Object-Oriented Programming

Akhia[1]

2020 年 10 月 1 日

[1]E-mail:akhialomgir362856@gmail.com

目录

Chapter 1

函数

1.1 引用

引用用 & 标记，用来为储存器提供**别名**。

```
int x;
int& ref = x;
```

分配了一个 int 单元，它拥有两个名字：x, ref。

1.1.1 引用调用

对函数参数用 & 作为引用参数，将获得**引用调用**。在引用调用中，引用参数将实际的实参传给函数，而不是实参的拷贝。而默认的调用方式为传值引用。

```
void swap(int& a,int& b){
    int t;
    t = a;
    a = b;
    b = t;
}
```

使用后，swap 函数的参数直接对应 main 中使用的 i 和 j 的储存空间，而不是拷贝。

1.1.2 引用返回

```
return expression;
```

默认情况下，函数返回时， expression 被求值，并将值拷贝到临时储存空间，即传值返回。

而引用返回中，返回值不再拷贝到临时储存空间，甚至 return 语句所用的储存单元对调用者而言都是可访问的。

```
int& val(){
    return i;
}
```

函数中返回引用 i 的引用，直接将值传入到调用者中。与传值返回不同，仅产生一个副本。

1.2 内联函数

关键字 **inline** 在函数**声明**（不能出现在定义部分）用来请求将函数用内联方式展开，即在每个调用函数的地方插入函数实现代码。

1.3 函数默认参数

所有没有默认值的参数都要放在函数列表的开始部分，然后才是有默认值的函数。

1.4 函数重载

在**参数个数或类型**有区别时，同一范围内允许使用相同签名的函数，称函数名被重载。重载函数用来对具有相似行为而数据类型不同的操作提供一个通用名称。

1.4.1 函数签名

重载函数有不同的**函数签名**：

1. 函数名
2. 参数个数、数据类型和顺序

1.5 new delete 操作符

1. new: 分配一个单元
2. new[]: 分配多个单元（数组）
3. delete: 释放 new 分配的单元
4. delete[]: 释放 new[] 分配的单元

new[] 分配了数组后，将第一个单元的地址保存到指向分配的储存空间的指针中。

1.6 友元

在当前类外定义的、不属于当前类中的函数也可以在类中声明为友元函数，可以访问当前类的**所有**成员。

1. 友元关系是单向的
2. 友元关系不能传递

1.7 异常处理

关键字：

1. try

2. catch

3. throw

catch 块定义在 try 块之后，因为例外在 try 块中抛出。例外和不同捕捉器的匹配依靠**类型**判断来进行。 catch 块提示用户例外后，块中的 continue 将重新返回到 try。 throw 不带任何参数，重新抛出异常给上级处理。

Chapter 2

类

2.1 构造函数

2.1.1 基本特征

构造函数是与类名相同的成员函数。

编译器默认添加：

1. 构造函数
2. 拷贝构造函数
3. 析构函数
4. 赋值运算符函数

1. 返回类型为void
2. 可以重载，但必须有不同的函数署名
3. 默认不带任何参数
4. 创建对象时会隐式调用
5. 用来初始化数据成员
6. 默认构造函数定义在类内
7. 带参构造函数定义在类外

2.1.2 拷贝构造函数

如果不提供，编译器会自动生成：将源对象所有数据成员的值逐一赋值给目标对象相应的数据成员。

```
1    Person(Person&);
2    Person(const Person&);
```

2.1.3 转型构造函数

关闭因转型构造函数导致的隐式类型转换，将运行期错误变成了编译器错误。

```
1    explicit Person(const string& n) {name = n;}
```

2.1.4 构造函数初始化

```
1    class C {
2        public:
3            C() {
4                x = 0;
5                c = 0; //ERROR(CONST)
6            }
7        private:
8            int x;
9            const int c;
10   }
```

对const类型初始化，只需要添加一个**初始化列表：**

```
1    class C {
2        public:
3            C() : c(0) {x = 0;}
4        private:
5            int x;
6            const int c;
7    }
```

