浙江大学
ZHEJIANG UNIVERSITY

2022 OOP辅学
五月网课

张道泽

2022.5.28

子类对象的构造销毁

当ctor( )遇到继承……

创建子类对象时，执行顺序：
① 父类成员变量初始化
② 父类ctor( )
③ 子类成员变量初始化
④ 子类ctor( )

当dtor( )遇到继承……

销毁子类对象时，执行顺序：
① 子类dtor( )
② 子类成员变量销毁
③ 父类dtor( )
④ 父类成员变量销毁

The output of the code below is:

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

class A {
public:
        A() { cout << 1; }
} a;

int main()
{
        cout << 2;
        A a;

        return 0;
}
```

The output of the code below is:

```
#include<iostream>
using namespace std;
class AA {
public:
    AA() { cout << 1; }
    ~AA() { cout << 2; }
};
class BB: public AA {
    AA aa;
public:
    BB() { cout << 3; }
    ~BB() { cout << 4; }
};
int main() {
    BB bb;
    return 0;
}
```

virtual function

虚函数：函数调用与函数体之间的绑定关系，在运行时才建立

在运行时才决定如何动作（dynanmic binding 动态联编/动态绑定）

虚函数是动态联编的基础

声明：`virtual<函数类型><函数名>(<arg list>)`

动态绑定的条件：

- **父类声明虚函数**，**子类重写**了它
- 通过**指针/引用**来调用该函数

virtual function

```
class Grandam {
    public:
        virtual void introduce_self()
        { cout<<"I am grandam."<<endl; }
};

class Mother:public Grandam {
    public:
        void introduce_self
        { cout<<"I am mother."<<endl;}
};

class Daughter:public Mother {
    public:
        void introduce_self()
        { cout<<"I am daughter."<<endl;}
};
```

```
void main()
{
    Grandam* ptr;
    Grandam g;
    Mother m;
    Daughter d;

    ptr=&g;
    ptr->introduce_self();

    ptr=&m;
    ptr->introduce_self();

    ptr=&d;
    ptr->introduce_self();
}
```

virtual function

```
class Grandam {
    public:
        virtual void introduce_self()
        { cout<<"I am grandam."<<endl; }
};

class Mother:public Grandam {
    public:
        void introduce_self
        { cout<<"I am mother."<<endl;}
};

class Daughter:public Mother {
    public:
        void introduce_self()
        { cout<<"I am daughter."<<endl;}
};
```

```
void main()
{
    Grandam* ptr;
    Grandam g;
    Mother m;
    Daughter d;

    ptr=&g;
    ptr->introduce_self();

    ptr=&m;
    ptr->introduce_self();

    ptr=&d;
    ptr->introduce_self();
}
```

```
I am grandam.
I am mother.
I am daughter.
```

virtual function

```
class Grandam {
    public:
        virtual void introduce_self()
        { cout<<"I am grandam."<<endl; }
};

class Mother:public Grandam {
    public:
        void introduce_self
        { cout<<"I am mother."<<endl;}
};

class Daughter:public Mother {
    public:
        void introduce_self()
        { cout<<"I am daughter."<<endl;}
};
```

```
void main()
{
    Grandam* ptr;
    Grandam g;
    Mother m;
    Daughter d;

    ptr=&g;
    ptr->introduce_self();

    ptr=&m;
    ptr->introduce_self();

    ptr=&d;
    ptr->introduce_self();
}
```

virtual function

```
class Grandam {
    public:
        virtual void introduce_self()
        { cout<<"I am grandam."<<endl; }
};

class Mother:public Grandam {
    public:
        void introduce_self
        { cout<<"I am mother."<<endl;}
};

class Daughter:public Mother {
    public:
        void introduce_self()
        { cout<<"I am daughter."<<endl;}
};
```

```
void main()
{
    Grandam* ptr;
    Grandam g;
    Mother m;
    Daughter d;

    ptr=&g;
    ptr->introduce_self();

    ptr=&m;
    ptr->introduce_self();

    ptr=&d;
    ptr->introduce_self();
}
```

```
I am grandam.
I am grandam.
I am grandam.
```

virtual function

“同一类族中不同类的对象，对同一函数调用作出不同的响应”

Note：

- 子类重写父类的虚函数时，virtual可加可不加
- 使用 object.method() 也可以调用虚函数，但只能静态联编，不构成多态
- 虚函数是父类的非static的成员函数
- 内联函数、构造函数不能是虚函数；析构函数可以是虚函数

纯虚函数

纯虚函数：

- 特殊的虚函数
- 在父类里：没有实现
- 在子类里：要么子类实现它，要么子类继续声明它是纯虚函数

声明：`virtual<函数类型><函数名>(arg list) = 0`

一个具有纯虚函数的类称为抽象类，抽象类不能被实例化

纯虚函数

