C++核心编程

1 内存分区模型

1.1 概述

C++是一种面向对象的编程语言，他吃苹果面向对象，吃苹果面向过程。C语言的基本单元是函数（一个过程），C++基本单元是类（一个对象）。

在VS下，依然是，项目+源文件。

#include<iostream>//头文件，输入输出流

#include<string>

using namespace std;//命名标准使用空间

int main()

{

string str = "hello";//定义变量

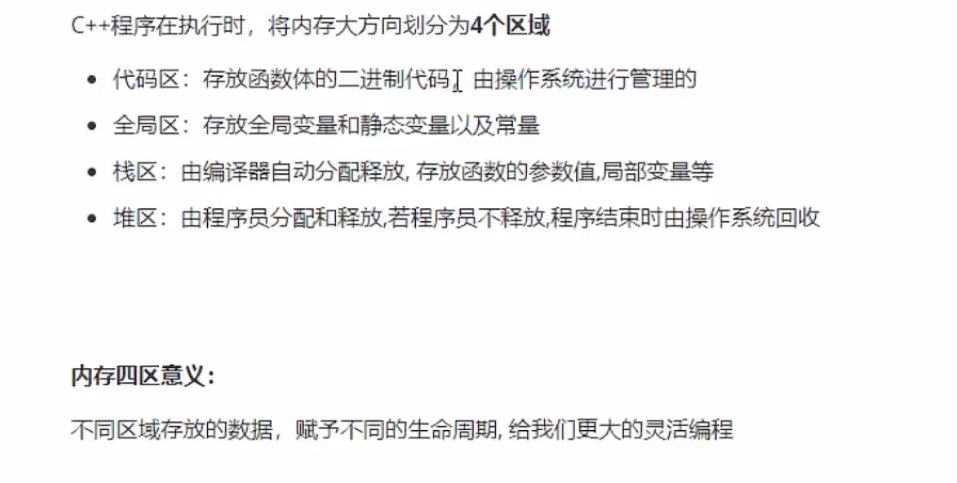
cout << "请输入字符串" << endl;//输出格式

cin >> str;//输入格式

cout << "字符串str=" << str << endl;

return 0;

}



全局区又叫静态区。

1.2 程序运行前

在程序编译后，生成exe可执行程序，为执行程序前分为两个区域：

（1）代码区

存放CPU执行的机器指令。

代码区是共享的，共享目的是防止重复执行相同的程序；

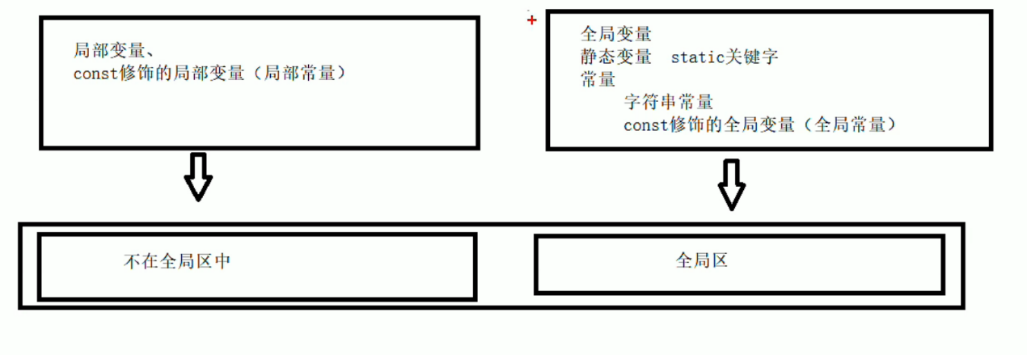
代码区是只读的，目的是防止程序意外的修改了指令。

（2）静态区

静态变量和全局变量的存放；

静态区还包括了常量区，字符串常量和其他常量（const修饰的全局变量才在）也存放在此；

该区域的数据在程序结束后由操作系统释放。



#include<iostream>

using namespace std;

//静态区（全局区）存放的数据

int g\_a = 10;//全局变量global

int g\_b = 10;

const int c\_g\_a = 10;//const修饰的全局变量

const int c\_g\_b = 10;

int main()

{

int a = 10;//局部变量

int b = 10;

cout << "局部变量a的地址：" << &a << endl;

cout << "局部变量b的地址：" << &b << endl;

const int c\_l\_a = 10;//const修饰的局部变量local

const int c\_l\_b = 10;

cout << "const修饰的局部变量c\_l\_a的地址：" << &c\_l\_a << endl;

cout << "const修饰的局部变量c\_l\_b的地址：" << &c\_l\_b << endl;

static int s\_a = 10;//静态变量

static int s\_b = 10;

cout << "静态变量s\_a的地址：" << &s\_a << endl;

cout << "静态变量s\_b的地址：" << &s\_b << endl;

cout << "全局变量g\_a的地址：" << &g\_a << endl;

cout << "全局变量g\_b的地址：" << &g\_b << endl;

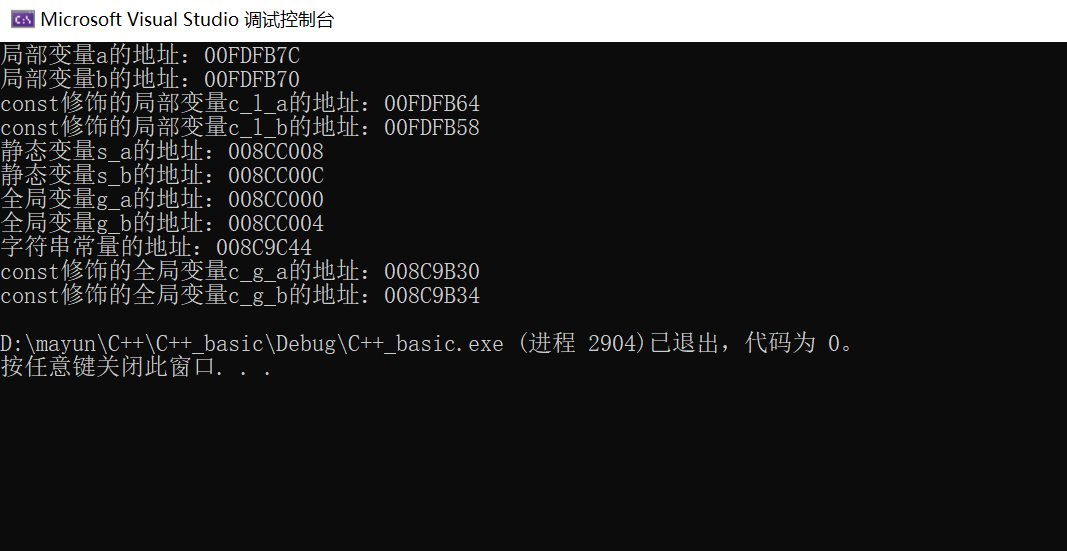
cout << "字符串常量的地址：" << &"hello world!!" << endl;

cout << "const修饰的全局变量c\_g\_a的地址：" << &c\_g\_a << endl;

cout << "const修饰的全局变量c\_g\_b的地址：" << &c\_g\_b << endl;

return 0;