初始化段由冒号:开始，c为需要初始化的数据成员，()内是初始值，这是初始化const的**唯一方法**。 **初始化列表**仅在**构造函数**中有效。数据成员的顺序仅取决于类中的顺序，与初始化段中的顺序无关。

2.1.5 new

```
1
2     new constructor[([arguments])]
```

2.2 析构函数

析构函数当对象被**销毁**时，自动调用。

```
1   class C {
2       public:
3           C() {}
4           ~C() {}
5       private:
6           int x;
```

没有**参数**和**返回值**，不能重载。

2.3 指向对象的指针

名称	符号	用途
成员选择操作符	.	对象和对象引用
指针操作符	->	指针访问成员，专用于对象指针

用途：

1. 作为参数传递给函数，通过函数返回

2. 使用new(])操作符动态创建对象，然后返回对象的指针

2.3.1 常量指针 this

常量指针：不能赋值、递增、递减，不能在static成员函数中使用。

避免命名冲突：

```
1    void setID (const string& id) {this->id=id;}
```

2.4 类数据成员和类成员函数

如果不使用 static(静态) 关键字，数据成员和成员函数都是属于**对象**的。而使用 static 则可以创建**类成员**：分为对象成员和实例成员。

2.4.1 类数据成员

静态成员只与类本身有关，与对象无关。它对整个类而言只有一个，而且必须在**任何程序块外**定义。

2.4.2 类成员函数

static 成员函数只能访问其他 static 成员，而非 static 成员都可以访问。同时 static 成员函数也可以是 inline 函数。

可以通过两种方式访问：

```
1    C c1;
2    c1.Meth();
3    C::Meth();
4    unsigned x = c1.var;
5    unsigned y =C::var;
```

但首选用类直接访问，为了说明静态成员直接与类关联。

2.4.3 成员函数中的静态变量

如果将成员函数中的局部变量定义为静态变量，**类的所有对象在调用这个成员函数共享这个变量**。

2.5 友元（friend）类

友元类中的所有成员函数都是目标类中的友元函数。

Chapter 3

继承

3.1 访问控制

	private(default)	protected	public
私有派生	不可访问	私有	私有
受保护派生		受保护	受保护
公有派生		受保护	公有

3.1.1 改变访问限制

使用 using 声明改变访问限制：

```
1  class D : B {
2  private:
3      float y;
4      using B::set_x(); //set_x() is public in B
5  }
```

3.1.2 同名覆盖

又称名字隐藏，如果派生类中添加的数据成员和基类中的数据成员（函数）同名，**派生类中的优先**，并隐藏了继承来的数组成员（函数），且可继承。

3.1.3 间接继承

数据成员可以通过继承链路继承，继承链路包括了基类和所有派生类。继承可以通过直接基类派生，也可以从间接基类派生。

3.2 继承机制下的构造和析构函数

构造函数	基类	$->$	派生类
析构函数	派生类	$->$	基类

3.2.1 继承机制下的构造函数

创建派生对象时，基类的构造函数将被自动调用，对**派生类对象中的基类部分**初始化。而派生类自己的构造函数则负责对象中的**派生类部分**的初始化。

默认构造函数：

1. 没有带明显形参的构造函数

2. 提供了默认实参的构造函数

创建派生类对象时，必须显式（带括号）或隐式（无括号）调用。如果基类有构造函数但没有默认构造函数，派生类的构造函数**必须**显式调用基类的某个构造函数。

3.2.2 继承机制下的析构函数

执行次序与构造函数相反，从派生类到基类。

3.3 多继承

派生类有多个基类，是所有基类的组合体。

3.3.1 虚基类

为解决从同一间接基类继承多次而造成浪费和混淆的问题，使用虚基类继承，从而仅将同名数据成员的一份拷贝发送到派生类。

3.4 继承方式

3.4.1 保护继承

1. 基类中所有公有成员在派生类中是保护成员
2. 基类中所有私有成员在派生类中是保护成员
3. 基类中所有私有成员仅在基类中可见

3.4.2 私有继承

1. 基类中所有公有成员在派生类中是私有成员
2. 基类中所有私有成员在派生类中是私有成员
3. 基类中所有私有成员仅在基类中可见

Chapter 4

多态

4.1 运行期绑定和编译器绑定

多态的函数的绑定是**运行期绑定**而不是**编译期绑定**。

4.1.1 多态的前提

1. 必须存在一个继承体系结构
2. 继承体系结构中的一些类必须拥有同名的 virtual 函数
3. 至少有一个基类类型的指针或基类类型的引用，这个指针或引用可用来对 virtual 成员函数进行调用

