

C++基础课程大纲

1、c 到 c++（课时-5小时）

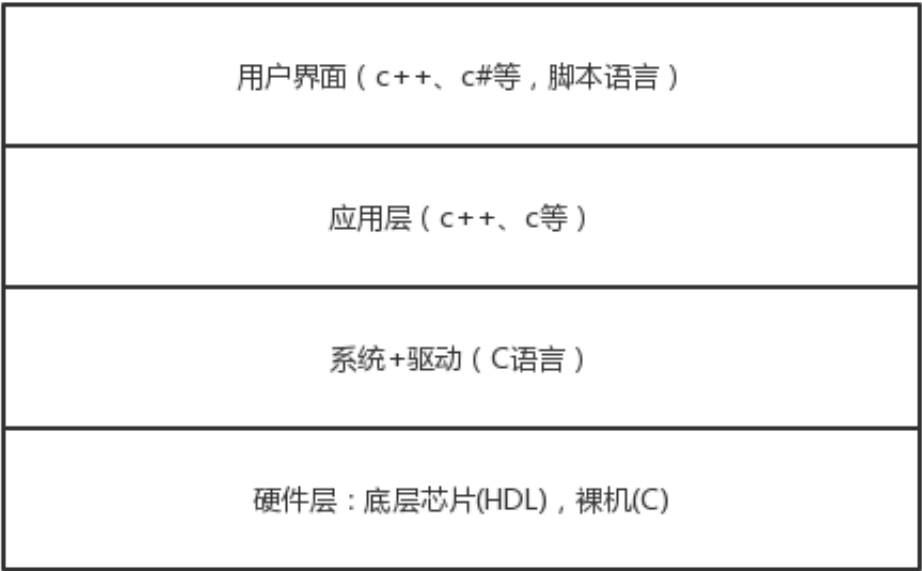
1.1 学习C++的意义

课程举例：

第一块：C++怎么来的

1.学习C语言够吗？一种语言能支持我们的整个职业生涯吗？

首先看当今产品架构



当代软件新特性

- 1、多语言
- 2、适应甲方爸爸随时变更的需求
- 3、可移植性要好
- 4、便于维护
- 5、便于随时更新

举个实际例子：下位机：板卡用于图像数据的获取和前处理（降噪，滤波和压缩等操作），得到图像原始数据，通过网络或者usb等协议将原始数据传给上位机（电脑或者服务器），上位机对图像数据进行高等处理（机器学习相关算法），处理完成最后在电脑上写一个用户界面软件用来显示原始图像和处理后的图像

2.最早的C语言被开发出来是用来解决科学计算问题

科学计算是一步一步进行求解结果的，所以C语言天生适合面向过程的编程思想，而随着科技进步（不论软件还是硬件），计算机走进生活中每一个领域，解决问题越来越复杂，面向过程解决复杂问题不利于代码维护和移植。

人们在想可不可以把生活中的解决问题的方法对应到程序中，比如由现实世界简历软件模型，将现实世界中存在的实物直接对应到软件设计中。

这样做的好处：

1、直接分析用户需求个体

2、代码直接描述现实世界，现实世界中如何解决一个问题，代码中就如何解决

3、这样有利于移植和协同工作

4、更适应多变的用户需求

这样就产生了面向对象的编程思想

3.linus说学习c++一无是处

生活中，能成为linus那样的神人，几率太低了，so，我们还是好好多学习几门语言来支撑我们职业生涯

第二块：由新思想（面向对象）+ C能产生什么样的效果

因为C语言没有语法特性支持面向对象，所以人们就像C+面向对象，这样就产生生了两种新的语言

C++， object-c

我们这门课程重点讲解c++，不讲解object-c

产生了新语言又带来了新的问题：以前写用C的软件都要在重写一遍吗？当然不是，所以要求C++天生要完全支持C中所有的特性

那么C++在C的基础之上到底增加了什么呢？

最显著的特点：

1、+：数据类型的强化，C++是一门强类型的语言，必须加强对类型的理解

2、+：对面向对象的支持。有了类和对象的概念

3、兼容C语言的执行效率

举例：

C->C++

1、编译器从gcc变成了g++

```
gcc 源文件.c  
g++ 源文件.cpp
```

2、变量定义更加灵活：c语言，用之前声明，c++在用的时候声明就行

代码演示：

```

//c语言
#include <stdio.h>
int main() {
    int i = 0, j = 0;
    for(i = 0; i < 5; ++i) {
        for(j = 0; j < 3; ++j) {
            printf("hello world\n");
        }
    }
    return 0;
}

//c++
#include <iostream>
using std::cin;
using std::cout;
int main() {
    for(int i = 0; i < 5; ++i) {
        for(int j = 0; j < 3; ++j) {
            cout << "hello world" << endl;
        }
    }
    return 0;
}

```

3、C语言可以有多个全局变量名，C++中不允许声明多个同名变量

4、struct的升级，C语言中struct只是不同变量的集合，C++中的struct是类的定义

代码演示：

```

//C语言-gcc
#include<stdio.h>
struct A {
    int a;
    char b;
    double c;
    // 不能在struct中定义实现函数
    /* // erro
    int get() {
        return a;
    }
    */
};

int main() {

    return 0;
}

// C++ -g++
#include <iostream>

```

```
using namespace std;
struct A {
    int a;
    int get() { // no erro
        return a;
    }
};
int main() {

    return 0;
}
```

5、C语言中允许默认类型

代码演示

```
//c语言，允许默认类型
#include <stdio.h>
f() { // gcc no erro
    return 5;
}
void g() {
    return;
}
void fun(void) {

}
int main() {
    g(1,2,3,45); // gcc no erro
    func(); // void 是没有参数，()空是可以有无限个参数
    return 0;
}
// c++
#include <iostream>
using namespace std;
f() { // g++ erro

}
void g(void) { // void g(void ) 和 void g() 一样，表示没有参数

}
int main() {

    return;
}
```

第三块：学习C++意义 1、C++是C语言的超集

2、学习c++同时可以掌握更多软件设计方法，同样可以快速上手其他语言

3、C++支持四种变成范式

4、找工作更容易

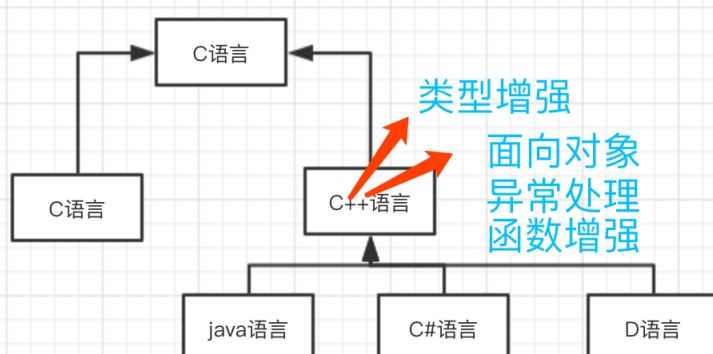
C 到 C++ 升级

1、继承了C语言的所有特性

2、C++提供更多的语法和特性

3、兼容C的开发效率

5、对后续语言的学习有很大帮助



1.2 c语言和c++语言中一些语法升级

1、const

引导：C语言中有真整的常量？

正确答案：enum变量

引导同学们说const

cosnt 在c语言中只是给变量增加只读属性，**read-only**变量不能做左值，虽然被**const**修饰，但是还是变量，我们通过指针改变**const**的值

```
// 通过指针修改const变量的值 -c语言
#include <stdio.h>
int main() {
    const int a = 0;
    int *p = (int *) &a;
    *p = 15; // 通过指针去修改a的值，成功，说明a不是一个真的常量，const 修饰后，只是一个
read-only变量
    printf("%d\n", a);
    return 0;
}
```

