Thursday, 14 December 2017

面向对象语言程序设计(C++)

一些容易忘记/混淆的概念

1. 左值和右值/前置++和后置++：

左值是对应内存中有确定存储地址的对象的表达式的值；右值是所有不是左值的表达式的值。
一般来说，左值是可以放到赋值符号左边的变量。但，能否被赋值不是区分左值与右值的依据。比如，C++的const左值是不可赋值的；而作为临时对象的右值可能允许被赋值。
左值与右值的根本区别在于是否允许用取地址运算符&，去获得对应的内存地址。

```
int i = 0;
int *p1 = &(++i); //正确
int *p2 = &(i++); //错误
++i = 1; //正确
```

(++i返回的是一个引用，引用是左值；i++返回的是一个临时变量(或者对象的一个临时副本)，临时变量是右值)

```
// 前缀形式：
int& int::operator++() //这里返回的是一个引用形式，就是说函数返回值也可以作为一个左值使用
{//函数本身无参，意味着是在自身空间内增加1的
  *this += 1;  // 增加
  return *this;  // 取回值
}

//后缀形式：
const int int::operator++(int) //函数返回值是一个非左值型的，与前缀形式的差别所在。
{//函数带参，说明有另外的空间开辟
 int oldValue = *this;  // 取回值
  ++(*this);  // 增加
  return oldValue;  // 返回被取回的值
}
```

2. const与static关键字的作用总结：
2.1 const：
对于既需要共享、又需要防止改变的数据应该声明为**常类型**（用const进行修饰）。
常对象：必须进行初始化，且初始化后不能再被更新。const 类名 对象名
常引用：被引用的对象不能被更新。const 类型说明符 &引用名。如果用常引用做形参，便不会意外地发生对实参的更改。常引用的声明形式如下：
const 类型说明符 &引用名;
常数组：数组元素不能被更新。类型说明符 const 数组名[size]
常指针：指向常量的指针

2.2 static：
静态生存期：对象的生存期与程序的运行期相同。在文件作用域(全局变量)中声明的对象同样具有静态生存期。

在函数内部声明静态生存期对象，要冠以关键字static。在函数体内声明一个静态局部变量，它会拥有局部可见性和全局生命期，同时它只在第一次进入函数时被初始化。

3. 常成员函数和静态成员函数：

3.1 常成员函数：
<类型标识符> 函数名 (形参列表) const
常成员函数**不**更新类中任何数据成员的值， 对于不改变对象状态的成员函数应该声明为**常函数**。
常成员函数不可以调用类中没有用const修饰的成员函数，通过常对象只能调用它的常成员函数，但是常成员函数可以被其他成员函数调用

注意：const关键字可以用于对重载函数的区分

3.2 静态成员函数：
非静态函数可以知道是通过哪个对象调用了自己(this指针)，静态函数不知道是通过哪个对象调用了自己。
静态成员函数一般不用来处理属于对象的数据。静态成员函数主要用于处理该类的静态数据成员，可以直接调用静态成员函数。
类外代码可以使用类名和作用域操作符来调用静态成员函数。
如果一定要用静态成员函数访问非静态成员，要通过对象来访问。

4. this指针
一个对象的this指针并不是对象本身的一部分。this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员时，编译器会自动将对象本身的地址作为一个隐含的参数传递给函数。this指针是类的一个自动生成、自动隐藏的私有成员，它存在于类的非静态成员函数中，指向被调用函数所在的对象。
全局仅有一个this指针，当一个对象被创建时，this指针就存放指向对象数据的首地址。

C++ 11的一些特性

- 可用using语句继承基类构造函数()
- override 和 final
- lambda表达式
- 使用initializer_list标准库类型解决可变形式参数
  - 1. 函数原型中使用实例化initializer_list模板代表可变参数列表； 2. 使用迭代器访问initializer_list中的参数； 3. 传入实参写在{}之内。
  - 与vector等容器一样initializer_list也支持**begin()和end()**操作，返回指向首元素的迭代器和尾后迭代器。initializer_list在同名头文件中声明，其实现由编译器支持。