```
class Polygon {
protected:
        int width, height;
public:
        void set_values (int a, int b) { width=a; height=b; }
        virtual int area (void) =0;
};

class Rectangle: public Polygon {
public:
        int area (void) { return (width * height); }

};

class Triangle: public Polygon {
public:
        int area (void) { return (width * height / 2); }
};
```

2-2 About virtual function, which statement below is correct? (2分)

A. Virtual function is a static member function
B. Virtual function is not a member function
C. Once defined as virtual, it is still virtual in derived class without virtual keyword,.
D. Virtual function can not be overloaded.

1-4 Dynamic binding is used as default binding method in C++. (1分)

1-2 An abstract class is a class with at least one pure virtual function. (1分)

2-6 Given:

```
class A {
        A() {}
        virtual f() = 0;
        int i;
};
```

which statement below is **NOT** true: (2分)

A. i is private
B. Objects of class A can not be created
C. i is a member of class A
D. sizeof(A) == sizeof(int)

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

class A {
    A() {};
    virtual int f() {};
    int i;
};
int main()
{
    cout << sizeof(A) << endl;
    cout << sizeof(int) << endl;
}
```

```
C:\WIND
8
4
请按任意键
```

```
enum NOTE { middleC, Csharp, Cflat };
class Instrument {
public:
  virtual void play(NOTE) const = 0;
  virtual char* what() const = 0;
  virtual void adjust(int) = 0;
};

class Wind : public Instrument {
public:
  void play(NOTE) const {
    cout << 1 << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(NOTE) const {
    cout << 2 << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};
```

```
class Stringed : public Instrument {
public:
  void play(NOTE) const {
    cout << 3 << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(NOTE) const {
    cout << 11 << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(NOTE) const {
    cout << 12 << endl;
  }
  char* what() const { return "Woodwind"; }
};
```

```
void tune(Instrument& i) {
  i.play(middleC);
}

void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
  return 0;
}
```

```cpp
#include <iostream>

struct Base
{
  virtual ~Base()
  {
    std::cout << "Destructing Base" << std::endl;
  }
  virtual void f()
  {
    std::cout << "I'm in Base" << std::endl;
  }
};

struct Derived : public Base
{
  ~Derived()
  {
    std::cout << "Destructing Derived" << std::endl;
  }
  void f()
  {
    std::cout << "I'm in Derived" << std::endl;
  }
};
```

```cpp
int main()
{
  Base *p = new Derived();
  (*p).f();
  p->f();
  delete p;
}
```

算符重载

```
class Time
{
private:
    int hour, minute, second;
public:
    Time(int h = 0, int m = 0, int s = 0);

    void Show();
    Time& operator++();       //重载前置++运算符
    Time operator++(int);     //重载后置++运算符   后置++作区分
    Time& operator=(const Time& other);    //重载=运算符
};

Time::Time(int h, int m, int s) {
    hour = h; minute = m; second = s;
}
void Time::Show() {
    cout << hour << ":" << minute << ":" << second << endl;
}


------------------------------------------------------------
int main()
{
    Time t1(10, 25, 52), t2, t3;
    cout << "t1   = "; t1.Show();
    cout << "++t1 = "; (++t1).Show();
    cout << "t1++ = "; (t1++).Show();
    cout << "t1   = "; t1.Show();
}
```

```
Time& Time::operator++() {
    second++;
    if (second == 60) {
        second = 0; minute++;
        if (minute == 60) {
            minute = 0; hour++;
            if (hour == 24) hour = 0;
        }
    }
    return *this;
}
Time Time::operator++(int) {
    Time temp = *this;    // 保存原值
    second++;
    if (second == 60) {
        second = 0; minute++;
        if (minute == 60) {
            minute = 0; hour++;
            if (hour == 24) hour = 0;
        }
    }
    return temp;
}
Time& Time::operator=(const Time& other) {
    if (this == &other)
        return *this;
    this->hour = other.hour;
    this->minute = other.minute;
    this->second = other.second;
    return *this;
}
```