在C语言const特性

1、const修饰的变量，只是这个变量具有只读属性，不能做左值，本质还是变量，可以通过指针去修改这个值

2、const修饰的局部变量，这个变量还是在栈空间分配存储空间

3、const修饰全局变量，空间分配在静态存储区

4、const只在编译器有效，它只是告诉编译器该变量不能出现在赋值符号的左侧

结论：

const在C语言中修饰的变量不是真的常量，只是只读变量，那么const在c++表现什么行为呢？

C++中const的行为

代码：

```
//c++ 尝试const修饰的值是否可以被改变
#include <iostream>
using namespace std;
int main() {
    const int a;
    int *p = (int *) &a; // 看看啊的值能不能被改变？
    *p = 15;
    cout << a << endl; // 结果是能改变
    return 0;
}
// const 和 define区别
// 分别用gcc 和 g++ 编译
#include <iostream>
using namespace std;
void f() {
    #define a = 3
    const int b = 4;
}
void g() {
    cout << "a = " << a << endl;
}
```

```
int main() {
    const int A = 1;
    const int B = 2;
    int array[A + B] = {0};
    for(int i = 0; i < A + B; ++i) {
        cout << i << endl;
    }
    f();
    g();
    return 0;
}
```

符号表

	变量名	值
	
const	value	10
	

const在c++中的行为

- 1、当编译到`int *p = (int *) &a;`时，编译器还是会为p分配空间。
- 2、当运行到`cout << a << endl;`时，回去符号表中去找a的值，而不是去找p指向位置的值

c++中这种行为，只是为了兼容C语言

在c++中：const修饰的变量，用取地址符号（&）时，编译器还会分配空间，但是在使用时候，不会使用该空间的值，而是去符号表中去寻找符号表中同名的变量

2、bool类型和引用

bool类型和引用（&）

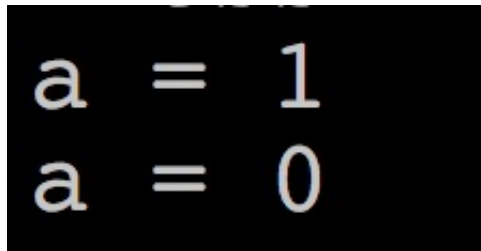
2.1 bool基本概念

bool类型是对c重的int类型的补充，在c语言中，我们用0和1表示逻辑真假。其中有两个关键字：false和true，其中0就是false，非0就是true，负数也是true

代码演示-1.cpp

```
#include <iostream>
using std::endl;
using std::cout;
int main() {
    bool a = true;
    a = -1;
    cout << "a = " << a << endl;
    a = 0;
    cout << "a = " << a << endl;
    return 0;
}
```

编译结果



从结果中可以看出，bool类型只有两个值，0和1；

2.2 bool类型的本质

首先用bool类型参与运算得到的结果是什么呢？

代码演示-2.cpp

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
    bool a, b;
    a = true;
    b = true;
    cout << "sizeof(bool) = " << sizeof(bool) << endl;
    a = a + 1;
    cout << "a = " << a << endl;
    a = a + b;
    cout << "a = " << a << endl;
    b = 0;
    cout << "b = " << b << endl;
    b -= 1;
    cout << "b = " << b << endl;
    return 0;
}
```

编译结果：


```
Press ENTER or type command to continue
sizeof(bool) = 1
a = 1
a = 1
b = 0
b = 1
```

不论怎么运算，bool类型就是0 和 1，bool类型大小为1byte，编译器是怎么对待bool类型的呢？bool类型的本质就是int型，最后编译器会把非0值转成1

引用

引用：通常所说的引用是指“左值引用”，也就是给一个变量起一个别名（小名），操作引用等于操作变量本身。

用法：

type name1;

type &name2 = name1;

使用规则：

- 1、变量引用必须初始化
- 2、type类型必须相同
- 3、不能给字面常量起别名，想给常量起别名，只能是const类型，别的都不行，形如：**const int &a = 1;**
- 4、引用做为参数，不用初始化，初始化过程发生在传参过程中
- 5、不可以返回局部变量的引用
- 6、const 修饰引用，**const type &name2 = name1;** 只是给name2增加了只读属性

代码演示-3.cpp

```
#include <iostream>
using std::endl;
using std::cout;
using std::cin;
int& demo1() {
    int a = 1;
    cout << "a = " << a << endl;
    return a; // 不能返回局部变量的引用
}
void demo2(int &a, int &b) { // 传引用
    int temp;
    temp = a;
    a = b;
    b = temp;
    return ;
}
```

```

}
//demo3,demo4表示const修饰引用，只是给变量加上了只读属性，不是一个真正的变量
void demo3() {
    int a = 5;
    const int &b = a;
    int *p = const_cast<int*>(&b);
    *p = 6;
    cout << "a = " << a << " b = " << b << endl;
    return ;
}

void demo4() {
    const int &a = 1;
    int *p = const_cast<int*>(&a);
    *p = 6;
    cout << "a = " << a << endl;
    return ;
}
// static 引用
void demo5() {
    static int &a = 1; // 报错，非const类型左值不能绑定一个常量
    return ;
}
int main() {
    int a = 1;
    int &b = a; //操作b等于操作a
    b = 3;
    cout << "a = " << a << " b = " << b << endl;
    //int &c = 1; //报错，不能给字面常量起别名
    int d = demo1();
    int e = 1, f = 2;
    demo2(e, f);
    cout << "e = " << e << " f = " << f << endl;
    demo3();
    demo4();
    demo5();
    return 0;
}

```

3、函数的升级

函数参数的默认值：函数参数可以传默认值

分别用gcc和g++去编译下面的代码

```

//c-gcc c代码
#include <stdio.h>
void f(int i = 0) {
    printf("i = %d\n",i);
    return ;
}

```

```

}
int main() {
    f();
    return 0;
}
// c++ -g++ c++代码
#include <iostream>
using namespace std;
void f(int i = 0) {
    cout << "i = " << i << endl;
    return ;
}
int main() {
    f();
    return 0;
}

```

c语言不允许函数参数有默认值，c++允许函数有默认值

C does not support default arguments

c++中允许函数参数有默认值，规则如下：

默认值设计规则：从右向左考虑

调用：从左向右赋值

```

// g++
#include <iostream>
using namespace std;
int add(int x, int y = 1, int z = 3) {
    return x + y + z;
}
int main() {
    add(0); // x = 0, y = 1, z = 3
    add(1, 2); // x = 1, y = 2, z = 3
    add(1, 2, 3); // x = 1, y = 2, z = 3
    return 0;
}

```

函数重载：同意作用域中的几个函数，满足：函数名相同，函数的参数不同，就构成函数重载关系，与返回值没有关系，切记，与返回值无关！！！！

分别用gcc和g++编译下面代码

```

//c gcc
#include <stdio.h>
/*c语言是编译不过的，改成f1, f2, f3即可*/
void f() {

```

```

    printf("hello world");
    return ;
}
void f(int i) {
    printf("i = %d\n", i);
    return ;
}
void f(int i, int j) {
    printf("i = %d\n", i);
    printf("j = %d\n", j);
    return ;
}
int main() {
    // c 语言不允许定义同名函数
    f();
    f(4);
    f(4, 5);
    return 0;
}
// c++ g++
#include <iostream>
using namespace std;
void f() {
    cout << "hello world" << endl;
    return ;
}
void f(int i) {
    cout << "i = " << i << endl;
    return ;
}
void f(int i, int j) {
    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    return ;
}
int main() {
    /*函数重载*/
    f();
    f(4);
    f(4, 5);
    return 0;
}

```

函数重载的意义：同一个动作加不同名词，将这种语言引入程序语言中，在实际调用中，是根据参数的个数和参数的类型去调用的。

编译器调用重载函数的规则：

§1.所有同名函数都是备选函数

S2.尝试寻找可执行函数，此时进行函数参数的匹配，切记不要出现调用二义性！！

重载函数的本质是：不同函数