继承与派生 Inheritance and Derivative

- 单继承派生：

```
class Derived: public
Base{
public:
      Derived ();
      ~Derived ();
};
```

- 多继承派生：

```
class Derived: public
Base1, private Base2{
public:
      Derived ();
      ~Derived ();
};
```

- 公有继承：
  - 基类的public和protected成员：访问属性在派生类中保持不变
  - 基类的private成员：不可以直接访问
- 私有继承：

  - 基类的public和protected成员：都以private身份出现在派生类中
  - 基类的private成员：不可以直接访问

- 保护继承：
  - 基类的public和protected成员：都以protected身份出现在派生类中
  - 基类的private成员：不可以直接访问

- protected成员的特点与作用：
  - 对建立其所在类对象的模块来说，它与private成员的性质相同
  - 对于其派生类来说，它与public成员的性质相同
  - 既实现了数据隐藏，又方便继承，实现代码重用
  - 如果派生类有多个基类，也就是多继承时，可以用不同的方式继承每个基类。
  - 如果基类和派生类都是一个人或一个团队写的，那么使用保护继承没有问题。但是如果是不同团队完成的，甚至是购买来的类库，那么使用保护继承就不太方便了（修改接口非常麻烦）。

- 三种继承方式的归纳

	派生类的成员函数对基类成员的访问权限	派生类的对象对基类成员的访问权限
公有继承 public inheritance	可以直接访问基类中的公有成员和保护成员，但不能直接访问基类中的私有成员	只能访问公有成员
私有继承 private inheritence	可以直接访问基类中的公有成员和保护成员，但不能直接访问基类中的私有成员	不能访问从基类继承过来的任何成员
保护继承 protected inheritance	可以直接访问基类中的公有成员和保护成员，但不能直接访问基类中的私有成员	不能访问从基类继承过来的任何成员

- 基类与派生类的相互转换：

- 公有派生类对象可以被当作基类的对象使用，反之则不可以
  - 派生类的对象可以隐含转换为基类对象
  - 派生类的对象可以初始化基类的引用
  - 派生类的指针可以隐含转换为基类的指针
- 通过基类对象名，指针只能使用从基类继承的成员

- 派生类的构造和析构
  - 默认情况
    - 基类的构造函数不被继承;
    - 派生类需要定义自己的构造函数。
  - C++11规定
    - 可用using语句继承基类构造函数。
    - 但是只能初始化从基类继承的成员。派生类的新增成员只能按类内初始值或者默认方式进行初始化。(因此这个新特性只适用于派生类很少或没有增加新数据成员的情况)
      - 派生类新增成员可以通过类内初始值进行初始化。
  - 语法形式：
    - using B::B; (基类名)(作用域分辨符)(基类构造函数名)
    - 自行设计构造函数：如果不继承基类的构造函数
    - 派生类新增成员：派生类定义构造函数进行初始化；
    - 继承来的成员：自动调用基类的构造函数进行初始化；
    - 派生类的构造函数需要给基类的构造函数传递参数。
  - 思考：派生类如果没有写构造函数会怎么样?
    - 当基类有默认构造函数时
      - 派生类的构造函数可以不向基类构造函数传递参数
      - 构造派生类的对象时，基类的默认构造函数将被调用

- 如需执行基类中带参数的构造函数
  - 派生类构造函数应该为基类构造函数提供参数
- 构造函数的执行顺序
  - 1. 调用基类构造函数。
    - **初始化的顺序按照它们被继承时声明的顺序（从左向右），即使没给某个基类传参数，只要这个基类被继承了，就会调用它的默认构造函数。**
  - 2. 对初始化列表中的成员进行初始化
    - **初始化的顺序按照它们在类中定义的顺序（即使没给这个类内的成员对象传参数，只要在类中被定义了，也会调用它的构造函数）**
    - 对象成员(这里主要是指派生类新增的成员，且这些类内的成员是其他类的对象)在初始化时自动调用其所属类的构造函数，由初始化列表提供参数
  - 3. 执行派生类的构造函数体中的内容。
- 派生类的复制构造函数/析构函数
  - 注意：析构顺序与构造顺序成“镜像”相反