算符重载

```
class Time
{
private:
    int hour, minute, second;
public:
    Time(int h = 0, int m = 0, int s = 0);

    void Show();
    Time& operator++();       //重载前置++运算符
    Time operator++(int);     //重载后置++运算符   后置++作区分
    Time& operator=(const Time& other);    //重载=运算符
};

Time::Time(int h, int m, int s) {
    hour = h; minute = m; second = s;
}
void Time::Show() {
    cout << hour << ":" << minute << ":" << second << endl;
}


-------------------------------------------------------------
int main()
{
    Time t1(10, 25, 52), t2, t3;
    cout << "t1   = "; t1.Show();
    cout << "++t1 = "; (++t1).Show();
    cout << "t1++ = "; (t1++).Show();
    cout << "t1   = "; t1.Show();
}
```

```
Time& Time::operator++() {
    second++;
    if (second == 60) {
        second = 0; minute++;
        if (minute == 60) {
            minute = 0; hour++;
            if (hour == 24) hour = 0;
        }
    }
    return *this;
}
Time Time::operator++(int) {
    Time temp = *this;    // 保存原值
    second++;
    if (second == 60) {
        second = 0; minute++;
        if (minute == 60) {
            minute = 0; hour++;
            if (hour == 24) hour = 0;
        }
    }
    return temp;
}
Time& Time::operator=(const Time& other) {
    if (this == &other)
        return *this;
    this->hour = other.hour;
    this->minute = other.minute;
    this->second = other.second;
    return *this;
}
```

```
t1   = 10:25:52
++t1 = 10:25:53
t1++ = 10:25:53
t1   = 10:25:54
```

算符重载

new, delete算符重载

```
class rect
{
private:
    int  length, width;
public:
    rect(int l, int w)
    { length = l; width = w; }
    void *operator new ( size_t  size)  //size_t即unsigned integer
    { return  malloc( size ) ; }

    void operator delete( void *p )
    { free( p ) ; }

    void disp( )
    { cout << "area: " << length * width << endl; }
};

void main()
{
    rect  *p;
    p = new rect(5, 9);
    p->disp();
    delete  p;
}
```

算符重载

new, delete算符重载

```
class rect
{
private:
    int  length, width;
public:
    rect(int l, int w)
    { length = l; width = w; }
    void *operator new ( size_t  size)  //size_t即unsigned integer
    { return  malloc( size ) ; }

    void operator delete( void *p )
    { free( p ) ; }

    void disp( )
    { cout << "area: " << length * width << endl; }
};

void main()
{
    rect  *p;
    p = new rect(5, 9);
    p->disp();
    delete  p;
}
```

```
area: 45
```

算符重载

- 无法重载的算符
  - 二元解析运算符 ::
  - 三元条件运算符 ？：

- 只可以重载已有的算符，不能自己新造算符

2-8 Which operator below can not be overloaded? (2分)

A. &&
B. []
C. ::
D. <<

1-3 The operator `::` can not be overloaded. (1分)

```
#include<iostream>
using namespace std;
class A{
public:
        A& operator=(const A& r)
        {
                cout << 1 << endl;
                return *this;
        }
};
class B{
public:
        B& operator=(const B& r)
        {
                cout << 2 << endl;
                return *this;
        }
};
class C{
private:
        B b;
        A a;
        int c;
};

int main()
{
        C m,n;
        m = n;
        return 0;
}
```

```
class counter{
private:
        int value;
public:
        counter():value(0) {}
        counter& operator++();
        int operator++(int);
        void reset()
        {
            value = 0;
        }
        operator int() const
        {
            return value;
        }
};
```

```
counter& counter::operator++()
{
        if (3 == value)
        value = 0;
        else
            value += 1;
        return *this;
}

int counter::operator++(int)
{
        int t = value;
        if (3 == value)
        value = 0;
        else
            value += 1;
        return t;
}
```

```
int main()
{
        counter a;
        while (++a)
             cout << "***\n";
        cout << a << endl;
        while (a++)
             cout << "***\n";
        cout << a << endl;
        return 0;
}
```

```
class String {
private:
    char *m_ptr;
public:
    String(const char *ptr)
    {
        m_ptr = new                                                    (1分);
        strcpy(m_ptr, ptr);
    }
    ~String()
    {
                                                                       (1分);
    }
    String &operator+=(const String &str)
    {
        char *s = new                                                  (1分);
        if (m_ptr)
        {
            strcpy(s, m_ptr);
                                                                       (1分) m_ptr;
        }
       strcat(s, str.m_ptr); // appends str.m_ptr to s
                                                                       (1分) = s;
       return                                                          (1分);
    }
```

Exception

```
int Div(int x, int y)
{
        if (y == 0) throw y;
        return (x / y);
}

int main(void)
{
        try {
                cout << "5/2=" << Div(5, 2) << endl;
                cout << "8/0=" << Div(8, 0) << endl;
                cout << "7/1=" << Div(7, 1) << endl;
        }
        catch (int) {
                cout << "exception of dividing zero." << endl;
        }
        cout << "after catch block." << endl;
        return 0;
}
```

Exception

```
int Div(int x, int y)
{
        if (y == 0) throw y;
        return (x / y);
}

int main(void)
{
        try {
                cout << "5/2=" << Div(5, 2) << endl;
                cout << "8/0=" << Div(8, 0) << endl;
                cout << "7/1=" << Div(7, 1) << endl;
        }
        catch (int) {
                cout << "exception of dividing zero." << endl;
        }
        cout << "after catch block." << endl;
        return 0;
}
```

```
5/2=2
exception of dividing zero.
after catch block.
```