}



1.3 程序运行后

（1）栈区

由编译器自动分配和释放，存放函数的参数值，局部变量等。

注：不要返回局部变量的地址，因为栈区开辟的空间由编译器自动释放，局部变量出函数就释放了，不能获取到该地址的内容了。

#include<iostream>

using namespace std;

int\* func(int b)//函数的参数和局部变量存放在栈区

{

b = 100;

int a = 10;

return &a;//不能返回局部变量的地址

}

int main()

{

int\* p = func(1);

cout << \*p << endl;//第一次打印正确是编译器的保留

cout << \*p << endl;//返回局部变量的地址，该地址的内容在出函数时已经被编译器释放了

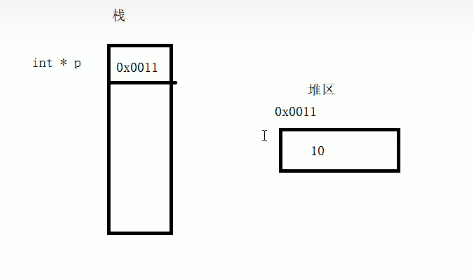
return 0;

}

（2）堆区

由程序员分配和释放，程序结束后由操作系统回收内存。

在C++中主要利用new操作符在堆区开辟内存。



int\* func()

{

int\* p = new int(10);//new操作符在堆区开辟内存，返回内存的地址；括号内是内存中的初始值

return p;//返回的是局部变量p的内容是一个地址

}

int main()

{

int\* p = func();//返回的一个地址，由于地址内存开辟在堆区，在程序运行时不被释放

cout << \*p << endl;

cout << \*p << endl;//在没有释放之前可以一直访问成功

return 0;

}

1.4 new操作符

（1）new的基本语法

int\* p = new int(10);//new操作符在堆区开辟内存，返回内存的地址；括号内是内存中的初始值

如果想释放堆区的数据，用关键字delete。

int\* func()

{

int\* p = new int(10);//new关键字在堆区开辟内存，返回内存的地址；括号内是内存中的初始值

return p;//返回的是局部变量p的内容是一个地址

}

int main()

{

int\* p = func();//返回的一个地址，由于地址内存开辟在堆区，在程序运行时不被释放

cout << \*p << endl;

cout << \*p << endl;//在没有释放之前可以一直访问成功

delete p;//已经释放堆区的内存

cout << \*p << endl;//非法访问冲突

cout << \*p << endl;

return 0;

}

（2）在堆区利用new开辟数组

void test1()//堆区创建一个数组

{

int\* arr = new int[10];//数组用[]，括号中代表元素的个数。返回数组的首地址

for (int i = 0; i < 10; i++)

{

arr[i] = i + 100;

}

for (int i = 0; i < 10; i++)

{

cout << arr[i] << endl;

}

delete[] arr;//释放数组时，要带[]

}

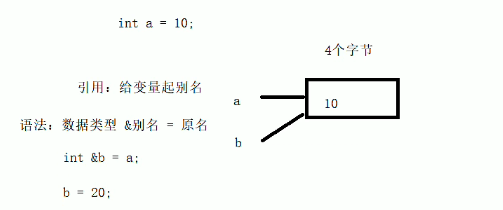
2 引用（注意内存分析）

2.1 引用的基本语法

引用的含义：给变量取别名。

语法：数据类型 &别名 = 原名；

如：int a = 10; // int &b = a;//则a,b都是同一块空间的名字。



int main()

{

int a = 10;

int& b = a;//引用的基本语法

cout << "a = " << a << endl;

cout << "b = " << b << endl;

b = 100;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

return 0;

}

2.2 引用的注意事项

（1）引用必须要初始化

在定义引用时必须要赋予初值，即必须要有原名。

Int &b;//没有初始化是错误的

（2）引用在初始化后，不可以改变。

即已经初始化引用了，则不可以作为其他变量的别名呢。

int a = 10;

int& b = a;//引用的基本语法

int c = 20;

b = c;//赋值操作

cout << "a = " << a << endl;

cout << "b = " << b << endl;

cout << "c = " << c << endl;

2.3 引用做函数参数

引用的作用：函数传参时，可以利用引用的技术达到形参改变实参的效果。

优点：可以简化指针修改实参。

（1）值传递

形参的改变不影响实参的值。

（2）址传递

用指针接收，形参的改变会改变实参的值。

（3）引用传递（避免了指针操作）

通过引用接收，形参的改变会影响实参。

void Swap03(int& a, int& b)//

{

//1

/\*int tmp = a;

a = b;

b = tmp;\*/

//2

/\*int tmp = \*a;

\*a = \*b;

\*b = tmp;\*/

//3

int tmp = a;

a = b;

b = tmp;

}

int main()

{

int a = 10;

int b = 20;

//Swap01(a, b);//1.值传递 值传递不改变实参的值

//Swap02(&a, &b);//2.址传递，指针接收，可以修饰实参

Swap03(a, b);//3.引用接收，可以修饰实参

cout << "a = " << a << endl;

cout << "b = " << b << endl;

return 0;

}

2.4 引用作为函数返回值

（1）不要返回局部变量的引用

局部变量出函数就内存释放了，会造成非法访问。

（2）函数的调用可以作为左值

当函数的返回值是引用时（全局或静态变量）可以作为赋值的左值被改变。

/int& test01()//(1)不要返回局部变量的引用

//{

// int a = 10;

// return a;

//}

int& test02()//静态变量全局区

{

static int a = 10;

return a;

}

int main()