```
//g++
#include <iostream>
using namespace std;
void f() {
    cout << "hello world" << endl;
    return ;
}
void f(int i) {
    cout << "i = " << i << endl;
    return ;
}
void f(int i, int j) {
    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    return ;
}
void (*ptr1)();
void (*ptr2)(int);
void (*ptr3)(int, int);
int main() {
    ptr1 = f;
    ptr2 = f;
    ptr3 = f;
    cout << (void *)ptr1 << endl << (void *)ptr2 << endl << (void *)ptr3 <<
endl;
    return 0;
}
```

函数重载+ 函数指针规则：

- 1、参数个数和参数类型必须相同
- 2、返回值类型也必须相同（体现c++是强类型语言，赋值符号两侧都是同类型）
- 3、重载函数不能通过函数名直接得到函数地址

函数参数默认值+函数的重载

切记二义性！！！！

看如下代码

```
// g++
#include <iostream>
using namespace std;
int add(int a, int b, int c = 0) {
    return a + b + c;
}
int add(int a, int b) {
    return a + b;
}
int main() {
    int c = add(1, 2); //erro 调用二义性，人都不知道调用哪个函数，机器更不知道
    return 0;
}
```

4、c++中新成员

4.1 动态内存申请：new

用法：

```
//单个变量的申请
type *name = new type; // type *name = new type(0); 用0初始化
delete name; //归还空间
//连续的空间申请
type *name = new type[len];
delete[] name; //归还申请的全部空间
```

new和delete成对使用

new是基于类型进行申请的，malloc 是基于字节申请的。

new和malloc区别

- 1、new是关键字，malloc是库函数（如果有的编译器不支持库文件，则用不了malloc）
- 2、new是按类型分配空间，malloc是根据字节大小申请
- 3、new申请单个变量的时候，可以初始化，malloc不具备初始化特性

区别：

```
// () 和 []区别
type *name = new type(0); // 单个初始化
type *name = new type[N]; // 长度为N的连续空间
```

4.2 命名空间：namespace

C语言只有一个作用域，而且C语言允许定义多个全局同名变量，这样就是有问题的。

namespace是将全局变量划分成多个区域，在不同的区域就可以存在多个同名变量。

用法：一般用法：

namespace name { int value; }

Namespace 嵌套namespace。

namespace name1 { namespace name2 { }}

用法：

using namespace name;：使用该命名空间中的全部东西

using name::value;：使用name命名空间中的一个变量 value

命名空间解决的问题就是：全局变量命名冲突问题

```
#include <iostream>
using namespace std; // 使用std命名空间
namespace Name1 {
    int value;
}
namespace Name2 {
    int value;
    namespace Name3 {
        int value;
    }
}
using namespace Name1;
using Name2::value;
int main() {
    Name1::value = 10;
    cout << Name::value << endl;
    Name2::Name3::value = 7;
    cout << Name2::Name3::value << endl;
    return 0;
}
```

2、类和对象

2.1 对象的声明和使用

1、看到类型，想到什么

1、该类型变量的在内存中大小

2、加在该类型上的运算

代码演示

```
//对类型上的运算
#include <iostream>
using namespace std;
int main() {
    // int 类型可以四则运算，布尔运算
    //double 类型可以四则运算，但是不能布尔运算
    return 0;
}
```

2、类的定义，对象的声明

举例：

归类：



, , , , , , , , , , , , 

答案：

动物：, , 

昆虫：, 

钱：, 

交通工具：, 

人：, 

国家：, 

对象：类的实例化

类：对象的抽象

解释下什么叫类的实例化，什么叫对象的抽象

类的定义和实现

对象的定义

代码演示：

```
//类和对象的定义和声明
#include <iostream>
using namespace std;
class Class_Name {}; // 类的声明和定义 关键字 公式：class + 类名 {};
int main() {
    Class_Name c; //对象的定义：类名 + 变量名;

    return 0;
}
```

2.2、类的封装特性

首先：对类的理解，可以理解成新定义的一个类型

类中又类属性和类方法，类属性就是我们通常说的变量，类方法就是我们说的函数

封装就是将一些信息隐藏起来，不让外界去访问这些信息

用手机去举例

使用手机的人：不需要知道手机怎么去开发

做手机的人：一定要知道手机软件怎么做，硬件怎么做

类也是一样，当我们去使用一个类的时候，我们不许知道类具体的实现（一种封装表现）

当我我们去开发一个类的时候，我们一定要知道他里面的细节。

男孩女孩都有不想让人知道小秘密

男孩：不想让人知道他的工资

女孩：不想让人知道他的年龄和体重

面向对象是将生活中事物直接对应到程序中。

那么封装怎么去实现的呢？

封装：C++有三个关键字。用来决定类外部是否可以访问类内部的方法和属性

关键字	性质
public	公有：外部可以访问
protected	被保护：外部不可以访问，但是子类可以访问
private	私有：外部和子类都不可以访问

访问级别是相随类外的，类本身是一个作用域，类外部对类内的数据或者方法进行访问的时候，需要看访问属性，但是在类的内部，是没有访问属性的概念的

类成员的作用域：

- 1、类作用域，类的成员方法可以访问类中所有成员，外部无法直接访问类中的私有和被访问属性的成员
- 2、类中的成员函数可以访问类的成员函数
- 3、类外部只能访问公有的成员
- 4、类的作用域与访问级别没有关系

2.3 const 成员属性和成员方法

const成员属性：在声明属性的时候，在属性前增加const。

const成员方法：在正常的成员属性后面用const去修饰。const方法内部的变量是不可以改变的（代码演示）

```

class Name {
    public:
        const type var_name; // const 属性
    public:
        type func_name() const; // const 方法
};

```

const属性使用：当类属性被const修饰后，该属性只是具有了只读属性，而不是真正的常量（代码演示）。

```

//代码演示const属性只具有只读性质
#include <iostream>
using namespace std;
class Test {
    public:
        const int a;
        Test() : a(0) {

        }
        int getA() {
            return a;
        }
};
int main() {
    Test t;
    cout << t.getA() << endl;
    int *p = const_cast<int *>(&t.a);
    *p = 10;
    cout << t.getA() << endl;
    return 0;
}

```

const 的引用：

我们知道，引用必须初始化，const 的引用如何去使用？

1、非const的引用只能用非const对象去初始化

2、const引用可以用**const对象初始化**，也可以用非const对象初始化

涉及到**const对象概念**：被const修饰的对象，内部属性只是具有只读属性，**const对象只能调用const方法**，普通成员既可以调用const方法，也可以调用普通成员方法

```

//代码演示：const成员方法(调用方式：普通成员，const成员)，const 方法内部变量不可以改变值
#include <iostream>
using namespace std;
class Test {
    private:
        int mi;

```

```

public:
    const int a;
    Test() : a(0) {
        mi = 0;
    }
    int getA() const{
        return a;
    }
    int getMi() const {
        cout << "const :" << endl;
        //mi = 1000; // erro
        return mi;
    }
    int getMi() {
        cout << "no-const:" << endl;
        mi = 100; // no erro
        return mi;
    }
};

int main() {
    Test t;
    const Test t1;
    cout << t.getMi() << endl;
    cout << t1.getMi() << endl;
    return 0;
}

```

```

//const 引用代码演示
#include <iostream>
using namespace std;
class Test {

};

int main() {
    Test tp;
    const Test tq;
    const Test &t1 = tp;
    const Test &t2 = tq;
    Test &t3 = tp;
    Test &t4 = tq; // erro
    return 0;
}

```

2.4 静态成员方法和静态成员属性

知识点的引出例子：

小要求：

- 1、统计某类对象数目
- 2、保证安全性（不能）
- 3、随时获取对象的数目

隐含：

- 1、不依赖对象去访问的静态成员函数和成员变量
- 2、必须保证静态成员的安全性（私有的）
- 3、方便快捷访问静态成员变量（静态成员方法）

静态成员属声明和使用：

