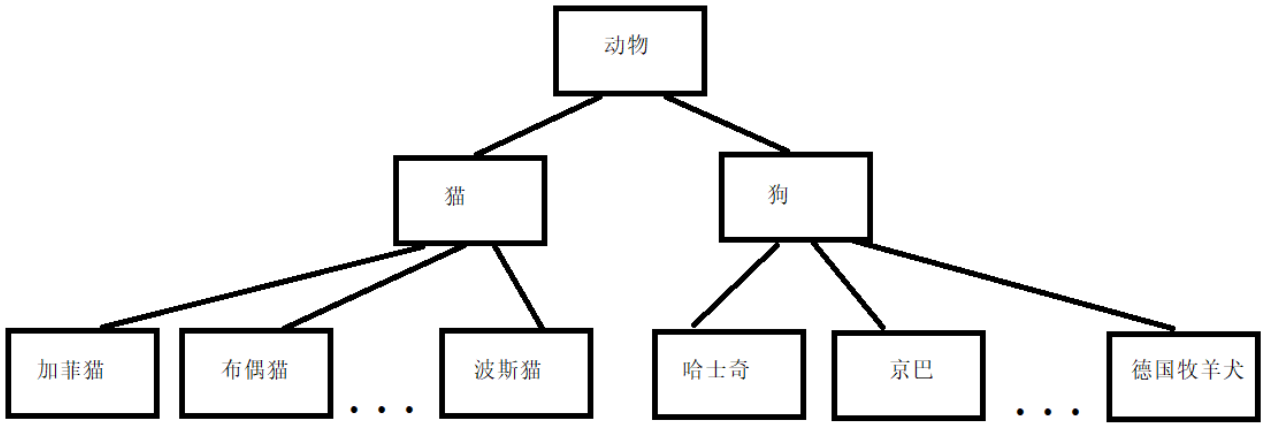
示例:

```
class MyPrint{
    public:
        void operator()(string text){
            cout<<text<<endl;
        }
};

class MyAdd{
    public:
        int operator()(int v1, int v2){
            return v1 + v2;
        }
};

int main(){

    // 重载的()操作符 也称为仿函数
    MyPrint myFunc;
    myFunc("hello world");  // hello world

    MyAdd add;
    int ret = add(10, 10);
    cout<<"ret = "<<ret<<endl;  //  ret = 20
    // 匿名对象调用
    cout<<"MyAdd()(100,100) = "<<MyAdd()(100, 100)<<endl;  // MyAdd()(100,100) = 200

    return 0;
}
```

4.6 继承

继承是面向对象三大特性之一

有些类与类之间存在特殊的关系, 例如下图中:



我们发现, 定义这些类时, 下级别的成员除了拥有上一级的共性, 还有自己的特性。

这个时候我们就可以考虑利用继承的技术, 减少重复代码

4.6.1 继承的基本语法

例如我们看到很多网站中, 都有公共的头部, 公共的底部, 甚至公共的左侧列表, 只有中心内容不同

接下来我们分别利用普通写法和继承的写法来实现网页中的内容, 看一下继承存在的意义以及好处

普通实现:

```
// Java页面
class Java{
    public:
        void header(){
            cout<<"首页、公开课、登录、注册...(公共头部)"<<endl;
        }
        void footer(){
            cout<<"帮助中心、交流合作、站内地图...(公共底部)"<<endl;
        }
        void left(){
            cout<<"Java,Python,C++...(公共分类列表)"<<endl;
        }
        void content(){
            cout<<"JAVA学科视频"<<endl;
        }
};
// Python页面
class Python{
    public:

        void header(){
```

```
            cout<<"首页、公开课、登录、注册...(公共头部)"<<endl;
        }
        void footer(){
            cout<<"帮助中心、交流合作、站内地图...(公共底部)"<<endl;
        }
        void left(){
            cout<<"Java,Python,C++...(公共分类列表)"<<endl;
        }
        void content(){
            cout<<"Python学科视频"<<endl;
        }
};
// C++页面
class CPP{
    public:
        void header(){
            cout<<"首页、公开课、登录、注册...(公共头部)"<<endl;
        }
        void footer(){
            cout<<"帮助中心、交流合作、站内地图...(公共底部)"<<endl;
        }
        void left(){
            cout<<"Java,Python,C++...(公共分类列表)"<<endl;
        }
        void content(){
            cout<<"C++学科视频"<<endl;
        }
};

int main(){

    // Java页面
    cout<<"Java下载视频页面如下:  "<<endl;
    Java ja;
    ja.header();
    ja.footer();
    ja.left();
    ja.content();
    cout<<"--------------------"<<endl;

    // Python页面
    cout<<"Python下载视频页面如下:  "<<endl;
    Python py;
    py.header();
    py.footer();
    py.left();
    py.content();
    cout<<"--------------------"<<endl;

    // C++页面
    cout<<"C++下载视频页面如下:  "<<endl;
    CPP cp;

    cp.header();
```

```
    cp.footer();
    cp.left();
    cp.content();

    return 0;
}
```