基类类型的指针可以指向任何基类对象和派生类对象。虚函数在系统中的运行期才对调用动作绑定。

4.1.2 虚函数成员继承

为了调用同一基类的类族的同名函数，不用定义多个指向各派生类的指针变量，而是使用虚函数用同一方式调用同一类族中不同类的所有同名函数。和普通成员函数相同，在派生类中虚成员函数也可以从基类中继承。

4.1.3 运行期绑定与虚成员函数表

vtable 虚成员函数表用来实现虚成员函数的运行期绑定。用途是支持运行期查询，使系统可以将一函数名绑定到虚成员函数表的特定入口地址。而如果在基类中重新定义，则会调用基类中的成员函数，无法达到多态效果。虚成员函数表需要额外的空间，查询也需要时间。构造函数不能是虚函数，析构函数可以是虚函数。

4.2 重载、覆盖和隐藏

只有虚函数在运行期绑定，是真正的多态。所以编译期多态机制与运行期多态机制相似，但也有区别。

	有效域	函数	同	异
重载	层次类	虚 or 实	函数名	函数签名
覆盖	层次类	虚	函数名 and 签名	
隐藏	顶层 or 同类内成员函数	实	函数名	函数签名
		虚	函数名 and 签名	

1. 重载同类中的成员函数或顶层函数签名不同而同名则称重载，属于编译器绑定。
2. 覆盖派生类中虚成员函数与基类中函数成员同名且签名相同则称覆盖。
3. 隐藏（要避免）同名而函数签名不同的层次类。

4.2.1 名字共享

1. 重载函数名的顶层函数
2. 重载构造函数
3. 同一类中非构造函数名字相同
4. 继承层次中的同名函数（特别是虚函数）

4.3 抽象基类

抽象基类不能实例化抽象基类的对象，用来指明派生类必须覆盖某些虚函数才能拥有对象。

抽象基类：必须拥有至少一个纯虚函数。

4.3.1 纯虚函数

在虚成员函数声明的结尾加上= 0即可将其定义为纯虚函数。

```
virtual void open() = 0;
```

4.3.2 公共接口

抽象基类定义了一个公共接口，被所有从抽象基类派生的类共享。由于抽象基类只含有 public 成员，通常使用关键字 struct 声明抽象基类。

4.4 运行期类型识别（RTTI）

1. 在运行期对类型转换进行检查
2. 在运行期确定对象类型
3. 拓展 RTTI

4.4.1 dynamic_cast 操作符

在运行期对可能引发错误的转型操作进行测试。

static_cast 不做运行时检查，可能不安全。

```
class B {}
class D : public B {}
D* p;
p = new B;
p = static_cast<D*>(new B);
```

而 dynamic_cast 虽然语法相同，但仅对多态类型（至少有一个虚函数）有效。 dynamic_cast 可以实现向上、向下转型。

static_cast 可以施加于任何类型，无论是否有多态性。 dynamic_cast 只能施加于具有多态性的类型，转型的目的类型只能是指针或引用。 dynamic_cast 虽然运行范围没有 static_cast 广，但是只有 dynamic_cast 能进行运行期安全检查。

4.4.2 typeid 操作符

确定**表达式**的类型。

```
1 #inlcude <typeinfo>
```

Chapter 5

操作符重载

5.1 基本操作符重载

可以重载的操作符：

```
new new[]   delete   delete[]
+   -   *   /   %   ^   &
|   ~   !   =   <   >

+=  -=  *=  /=  %=  ^=  &=  |=
<<  >>  <<= >>= ==  !=  <= >=

&&  ||  ++  --  ,   ->  ->*

()  []
```