{

/\*int& ret1 = test01();

cout << "ret1 = " << ret1 << endl;

cout << "ret1 = " << ret1 << endl;\*/

int& ret2 = test02();

cout << "ret2 = " << ret2 << endl;

cout << "ret2 = " << ret2 << endl;

test02() = 1000;//(2)返回引用的函数可以作为左值 内存分析

cout << "ret2 = " << ret2 << endl;

cout << "ret2 = " << ret2 << endl;

cout << "ret2 = " << ret2 << endl;

return 0;

}

2.5 引用的本质

本质：引用的本质在C++内部实际上是一个指针常量（它指向的位置不能被改变，但其指向的内容可以被改变）。int a = 10; int\* const p = &a; int& p = a;

形式上就是一个变量别名，所以简化了指针的定义和解引用操作等。

2.6 常量引用

与const一起使用：用来修饰形参，防止误操作。因为引用作为形参会改变实参的值，所以在一些不需要修改实参的情况下，可以加const修饰形参的引用。

void ShowValue(const int& val)//用const修饰引用，防止误操作

{

//val = 10000;//防止误操作

cout << "val = " << val << endl;

}

int main()

{

int a = 100;

ShowValue(a);

cout << "a = " << a << endl;//引用做形参会改变实参的值

return 0;

}

3 函数的提高

3.1 函数的默认参数

（1）基本格式

返回值 函数名(形参类型 形参名 = 默认值){}

如果使用函数时，没传相应参数的值，则采用默认值；如果传了值，则用传入的值。

（2）注意事项

a.如果某个位置已经有了默认参数，则这个位置之后的所以参数都必须要有默认值。

b.如果函数声明有了默认参数，则函数的实现不能写默认值。声明和实现只能有一个有默认参数。

3.2 函数的占位参数（讲的不全）

（1）基本格式

返回类型 函数名(数据类型){}

只写数据类型的形参就是占位参数，传入时必须按照类型传。

（2）占位参数也可以有默认参数(数据类型 = 10)

3.3 函数的重载

3.3.1 函数重载概述

作用：C++中函数名可以相同，提高代码的复用性。

函数重载的前提条件：（注意函数调用出错）

（1）同一作用域下，比如都是全局的。

（2）函数名相同

（3）**函数参数**类型不同或者个数不同或者顺序不同

注：函数的返回值不可以作为函数重载的条件。

3.3.2 函数重载的注意事项

（1）引用作为重载条件

函数参数加const 和不加const也是可以作为函数重载条件的，如果传入参数是变量则走没加const的函数，如果传入是常量，则走加了const 的函数。

（2）函数重载碰到默认参数

容易出现函数调用出错，尽量避免。

#include<iostream>

using namespace std;

//1.引用与函数重载

void func1(int& a)

{

cout << "函数 func1(int& a)的调用" << endl;

}

void func1(const int& a)

{

cout << "函数 func1(const int& a)的调用" << endl;

}

//2.函数重载与默认参数

void func2(int a, int b = 10)

{

cout << "函数 func2(int a, int b = 10)的调用" << endl;

}

void func2(int a)

{

cout << "函数 func2(int a)的调用" << endl;

}

int main()

{

int a = 10;

func1(a);//传入变量，走无const的函数

func1(10);//传入常量，走有const的函数

//func2(10);//函数重载碰到默认参数，容易导致函数调用出错，尽量避免

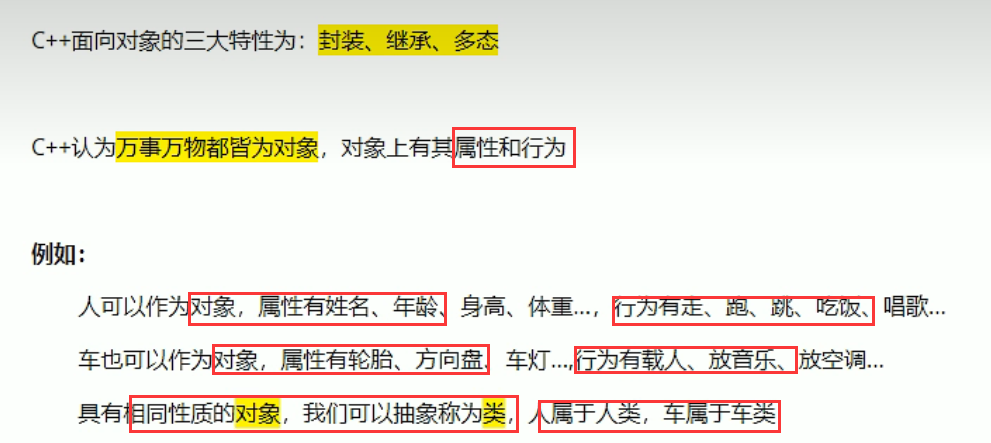
func2(10, 10);

return 0;

}

4 类和对象

**概述：**



4.1 封装

4.1.1 封装的意义

封装是C++面向对象的三大特点之一。

封装的意义：

（1）将属性和行为作为一个整体，表现生活中的事物

（2）将属性和行为加以权限控制

**在设计类的时候，将属性（成员属性或成员变量）和行为（成员函数或成员方法）写在一起，表现事物。**

语法：class 类名{ 访问权限: 属性 / 行为};

例：设计一个圆类，球圆的周长

#include<iostream>

using namespace std;

const double PI = 3.14;

//类和对象

//例：设计一个圆类，可以获取其周长C = 2\*PI\*半径

//封装 类

class Circle

{

//(1)访问权限 如公共权限

public:

//(2)属性 变量 如圆的半径

double m\_r;

//(3)行为 函数表达 如圆的周长

double Circle\_C()

{

return 2 \* PI \* m\_r;

}

};

int main()

{

//实例化 用类创建一个具体的对象，如用圆类创建一个圆

Circle c1;

c1.m\_r = 10;

cout << "圆的周长为" << c1.Circle\_C() << endl;

return 0;

}

例：创建一个学生类，有姓名和学号，可以打印姓名和学号。

**也可以用类的行为给属性赋值**。

#include<iostream>

#include<string>

using namespace std;

class Student

{

public:

string name;

string id;

void GetName()

{

cout << "学生姓名为" << name << endl;

}

void GetID()

{

cout << "学生学号为" << id << endl;

