类和对象

基础

调用

- 访问对象成员变量和函数的方式有直接、通过指针或引用

```
class AClass{
public:
  int a=0;
};

int main(){
  AClass c;
  //直接访问
  cout<<c.a<<endl;
  //通过指针访问
  AClass * cp=&c;
  cout<<cp->a<<endl;
  //通过引用访问
  AClass & cr=c;
  cout<<cr.a<<endl;
  return 0;
}
```

- 成员函数的缺省参数可以使新增函数参数时不必修改原来调用函数的部分

  可以使后面的参数缺省，不能使前面的参数缺省，即不能`FUN(,1)`

  缺省构造函数要避免调用时和重载函数冲突

```
class AClass{
public:
  FUN(int a=0,int b=1,int c=2){}
}
int main(){
  AClass c;
  c.FUN(1);//等价于c.FUN(1,1,2)
  return 0;
}
```

- 成员函数声明的关键字可以只在类内部说明，类外部写函数时可以只写类型

- 静态成员可以直接通过`ClassName::FUN()`进行访问，也可以通过普通方式访问（但没有实际意义，因为并不属于某个对象）

  静态成员函数中不能有非静态成员变量和非静态成员函数，因为静态成员函数不属于某个对象，而属于同一类共有

- 常量成员函数的`const`关键字在函数名后面`void FUN()const{}`，其内不能改变属性的值，也不能调用非常量成员函数；`const`在函数名后面的情况只对成员函数

  常量对象不能改变属性值，同一个函数加或不加`const`属性算重载，区别在于常量对象优先调用常量成员函数

  常量成员变量或对象可以通过非常量的强制转换使用改变

  常量成员函数的`const`如果写在前面，相当于返回了常量值，没有意义；而`static`写在前面或者后面没有区别，因为`static`不会用来声明返回值

类成员可访问范围

- 类和结构体的区别：类成员默认`private`，结构体默认`public`
- `private`可访问范围
  - 基类成员函数
  - 基类友元函数
- `protected`可访问范围
  - 基类成员函数
  - 基类友元函数
  - 派生类成员函数（可访问当前已定义好的派生类对象的基类保护成员）
- `public`可访问范围
  - 基类成员函数
  - 基类友元函数
  - 派生类成员函数
  - 派生类友元函数
  - 其他函数

构造函数和析构函数

构造函数

- 函数名和类名相同，没有返回值，可以多个重载
- 如果没有任何构造函数，编译器自动生成一个无参构造函数和一个复制构造函数

  如果编写了构造函数，编译器不再生成无参构造函数，仍会生成复制构造函数

  如果编写了复制构造函数，编译器不会编译任何构造函数，因此一定至少还需要一个构造函数进行初始化

  初始化对象时可以有缺省参数

- 可以通过初始化列表直接对成员进行赋值

```
AClass():n(i){}
```

- 封闭类的成员对象，引用成员和`const`常量成员必须在初始化列表中初始化

转换构造函数

- 即普通构造函数，在其他类型变量对该类对象赋值时，先用该变=变量生成临时对象，再按位拷贝（**不是调用复制构造函数**）

复制构造函数

- 一般写`ClassName(const ClassName & c){}`

  传递引用是因为只能传递引用，因为传值会造成递归调用复制构造函数

- 三种情况调用
  - 用一个对象去初始化另一个类的对象
  - 函数形参是类的对象，可以通过传引用避免复制构造函数的调用
  - 函数返回值是类的对象
  - **等号赋值除非重载否则不是复制构造函数而是按位拷贝**

析构函数

- 只能有一个析构函数，定义了则不生成缺省析构函数，否则生成什么都不做的缺省析构函数

其他成员

成员对象（封闭类）

- 其类没有无参构造函数的成员对象必须在封闭类定义的构造函数里初始化

  成员对象先与该封闭类初始，且初始顺序和声明顺序相同，和初始列表中的顺序无关

  析构时先执行封闭类的析构函数再析构成员对象

```
class AClass{
private:
  int a;
public:
  AClass(int n){
    a=n;
  }
};
class BClass{
private:
  AClass c;
public:
  BClass():c(1){};
};
```