不能重载的操作符：

```
.   ::   .*   ?:   sizeof
```

调用函数的对象是操作符重载函数的第一个参数，所以：

1. 一元操作符不需要参数
2. 二元操作符需要一个参数

```
class C{
    C operator+(const C&) const;
}
C C::operator+(const C& c) const{
    //~~~
    }
```

5.1.1 操作符优先级和语法

重载不能改变操作符的优先级和语法。

5.2 顶层函数进行操作符重载

被重载的操作符：

1. 类成员函数

2. 顶层函数

以顶层函数的形式重载操作符时，二元操作符重载函数必须有两个参数，一元操作符重载必须有一个参数。

顶层函数实现时，不能直接调用类里面的私有成员，改进的方式是把顶层函数设为该类的友元函数。

使用顶层函数，非对象操作数可以出现在操作符的左边，而使用成员函数时，第一个操作数必须是类的对象。

```
1  int main()
2  {
3      complex c1, c2(15.5, 23.1);
4      c1 = c2 + 13.5;
5      c1 = 13.5 + c2; // 用成员函数重载会报错
6      return 0;
7  }
```

成员函数重载：

```
class complex
{
public:
    complex();
    complex(double a);
    complex(double a, double b);
    complex operator+(const complex & A)const;
    complex operator-(const complex & A)const;
    complex operator*(const complex & A)const;
    complex operator/(const complex & A)const;
    void display()const;
private:
    double real;    //复数的实部
    double imag;    //复数的虚部
};

//~~~

//重载加法操作符
complex complex::operator+(const complex & A)const
{
    complex B;
    B.real = real + A.real;
    B.imag = imag + A.imag;
    return B;
}
```

顶层函数重载：

```
class complex
{
public:
    complex();
    complex(double a);
    complex(double a, double b);
    double getreal() const { return real; }
    double getimag() const { return imag; }
    void setreal(double a){ real = a; }
    void setimag(double b){ imag = b; }
    void display()const;
private:
    double real;    //复数的实部
    double imag;    //复数的虚部
};

//~~~

//重载加法操作符
complex operator+(const complex & A, const complex &B)
{
    complex C;
    C.setreal(A.getreal() + B.getreal());
    C.setimag(A.getimag() + B.getimag());
    return C;
}
```

以类成员函数的形式进行操作符重载，操作符左侧的操作数必须为类对象；而以顶层函数的形式进行操作符重载，只要类中定义了相应的转型构造函数，操作符左侧或右侧的操作数均可以不是类对象，但其中必须至少有一个类对象，否则调用的就是系统内建的操作符而非自己定义的操作符重载函数。

友元函数重载：

```
1   class complex
2   {
3   public:
4       complex();
5       complex(double a);
6       complex(double a, double b);
7       friend complex operator+(const complex & A,
            const complex & B);
8       friend complex operator-(const complex & A,
            const complex & B);
9       friend complex operator*(const complex & A,
            const complex & B);
10      friend complex operator/(const complex & A,
            const complex & B);
11      void display()const;
12  private:
13      double real;    //复数的实部
14      double imag;    //复数的虚部
15  };
16
17  //重载加法操作符
18  complex operator+(const complex & A, const complex
        &B)
19  {
20      complex C;
21      C.real = A.real + B.real;
22      C.imag = A.imag + B.imag;
23      return C;
24  }
```

采用友元函数的形式进行操作符重载，如此实现既能继承操作符重载函数是顶层函数的优势，同时又能够使操作符重载函数实现起来更简单。

5.3 输入输出操作符的重载

>> 的第一个操作数是系统类的对象，而重载函数是以类成员函数的形式实现的。为了不对系统类的源码进行修改，只能将 >> 重载函数设计为顶层函数。

5.4 赋值运算符重载

拷贝构造函数和赋值操作符（=），都用来拷贝一个类的对象给另一个同类型的对象。拷贝构造函数将一个对象拷贝到另一个**新**的对象，赋值操作符将一个对象拷贝到另一个**已存在**的对象。编译器会自动生成拷贝构造函数和赋值运算符，但也可能会出现相应的问题。可以使用私有成员函数来让拷贝构造函数和赋值操作符函数共享复制的功能。