继承实现:

```
// 公共页面
class BasePage{
    public:
        void header(){
            cout<<"首页、公开课、登录、注册...(公共头部)"<<endl;
        }
        void footer(){
            cout<<"帮助中心、交流合作、站内地图...(公共底部)"<<endl;
        }
        void left(){
            cout<<"Java,Python,C++...(公共分类列表)"<<endl;
        }
};

// Java页面
class Java: public BasePage{
    public:
        void content(){
            cout<<"JAVA学科视频"<<endl;
        }
};
// Python页面
class Python: public BasePage{
public:
    void content(){
        cout<<"Python学科视频"<<endl;
    }
};
// C++页面
class CPP: public BasePage{
    public:
        void content(){
            cout<<"C++学科视频"<<endl;
        }
};

int main(){

    // Java页面
    cout<<"Java下载视频页面如下:  "<<endl;
    Java ja;
    ja.header();

    ja.footer();
```

```
    ja.left();
    ja.content();
    cout<<"--------------------"<<endl;

    // Python页面
    cout<<"Python下载视频页面如下:  "<<endl;
    Python py;
    py.header();
    py.footer();
    py.left();
    py.content();
    cout<<"--------------------"<<endl;

    // C++页面
    cout<<"C++下载视频页面如下:  "<<endl;
    CPP cp;
    cp.header();
    cp.footer();
    cp.left();
    cp.content();

    return 0;
}
```

总结:

继承的好处: 可以减少重复的代码

class A : public B;

A 类称为子类 或 派生类

B 类称为父类 或 基类

派生类中的成员, 包含两大部分:

- 一类是从基类继承过来的, 一类是自己增加的成员。
- 从基类继承过过来的表现其共性, 而自己新增的成员体现了其个性。

4.6.2 继承方式

继承的语法: `class 子类 : 继承方式 父类`

继承方式一共有三种:

- 公共继承: 从父类继承的public属性仍为public, protected属性仍为protected
- 保护继承: 从父类继承的全部public和protected属性均变为protected属性
- 私有继承: 从父类继承的全部public和protected属性均变为private属性

示例:

```
class Base{
    public:
        int m_A;

    protected:
```

```
        int m_B;
    private:
        int m_C;
};

// 公共继承
class Son1: public Base{
    public:
        void func(){
            m_A; // 类内可访问 public权限
            m_B; // 类内可访问 protected权限
            // m_C; // 类内不可访问 private权限
        }
};
void myClass(){
    Son1 s1;
    s1.m_A; // 类外只能访问到public权限
}

// 保护继承
class Son2: protected Base{
    public:
        void func(){
            m_A; // 类内可访问 protected权限
            m_B; // 类内可访问 protected权限
            // m_C; // 类内不可访问 private权限
        }
};
void myClass2(){
    Son2 s;
    // s.m_A; // 类外不可访问 protected权限
}

// 私有继承
class Son3: private Base{
    public:
        void func(){
            m_A; // 类内可访问 protected权限
            m_B; // 类内可访问 protected权限
            // m_C; // 类内不可访问 private权限
        }
};
class GrandSon: public Son3{
    public:
        void func(){
            // Son3是私有继承，所以继承Son3的属性在GrandSon3中都无法访问到
        }
};
```

继承访问不到父类的private属性

类内的三种权限

- public : 类内类外都可访问

- protected : 类内可以访问, 类外不可访问
- private : 类内可以访问, 类外不可访问, 子类无法访问(无法访问并不等于没有继承)

示例:

```
class Base{
    public:
        int m_A;
    protected:
        int m_B;
    private:
        int m_C; // 私有成员只是被隐藏了，但是还是会继承下去
};

// 公共继承
class Son: public Base{
    public:
        int m_D;
};

int main(){

    // 父类中所有成员属性都会被继承
    cout<<"sizeof Son = "<<sizeof(Son)<<endl;
    // sizeof Son = 16 (4 * 4(int))

    return 0;
}
```

> 结论: 父类中私有成员也是被子类继承下去了, 只是由编译器给隐藏后访问不到

4.6.3 继承中构造和析构顺序

子类继承父类后, 当创建子类对象, 也会调用父类的构造函数

问题: 父类和子类的构造和析构顺序是谁先谁后?