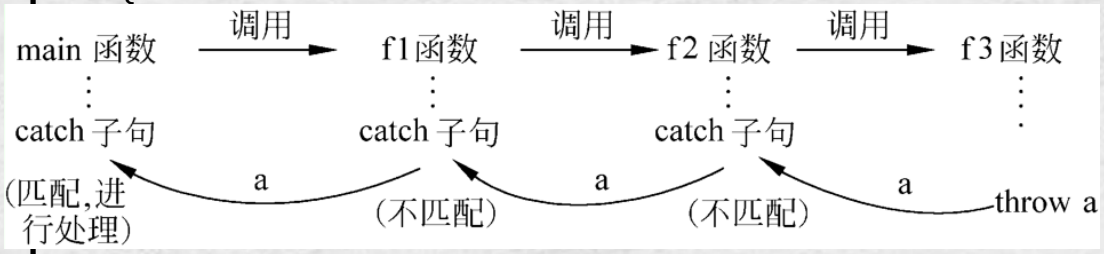
Exception

多层嵌套exception的捕获

```
int main()
{
    void f1();
    try {
        f1();
    }
    catch (double) {
        cout << "catched double exception." << endl;
    }
    cout << "after catch block. (double)" << endl;
    return 0;
}
void f1()
{
    try {
        f2();
    }
    catch (char) {
        cout << "catched char exception.";
    }
    cout << "after catch block. (char)" << endl;
}
```

```
void f2()
{
    try {
        f3();
    }
    catch (int) {
        cout << "catched int exception." << endl;
    }
    cout << "after catch block. (int)" << endl;
}
void f3()
{
    double a = 0;
    try {
        throw a;  // throw double
    }
    catch (float) {
        cout << "catched float exception." << endl;
    }
    cout << "after catch block. (float)" << endl;
}
```

Exception




```
int main()
{
    void f1();
    try {
        f1();
    }
}
void f1()
{
    try {
        f2();
    }
    catch (char) {
        cout << "catched char exception.";
    }
    cout << "after catch block. (char)" << endl;
}
```

```
void f2()
{
    try {
        f3();
    }
    catch (int) {
        cout << "catched int exception." << endl;
    }
    cout << "after catch block. (int)" << endl;
}
void f3()
{
    double a = 0;
    try {
        throw a;  // throw double
    }
    catch (float) {
        cout << "catched float exception." << endl;
    }
    cout << "after catch block. (float)" << endl;
}
```

Exception

```
int main()
{
    void f1();
    try {
        f1();
    }
    catch (double) {
        cout << "catched double exception." << endl;
    }
    cout << "after catch block. (double)" << endl;
    return 0;
}
void f1()
{
    try {
        f2();
    }
    catch (char) {
        cout << "catched char exception.";
    }
    cout << "after catch block. (char)" << endl;
}
```

```
void f2()
{
    try {
        f3();
    }
    catch (int) {
        cout << "catched int exception." << endl;
    }
    cout << "after catch block. (int)" << endl;
}
void f3()
{
    double a = 0;
    try {
        throw a;  // throw double
    }
    catch (float) {
        cout << "catched float exception." << endl;
    }
    cout << "after catch block. (float)" << endl;
}
```

```
catched double exception.
after catch block. (double)
```

Exception

不可达的catch

```
int main(void)
{
	try {
		int a = 0;
		throw a;
	}
	catch (...) {				// any exception
		cout << "exception of any type." << endl;
	}
	catch (char* str) {			// unreachable, error
		cout << "exception of char*: " << str << endl;
	}
	catch (int e) {				// unreachable, error
		cout << "exception of int: " << e << endl;
	}
}
```

Exception

```
int main(void)
{
        try {
                int a = 0;
                throw a;
        }
        catch (...) {                   // any exception
                cout << "exception of any type." << endl;
        }
        catch (char* str) {             // unreachable, error
                cout << "exception of char*: " << str << endl;
        }
        catch (int e) {                 // unreachable, error
                cout << "exception of int: " << e << endl;
        }
}
```

error…

1-1 catch (type p) acts very much like a parameter in a function. Once the exception is caught, you can access the thrown value from this parameter in the body of a catch block.。 (2分)

1-5 If you are not interested in the contents of an exception object, the catch block parameter may be omitted.。 (2分)

2-4 Suppose that statement3 throws an exception of type Exception3 in the following statement: (2分)

```
try {
statement1; statement2; statement3; }
catch (Exception1 ex1) { }
catch (Exception2 ex2) { }
catch (Exception3 ex3) { statement4; throw; }
statement5;
```

Which statements are executed after statement3 is executed?

- A. statement2
- B. statement3
- C. statement4
- D. statement5