```
#include <iostream>
using namespace std;
class Test {
    public:
        static int a; // 类中声明
        int getA() {
            return a;
        }
};
int Test::a = 10; // 类外初始化
int main() {
    Test t;
    cout << t.getA() << endl;
    return 0;
}
```

静态成员函数特性：

- 1、是类中的特殊的函数
- 2、静态成员函数隶属于整个类
- 3、可以直接通过类名直接访问公有的静态成员函数
- 4、可以通过对象名取访问公有的静态成员函数
- 5、静态成员函数不能直接访问成员变量
- 6、静态成员函数，没有this指针，所以不能直接访问类中成员变量

```
// 静态成员代码演示
#include <iostream>
using namespace std;
class Test {
    private:
        static int a;
    public:
        int getA() {
```

```

        return a;
    }
};
int Test::a = 10; //静态成员变量使用前必须初始化，初始化的方式是在类外
int main() {
    Test t;
    cout << t.getA() << endl;
    return 0;
}

```

静态成员函数与普通成员函数的区别：

	类方法	成员方法
所有对象共享	是	是
隐藏this指针	否	是
访问普通函数	否	是
访问静态成员函数	是	是
通过类名引用	是	否
通过对象名引用	是	是

2.5 构造函数

知识点引出：对象的属性初值是多少？

- 1、栈对象
- 2、堆对象
- 3、全局对象

编程验证：

```

//全局对象成员变量初值是0， 堆对象和栈对象初始值是随机值
#include <iostream>
using namespace std;
class Test {
    private:
        int a;
    public:
        int get() {
            return a;
        }
};
Test t1;
int main() {

```

```

    Test t2;
    cout << t1.get() << endl;
    cout << t2.get() << endl;
    Test *t3 = new Test;
    cout << t3->get() << endl;
    delete t3;
    return 0;
}

```

想要给对象中的属性一个初始值，怎么做呢？

1、提供一个成员方法 init();

缺点：1、每次初始化的时候都要立即调用

2、调用顺序不同，结果也不同

举例：先生孩子后结婚

2、有没有办法能不能自动调用？

构造函数：在对象创建的时候，自动调用这个函数。访问属性：public（目前）

函数：返回值 函数名 参数列表 函数体

构造函数是一种特殊的函数：

```

//构造函数
/*
1、函数没有返回值
2、函数名与类名相同
3、在对象创建时候，自动调用
*/
class Test {
public:
    Test() { //构造函数的声明和使用

    }
};

```

在c++中，对象的构造是通过构造函数完成的

一个空类中，编译器会默认提供一个空的构造函数，这个空的构造函数没有参数，是一个空的函数体

会使用构造函数，然后我们研究函数的重载：

1、带参数的构造函数

想一下为什么要有带参数的构造函数：不同对象->出生的状态不同

代码演示：

```

//带参数的构造函数

```

```

#include <iostream>
using namespace std;
class Test {
private:
    int i;
public:
    Test() {

    }

    Test(int v) {
        i = v;
    }

    int get() {
        return i;
    }
};

int main() {
    Test t;
    cout << "t.i = " << t.get() << endl;
    Test t1(10);
    cout << "t1.i = " << t1.get() << endl;
    return 0;
}

```

初始化和赋值区别：在C++中对象的初始化，要调用构造函数，赋值则不调用构造函数，初始化和赋值是两个不同的意义

对象的构造过程中遇上内存操作

怎么用一个对象去初始化另一个对象

2个特殊的构造函数：

1、无参数构造函数

2、拷贝构造函数：拷贝构造函数是一种特殊的带参数的构造函数，参数是：const 类名 &

```

//拷贝构造函数
#include <iostream>
using namespace std;
class Test {
private:
    int i;
public:
    Test(int v = 0) {
        i = v;
    }

    Test(const Test &obj) {
        i = obj.i;
    }

    int get() {

```

```

        return i;
    }
};

int main() {
    Test t(10);
    Test t1 = t;
    cout << "t1.i = " << t1.get() << endl;
    return 0;
}

```

经典面试问题：空类中真的是空的吗？

```

//空类中是不是真的空的吗？
#include <iostream>
using namespace std;
class Test {};
int main() {
    Test t; // 为什么能编译过？
    Test t1(t);
    return 0;
}

```

编译器会为空类提供一个无参构造函数，一个空的拷贝构造函数

若类中有构造函数时，编译器就不会为类中提供无参数构造函数

代码演示

```

//编译器对构造函数的行为
#include <iostream>
using namespace std;
class Test {
private:
    int i;
public:
    Test(int v) {
        i = v;
    }
};

int main() {
    Test t(10);
    //Test t; // erro
    Test t3(t);
    return 0;
}

```

以上代码表示：

1、编译器不会提供一个默认无参数的构造函数

2、编译器还会提供默认的拷贝构造函数，代码中Test t3(t)中表示

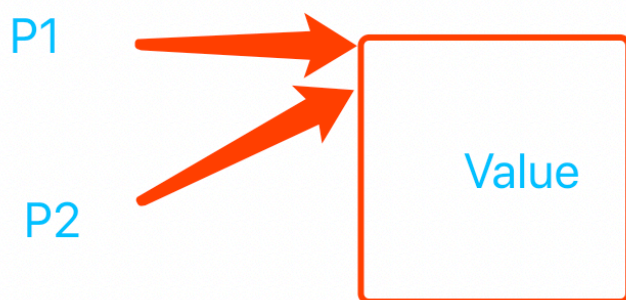
拷贝构造函数：

```
//c++拷贝构造函数
class Test {
public:
    Test(const Test &obj){}; //拷贝构造函数生命
};
```

提到拷贝：分深拷贝，浅拷贝

同时，编译器会为类提供一个默认的拷贝构造函数，这个默认的拷贝构造函数是浅拷贝

浅拷贝：拷贝之后，屋里状态相同——内存相同，值相同



深拷贝：拷贝之后，逻辑状态相同——内存不同，值相同



当涉及到内存相关操作时候，如果不执行深拷贝，在对象释放的时候，会出现同一个指针释放两次，程序崩溃

所以当我们手动实现拷贝构造函数的时候，需要完成深拷贝

```
//深拷贝代码实现
#include <iostream>
using namespace std;
class Test {
private:
    int i;
    int j;
    int *p;
public:
    Test(int v1 = 0, int v2 = 0, int v3 = 0) {
        i = v1;
        j = v2;
        p = new int;
        *p = v3;
    }
    // 深拷贝
    Test(const Test &obj) {
        i = obj.i;
        j = obj.j;
        p = new int;
        *p = *obj.p;
    }
    int* getP() {
        return p;
    }
    int getI() {
        return i;
    }
    int getJ() {
        return j;
    }
    void free() {
        delete p;
    }
};

int main() {
    Test t1;
    Test t2 = t1;
    cout << t1.getI() << " " << t1.getJ() << endl;
    cout << t2.getI() << " " << t2.getJ() << endl;
    cout << t1.getP() << " " << t2.getP() << endl;
    t1.free();
    t2.free();
    return 0;
}
```

2.6 初始化列表

初始化列表的引入：类中成员属性可以定义成const或者引用？

```
//代码演示，如果类中属性定义成const or 引用
#include <iostream>
using namespace std;
class Test {
    private:
        int i;
        const int ci;
        int &ri;
    public:
        Test(int ii) { // 是否报错？结论是一定报错

        }
};
int main() {
    return 0;
}
```

问题一：这个程序有问题吗？（有）

问题二：ci的值是多少？是不是真正的常量？（不是，只读变量）

```
//代码演示
#include <iostream>
using namespace std;
class Test {
    private:
        const int ci;
        int &ri;
        int i; // 只读属性
    public:
        Test(int v = 0) : ci(v), ri(i), i(v) {

        }

        void set(int v) {
            int *p = const_cast<int*>(&ci);
            *p = v;
            return ;
        }

        int get() {
            return ci;
        }
};
```

```
int main() {
    Test t(10);
    cout << t.get() << endl;
    t.set(100);
    cout << t.get() << endl;
    return 0;
}
```

const 属性必须初始化，引用使用之前必须初始化

属性的初始化：1、构造函数->报错，2、初始化列表

初始化列表使用