示例:

```
class Base {
    public:
        Base(){
            cout<<"Base构造函数!"<<endl;
        }
        ~Base(){
            cout<<"Base析构函数!"<<endl;
        }
};

class Son: public Base{
    public:
        Son(){
            cout<<"Son构造函数!"<<endl;
```

```
        }
        ~Son(){
            cout<<"Son析构函数!"<<endl;
        }
};

int main(){

    // 继承中先调用父类构造函数，再调用子类构造函数，析构顺序与构造相反
    Son s;

    return 0;
}
```

> 总结: 继承中 先调用父类构造函数, 再调用子类构造函数, 析构顺序与构造相反

4.6.4 继承同名成员处理方式

问题: 当子类与父类出现同名的成员, 如何通过子类对象, 访问到子类或父类中同名的数据呢?

- 访问子类同名成员->直接访问即可
- 访问父类同名成员->需要加作用域

示例:

```
class Base {
    public:
        int m_A;
        Base(){
            m_A = 100;
        }
        void func(){
            cout<<"Base - func()调用"<<endl;
        }
        void func(int a){
            cout<<"Base - func(int a)调用"<<endl;
        }
};

class Son: public Base {
    public:
        int m_A;
        Son(){
            m_A = 200;
        }
        void func(){
            cout<<"Son - func()调用"<<endl;
        }
};

int main(){


    Son s;
```

```
    // 当子类与父类拥有同名的成员函数，子类会隐藏父类中所有版本的同名成员函数
    // 如果想访问父类中被隐藏的同名成员函数，需要加父类的作用域
    cout<<"Son下的m_A = "<<s.m_A<<endl;
    cout<<"Base下的m_A = "<<s.Base::m_A<<endl;

    s.func();
    s.Base::func();
    s.Base::func(10);

    return 0;
}
```

总结:

1. 子类对象可以直接访问到子类中同名成员
2. 子类对象加作用域可以访问到父类同名成员
3. 当子类与父类拥有同名的成员函数, 子类会隐藏父类中同名成员函数, 加作用域可以访问到父类中同名函数

4.6.5 继承同名静态成员处理方式

问题: 继承中同名的静态成员在子类对象上如何进行访问?

静态成员和非静态成员出现同名, 处理方式一致

- 访问子类同名成员->直接访问即可
- 访问父类同名成员->需要加作用域

示例:

```
class Base{
    public:
        static void func(){
            cout<<"Base - static void func()"<<endl;
        }
        static void func(int a){
            cout<<"Base - static void func(int a)"<<endl;
        }
        static int m_A;
};
int Base::m_A = 100;

class Son: public Base{
    public:
        static void func(){
            cout<<"Son - static void func()"<<endl;
        }
        static int m_A;
};
int Son::m_A = 200;

int main(){

    // 同名成员属性
    // 通过对象访问
```

```
    cout<<"通过对象访问: "<<endl;
    Son s;
    cout<<"Son  下 m_A = "<<s.m_A<<endl;
    cout<<"Base 下 m_A = "<<s.Base::m_A<<endl;
    // 通过类名访问
    cout<<"通过类名访问: "<<endl;
    cout<<"Son  下 m_A = "<<Son::m_A<<endl;
    cout<<"Base 下 m_A = "<<Son::Base::m_A<<endl;

    // 同名成员函数
    // 通过对象访问
    cout<<"通过对象访问: "<<endl;
    Son s;
    s.func();
    s.Base::func();
    cout<<"通过类名访问: "<<endl;
    Son::func();
    Son::Base::func();
    // 出现同名，子类会隐藏掉父类中所有同名成员函数，需要加作作用域访问
    Son::Base::func(100);

    return 0;
}
```

> 总结: 同名静态成员处理方式和非静态处理方式一样, 只不过有两种访问的方式(通过对象、通过类名)

4.6.6 多继承语法

C++允许 **一个类继承多个类**

语法: `class 子类 : 继承方式 父类1 , 继承方式 父类2...`

多继承可能会引发父类中有同名成员出现, 需要加作用域区分

C++实际开发中不建议用多继承

示例:

```
class Base1{
    public:
        int m_A;
        Base1(){
            m_A = 100;
        }
};

class Base2{
    public:
        int m_A;
        Base2(){
            m_A = 200;
        }
};
```

```
// 语法: class 子类 : 继承方式 父类1, 继承方式 父类2
class Son: public Base2, public Base1{
    public:
        int m_C;
        int m_D;
        Son(){
            m_C = 300;
            m_D = 400;
        }

};

int main(){

    // 多继承容易产生成员同名的情况
    // 通过使用类名作用域可以区分调用哪一个基类的成员
    Son s;
    cout<<"sizeof Son = "<<sizeof(s)<<endl;
    cout<<s.Base1::m_A<<endl;
    cout<<s.Base2::m_A<<endl;

    return 0;
}
```