友元

- 友元函数的意义是允许成员函数以外的函数访问该类的属性
- 友元类允许该类的所有成员函数访问其私有变量（所有变量）

运算符重载

重载类型

- 重载为普通函数：左操作数为非类对象，且只访问公有元素

  参数个数为运算符目数

- 重载为成员函数：左操作数为类对象

  参数个数为运算符目数减一

- 重载为友元函数：左操作数为非类对象，且需要访问`private`对象

  参数个数为运算符目数

实例

- 强制类型转换符：重载为成员函数，在运行中需要调用的时候自动强制转换当前类类型；无返回值类型，有返回值；形参为空

  因为通常不改变对象，可以声明为常量成员函数

```
operator int() const {
  return n;
}
```

- 函数调用运算符：将对象当作函数名进行函数调用计算；初始化时不为函数调用

```
type operator(){}
```

- 下标运算符：对类成员进行数组式的读写

  返回引用保证写入（赋值）的时候做操作数为类内数组对象该元素的地址

```
type & operator[](type n){
  return pointer[n];
}
```

- 赋值运算符：一般为了方便连续赋值，返回当前类（对象）的引用

  此时类做了返回值（以及某些情况函数参数），需要复制构造函数

  最好使用`const`，若右侧为返回的临时变量，必须用`const`进行匹配（`return`空间不能修改）

```
const AClass & operator=(const type n){
  return *this;
}
```

  除此之外，还需要对类之间赋值重载赋值运算符（使其只是内容相同即深赋值），否则两个类指向的同一个地址（赋值只是赋值地址，浅复制）；此时需要判断`n==c.n`先终止实参赋给形参的再次调用（所有现成的如`int`类内部都是已经实现了深复制的）

```
AClass & operator=(const AClass &c){
  if(n==c.n) return *this;
  //
  return *this;
}
```

- 流运算符：输入和输出

  可以在类内重载为友元函数或类外重载为普通函数（此时访问成员必须全部是`public`）

  必须传入和返回引用，因为`cin`和`cout`不能复制

```
friend istream & operator>>(istream &i,AClass &c){//不能const
  i>>c.n;
  return i;
}

friend ostream & operator<<(ostream &o,const AClass &c){
  o<<c.n<<endl;
  return o;
}
```

- 自增/自减运算符：前置运算符为一元运算符，后置运算符为二元运算符（第二个参数无用）；即作为成员函数时，前置不带参数，后置带一个无用的`int`参数

  后置运算符因为返回原值，同时还要修改现值，故返回一个复制构造函数构造的临时对象

```
AClass & operator++(){
  n++;
  return *this;
}
AClass & operator++(int u){
  AClass tmp(*this);
  n++;
  return tmp;
}
```

注意

- 不能重载的运算符：`.`、`.*`、`::`、`?:`、`sizeof`、`#`、`##`等
- 必须重载为成员函数：`()`、`[]`、`->`、`=`
- 对于双目运算符，既要支持做操作数为类的对象，又要支持为非类对象，则前者重载为成员函数，后者重载为友元函数

继承

基础

-
```
class BClass{

};
class DClass:public BClass{

};
```

- 执行派生类构造函数前要先执行基类构造函数（再执行其他成员如封闭类的构造函数），如果涉及基类`private`成员则要在初始化列表初始化

  构造函数不会自动继承

  如要使派生类对象具有基类的全部功能（可以当作基类使用），则在初始化列表初始化；即具有了一个类型形式是派生类的基类对象，派生类中其他功能也可以使用

  析构时先析构派生类再析构基类

- 一般公有派生情况的赋值兼容
  - 派生类对象赋值给基类对象
  - 派生类对象初始化基类引用

    - 派生类对象地址赋值给基类指针

      此时指针指向派生类对象（中的基类对象），不能访问派生类对象中的其他成员，但可以通过强制类型转换为派生类指针访问

- 派生类函数与基类函数关系（返回值无关）
    - 重载：同一类（同一范围中）定义，名字相同，参数不同
    - 覆盖：基类虚函数和派生类函数，名字相同，参数相同
    - 重写：基类函数和派生类函数，名字相同，参数不同；基类非虚函数和派生类函数，名字相同，参数相同（只能通过指针类型不同调用，无多态性）

      重写如果不使用指针，只能通过`derived.base::Func()`调用
- 多层继承只需要继承直接基类，直接基类会自动继承间接基类
- 派生类中的基类和封闭类中的成员对象在空间上是相似的，但是逻辑概念不同（独立性不同、访问主体不同）

  和封闭类的区别是，再定义一个封闭类，需要通过当前对象间接访问其中的封闭类对象；直接调用当前对象会先检查调用基类的函数，没有初始化则拥有无参构造的基类对象并执行其函数等