}

//用类的行为对类的属性赋值

void SetName(string s\_name)

{

name = s\_name;

}

void SetID(string s\_id)

{

id = s\_id;

}

};

int main()

{

Student stu1;

stu1.SetName("张三");

stu1.SetID("123456");

stu1.GetName();

stu1.GetID();

return 0;

}

4.1.2 访问权限

（1）公共权限(public)

成员 类内可以访问，类外也可以访问

（2）保护权限(protected)

成员 类内可以访问，类外不可以访问。且继承时，儿子可以访问父亲中保护的内容，如车是可以使用的。

（3）私有权限(private)

成员 类内可以访问，类外不可以访问。且继承时，儿子不可以访问父亲中私有的内容，如银行卡密码是不可以使用的。

#include<iostream>

#include<string>

using namespace std;

class Person

{

public:

string name;

protected:

string car;

private:

int password;

public:

void fun()

{

name = "张三";

car = "三轮车";

password = 12345;

}

};

int main()

{

Person p1;

p1.name = "里斯";//公共权限下，类内类外都可以访问

//p1.car = "宝马";//保护权限下成员类外不可以访问

//p1.password = 123;//私有权限下成员类外不可以访问

return 0;

}

**注：C++中struct和class的区别，其实并没有什么区别，主要的区别在于它们的默认权限不同。Struct默认权限为公共权限，class默认权限为私有权限。**

4.1.3 成员属性私有化

好处：

（1）可以自己控制读写权限。

（2）对于写操作，可以检测写入数据的有效性。

#include<iostream>

#include<string>

using namespace std;

//(1)成员属性设置为私有，可以自己控制读写权限

class Person

{

private://成员属性设置为私有

string m\_Name;

int m\_Age;

string m\_Lover;

public://以成员函数来控制属性的读写权限

//姓名 可读可写

void SetName(string name)

{

m\_Name = name;

}

string GetName()

{

return m\_Name;

}

//年龄只可读

int GetAge()

{

return m\_Age;

}

//(2)对于写操作可以检测传入数据的有效性，如年龄在0-150之间

//设置年龄

void SetAge(int age)

{

m\_Age = 0;

if (age >= 0 && age <= 150)

{

m\_Age = age;

}

else

{

cout << "写入年龄不符规定！！ age:" << age << endl;

m\_Age = 0;

}

}

//爱人只写

void SetLover(string lover)

{

m\_Lover = lover;

}

};

int main()

{

Person p1;

p1.SetName("kuikui");//设置姓名

p1.SetAge(10);

cout << "姓名为: " << p1.GetName() << endl;//读取姓名

cout << "年龄为: " << p1.GetAge() << endl;//读取年龄

p1.SetLover("lanlan");

p1.SetAge(100);

cout << "年龄为: " << p1.GetAge() << endl;//读取年龄

p1.SetAge(180);

return 0;

}

案例一：设计一个立方体类，可以获得面积体积；分别用全局函数和成员函数判断两个立方体是否相等。

//1.创建一个立方体类，并可获取其面积体积；

//由于属性尽量设置成私有，则还需要设置属性的读写成员函数

class Cube

{

//长宽高属性

private:

int m\_L;

int m\_W;

int m\_H;

//成员函数

public:

//设置和获取长宽高

void SetL(int l)

{

m\_L = l;

}

void SetW(int w)

{

m\_W = w;

}

void SetH(int h)

{

m\_H = h;

}

int GetL()

{

return m\_L;

}

int GetW()

{

return m\_W;

}

int GetH()

{

return m\_H;

}

//获取面积和体积

int Calculate\_S()

{

return (m\_L \* m\_W + m\_W \* m\_H + m\_L \* m\_H) \* 2;

}

int Calculate\_V()

{

return m\_L \* m\_W \* m\_H;

}

//2.2定义一个成员函数判断两个立方体是否相等

bool IsEquel\_c(Cube& c2)//传入另一个立方体与本立方体比较

{

if (m\_L == c2.GetL()

&& m\_W == c2.GetW()

&& m\_H == c2.GetH())

{

return true;

}

return false;

}

};

//2.1定义一个全局函数判断两个立方体是否相等

bool IsEquel\_g(Cube& c1, Cube& c2)//引用的方式，不另外开辟空间

{

if (c1.GetL() == c2.GetL()

&& c1.GetW() == c2.GetW()

&& c1.GetH() == c2.GetH())

{

return true;

}

return false;

}

int main()

{

//1.定义立方体类，设置长宽高，获取面积和体积

Cube c1;

c1.SetL(10);

c1.SetW(9);

c1.SetH(8);

cout << "c1立方体的面积为：" << c1.Calculate\_S() << endl;

cout << "c1立方体的体积为：" << c1.Calculate\_V() << endl;

Cube c2;

c2.SetL(10);

c2.SetW(9);

c2.SetH(8);

cout << "c2立方体的面积为：" << c2.Calculate\_S() << endl;

cout << "c2立方体的体积为：" << c2.Calculate\_V() << endl;

if (IsEquel\_g(c1, c2))

{

cout << "全局函数：c1立方体等于c2立方体" << endl;

}

else

{

cout << "全局函数：c1立方体不等于c2立方体" << endl;

}

if (c1.IsEquel\_c(c2))

{

cout << "成员函数：c1立方体等于c2立方体" << endl;

}

else

{

cout << "成员函数：c1立方体不等于c2立方体" << endl;

}

return 0;

}

**案例二：**设计一个圆类，包含圆心与半径，再判断一个点与圆的位置关系。点与圆心的距离与半径的关系。重点知识：

（1）圆类中包含点类，类的包含；

（2）程序的分文件编写。.h文件中实现类的定义，成员函数和全局函数的声明。.cpp文件中实现成员函数和全局函数的定义。主函数中实现参数的定义及成员函数和全局函数的调用与传参。

4.2 对象的初始化和清理

概述：生活中的一些电子产品会有恢复出产设置或者数据的删除与清理。

同样，C++中的面向对象来源于生活，每个对象也都会有初始设置以及对象销毁前的清理数据的设置。引入构造函数和析构函数的概念。

4.2.1 构造函数和析构函数

对象的初始化和清理是两个非常重要的安全问题。C++利用构造函数和析构函数解决上述问题，这两个函数由编译器自动调用，如果我们自己写了，则系统调用我们写的函数；否则系统调用默认的空实现。