> 总结: 多继承中如果父类中出现了同名情况, 子类使用时候要加作用域

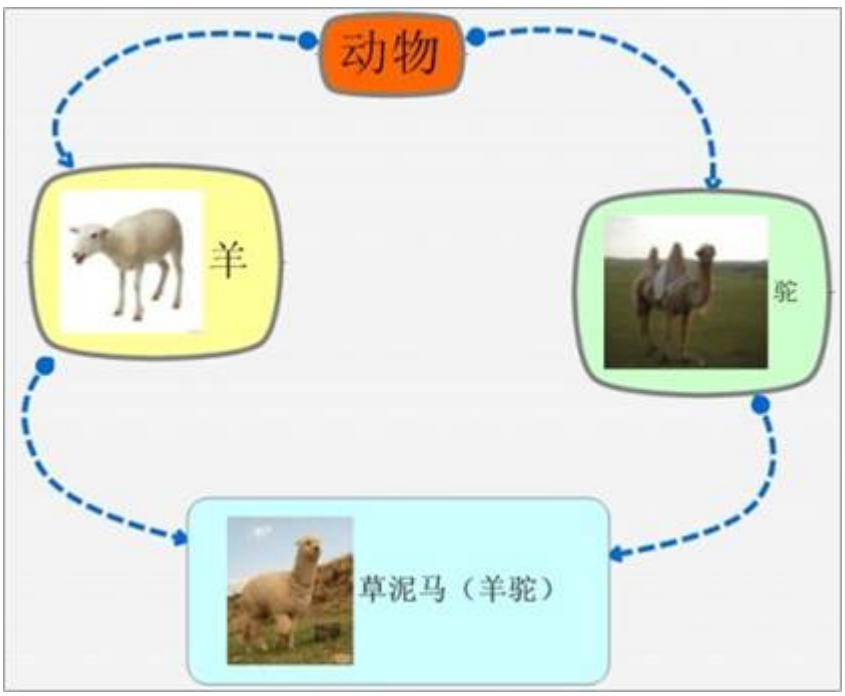
4.6.7 菱形继承(砖石继承)

菱形继承概念:

两个派生类继承同一个基类

又有某个类同时继承者两个派生类

典型的菱形继承案例:



菱形继承问题:

1. 羊继承了动物的数据, 驼同样继承了动物的数据, 当草泥马使用数据时, 就会产生二义性。
2. 草泥马继承自动物的数据继承了两份, 其实我们应该清楚, 这份数据我们只需要一份就可以。

示例:

```
class Animal{
    public:
        int m_Age;
};

// 继承前加virtual关键字后, 变为虚继承
// 此时公共的父类Animal称为虚基类
class Sheep:virtual public Animal{};
class Camel:virtual public Animal{};
class Alpaca: public Sheep, public Camel{};

int main(){

    Alpaca a;
    a.Sheep::m_Age = 18;
    a.Camel::m_Age = 28;
    // 使用虚继承后m_Age只有一份
    cout<<"a.Sheep::m_Age = "<<a.Sheep::m_Age<<endl; // a.Sheep::m_Age = 28
    cout<<"a.Camel::m_Age = "<<a.Tuo::m_Age<<endl; // a.Camel::m_Age = 28
    cout<<"a.m_Age = "<<a.m_Age<<endl; // a.m_Age = 28

    return 0;
}
```

总结:

- 菱形继承带来的主要问题是子类继承多分相同的数据, 导致资源浪费以及毫无意义
- 利用虚继承可以解决菱形继承问题

4.7 多态

4.7.1 多态的基本概念

多态是C++面向对象三大特性之一

多态分为两类

- 静态多态: 函数重载 和 运算符重载属于静态多态, 复用函数名
- 动态多态: 派生类和虚函数实现运行时多态

静态多态和动态多态区别:

- 静态多态的函数地址早绑定 - 编译阶段确定函数地址
- 动态多态的函数地址晚绑定 - 运行阶段确定函数地址

如果函数地址在编译阶段就能确定, 那么静态联编 如果函数地址在运行阶段才能确定, 就是动态联编

下面通过案例进行讲解多态

```cpp
// 多态满足条件:
// 1、有继承关系
// 2、子类重写父类中的虚函数(父类必须有virtual关键字，重写除了函数名外参数也得完全相同)
// 多态使用:
// 父类指针或引用指向子类对象
class Animal{
    public:
        // 函数前面加上virtual关键字变成虚函数，那么编译器在编译的时候就不函数调用
        // 所以在运行时才进行地址绑定
        virtual void speak(){
            cout<<"动物在说话"<<endl;
        }
};

class Cat: public Animal{
    public:
        void speak(){
            cout<<"小猫在说话"<<endl;
        }
};

class Dog: public Animal{
    public:
        void speak(){
            cout<<"小狗在说话"<<endl;
        }
};

// 父类引用可以接受子类对象
void DoSpeak(Animal &animal){
    animal.speak();
}
int main(){

    // 如果父类中speak()函数不是虚函数,则在编译时就进行了地址绑定
    // 从而此处的输出均为 动物在说话
    Cat cat;
    DoSpeak(cat); // 小猫在说话

    Dog dog;
    DoSpeak(dog); // 小狗在说话

    cout<<"sizeof Animal = "<<sizeof(Animal)<<endl;
    // sizeof Animal = 4  空类大小为1，但加了virtual后变为4了
    // 4为一个四字节指针(vfptr)的大小 virtual function ptr

    return 0;
}
```

总结:

多态满足条件

- 有继承关系
- 子类重写父类中的虚函数

多态使用条件

- 父类指针或引用指向子类对象

重写: 函数返回值类型 函数名 参数列表 完全一致

4.7.2 多态案例一-计算器类

案例描述:

分别利用普通写法和多态技术, 设计实现两个操作数进行运算的计算器类

多态的优点:

- 代码组织结构清晰
- 可读性强
- 利于前期和后期的扩展以及维护

示例:

```
// 抽象计算器类
// 多态优点: 代码组织结构清晰, 可读性强, 利于前期和后期的扩展以及维护
class AbstractCalculator{
    public:
        int m_Num1;
        int m_Num2;
        virtual int getResult(){
            return 0;
        }
};

// 加法计算器
class AddCalculator: public AbstractCalculator{
    public:
        int getResult(){
            return m_Num1 + m_Num2;
        }
};
// 减法计算器
class SubCalculator: public AbstractCalculator{
    public:
        int getResult(){
            return m_Num1 - m_Num2;
        }
};

// 乘法计算器
class MulCalculator: public AbstractCalculator{
    public:
        int getResult(){
            return m_Num1 * m_Num2;
        }
```

```
};

int main(){

    // 创建加法计算器
    AbstractCalculator *abc = new AddCalculator;
    abc->m_Num1 = 10;
    abc->m_Num2 = 10;
    cout<<abc->m_Num1<<" + "<<abc->m_Num2<<" = "<<abc->getResult()<<endl;
    delete abc;  // 用完了记得销毁

    // 创建减法计算器
    abc = new SubCalculator;
    abc->m_Num1 = 10;
    abc->m_Num2 = 10;
    cout<<abc->m_Num1<<" - "<<abc->m_Num2<<" = "<<abc->getResult()<<endl;
    delete abc;

    // 创建乘法计算器
    abc = new MulCalculator;
    abc->m_Num1 = 10;
    abc->m_Num2 = 10;
    cout<<abc->m_Num1<<" * "<<abc->m_Num2<<" = "<<abc->getResult()<<endl;
    delete abc;

    return 0;
}
```

> 总结: C++开发提倡利用多态设计程序架构, 因为多态优点很多

4.7.3 纯虚函数和抽象类

在多态中, 通常父类中虚函数的实现是毫无意义的, 主要都是调用子类重写的内容

因此可以将虚函数改为**纯虚函数**

纯虚函数语法: `virtual 返回值类型 函数名 (参数列表) = 0 ;`

当类中有了纯虚函数, 这个类也称为抽象类

抽象类特点:

- 无法实例化对象
- 子类必须重写抽象类中的纯虚函数, 否则也属于抽象类

示例:

```
class Base{
    public:
        // 纯虚函数
        // 类中只要有一个纯虚函数就称为抽象类
        // 抽象类无法实例化对象
        // 子类必须重写父类中的纯虚函数，否则也属于抽象类

        virtual void func() = 0;
```

```
};

class Son: public Base{
    public:
        virtual void func(){
            cout<<"func调用"<<endl;
        };
};

int main(){

    Base *base = NULL;
    // base = new Base; // 错误，抽象类无法实例化对象
    base = new Son;
    base->func();
    delete base;// 记得销毁

    return 0;
}
```