- 派生类成员的标识与访问
  - 访问从基类继承的成员/避免二义性和冗余的情况
  - 虚基类：从第一级继承开始使用虚继承方式，可以避免因继承过程中继承了公共基类而产生冗余的问题。在虚继承的情况下，调用基类构造函数时，只有最远派生类传递的参数起实际作用。

多态性 Polymorphism

- 多态：操作接口具有表现多种不同形态的能力。在不同的环境下，对不同的对象具有不同的处理方式。
  - 绑定(实现多态的方式)
    - 编译时的多态性(早绑定/静态绑定)：函数重载是静态多态性的一种体现

  - 运行时的多态性(晚绑定/动态绑定)

- 运算符重载(静态多态性)
  - 运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时导致不同的行为
  - C++ 几乎可以重载全部的运算符，而且只能够重载C++中已经有的
  - 不能重载的运算符：“.”、“.*”、“::”、“?:”
    - 重载之后运算符的优先级和结合性都不会改变
  - 双目运算符重载为成员函数
    - +的重载：实现Complex Number + Complex Number 或 Complex Number + Real Number可以通过将运算符重载为成员函数实现
    - 如果要重载 B 为类成员函数，使之能够实现表达式 oprd1 B oprd2，其中 oprd1 为A 类对象，则 B 应被重载为 A 类的成员函数，形参类型应该是 oprd2 所属的类型
    - 经重载后，表达式 oprd1 B oprd2 相当于 oprd1.operator B(oprd2)
  - 单目运算符重载为成员函数
    - 前置单目运算符重载规则：如果要重载 U 为类成员函数，使之能够实现表达式 U oprd，其中 oprd 为A类对象，则 U 应被重载为 A 类的成员函数，无形参，经重载后，表达式 U oprd 相当于 oprd.operator U()
    - 后置单目运算符 ++和 -- 重载规则: 如果要重载 ++或--为类成员函数，使之能够实现表达式 oprd++ 或 oprd-- ，其中 oprd 为A类对象，则 ++或-- 应被重载为 A 类的成员函数，且具有一个 int 类型形参，经重载后，表达式 oprd++ 相当于 oprd.operator ++(0)
  - 运算符重载为非成员函数
    - 两种情况
      - +的重载：实现Real Number + Complex Number 无法通过将运算符重载为成员函数实现，需要将运算符重载为全局函数

    - 如果左操作数是一个对象，但这个对象不是自己设计的类的对象，又想能够让这个对象和我们自定义的另一个类的对象去运算，那么只能将运算符重载为类外的非成员函数

  - 运算符重载为非成员函数的规则
    - 函数的形参代表依自左至右次序排列的各操作数
    - 重载为非成员函数时
    - 参数个数=原操作数个数（后置++、—除外）
    - 至少应该有一个自定义类型的参数
    - 后置单目运算符 ++和—的重载函数，形参列表中要增加一个int，但不必写形参名
    - 如果在运算符的重载函数中需要操作某类对象的私有成员，可以将此函数声明为该类的友元

- 虚函数简介：
  - 用virtual关键字说明的函数
  - 虚函数是实现运行时多态性基础
  - C++中的虚函数是动态绑定的函数
  - 虚函数必须是非静态的成员函数，虚函数经过派生之后，就可以实现运行过程中的多态。**虚函数是属于对象的，而不是属于整个类的，虚函数需要在运行的时候通过指针定位到它指向的对象是谁，然后决定调用哪个函数体，因此虚函数是属于对象的而不是属于类的函数。**虚表是属于类的，虚指针是属于对象的。
  - 一般成员函数可以是虚函数
  - 构造函数不能是虚函数
  - 析构函数可以是虚函数
- 虚函数注意事项：
  - 建议：不要重新定义继承而来的非虚函数