```
className::ClassName():m1(v1), m2(v2), m3(v3) {
    // 构造函数函数体
}

// 类外实现构造函数
class Test {
    Test();
};
Test::Test() : m1(), m2(), m3() {

}

class Test1 {
    Test1() : m1(), m2(), m3(),.....{

    }
};
```

初始化列表使用注意事项

- 1、成员属性初始化顺序与成员属性的声明顺序相同
- 2、成员初始化顺序与初始列表中的位置无关
- 3、初始化列表先于构造函数函数体执行

注意：第一、当执行到构造函数的函数体时，对象已经创建成功，执行构造函数，是给对象属性赋初值

第二：类中的const属性会被分配内存空间，这个属性在内存中的位置与具体对象相同，所以const修饰只是一个只读变脸，还是可以通过指针去修改这个属性的值

初始化与赋值的不同

初始化：对正在创建的对象进行初值的设置

赋值：对已经存在对象进行初值的设置

初始化列表可以初始化的属性

- 1、普通成员属性

2、const 属性

3、类成员属性：该类没有提供默认的构造函数，必须通过初始化列表中对该类成员属性初始化

2.7 对象的构造顺序

对象的构造顺序

1、局部对象构造：执行流达到对象声明语句时，对象就构造了（调用构造函数时）

```
#include <iostream>
class Test {
private:
    int mi;
public:
    Test(int v = 0) {
        cout << "Test(int)" << endl;
        mi = v;
    }

    int get() {
        return i;
    }
};

int main() {
    Test t1(10);
    int i = 0;
    while(i < 3) Test t2 = (++i);
    return 0;
}
```

不要顺便改变程序的执行顺序

2、堆对象的构造：遇到new这个关键字，就创建对象

代码演示：

```
//new 代码演示
#include <iostream>
using namespace std;
class Test {
private:
    int mi;
public:
    Test(int v = 0) {
        cout << "Test(int)" << endl;
    }
};

int main() {
    Test *t1 = new Test(10);
}
```

```
delete t1;
return 0;
}
```

3、全局对象构造顺序：构造顺序是不确定，c++标准中没有定义全局对象的构造顺序，不同的编译器有不同构造顺序，因此：**c++中避免全局变量的使用**

2.8 析构函数的使用

析构函数：作用对象的销毁

生活中：对象要初始化才能使用，当使用完成之后，一般都要将对象销毁。类中能否有销毁对象的方法

第一种方法：写一个销毁函数，用的时候手动调用

优点：可以销毁对象

确定：要手动调用

第二种方法：析构函数

优点：对象销毁的时候，自动调用

析构函数的用法：

```
#include <iostream>
using namespace std;
class Test {
    private:
        int mi;
    public:
        Test() { // 构造函数
                // 函数体
            }
        ~Test() { // 析构函数
                // 函数体
            }
};

int main() {

    return 0;
}
```

析构函数：没有参数，没有返回值，所以不能重载

析构函数定义的准则

类中定义析构函数，并且在构造过程中使用了系统资源，如文件打开，内存申请，设备打开等，则要在析构函数中进行文件关闭，内存归还，设备关闭操作。

析构函数的作用：

- 1、在对象销毁时，进行清理工作
- 2、对象销毁时自动调用
- 3、是对象释放系统资源的保证

2.9 对象构造顺序与析构顺序

1、单个对象创建的构造顺序

- 1、先调用父类的构造函数
- 2、调用对象成员的构造函数
- 3、调用自己的构造函数

析构函数调用的顺序与构造函数调用的顺序相反

2、多个对象的构造顺序

构造顺序：对象A->对象B->对象C

析构顺序：对象C->对象B->对象A

栈对象：类似于入栈出栈

堆对象：堆对象的析构与delete使用顺序有关

代码演示

```
//构造函数和析构函数调用顺序演示
#include <iostream>
using namespace std;
class A {
    private:
        int i;
    public:
        A(int v = 0) {
            i = v;
            cout << "A(int)" << endl;
        }
        ~A() {
            cout << "~A()" << endl;
        }
};
class Test {
    private:
        A a1;
        A a2;
        int i;
    public:
        Test(int v = 0) {
            cout << "Test(int)" << endl;
        }
}
```

```

        ~Test() {
            cout << "~Test()" << endl;
        }
};
int main() {
    Test t;
    return 0;
}

```

2.10 const成员函数

const修饰对象叫const对象

const 对象有以下特点：

1. 该对象是只读对象
2. 只读对象的成员属性是只读的

const 成员函数

声明const成员函数

```

// type className::Name(参数1, 参数2, ..... ) const {} // const 成员函数
#include <iostream>
using namespace std;
class Test {
    private:
        int i;
    public:
        int get() { // 普通成员函数
            return i;
        }
        int get() const { // const 成员函数
            return i;
        }
};
int main() {

    return 0;
}

```

const 成员函数特性

1. const 对象只能调用const成员函数
2. const 成员函数只能调用const成员函数
3. const 成员函数中不能改变变量的值

代码演示

```

#include <iostream>

```



```

using namespace std;
class Test {
private:
    int i;
public:
    Test(int v = 0) {
        cout << "Test(int)" << endl;
    }
    int get() {
        cout << "get()" << endl;
        print();
        return i;
    }
    int get() const {
        cout << "get() const" << endl;
        print();
        return i;
    }
    void print() {
        cout << "print()" << endl;
    }
    void print() const {
        cout << "print() const" << endl;
    }
    ~Test() {
        cout << "~Test()" << endl;
    }
};

int main() {
    Test t1;
    const Test t2;
    cout << t1.get() << endl;
    cout << t2.get() << endl;
    return 0;
}

```

2.11 成员函数和成员变量是隶属于具体的对象？

每一个对象有自己的成员变量，共用一套成员函数

属性

方法

从程序运行角度来看，对象是由数据和函数组成

数据存在栈区，堆区和全局数据区

函数在代码中存在代码段中，代码段是指程序编译完成了，就不能改变了。

this指针：表示指向自己的指针

成员函数只有一套，他能访问所有类的任何对象的成员变量，成员函数可以直接访问对象的成员属性

代码演示

```
//打印不同对象的this指针
#include <iostream>
using namespace std;
class Test {
public:
    Test* get() {
        return this;
    }
};
int main() {
    Test t1, t2;
```

```

    cout << t1.get() << endl;
    cout << t2.get() << endl;
    return 0;
}

```

2.12 数组类实现

```

#include <iostream>
using namespace std;
class Array {
private:
    int m_len;
    int *m_data;
public:
    Array(int v = 0) : m_len(v) {
        m_data = new int[m_len];
        for(int i = 0; i < m_len; ++i) {
            m_data[i] = 0;
        }
    }

    Array(const Array &obj) : m_len(obj.m_len) {
        m_data = new int[obj.m_len];
        for(int i = 0; i < obj.m_len; ++i) {
            m_data[i] = obj.m_data[i];
        }
    }

    int getlen() {
        return m_len;
    }

    bool get(int index, int &value) {
        bool ret = ((index >= 0) && (index < m_len));
        if(ret) {
            value = m_data[index];
        }
        return ret;
    }

    bool set(int index, int &value) {
        bool ret = ((index >= 0) && (index < m_len));
        if(ret) {
            m_data[index] = value;
        }
        return ret;
    }
}
/*

```

```

        void free() {
            delete[] m_data;
        }
    */
    ~Array() {
        delete[] m_data;
    }
};

int main() {
    Array a(10);
    for(int i = 0; i < a.getlen(); ++i) {
        int v = i + 1;
        a.set(i, v);
    }
    for(int i = 0; i < a.getlen(); ++i) {
        int v = 0;
        a.get(i, v);
        cout << v << endl;
    }
    return 0;
}

```

2.13 静态成员方法和静态成员属性

1、静态成员方法

知识点引入：

- 1、统计对象数目
- 2、保证程序的安全性
- 3、随时获取对象数目

问题分析：

不依赖对象访问静态成员变量

保证静态成员的安全性

方便、快捷访问静态成员变量

引出：静态成员函数

对象中的成员方法是每个对象专有的，在对象中不能共享

静态成员函数使用：

- 1、用static修饰
- 2、在类外单独分配空间，不隶属于任何一个对象
- 3、存储在全局变量区