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。

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。

（1）构造函数

语法： 函数形式：类名(){}

a.没有返回值也不写void

b.函数名称与类名相同

c.构造函数可以有参数，因此可以发生重载

d.程序在调用对象时会自动调用构造，无须手动调用，且只调用一次。

（2）析构函数

语法： 函数形式：~类名(){}

a.没有返回值也不写void

b.函数名称与类名相同，在名称前加上符号~

c.析构函数没有参数，因此不可以发生重载

d.程序在对象销毁时会自动调用析构，无须手动调用，且只调用一次。

#include<iostream>

//构造函数和析构函数

using namespace std;

class Person

{

public://设置为公共权限

//1.构造函数

Person()

{

cout << "Person构造函数的调用" << endl;

}

//2.析构函数

~Person()

{

cout << "Person析构函数的调用" << endl;

}

};

void test()

{

Person p;//创建对象会自动调用构造函数 栈区，对象销毁会自动调用析构函数

}

int main()

{

test();

return 0;

}

4.2.2 构造函数的分类及调用

（1）分类

a.无参构造（默认构造）和有参构造

b.普通构造和拷贝构造，拷贝构造将传入对象的所有属性拷贝到次类中。表达形式： 类名(const 某一类名 &p){}//const修饰，且用引用的方式传递。

//拷贝构造 将其他对象的属性全部拷贝到本类中

Person(const Person& p)

{

age = p.age;

}

（2）调用

#include<iostream>

//构造函数和析构函数

using namespace std;

class Person

{

public://设置为公共权限

//1.构造函数 无参构造 普通构造

Person()

{

cout << "Person无参构造函数的调用" << endl;

}

Person(int a)//有参构造

{

cout << "Person有参构造函数的调用" << endl;

age = a;

}

//拷贝构造 将其他对象的属性全部拷贝到本类中

Person(const Person& p)

{

cout << "Person拷贝构造函数的调用" << endl;

age = p.age;

}

//2.析构函数

~Person()

{

cout << "Person析构函数的调用" << endl;

}

int age;

};

void test()

{

//构造函数调用的方法

//1.括号法

//Person p1;//无参构造 注1：调用无参构造时，不能加括号，因为会被认为是一个函数的声明

//Person p2(10);//有参构造

//Person p3(p2);//拷贝构造

//2.显示法

//Person p1;

//Person p2 = Person(10);//Person(10)匿名对象，本语句结束后匿名对象就销毁了 p2给了它一个名字

//Person p3 = Person(p2);//注2：不要用拷贝构造函数创建匿名对象，编译器认为Person(p3) === Person p3;

//3.隐式转换法

Person p2 = 10;//自动转换成有参构造

Person p3 = p2;//自动转换成拷贝构造

cout << "age为" << p3.age << endl;

}

int main()

{

test();

return 0;

}

4.2.3 拷贝构造函数调用的时机

（1）用已有的对象创建一个新的对象，拷贝属性

（2）函数的传值调用会有拷贝数据

（3）函数的返值时也有数据的拷贝。

4.2.4 构造函数的调用规则

（1）创建一个类，编译器自动产生三个默认函数，默认无参构造函数（空实现）、析构函数（空实现）、拷贝构造函数（值拷贝）。

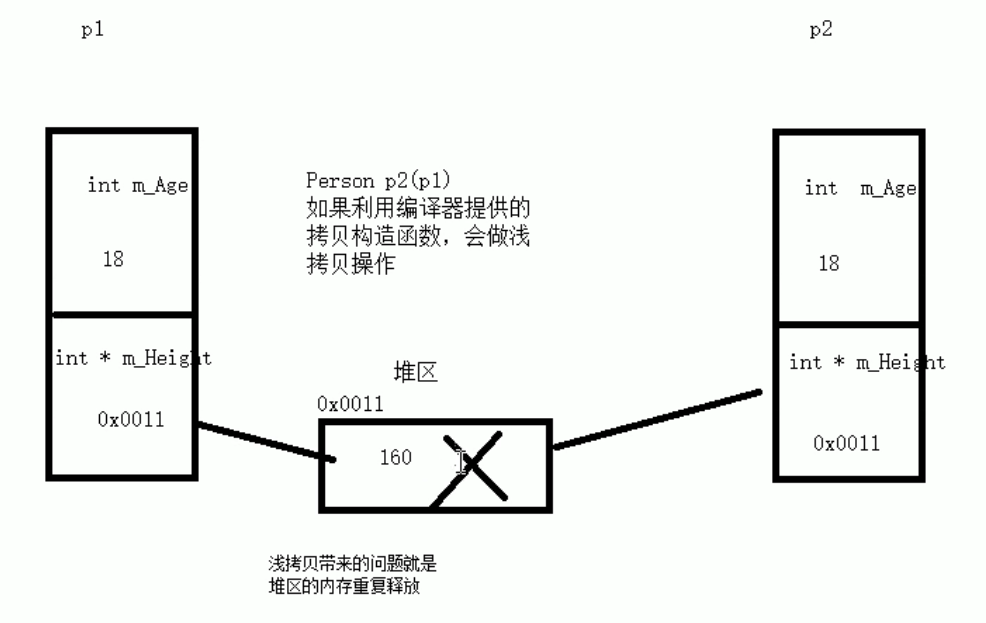
（2）如果我们写了有参构造函数，则编译器不再生成默认无参构造函数，但会生成拷贝构造函数；如果我们写了拷贝构造函数，则编译器不再生成默认构造函数和有参构造函数。

4.2.5 构造函数的浅拷贝与深拷贝

深浅拷贝问题是面试的经典问题，也是常见的一个坑。

浅拷贝：简单的赋值操作，值拷贝。

深拷贝：在堆区重新申请空间，进行拷贝操作。



//浅拷贝与深拷贝

class Person

{

public://设置为公共权限

//1.构造函数 无参构造 普通构造

Person()

{

cout << "Person无参构造函数的调用" << endl;

}

Person(int age, int height)//有参构造

{

cout << "Person有参构造函数的调用" << endl;

m\_age = age;

//堆区申请内存存放高度信息

m\_height = new int(height);

}

//如果不定义拷贝构造函数，编译器会自动生成一个值拷贝构造函数

//自己定义一个深拷贝的构造函数

Person(const Person& p)