4.7.4 多态案例二-制作饮品

案例描述:

制作饮品的大致流程为: 煮水 - 冲泡 - 倒入杯中 - 加入辅料

利用多态技术实现本案例, 提供抽象制作饮品基类, 提供子类制作咖啡和茶叶

冲咖啡

冲茶叶

示例:

```
// 抽象制作饮品
class AbstractDrinking{
    public:
        // 烧水
        virtual void Boil() = 0;
```

```
        // 冲泡
        virtual void Brew() = 0;
        // 倒入杯中
        virtual void PourInCup() = 0;
        // 加入辅料
        virtual void PutSomething() = 0;
        // 规定流程
        void MakeDrink(){
            Boil();
            Brew();
            PourInCup();
            PutSomething();
        }
};

// 制作咖啡
class Coffee: public AbstractDrinking{
    public:
        // 烧水
        virtual void Boil(){
            cout<<"煮农夫山泉!"<<endl;
        }
        // 冲泡
        virtual void Brew(){
            cout<<"冲泡咖啡!"<<endl;
        }
        // 倒入杯中
        virtual void PourInCup(){
            cout<<"将咖啡倒入杯中!"<<endl;
        }
        // 加入辅料
        virtual void PutSomething(){
            cout<<"加入牛奶!"<<endl;
        }
};

// 制作茶水
class Tea: public AbstractDrinking{
    public:
        // 烧水
        virtual void Boil(){
            cout<<"煮自来水!"<<endl;
        }
        // 冲泡
        virtual void Brew(){
            cout<<"冲泡茶叶!"<<endl;
        }
        // 倒入杯中
        virtual void PourInCup(){
            cout<<"将茶水倒入杯中!"<<endl;
        }
        // 加入辅料

        virtual void PutSomething(){
```

```
            cout<<"加入枸杞!"<<endl;
        }
};

// 业务函数
void DoWork(AbstractDrinking* drink){
    drink->MakeDrink();
    delete drink;
}

int main(){

    DoWork(new Coffee);
    cout<<"--------------"<<endl;
    DoWork(new Tea);

    return 0;
}
```

4.7.5 虚析构和纯虚析构

多态使用时, 如果子类中有属性开辟到堆区, 那么父类指针在释放时无法调用到子类的析构代码

解决方式: 将父类中的析构函数改为**虚析构**或者**纯虚析构**

虚析构和纯虚析构共性:

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

虚析构和纯虚析构区别:

- 如果是纯虚析构, 该类属于抽象类, 无法实例化对象

虚析构语法:

`virtual ~类名(){}`

纯虚析构语法:

`virtual ~类名() = 0;`

`类名::~类名(){}`

示例:

```
class Animal {
    public:
        Animal(){
            cout<<"Animal 构造函数调用!"<<endl;
        }
        virtual void Speak() = 0;

        // 析构函数加上virtual关键字, 变成虚析构函数
        // virtual ~Animal(){
        //  cout<<"Animal虚析构函数调用!"<<endl;
```

```
        // }
        // 纯虚析构
        virtual ~Animal() = 0;
};

Animal::~Animal(){
    cout<<"Animal 纯虚析构函数调用!"<<endl;
}

// 和包含普通纯虚函数的类一样，包含了纯虚析构函数的类也是一个抽象类。不能够被实例化。

class Cat: public Animal{
    public:
        string *m_Name;
        Cat(string name){
            cout<<"Cat构造函数调用!"<<endl;
            m_Name = new string(name);
        }
        virtual void Speak(){
            cout<<*m_Name<< "小猫在说话!"<<endl;
        }
        ~Cat(){
            cout<<"Cat析构函数调用!"<<endl;
            if (this->m_Name != NULL){
                delete m_Name;
                m_Name = NULL;
            }
        }
};

int main(){

    Animal *animal = new Cat("Tom");
    animal->Speak();

    // 通过父类指针去释放，会导致子类对象可能清理不干净，造成内存泄漏
    // 怎么解决？给基类增加一个虚析构函数
    // 虚析构函数就是用来解决通过父类指针释放子类对象
    delete animal;

    return 0;
}
```

总结:

1. 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象
2. 如果子类中没有堆区数据, 可以不写为虚析构或纯虚析构
3. 拥有纯虚析构函数的类也属于抽象类#### 4.7.6 多态案例三-电脑组装

4.7.6 多态案例三-组装电脑

案例描述:

电脑主要组成部件为 CPU(用于计算), 显卡(用于显示), 内存条(用于存储)

