析构函数没有返回值，没有任何参数，由编译器自动调用。

析构函数的执行顺序与构造函数相反。

析构函数可以是虚函数。

类对象撤销时会自动调用析构函数。

void func()

{

Student s;

}

在遇到右花括号时，会运行类对象s的析构函数。

动态分配的类指针（通过new操作符实现），只有删除指针（delete）时才会调用析构函数，否则不会运行析构函数，从而导致内存泄露。

例：程序class\_test12

// 析构函数

#include <iostream>

class A

{

public:

A()

{

std::cout << "执行构造函数" << std::endl;

}

~A()

{

std::cout << "执行析构函数" << std::endl;

}

};

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

{

A\* p = new A;

delete p; // 删除p时执行析构函数，否则编译器不会自动执行析构函数

p = NULL;

return 0;

}

vector容器中存放类对象时，构造函数和析构函数的执行要复杂很多。

#include <iostream>

#include <vector>

using std::vector;

class A

{

public:

A()

{

std::cout << "执行构造函数" << std::endl;

}

A(const A& a)

{

std::cout << "执行拷贝构造函数" << std::endl;

}

~A()

{

std::cout << "执行析构函数" << std::endl;

}

};

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

{

A a1;

vector<A> vec;

vec.push\_back(a1);

std::cout << "vec size = " << vec.size() << std::endl;

std::cout << "vec capacity = " << vec.capacity() << std::endl;

vec.push\_back(a1);

std::cout << "vec size = " << vec.size() << std::endl;

std::cout << "vec capacity = " << vec.capacity() << std::endl;

vec.push\_back(a1);

std::cout << "vec size = " << vec.size() << std::endl;

std::cout << "vec capacity = " << vec.capacity() << std::endl;

return 0;

}

输出的结果如下：

执行构造函数 // 构造类对象a1

执行拷贝构造函数 // 第一次执行push\_back操作，调用拷贝构造函数，在// vector容器中存储的是对象的副本。

vec size = 1

vec capacity = 1

执行拷贝构造函数 // 第二次执行push\_back操作时，由于capacity为1，// 所以vector容器会重新申请内存空间，并将原来的

// 元素拷贝到新的内存块中，然后销毁掉原内存块中

// 的对象，所以会调用两次拷贝构造函数，并调用一

// 次析构函数

执行拷贝构造函数

执行析构函数

vec size = 2

vec capacity = 2

执行拷贝构造函数 // 第三次执行push\_back操作时，由于capacity为2，// 所以vector容器会重新申请内存空间，并将原来的

// 元素拷贝到新的内存块中，然后销毁掉原内存块中

// 的对象，所以会调用三次拷贝构造函数，并调用二

// 次析构函数

执行拷贝构造函数

执行拷贝构造函数

执行析构函数

执行析构函数

vec size = 3

vec capacity = 3

执行析构函数 // 析构容器中的vec[2]对象

执行析构函数 // 析构容器中的vec[1]对象

执行析构函数 // 析构容器中的vec[0]对象

执行析构函数 // 析构掉a1对象