{

cout << "Person深拷贝构造函数的调用" << endl;

m\_age = p.m\_age;

m\_height = new int(\*p.m\_height);

}

//2.析构函数

~Person()

{

//析构函数中对堆区申请的内存进行释放

if (m\_height != NULL)

{

delete m\_height;

m\_height = NULL;

}

cout << "Person析构函数的调用" << endl;

}

int m\_age;

int\* m\_height;//高度的指针，定义在堆区

};

void test()

{

Person p1(18, 160);

cout << "age = " << p1.m\_age << " height = " << \*p1.m\_height << endl;

//Person p2(p1);//默认的拷贝构造函数 值拷贝，浅拷贝，将指针的值拷贝到另一个对象

// 当对象析构时，会造成内存的重复释放，发生错误

//cout << "age = " << p2.m\_age << " height = " << \*p2.m\_height << endl;

Person p2(p1);//

cout << "age = " << p2.m\_age << " height = " << \*p2.m\_height << endl;

}

int main()

{

test();

return 0;

}

4.2.6 初始化列表

class Person

{

public:

//初始化列表语法

Person(int a, int b, int c) : m\_a(a), m\_b(b), m\_c(c)

{

cout << "Person 有参构造函数的调用 " << endl;

}

int m\_a;

int m\_b;

int m\_c;

void Print()

{

cout << "m\_a = " << m\_a << endl;

cout << "m\_b = " << m\_b << endl;

cout << "m\_c = " << m\_c << endl;

}

};

int main()

{

Person p(10, 20, 30);

p.Print();

return 0;

}

4.2.7 其他类的对象作为本类的成员

构造时，先构造其他类，再构造本类；析构时，与构造相反。

//手机类

class Phone

{

public:

Phone(string Pname) :m\_pname(Pname)

{

cout << "Phone 构造函数调用" << endl;

}

~Phone()

{

cout << "Phone 析构函数调用" << endl;

}

string m\_pname;

};

//其他类对象作为本类的成员

//人类

class Person

{

public://Phone m\_Pname = pname;//隐式转换法

Person(string name, string pname) :m\_name(name), m\_Pname(pname)

{

cout << "Person 构造函数调用" << endl;

}

~Person()

{

cout << "Person 析构函数调用" << endl;

}

string m\_name;

Phone m\_Pname;

};

int main()

{

Person p("张三", "苹果X");

cout << p.m\_name << "拿着一个" << p.m\_Pname.m\_pname << endl;

return 0;

}

4.2.8 静态成员

由static修饰的类的成员变量和成员函数。

（1）static修饰成员变量

a.类定义的所有对象的这一成员共享（共用一块内存空间）（对象访问或类名访问）

b.在编译阶段分配内存，因为改变了变量的存储区，静态区，代码运行前分配内存。

c.类内声明，类外初始化，同样受访问权限的影响。

//类的静态成员变量

class Person

{

public:

static int m\_A;

};

//类内声明，类外初始化

int Person::m\_A = 100;

int main()

{

//静态成员变量，共用一块空间，一改变则都改变

Person p;

cout << "m\_A = " << p.m\_A << endl;

Person p2;

p2.m\_A = 300;

cout << "m\_A = " << p.m\_A << endl;

//由于内存共用，所以静态成员变量的访问方式有两种

//1.对象访问

cout << "m\_A = " << p2.m\_A << endl;

//2.类名访问

cout << "m\_A = " << Person::m\_A << endl;

return 0;

}

（2）静态成员函数

a.所有对象共享同一个函数（对象访问或类名访问）

b.静态成员函数只能访问静态成员变量（共享函数）

c.受访问权限影响。

4.3 C++对象模型和this指针

4.3.1 对象特性

成员变量与成员函数是分开存储的

注：只有非静态成员变量属于类对象，静态成员变量、非静态成员函数和静态成员函数都不属于类对象，即对象的内存大小不包含这些。非静态函数只会诞生一份函数实例，即多个对象共用一块代码，用this指针区分。

class Person

{

int m\_A;//非静态成员变量

static int m\_B;//静态成员变量不属于类对象

void func1() {}//非静态成员函数不属于类对象

static void func2(){}//静态成员函数不属于类对象

};

//类内声明，类外初始化

int Person::m\_B = 100;

int main()

{

Person p;

//空对象占用内存空间为1，编译器会给每个空对象分配一个字节空间，以区分空对象占用的位置

//每一个空对象应该有独一无二的空间

/\*cout << "sizeof p = " << sizeof(p) << endl;\*/

cout << "sizeof p = " << sizeof(p) << endl;//类似结构体内存大小

return 0;

}

4.3.2 this指针概念

C++通过提供特殊的对象指针，解决成员函数调用对象不确定的问题。This指针指向**被调用的成员函数**所属的对象。

**注：**this指针是一个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指针是一个指针向量，指向不能被修改，指向的内容可以修改。

作用：

（1）当形参与成员变量同名使，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，可用return \*this，并用引用的方式返回，可以保持对象本身。如果不是引用方式，则只是值拷贝一份新的对象。

class Person

{

public:

Person(int age)

{

this->age = age;//用this指针解决成员变量与形参同名问题，this指针相当于对象指针。

}

int age;

Person& PersonAddAge(int age)

{

this->age += age;

return \*this;//引用的方式传回\*this，获得对象本身。

}

};

int main()

{

Person p1(10);

p1.PersonAddAge(10).PersonAddAge(10).PersonAddAge(10).PersonAddAge(10).PersonAddAge(10);//链式编程思想，追加

cout << "p1的年龄为：" << p1.age << endl;//也是一种特殊的链式编程追加

return 0;

}

4.3.3 空指针调用成员函数

C++允许空指针也调用成员函数，但要注意this指针。

class Person

{

public:

void PersonShowClass()

{

cout << "这是一个Person类" << endl;

}

void ShowAge()

{

if (this == NULL)

{

cout << "this指针为空" << endl;

return;

}

cout << "m\_Age = " << this->m\_Age << endl;//当空指针调用函数时，相当于this指针为空，访问出错

}

int m\_Age;

};

int main()

{

Person\* p = NULL;//类指针

p->PersonShowClass();//允许空指针访问

p->ShowAge();

return 0;