多态

多态的实现

- 普通情况下，函数调用通过指针类型决定调用基类还是派生类中的同名函数

  多态情况下，将基类中的同名函数声明为`virtual`（且只在声明中使用），可以通过基类指针引用的对象类型实现基类或派生类的函数调用

```cpp
class AClass{
public:
  virtual void SayHiV(){cout<<"AClassV"<<endl;}
  void SayHi(){cout<<"AClass"<<endl;}
};

class BClass:public AClass{
public:
  void SayHiV(){cout<<"BClassV"<<endl;}
  void SayHi(){cout<<"BClass"<<endl;}
};

void FuncPa(AClass *pa){
  pa->SayHiV();
  pa->SayHi();
}

void FuncPb(BClass *pb){
  pb->SayHi();
}

int main(){
  AClass oa;
  BClass ob;
  FuncPa(&oa); //AClassV \n AClass
  FuncPa(&ob); //BClassV \n AClass
  //FuncPb(&oa); 不能将原生基类对象强制赋给派生类指针!
  FuncPb(&ob); //BClass
  return 0;
}
```

- 多态实际上是对具有继承和派生关系的量，在需要函数重载（需要对这些派生类做同一种处理）的时候，可以统一抽象处理（使基类指针在不同情况指向不同派生类对象），原来重载的函数就可以不用重复写了

多态注意点

- 访问权限：根据指针类型检查，因此基类虚函数需要声明为公有（派生类同名覆盖函数为私有则没有关系，因为多态时为基类指针）
- 基类指针指向派生类对象，会优先选择符合多态的函数

- ```
  class AClass{
  public:
    virtual void Func() const {cout<<"AFunc"<<endl;}
    virtual void Func(int n=0){cout<<"AFunc(int)"<<endl;}
    //两个非虚函数可以这样重载 但不能调用Func()
  };

  class BClass{
  public:
    void Func(){cout<<"BFunc"<<endl;}
    void Func(int n){cout<<"BFunc(int)"<<endl;}
  }

  int main(){
    AClass oa,*pa;
    BClass ob;
    pa=&ob;
    pa->Func(); //AFunc(int) 因为Func() const 和Func()不满足多态
    return 0;
  }
  ```

- 先非多态地调用基类或派生类的函数，函数内再调用函数，仍然要根据多态性调用
- 基类析构函数一般一定要定义为虚函数（这样才会根据对象对派生类析构，再析构基类），以免通过基类指针销毁派生类对象时只调用了基类的析构函数
- 构造函数不能声明为虚函数

纯虚函数

- 纯虚函数：没有函数体的虚函数

  抽象类：包含纯虚函数的类

  ```
  class AClass{
  public:
    virtual void Func()=0;
  };
  ```

- 抽象类只能作为基类，不能创建抽象类对象；可以定义其指针/引用，但也只能指向派生类对象

  继承的派生类如果没有实现纯虚函数的多态，则仍然是抽象类

- 抽象类的成员函数可以调用纯虚函数，但构造/析构函数不能

模版

函数模版
  ```

```cpp
template<class T>
T Func(T &a,T &b){
  T tmp=a;
  a=b;
  b=tmp;
  return a;
}

template<typename T>
void Func(T &a){}
```

- 函数模版与重载的区别：重载是传入参数不同，模版要求参数相同（或参数有初始化时缺省参数）
- 函数匹配
  1. 找参数完全匹配的函数
  2. 找参数完全匹配的函数模版
  3. 无二义性的情况下，找参数经过自动转换后能够匹配的函数

类模版

```cpp
template<class T1,class T2,...>
class AClass{
public:
  T1 n;
  void Func(T2 a);
};
template<class T1,class T2,...>
void AClass<T1,T2,...>::Func(T2 a){}
```

- 同一个类模版的两个实例化不同的模版类不兼容
- 除了定义，任何时候使用类模版对象需要在类名后声明真是类型 `AClass<int,double,...> c(2.5);`
- 同一类实例化的模版类共享同样的静态成员
- 静态成员一定在类外初始化

```cpp
template<class T>
class AClass{
public:
  static int n1;
  static T n2;
};
template<>
double AClass<double>::n2=5.2;
template<class T>
int AClass<T>::n1=120;
```

- 非类型参数

  非类型参数必须实例化

  局部变量不能用作非类型参数

```cpp
template<class T,int n>
class AClass{
public:
  T array[n];
};

int main(){
  int n=10;
  AClass<int,10> c;
  //AClass<int,n> c2; 错误 局部变量不能用作非类型参数
  return 0;
}
```

类模版与继承

- 类模版派生类模版

```cpp
template<class T1,class T2>
class BClass{};

template<class T1,class T2>
class DClass:public BClass<T2,T1>{};
```

- 模版类派生类模版（相当于前者继承时实例化）

```cpp
template<class T1,class T2>
class BClass{};

template<class T1,class T2>
class DClass:public BClass<int,double>{};
```

- 普通类派生类模版
- 模版类派生普通类

I/O