```

//静态成员函数
static Type Name() {}
//可以直接通过类名去访问静态成员函数
className::Name();
//静态成员变量
static Type Name;
//初始化
Type className::Name = 0;

```

静态成员函数特点:

- 1、静态成员函数是类中特殊的函数
- 2、静态成员函数属于整个类
- 3、静态成员函数可以直接通过类名直接访问公有的静态成员函数
- 4、通过对象名可以直接访问公有的静态成员函数
- 5、静态成员函数不能直接访问成员变量
- 6、静态成员函数没有this指针，所以不能直接访问类中的成员变量

2.14 临时对象（返回值优化）

由一个历程引入

```

#include <iostream>
using namespace std;
class Test {
    private:
        int a;
    public:
        Test() {
            Test(10);
            //a = 10
        }
        Test(int v) {
            a = v;
        }
        void print() {
            cout << "a =" << a << endl;
        }
};
int main() {
    Test t;
    t.print();
    return 0;
}

```

看看结果是什么？

程序的意图是用0初始化a

运行结果：a的值是随机值。

构造函数，一般都是自动调用，现在是我们手动调用构造函数，就会产生临时对象（这是合法的）

//接上个程序是怎么执行的

Test t(10) 等价于 Test t = Test(10);

执行过程：1、生成临时对象，2 用临时对象去初始化新对象，调用拷贝构造函数

g++关闭返回值优化选项：-fno-elide-constructors

临时对象产生的一种情况：手动调用构造函数

临时对象特点

1 生命周期只有一条语句

2 作用于只有当前语句

3 是c++中灰色地带

三、继承

3.1 类封装

类的基本关系：

1、组合关系

2、继承关系

其中类和类最简单的关系就是：组合关系

一个类，从两方面看->类的实现，需要知道具体实现细节

从类的使用上看，只需要知道一个类怎么使用就可以，不需要知道类内部的细节

我们先从类的实现方面看一个类：复杂的实现过程要被封装起来

封装：封装的体现：属性和行为

语法体现：

public：外界可以使用

private：外界不可以使用

protected：继承时候讲

```
#include <iostream>
using namespace std;
class Test {
    private:
```

```

        int a;
    public:
        int b;
};
int main() {
    Test t;
    t.a = 0; // erro
    t.b = 10; // ok
    return 0;
}

```

实现一个计算类

```

// 实现一个计算类，联系权限
#include <iostream>
using namespace std;
class Compute {
    private:
        char op;
        int a;
        int b;
    public:
        void setValue(int v1, int v2);
        bool setOp(char c);
        bool res(int &v);
};

void Compute::setValue(int v1, int v2) {
    a = v1;
    b = v2;
    return;
}

bool Compute::setOp(char c) {
    bool ret = false;
    if((c == '+') || (c == '-') || (c == '*') || (c == '/')) {
        ret = true;
        op = c;
    } else {
        ret = false;
    }
    return ret;
}

bool Compute::res(int &v) {
    bool ret = false;
    if(op == '+') {
        v = a + b;
        ret = true;
    }
}

```

```

    } else if(op == '-') {
        v = a - b;
        ret = true;
    } else if(op == '*') {
        v = a * b;
        ret = true;
    } else if(op == '/') {
        if(b == 0) {
            ret = false;
            v = 0;
        } else {
            ret = true;
            v = a / b;
        }
    }
}

return ret;
}

int main() {
    Compute c;
    c.setValue(10, 5);
    int v = 0;
    c.setOp('+');
    c.res(v);
    cout << v << endl;
    return 0;
}

```

整个类是一个作用域，所以在同一个类，没有访问权限的限制，所以同一个类中成员函数可以访问类内部的成员属性

对于访问权限：是对类外部

对于类封装：区别作用域和访问权限概念

对于类的方法和属性的访问，一般通过对象去访问和使用，这个时候是类的外部。访问权限就起作用了

3.2 组合关系

讲继承之前内，先讲类的组合关系

类中组合关系：是整体与部分的关系

组合特点：

- 1、其他类对象是某个类的成员属性
- 2、当前类对象与成员对象生命周期相同
- 3、成员对象在用法上和普通对象相同

在实际使用过程中，因为组合关系是最简单，优先考虑组合关系，后考虑继承关系


```
//组合关系代码演示
#include <iostream>
using namespace std;
class Mem {
public:
    Mem() {
        cout << "Mem()" << endl;
    }
};
class Disk {
public:
    Disk() {
        cout << "Disk()" << endl;
    }
};
class CPU {
public:
    CPU() {
        cout << "Cpu()" << endl;
    }
};
class Computer {
private:
    Mem m;
    Disk d;
    CPU c;
public:
    Computer() {
        cout << "Computer()" << endl;
    }
    void powerdown() {
        cout << "power down" << endl;
    }
};
int main() {
    Computer c;
    c.powerdown();
    return 0;
}
```

3.3 继承关系

继承关系：是指类之间的父子关系

继承的特点：

- 1、子类拥有父类所有的属性和方法

- 2、子类是一种特殊的父类：例如我想买电脑，买了一个dell电脑
- 3、子类对象可以当作父类对象的使用
- 4、子类中可以添加父类中没有的方法

语法特性：

```
#include <iostream>
using namespace std;
class Father {
    private:
        int v;
    public:
        Father() {
            cout << "Father()" << endl;
        }
        void print() {
            cout << "v = " << v << endl;
        }
};
class Son : public Father {

};
int main() {
    Son s;
    s.print();
    return 0;
}
```

从上面代码可以看出继承的意义：代码复用，高级面向对象课程中重要前导知识

子类和父类的关系：

- 1、子类就是一个特殊父类
- 2、子类对象可以直接初始化父类对象
- 3、子类对象可以直接赋值给父类对象

代码演示

```
#include <iostream>
using namespace std;
class Parent {

};
class Child : public Parent {

};

int main() {
```

```

    Child c;
    Parent p = c;
    Parent p1;
    p1 = c;
    return 0;
}

```

继承小结：

- 1、继承是类和类之间的一种特殊关系
- 2、子类拥有父类的所有属性和方法
- 3、子类对象当作父类对象使用
- 4、子类可以添加新属性和新的方法，也可以改写父类中方法
- 5、继承是代码复用的重要手段

3.4 继承中的访问级别

问题引出：子类可以直接访问父类中的私有成员吗？

根据面向对象理论：继承，子类拥有父类中一切属性和方法得出子类可以访问父类中的私有成员属性和方法

根据c++理论，外界不能直接访问类内部的私有成员属性和方法

代码验证，哪个对？