}

4.3.4 const修饰成员函数

常函数：

a.成员函数后加const我们称这个函数为常函数

b.常函数内不可以修改成员属性

c.成员属性声明时加关键字mutable后，常函数中依然可以修改成员属性。

常对象：

a.声明对象前加const称改对象为常对象

b.常对象不可以修改成员属性，加上mutable就可以的

c.常对象只能调用常函数。

//常函数与常对象

class Person

{

public:

//成员函数后面加const变为常函数。this指针本质上是一个指针常量，即\* const this指向不能改变

//常函数的本质是将this的类型变为const \* const this即指向的值也不能被改变

//关键字mutable

void ChangeAge() const

{

//this->m\_Age = 100;

this->n\_Age = 100;

}

int m\_Age;

mutable int n\_Age;

void func1()

{

}

};

int main()

{

Person p;

p.ChangeAge();

cout << "p的年龄为: " << p.n\_Age << endl;

const Person p1;//常对象

p1.ChangeAge();//常对象只能调用常函数 因为常对象不能修改成员属性，

//如果成员函数中存在修改成员属性则与常对象性质冲突

//p1.func1();//

cout << "p1的年龄为: " << p1.n\_Age << endl;

return 0;

}

4.4 友元

生活中的家有客厅（public），也有卧室（private），但是卧室可以由好朋友或好闺蜜进入。

C++中，通过友元技术使一个函数或者类 访问另一个类中的私有成员。

友元的关键字为friend，友元的三种实现

（1）全局函数做友元

//1.全局函数做友元

class House

{

friend void GoodGay();//友元函数声明

public:

House()

{

sittingRoom = "客厅";

bedRoom = "卧室";

}

string sittingRoom;

private:

string bedRoom;

};

//全局函数

void GoodGay()

{

House h;

cout << "好朋友正在访问：" << h.sittingRoom << endl;

cout << "好朋友正在访问：" << h.bedRoom << endl;

}

int main()

{

GoodGay();

return 0;

}

（2）类做友元

//2.类做友元

class House

{

friend class GoodGay;

public:

House()

{

sittingRoom = "客厅";

bedRoom = "卧室";

}

string sittingRoom;

private:

string bedRoom;

};

//类

class GoodGay

{

public:

GoodGay()//构造函数 堆区创建一个House类的内存

{

house = new House;

}

House\* house;

void visit()

{

cout << "好朋友正在访问：" << this->house->sittingRoom << endl;

cout << "好朋友正在访问：" << this->house->bedRoom << endl;

}

};

int main()

{

GoodGay g;//默认构造一个House\* house; house又默认构造函数

g.visit();

return 0;

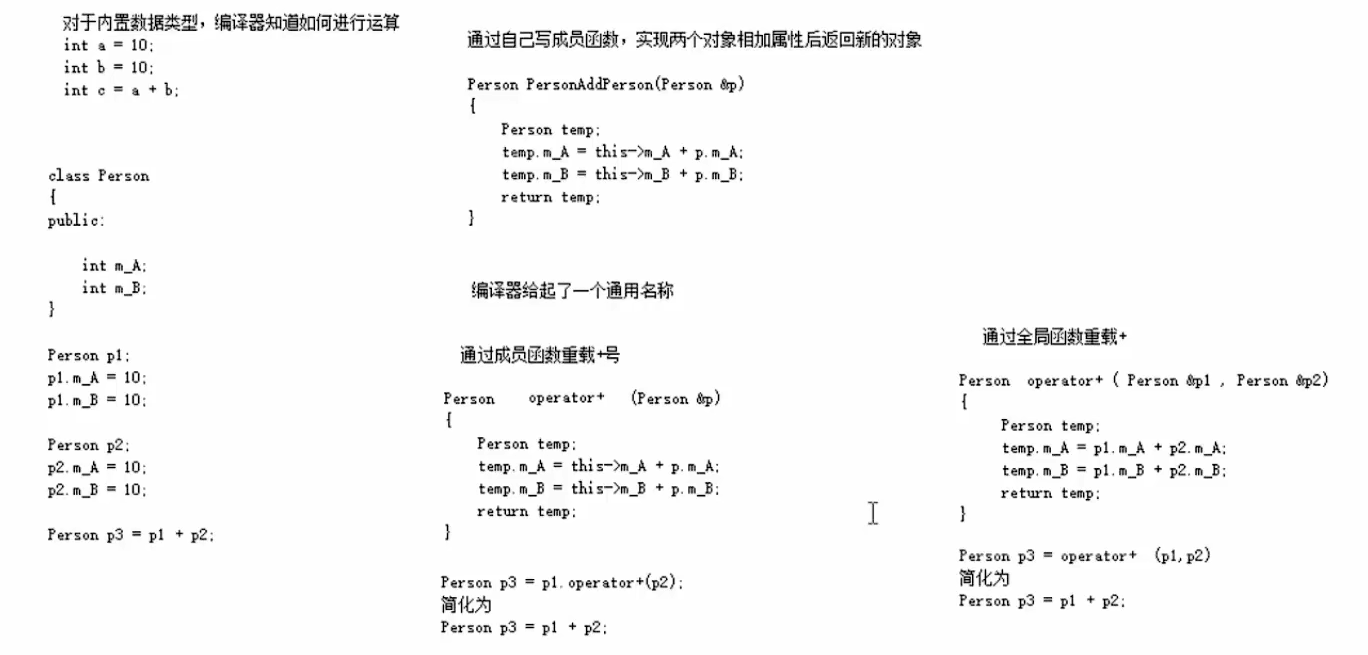
}

（3）成员函数做友元 friend void GoodGay::vist();//成员函数做友元

4.5 运算符的重载

将已经学到的运算符进行重新定义，以满足不同的数据类型。如：加法运算符可以实现两个整数的加法，但两个类的加法呢，就会用到运算符的重定义。

4.5.1 加法运算符的重载



通过成员函数进行重载，或者全局函数进行重载，函数体是我们实现的逻辑，函数名用编译器内置的operator+。运算符重载时也可以发生函数重载。

//加号运算符重载

class Person

{

public:

//1.成员函数重载

//Person PersonAddPerson(Person & p)

//Person operator+ (Person& p)//运算符通用函数名

//{

// Person tmp;

// tmp.m\_A = m\_A + p.m\_A;

// tmp.m\_B = m\_B + p.m\_B;

// return tmp;

//}

int m\_A;

int m\_B;

};