将每个零件封装出抽象基类, 并且提供不同的厂商生产不同的零件, 例如Intel厂商和Amd厂商

创建电脑类提供让电脑工作的函数, 并且调用每个零件工作的接口

测试时组装三台不同的电脑进行工作

示例:

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

// 抽象CPU类
class CPU{
    public:
        // 抽象的计算函数
        virtual void calculate() = 0;
};

// 抽象显卡类
class VideoCard{
    public:
        // 抽象的显示函数
        virtual void display() = 0;
};

// 抽象内存条类
class Memory{
    public:
        // 抽象的存储函数
        virtual void storage() = 0;
};

// 电脑类
class Computer{
    public:
        Computer(CPU *cpu, VideoCard *vc, Memory *mem){
            m_cpu = cpu;
            m_vc = vc;
            m_mem = mem;
        }

        // 提供工作的函数
        void work(){
            // 让零件工作起来，调用接口
            m_cpu->calculate();
            m_vc->display();
            m_mem->storage();
        }

        // 提供析构函数 释放3个电脑零件
        ~Computer(){

            // 释放CPU零件
```

```
            if (m_cpu != NULL){
                delete m_cpu;
                m_cpu = NULL;
            }
            // 释放显卡零件
            if (m_vc != NULL){
                delete m_vc;
                m_vc = NULL;
            }
            // 释放内存条零件
            if (m_mem != NULL){
                delete m_mem;
                m_mem = NULL;
            }
        }

    private:
        CPU *m_cpu; // CPU的零件指针
        VideoCard *m_vc; // 显卡零件指针
        Memory *m_mem; // 内存条零件指针
};

// 具体厂商
// Intel厂商
class IntelCPU: public CPU{
    public:
        virtual void calculate(){
            cout<<"Intel的CPU开始计算了!"<<endl;
        }
};

class IntelVideoCard: public VideoCard{
    public:
        virtual void display(){
            cout<<"Intel的显卡开始显示了!"<<endl;
        }
};

class IntelMemory: public Memory{
    public:
        virtual void storage(){
            cout<<"Intel的内存条开始存储了!"<<endl;
        }
};

// Amd厂商
class AmdCPU: public CPU{
    public:
        virtual void calculate(){
            cout<<"Amd的CPU开始计算了!"<<endl;
        }
};
```

```
class AmdVideoCard: public VideoCard{
    public:
        virtual void display(){
            cout<<"Amd的显卡开始显示了!"<<endl;
        }
};

class AmdMemory : public Memory{
    public:
        virtual void storage(){
            cout<<"Amd的内存条开始存储了!"<<endl;
        }
};


int main(){

    cout<<"第一台电脑开始工作: "<<endl;
    // 电脑三个intel零件指针
    CPU *intelCpu = new IntelCPU;
    VideoCard *intelCard = new IntelVideoCard;
    Memory *intelMem = new IntelMemory;
    // 创建第一台电脑
    Computer *computer1 = new Computer(intelCpu, intelCard, intelMem);
    computer1->work();
    delete computer1; // Computer类的析构函数中将三个零件在堆的地址释放了
    cout<<"-----------------------"<<endl;

    cout<<"第二台电脑开始工作: "<<endl;
    // 第二台电脑组装
    Computer *computer2 = new Computer(new AmdCPU, new AmdVideoCard, new AmdMemory);;
    computer2->work();
    delete computer2;
    cout<<"-----------------------"<<endl;

    cout<<"第三台电脑开始工作: "<<endl;
    // 第三台电脑组装
    Computer *computer3 = new Computer(new AmdCPU, new IntelVideoCard, new AmdMemory);;
    computer3->work();
    delete computer3;

    return 0;

}
```

5 文件操作

程序运行时产生的数据都属于临时数据, 程序一旦运行结束都会被释放

通过文件可以将数据 **持久化**

C++中对文件操作需要包含头文件 < fstream >

文件类型分为两种:

1. **文本文件** - 文件以文本的 **ASCII码** 形式存储在计算机中
2. **二进制文件** - 文件以文本的 **二进制** 形式存储在计算机中, 用户一般不能直接读懂它们

操作文件的三大类:

1. ofstream： 写操作
2. ifstream： 读操作
3. fstream ： 读写操作

5.1文本文件

5.1.1写文件

写文件步骤如下:

1. 包含头文件

   # include

2. 创建流对象

   ofstream ofs;

3. 打开文件

   ofs.open("文件路径", 打开方式);

4. 写数据

   ofs<<"写入的数据";

5. 关闭文件

   ofs.close(); 文件打开方式:

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置: 文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在, 先删除, 再创建
ios::binary	二进制方式

注意: 文件打开方式可以配合使用, 利用 | 操作符

例如: 用二进制方式写文件