```

#include <iostream>
using namespace std;
class Father {
    private:
        int a;
    public:
        Father(int v = 0) {
            a = v;
        }
        int get() {
            return a;
        }
};

class Child : public Father {
    public:
        int add() {
            return a + 10;
        }
};

int main() {
    Child c;
}

```

```

    cout << c.add() << endl;
    return 0;
}

```

```

/Users/wrf/Desktop/Test.cpp:5:10: note: declared private here
    int a;
    ^

```

结论：父类private成员不能被子类方法访问，即而，说是代码复用，其实不能复用，扯淡呢!!!
 引出新的知识点：protected

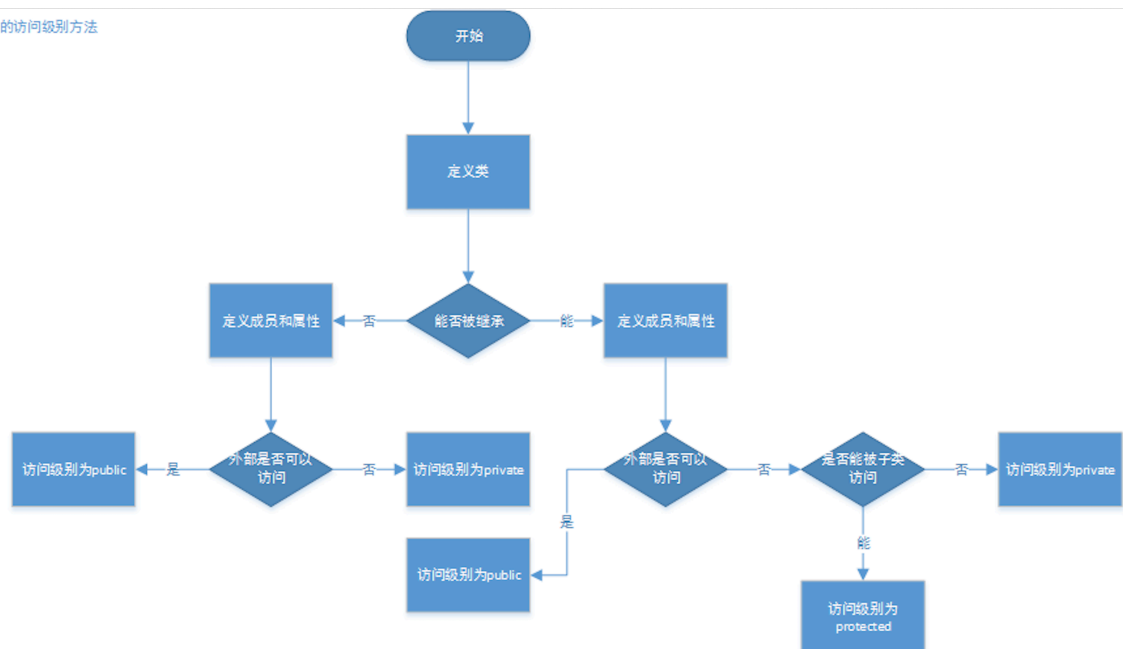
访问级别：protected：被保护的，修饰的成员变量不能被外部直接访问，修饰的成员变量可以直接被子类直接访问

为什么要有protected? 举例子：源于生活（匈奴举例）。

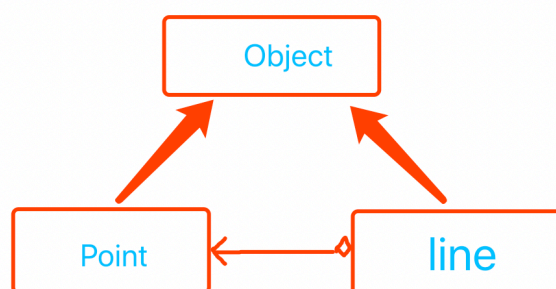
protected权限介于**private**和**public**之间

如何确定类属性的访问级别?

定义类中成员和属性的访问级别方法



3.5 完成一个类设计



```

#include <iostream>

```

```

#include <string>
#include <sstream>
using namespace std;
class Object {
    protected:
        string m_name;
        string m_info;
    public:
        Object() {
            m_name = "Object";
            m_info = "";
        }
        string name() {
            return m_name;
        }
        string info() {
            return m_info;
        }
};

class Point : public Object {
    private:
        int mx;
        int my;
    public:
        Point(int x = 0, int y = 0) {
            mx = x;
            my = y;
            m_name = "Point";
            ostringstream s;
            s << "P(" << mx << "," << my << ")";
            m_info = s.str();
        }
        int x() {
            return mx;
        }
        int y() {
            return my;
        }
};

class Line : public Object {
    private:
        Point mp1;
        Point mp2;
    public:
        Line(const Point &p1, const Point &p2) {
            mp1 = p1;
            mp2 = p2;
        }
};

```

```

        m_name = "Line";
        ostringstream s;
        s << "Line from " << mp1.info() << " to " << mp2.info() << ".";
        m_info = s.str();
    }
    Point begin() {
        return mp1;
    }
    Point end() {
        return mp2;
    }
};

int main() {
    Object o;
    cout << o.name() << endl;
    cout << o.info() << endl;
    Point p1, p2(3, 5);
    Line l(p1, p2);
    cout << p1.name() << endl;
    cout << p1.info() << endl;
    cout << p2.name() << endl;
    cout << p2.info() << endl;
    cout << l.name() << endl;
    cout << l.info() << endl;
    return 0;
}

```

3.6 不同的继承方式

继承是一种方式，代码格式

```

//继承方式
class A {};
class B : 继承方式 父类名称 {};

```

继承方式分为：public，protected，private

public继承方式：父类成员在子类中保持原有的访问级别

private继承方式：父类成员在子类中变成私有访问级别

protected继承方式：父类中共有成员变成protected，其他访问权限成员变量不变

		public	protected	private	父类中成员访问级别
继承方式	public	public	protected	private	父类中成员在继承到子类的访问级别
	protected	protected	protected	private	
	private	private	private	private	

父类中成员继承到子类之后访问属性变化：

父类中成员继承到子类之后访问属性 = max{ 继承方式, 父类成员访问属性}

C++中默认的继承方式是私有继承

一般性原则:

1、C++工程项目只用public继承方式

2、C++派生语言, 只支撑一种继承方式, public

3、private和protected继承方式带来的复杂性远大于工程实际应用特性

4、综上, 继承方式就是用public继承

3.7 继承中的构造和析构

子类中定义的构造函数原则:

1、必须对继承过来的成员进行初始化

2、初始化方式: 子类构造函数的初始化列表or子类构造函数的函数体中

3、父类成员的初始化, 直接调用父类的构造函数

存在一个问题, 父类中的公有成员在子类中是可以访问的, 但是父类中的私有成员, 在子类中是不能访问的, 所以, 给父类成员初始化, 直接调用父类的构造函数, 是最简单的方式

父类构造函数调用方式:

1、自动调用:

适用于: 父类中构造函数是无参构造和带有默认参数构造函数

子类在创建对象时候会自动调用父类的构造函数

代码演示

```
#include <iostream>
using namespace std;
class A {
    private:
        int mi;
    public:
        A(int v) : mi(v) {

        }
};
class B : public A {
    public:
        B() {
            cout << " B " << endl;
        }
};

int main() {
```

```

    B b; //报错
    return 0;
}

```

test.cpp:13:9: error: constructor for 'B' must explicitly initialize the base class 'A' which does not have a default constructor

报错信息，提示A类没有默认的构造函数，改正，添加一个无参构造函数，或者把int v 加上默认参数

2、手动调用

手动调用也成为显示调用，使用方式是，在子类构造函数的初始化列表中调用父类构造函数

代码演示

```

#include <iostream>
using namespace std;
class A {
    private:
        int mi;
    public:
        A(int v) : mi(v) {

        }
};
class B : public A {
    public:
        B() : A(5) {
            // 显示调用父类的构造函数
        }
};
int main() {
    B b;
    return 0;
}

```

继承构造规则： 1、子类对象在创建时候，会先调用父类的构造函数，调用方式：显示调用，隐式调用

2、先执行父类的构造函数，再执行子类的构造函数，所以初始化列表中，最先写父类构造函数，然后依次是子类的属性初始化

组合 + 继承的构造

- 1、先调用父类的构造函数
- 2、再调用成员对象的构造函数
- 3、最后调用自己的构造函数

口诀：先父母，再兄弟，后自己(可递归调用)，析构顺序与构造顺序相反

代码演示：

