在类的继承中，派生类的构造函数除了初始化自己的数据成员外，还要初始化基类。

向基类构造函数中传递实参：

例：程序class\_test20

// 继承中的构造函数

#include <iostream>

using namespace std;

class animal

{

public:

animal(int height,int weight) // 构造函数

{

cout << "animal construct" << endl;

}

~animal() // 析构函数

{

cout << "animal destruct" << endl;

}

public:

void eat()

{

cout << "animal eat" << endl;

//breathe();

}

void sleep()

{

cout<<"animal sleep"<<endl;

}

protected:

void breathe()

{

cout << "animal breathe" <<endl;

}

};

class fish: public animal

{

public:

fish() : animal(400,300)//基类的构造函数带有了参数，需要给基类的构造函数传参

{

cout << "fish construct" << endl;

breathe();

}

~fish()

{

cout << "fish destruct" << endl;

}

};

int main(int argc, char\* argv[])

{

fish fh;

return 0;

}

输出为：

animal construct

fish construct

animal breathe

fish destruct

animal destruct

可以看出：在创建派生类fish的对象时，首先调用基类animal的构造函数，然后调用fish的构造函数，而析构时，正好相反。

派生类对基类进行初始化时，只初始化自己的直接基类，不初始化在多重继承中的间接基类。

在类的继承中，派生类可以访问基类的public成员和protected成员，但不应该在派生类的构造函数中去初始化基类的public成员和protected成员。

拷贝构造函数、赋值运算符重载与继承

例：程序class\_test23

// 拷贝构造函数，赋值运算符重载与继承

#include <iostream>

using namespace std;

class Base

{

public:

Base()

{

number\_ = 1;

cout << "Base的构造函数" << endl;

}

Base(const Base& b)

{

number\_ = b.number\_;

cout << "Base的拷贝构造函数" << endl;

}

Base& operator=(const Base& rhs)

{

cout << "Base的赋值运算符重载" << endl;

return \*this;

}

void Print()

{

cout << "number = " << number\_ << endl;

}

Base& SetNumber(int n)

{

number\_ = n;

return \*this;

}

private:

int number\_;

};

class Derive : public Base

{

public:

Derive()

{

cout << "Derive的构造函数" << endl;

}

Derive(const Derive& d) : Base(d) // 派生类的拷贝构造函数应该显式调用基类的拷贝构造函数来初始化基类部分

{

cout << "Derive的拷贝构造函数" << endl;

}

// 如果不调用基类的拷贝构造函数，也不会报错，在拷贝构造时，

// 会调用基类的构造函数去构造派生类的基类部分，这是一个不完全的拷贝，

// 被拷贝对象的基类部分不会被拷贝过来

//Derive(const Derive& d)

//{

// cout << "Derive的拷贝构造函数" << endl;

//}

Derive& operator=(const Derive& rhs)

{

if (this != &rhs) // 防止自身赋值

{

Base::operator=(rhs); // 调用基类的赋值运算符初始化基类部分

}

cout << "Derive的赋值运算符重载" << endl;

return \*this;

}

};

int main(int argc, char\* argv[])

{

Base b;

Base b1(b);

Base b2;

b2 = b;

Derive d;

d.SetNumber(10);

Derive d1(d);

// 输出为

// 如果派生类Derive在拷贝构造函数不显式调用基类Base的拷贝构造函数，

// 则输出为1，d1对象就没有完全拷贝d

d1.Print();

Derive d2;

d2 = d;

return 0;

}

在C++继承中，派生类如果定义了拷贝构造函数和赋值运算符的重载，应该显式去调用基类的拷贝构造函数和赋值运算符重载，否则在使用时，就会发生不完全拷贝的现象，即被拷贝对象的基类部分不会被拷贝。

在类继承中，析构函数的执行顺序与构造函数正好相反，先执行派生类的析构函数，再执行基类的析构函数。

虚析构函数：

在类继承中，如果给一个基类指针动态分配了一个派生类对象，则基类的析构函数需要定义为虚函数，这样在删除这个基类指针时，能够调用派生类的析构函数，来删除其中的派生类部分。

用来作基类的类，才会将析构函数声明为虚函数。

当基类的析构函数为虚函数时，则派生类中的析构函数也为虚函数。

例：程序class\_test24

// 虚析构函数

#include <iostream>

class Base

{

public:

Base()

{

std::cout << "执行Base的构造函数" << std::endl;

}

virtual ~Base()

{

std::cout << "执行Base的析构函数" << std::endl;

}

};

class Derive : public Base

{

public:

Derive()

{

std::cout << "执行Derive的构造函数" << std::endl;

}

~Derive()

{

std::cout << "执行Derive的析构函数" << std::endl;

}

};

int main(int argc, char\* argv[])

{

Base\* b\_ptr = new Derive();

delete b\_ptr; // 动态创建的类对象，必须使用delete删除，否则不会调用析构函数

b\_ptr = NULL;

return 0;

}

输出为：

执行Base的构造函数

执行Derive的构造函数

执行Derive的析构函数

执行Base的析构函数

可以看出：在删除Base指针时，调用了Derive的析构函数，其会自动调用Base的析构函数。如果Base的析构函数不为虚函数，则不会调用Derive的析构函数，从而会导致资源泄漏。