5.5 特殊操作符重载

5.5.1 下标操作符重载

只能以成员函数形式重载。

可以修改对象：

```
returntype& operator[] (paramtype);
```

可以访问对象但不能修改：

```
const returntype& operator[] (paramtype);
```

5.5.2 函数调用操作符重载

语法：

```
returntype operator() (paramtype);
```

效果：

```
c(x,name); \\ 将被翻译为：
c.operator()(x,name);
```

5.5.3 自增自减操作符重载

重载内容：

1. 前置自增
2. 后置自增
3. 前置自减
4. 后置自减

前置自增：

```
C operator++(); // ++c
```

后置自增：

```
C operator++ (int); // c++
```

其中后置的 int 无实际意义，仅用来区分。

5.5.4 转型操作符重载

```
operator othertype();
```

5.6 内存管理操作符

内存管理操作符 new,new[],delete,delete[]，可以用成员函数也可以用顶层函数重载。

new,new[] 操作符重载函数的第一个操作必须是 size_t 类型（数值等于将被创建的对象的大小总和）。

delete,delete[] 操作符重载函数的第一个参数必须是 void* 类型的，返回类型必须是 void。

Chapter 6

模板

6.1 模板基础知识

模板头：

```
\\ class == typename
template<class T>
template<typename T>
template<typename T1,typename T2,typename T3>
```

模板类外定义函数：

```
template<class T>
Array<T>::Array(int s){
    a = new T[size = s];
}
```

模板实例：

```
Array<double> Array<int> etc
```

对象不属于模板类，只属于模板实例。

参数表中的模板类：模板类可以作为一种数据类型出现在参数表中。

模板的函数类型参数：

```
template<class T,int X,float Y>
```

6.2 断言

断言判断一个表达式，如果结果为假，输出诊断消息并中止程序。

```
void assert(
   int expression
);
```

参数：Expression (including pointers) that evaluates to nonzero or 0.（表达式【包括指针】是非零或零）

原理：assert 的作用是现计算表达式 expression ，如果其值为假（即为0），那么它先向 stderr 打印一条出错信息，然后通过调用 abort 来终止程序运行。

用法总结：

1. 在函数开始处检验传入参数的合法性如:

```
1  int resetBufferSize(int nNewSize)
2  {
3     //功能改变缓冲区大小:,
4     //参数:nNewSize 缓冲区新长度
5  //返回值缓冲区当前长度:
6  //说明保持原信息内容不变:      nNewSize表示清除
       缓冲区<=0
7  assert(nNewSize >= 0);
8  assert(nNewSize <= MAX_BUFFER_SIZE);
9  }
```

2. 每个 assert 只检验一个条件,因为同时检验多个条件时,如果断言失败,无法直观的判断是哪个条件失败

3. 不能使用改变环境的语句,因为assert只在DEBUG个生效

4. assert 和后面的语句应空一行,以形成逻辑和视觉上的一致感

5. 有的地方,assert 不能代替条件过滤

ASSERT 只有在 Debug 版本中才有效，如果编译为 Release 版本则被忽略掉。（在C中，ASSERT 是宏而不是函数），使用 ASSERT “断言”容易在 debug 时输出程序错误所在。而 assert() 的功能类似，它是 ANSI C 标准中规定的函数，它与 ASSERT 的一个重要区别是可以用在 Release 版本中。

使用 assert 的缺点是，频繁的调用会极大的影响程序的性能，增加额外的开销。在调试结束后，可以通过在包含 #include ¡assert.h¿ 的语句之前插入 #define NDEBUG 来禁用 assert 调用，示例代码如下：

```
1 #include <stdio.h>
2 #define NDEBUG
3 #include <assert.h>
```

6.3 STL 标准模板库