```

#include <iostream>

```



```

using namespace std;
class A {
public:
    A() {
        cout << "A" << endl;
    }
    ~A() {
        cout << "~A" << endl;
    }
};
class B {
public:
    B() {
        cout << "B" << endl;
    }
    ~B() {
        cout << "~B" << endl;
    }
};
class C : public B {
private:
    A a1;
public:
    C() : B(), a1() {
        cout << "C" << endl;
    }
    ~C() {
        cout << "~C" << endl;
    }
};
int main() {
    C c;
    return 0;
}

```

继承构造小结：

- 1、子类对象构造时候，先初始化父类的各属性，只要调用父类的构造函数
- 2、先执行父类的构造函数，再执行成员属性的构造函数，最后执行自己的构造函数
- 3、子类对象销毁的时候也需要调用父类和成员对象的析构函数
- 4、父类成员构造显示调用的时候，需要在子类构造函数的初始化列表中
- 5、继承的构造顺序与析构顺序相反

3.8 父子间冲突

子类中是否可以定义父类中同名的属性？

代码验证

```

#include <iostream>
using namespace std;
class A {
    private:
        int a;
    public:
        A(int v = 0) : a(v) {
            cout << "A::a = " << a << endl;
        }
        int getA() {
            cout << "A::a = ";
            return a;
        }
};
class B : public A {
    private:
        int a;
    public:
        B(int v = 1) : A(10), a(v) {
            cout << "B::a = " << a << endl;
        }

        int getA() {
            cout << "B::a = ";
            return a;
        }
};
int main() {
    B b(5);
    cout << b.getA() << endl;
    return 0;
}

```

程序运行结果：

```

A::a = 10
B::a = 5
B::a = 5

```

结果我们看到，输出的是B类中的a属性的值，结论优先选择子类中同名函数。

结论：

1、子类可以定义父类同名成员

2、子类中如果和父类中定义同名成员，编译器会将父类成员给隐藏（这个叫同名覆盖）

3、被隐藏的同名成员依然在子类中，可以通过作用域分辨符去访问

```
#include <iostream>
using namespace std;
class A {
    private:
        int a;
    public:
        A(int v = 0) : a(v) {
            cout << "A::a = " << a << endl;
        }
        int getA() {
            cout << "A::a = ";
            return a;
        }
};

class B : public A {
    private:
        int a;
    public:
        B(int v = 1) : A(10), a(v) {
            cout << "B::a = " << a << endl;
        }

        int getA() {
            cout << "B::a = ";
            return a;
        }
};

int main() {
    B b(5);
    cout << b.getA() << endl; // 默认访问子类
    cout << b.A::getA() << endl; // 显示访问父类成员函数
    return 0;
}
```

运行结果：

```
A::a = 10
B::a = 5
B::a = 5
A::a = 10
[Finished in 0.4s]
```

3.9 同名覆盖带来的问题

父类指针（引用）指向子类对象时候，编译器这个时候会怎么处理呢？

编译期间，编译器只能根据指针类型去判断，统一调用父类中的函数，解释为父类版本最安全

问题来源：子类重写父类中函数，因为父类中函数不满足需求，所以要在子类中重写，结果调用的时候就出问题了。

3.10 对象模型

成员属性：

对象模型讨论的问题是对象在内存中是如何存储的。

数据类型的本质是什么？数据类型的本质是固定内存大小的别名。

class是一种特殊的结构体

class与struct遵循相同的内存对其规则

class中的属性和方法是分开存放的

所有对象共享成员方法

c++中的对象并不包含成员函数

一个对象就是一个结构体，运行的时候，类退化成结构体

1、所有成员变量在内存中一次排序

2、成员变量可能存在内存空隙

3、通过内存地址可以直接访问成员变量

4、访问权限在编译期间有效，在运行期间无效

代码演示：

```
#include <iostream>
```

```

using namespace std;
class A {
public:
    int i;
    int j;
    char c;
    double d;
    int getI() {
        return i;
    }
    int getJ() {
        return j;
    }
    char getC() {
        return c;
    }
    double getD() {
        return d;
    }
};

typedef struct B {
    int a;
    int i;
    int j;
    char c;
    double d;
} B;

int main() {
    A a;
    B *b;
    b = reinterpret_cast<B *>(&a); // 重新调整内存
    b->i = 10;
    b->j = 100;
    b->c = 'C';
    b->d = 100.0;
    cout << a.getI() << endl;
    cout << a.getJ() << endl;
    cout << a.getC() << endl;
    cout << a.getD() << endl;
    b->i = 10000;
    b->j = 100;
    b->c = 'C';
    b->d = 100.1;
    cout << a.getI() << endl;
    cout << a.getJ() << endl;
    cout << a.getC() << endl;
    cout << a.getD() << endl;
    return 0;
}

```

成员方法：

1、类中成员方法在代码段中

2、对象调用成员函数，对象地址作为隐藏函数传给成员函数

3、成员函数通过对象地址访问成员变量

4、C++隐藏了对象地址传递过程

用C语言实现一个C++类

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
typedef void demo;
typedef struct class_demo {
    int i;
    int j;
} class_demo;
demo* creat_demo(int i, int j) {
    class_demo* ret = (class_demo*)malloc(sizeof(class_demo));
    if(ret != NULL) {
        ret->i = i;
        ret->j = j;
    }
    return ret;
}
int getI(demo* p) {
    class_demo *p1 = (class_demo *)p;
    return p1->i;
}
int getJ(demo* p) {
    class_demo *p1 = (class_demo *)p;
    return p1->j;
}
void free_demo(demo *p) {
    free(p);
    return ;
}

int main() {
    demo *d = creat_demo(1, 2);
    cout << getI(d) << endl;
    cout << getJ(d) << endl;
    //d->i = 100; // class 默认私有，外部不能访问
    free_demo(d);
    return 0;
}
```

继承情况下对象的存储代码演示：

```
#include <stdlib.h>
#include <stdio.h>
typedef void demo;
typedef void deverid;
demo* demo_init(int i, int j);
int getA(demo *pthis);
int getB(demo *pthis);
int add(demo *pthis, int value);
void free_demo(demo *pthis);
deverid* deverid_init(int i, int j, int k);
int getC(deverid *pthis);
int add1(deverid *pthis, int value);
void free_deverid(deverid *pthis);
typedef struct class_demo {
    int a;
    int b;
}class_demo;
typedef struct class_derived
{
    class_demo d;
    int c;
} class_derived;
demo* demo_init(int i, int j) {
    class_demo *p = (class_demo*) malloc(sizeof(class_demo));
    if(p != NULL) {
        p->a = i;
        p->b = j;
    }
    return p;
}

int getA(demo *pthis) {
    class_demo *p = (class_demo*) pthis;
    return p->a;
}

int getB(demo* pthis) {
    class_demo *p = (class_demo*)pthis;
    return p->b;
}

int add(demo *pthis, int value) {
    class_demo *p = (class_demo *)pthis;
    return p->a + p->b + value;
}

void free_demo(demo *pthis) {
```

```

    free(pthis);
    return ;
}

deverid* deverid_init(int i, int j, int k) {
    class_derived *p = (class_derived *) malloc(sizeof(class_derived));
    if(p != NULL) {
        p->d.a = i;
        p->d.b = j;
        p->c = k;
    }
    return p;
}

int getC(deverid *pthis) {
    class_derived *p = (class_derived *) pthis;
    return p->c;
}

int add1(deverid *pthis, int value) {
    class_derived *p = (class_derived *) pthis;
    return p->d.a + p->d.b + p->c + value;
}

void free_deverid(deverid *pthis) {
    free(pthis);
}

int main() {
    demo *d = demo_init(1, 2);
    printf("a = %d\n", getA(d));
    printf("b = %d\n", getB(d));
    printf("add = %d\n", add(d, 5));
    // d->a = 100; // 默认私有
    free_demo(d);
    deverid *d1 = deverid_init(1, 2, 3);
    printf("a = %d\n", getA(d1));
    printf("b = %d\n", getB(d1));
    printf("c = %d\n", getC(d1));
    printf("add1 = %d\n", add1(d1, 5));
    //d1->c = 100;
    free_deverid(d1);
    return

```

四、多态

五、模板

六、异常

七、重载和运算符重载