//2.全局函数重载

//Person PersonAddPerson1(Person& p1, Person& p2)

Person operator+ (Person& p1, Person& p2)

{

Person tmp;

tmp.m\_A = p1.m\_A + p2.m\_A;

tmp.m\_B = p1.m\_B + p2.m\_B;

return tmp;

}

int main()

{

Person p1;

p1.m\_A = 10;

p1.m\_B = 20;

Person p2;

p2.m\_A = 30;

p2.m\_B = 40;

//Person p3 = PersonAddPerson1(p1, p2);//Person + Person类型 加法重载

Person p3 = p1 + p2;//运算符重载后直接写即可

cout << "p3.m\_A = " << p3.m\_A << endl;

cout << "p3.m\_B = " << p3.m\_B << endl;

return 0;

}

4.5.2 左移运算符重载

作用：左移运算符重载，不同数据类型输出。如cout << Person p << endl;

只能通过全局函数重载左移运算符。

ostream &operator << (ostream &cout, Person p)

{

Return cout;//链式编程

}

配合友元可以访问私有成员

4.5.3 递增运算符重载++ --

作用：实现自己的整型数据运算。

使用对象运算，运算符重载，注意函数是否有返回值。返回引用链式编程。

MyInteger& operator++()//前置递增 返回引用

{

}

MyInteger operator++(int)//后置递增 返回值 int为占位参数

{

}

4.5.4 赋值运算符重载

作用：对属性进行值拷贝。Operator=

//赋值运算符重载 深拷贝

class Person

{

public:

Person(int a)//构造函数

{

m\_A = new int(a);

}

Person& operator=(Person& p)//防止重复释放同一块内存深拷贝

{

if (m\_A != NULL)//把原本堆区数据释放

{

delete m\_A;

m\_A = NULL;

}

m\_A = new int(\*p.m\_A);

return \*this;//返回对象本身

}

~Person()//析构函数

{

if (m\_A != NULL)

{

delete m\_A;

m\_A = NULL;

}

}

int\* m\_A;

};

int main()

{

Person p1(10);

Person p2(30);

Person p3(50);

p1 = p2 = p3;//重定义赋值运算符 链式编程

cout << "p1.m\_A = " << \*p1.m\_A << endl;

cout << "p2.m\_A = " << \*p2.m\_A << endl;

cout << "p3.m\_A = " << \*p3.m\_A << endl;

return 0;

}

4.5.5 关系运算符重载== >=

某种对象的关系对比。Bool operator==(Person& p){}

4.5.6 函数调用运算符重载()

函数调用运算符重载()----仿函数（类似函数，但又不完全是函数）。

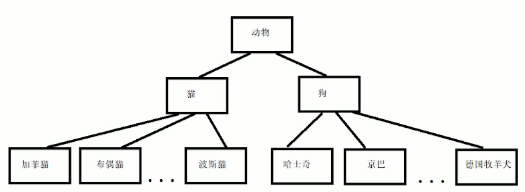
用对象名()作为仿函数。STL中存在。

匿名函数对象，类名+括号Person()()匿名仿函数对象

4.6 继承

概述：

继承是面向对象三大特性之一，类与类之间存在特殊的关系：



我们发现，定义这些类时，下级别的成员除了拥有上一级别的共性，还有自己的特性。这时候可以考虑用继承的技术，减少重复代码。

4.6.1 继承的基本语法

公共的内容可以继承。如网页的头，底部等。

**语法：Class 子类: public(继承方式) 父类**

子类也称派生类，子类中包含父类中继承的成员，也包括自身的特性成员，父类又称基类。

#include<iostream>

#include<string>

//继承

using namespace std;

//父类 基类 公共的内容

class BaseContent

{

public:

void header()

{

cout << "首页、登录、注册。。。（公共头部）" << endl;

}

void floor()

{

cout << "联系方式、交流中心。。。（公共底部）" << endl;

}

void left()

{

cout << "Java、Python、CPP。。。（公共导航栏）" << endl;

}

};

//子类 派生类

class Java :public BaseContent

{

public:

void content()

{

cout << "Java视频链接" << endl;

}

};

class Python :public BaseContent

{

public:

void content()

{

cout << "Python视频链接" << endl;

}

};

class CPP :public BaseContent

{

public:

void content()

{

cout << "CPP视频链接" << endl;

}

};

int main()

{

Java ja;

ja.header();

ja.floor();

ja.left();

ja.content();

cout << "-------------------------" << endl;

Python py;

py.header();

py.floor();

py.left();

py.content();

cout << "-------------------------" << endl;

CPP cpp;

cpp.header();

cpp.floor();

cpp.left();

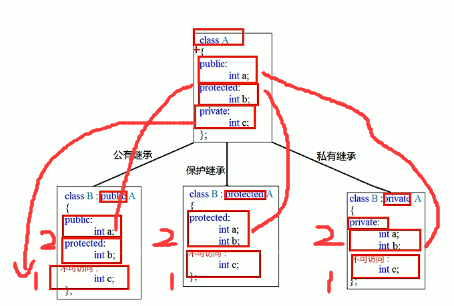
cpp.content();

return 0;

}

4.6.2继承方式

公共继承、保护继承、私有继承。



class Java :public BaseContent

class Java :protect BaseContent

class Java :private BaseContent

4.6.3 继承中的对象模型

父类中所有的非静态成员属性（不管访问权限）都会被继承下去，父类中的私有属性，是编译器默认隐藏了，访问不到，但确实是被继承下去了的。

利用开发人员命令提示工具查看对象模型。

4.6.4 继承中的构造和析构顺序

父类先构造，子类后构造，析构的顺序与构造的顺序相反。

4.6.5 继承中同名成员的处理方式

（1）访问子类同名成员 直接访问即可

子类名.成员

（2）访问父类中的同名成员 需要加作用域 父类名::

子类名.父类名::成员

注：如果出现子类和父类同名的成员，则子类的成员隐藏父类中的所有同名成员（包括重载），要想访问父类中的同名成员，必须加作用域。

4.6.6 继承中同名的静态成员的处理

（1）访问子类同名成员 直接访问即可

子类名.成员

（2）访问父类中的同名成员 需要加作用域 父类名::

子类名.父类名::成员

4.6.7 多继承

**语法：Class 子类: public(继承方式) 父类1，public(继承方式) 父类2**

实际项目中一般不用。