```
        ios::binary | ios:: out
```

示例:

```
// 1、包含头文件
#include<iostream>
#include<fstream>
using namespace std;
int main(){

    // 2、创建流对象
    ofstream ofs;
    // 3、打开文件
    ofs.open("test.txt", ios::out);
    // 4、写数据
    ofs<<"姓名：张三"<<endl;
    ofs<<"性别：男"<<endl;
    ofs<<"年龄：18"<<endl;
    // 5、关闭文件
    ofs.close();

    return 0;
}
```

总结:

- 文件操作必须包含头文件 fstream
- 读文件可以利用 ofstream, 或者 fstream 类
- 打开文件时候需要指定操作文件的路径, 以及打开方式
- 利用<<可以向文件中写数据
- 操作完毕, 要关闭文件

5.1.2读文件

读文件与写文件步骤相似, 但是读取方式相对于比较多

读文件步骤如下:

1. 包含头文件

   # include

2. 创建流对象

   ifstream ifs;

3. 打开文件并判断文件是否打开成功

   ifs.open("文件路径", 打开方式);

4. 读数据

   四种方式读取

5. 关闭文件

   ifs.close();

示例:

```
#include<iostream>
#include<fstream>
using namespace std;
#include<string>
int main(){

    ifstream ifs;

    ifs.open("test.txt", ios::in);
    if (!ifs.is_open()){
        cout<<"文件打开失败"<<endl;
        return;
    }

    // 第一种方式
    // char buf[1024] = {0}; // 创建一个字符数组，初始化全为0，ascii码中0表示空字符即null
    // while(ifs >> buf){ // 一个字符一个字符读，读到文件结尾
    //  cout<<buf<<endl;
    // }

    // 第二种
    // char buf[1024] = {0};
    // while(ifs.getline(buf, sizeof(buf))){ // 一行一行读到buf中，每一行最大读取sizeof(buf)个字符
    //  cout<<buf<<endl;
    // }

    第三种
    string st;
    while(getline(ifs, st)){ // 一行一行读到字符串中
        cout<<st<<endl;
    }

    // 第四种，不推荐用
    // char c;
    // while((c = ifs.get()) != EOF){ // 将每个字符读到字符c中,直到读至文件尾EOF
    //  cout<<c;
    // }

    ifs.close();

    return 0;
}
```

总结:

- 读文件可以利用 ifstream, 或者 fstream 类
- 利用is_open函数可以判断文件是否打开成功
- close 关闭文件

5.2 二进制文件

以二进制的方式对文件进行读写操作

打开方式要指定为 ios::binary

5.2.1 写文件

二进制方式写文件主要利用流对象调用成员函数 write

函数原型 : `ostream& write(const char *buffer, int len);`

参数解释: 字符指针buffer指向内存中一段存储空间, len是写的字节数

示例:

```
// 1、包含头文件
#include<iostream>
#include<fstream>
using namespace std;

class Person{
    public:
        char m_Name[64];
        int m_Age;
};

// 二进制文件  写文件
int main(){

    // // 2、创建输出流对象
    // ofstream ofs;
    // // 3、打开文件
    // ofs.open("person.txt", ios::out | ios::binary);

    // 步骤2和3可以合并
    ofstream ofs("person.txt", ios::out | ios::binary);

    Person p = {"张三"  , 18};

    // 4、写文件
    ofs.write((const char *)&p, sizeof(p));

    // 5、关闭文件
    ofs.close();

    return 0;
}
```

> 总结: 文件输出流对象 可以通过write函数, 以二进制方式写数据

5.2.2 读文件

二进制方式读文件主要利用流对象调用成员函数 read

函数原型: `istream& read(char *buffer, int len);`

参数解释: 字符指针buffer指向内存中一段存储空间, len是读的字节数

示例:

```
#include<iostream>
#include<fstream>
using namespace std;

class Person{
    public:
        char m_Name[64];
        int m_Age;
};

int main(){

    // 创建流对象并打开
    ifstream ifs("person.txt", ios::in | ios::binary);
    if(!ifs.is_open()){
        cout<<"文件打开失败"<<endl;
    }

    Person p;
    ifs.read((char *)&p, sizeof(p)); // 取到p的地址后强转为char *

    cout<<"姓名:  "<<p.m_Name<<" 年龄:  "<<p.m_Age<<endl;

    return 0;
}
```

> 总结: 文件输入流对象 可以通过read函数, 以二进制方式读数据