- 通过虚函数实现运行时多态：编译器事先不确定指针所指向的对象类型，在运行时根据实际地址确定。
- 为函数声明virtual关键字的作用是：指示编译器编译器不要做静态绑定，而要在运行时做动态绑定
- 注意：声明virtual关键字的函数，函数体不能写在类体中（因为不希望在编译阶段处理，所以不能写成inline形式），要在类外实现。
- 派生类可以不显式地用virtual声明虚函数，这时系统就会用以下规则来判断派生类的一个函数成员是不是虚函数：
  - 该函数是否与基类的虚函数有相同的函数原型（函数名称、参数个数、对应参数类型）
  - 该函数是否与基类的虚函数有相同的返回值或者满足类型兼容规则的指针、引用型的返回值。
  - 如果从名称、参数和返回值三个方面检查之后，派生类的函数满足上述条件，就会自动确定为虚函数。这时，派生类的虚函数便覆盖了基类的虚函数。
  - **注意！派生类中的虚函数会隐藏基类中同名函数的所有其他重载形式。**
  - 一般习惯于在派生类的函数中也使用virtual关键字，以增加程序的可读性。
- 虚表
  - 每个多态类有一个虚表（virtual table）虚表是属于类的
  - 虚表中有当前类的各个虚函数的入口地址
  - 每个对象有一个指向当前类的虚表的指针（虚指针vptr）虚指针是属于对象的
- 动态绑定的实现
  - 构造函数中为对象的虚指针赋值
  - 通过多态类型的指针或引用调用成员函数时，通过虚指针找到虚表，进而找到所调用的虚函数的入口地址
  - 通过该入口地址调用虚函数

- 抽象类

- **（注意：抽象类只能作为基类来使用，不能定义抽象类的对象。）**
  - 纯虚函数
    - 纯虚函数是一个在基类中声明的虚函数，它在该基类中没有定义具体的操作内容，要求各派生类根据实际需要定义自己的版本，纯虚函数的声明格式为：virtual 函数类型 函数名(参数表) = 0;
    - 带有纯虚函数的类称为抽象类
  - 抽象类
    - 带有纯虚函数的类称为抽象类:
      class 类名 { virtual 类型 函数名(参数表)=0; //其他成员……
  - 抽象类作用
    - 抽象类为抽象和设计的目的而声明
    - 将有关的数据和行为组织在一个继承层次结构中，保证派生类具有要求的行为。
    - 对于暂时无法实现的函数，可以声明为纯虚函数，留给派生类去实现。
- override与final
  - override：
    - 多态行为的基础：基类声明虚函数，继承类声明一个函数覆盖该虚函数
    - 覆盖要求： 函数签名（signatture）完全一致
    - 函数签名包括：函数名 参数列表 const
    - 使用显示函数覆盖：C++11 引入显式函数覆盖，在编译期而非运行期捕获此类错误。在虚函数显式重载中运用，编译器会检查基类是否存在一虚拟函数，与派生类中带有声明override的虚拟函数，有相同的函数签名（signature）；若不存在，则会回报错误。
  - final：C++11提供的final，用来避免类被继承，或是基类的函数被改写(防止被覆盖)
    - 例： struct Base1 final { };

- struct Derived1 : Base1 { }; // 编译错误：Base1为final，不允许被继承
- struct Base2 { virtual void f() final; };
- struct Derived2 : Base2 { void f(); // 编译错误：Base2::f 为final，不允许被覆盖 };

模板与群体数据

- 具有相同类型的数据称为群体，如果它们按一定次序排列，则称为线性群体
- 模版
  - 模板参数表的内容
    - 类型参数：class（或typename） 标识符
    - 常量参数：类型说明符 标识符
    - 模板参数：template <参数表> class标识符
- 类模版
  - 类模板的作用
    - 使用类模板使用户可以为类声明一种模式，使得类中的某些数据成员、某些成员函数的参数、某些成员函数的返回值，能取任意类型（包括基本类型的和用户自定义类型）
  - 类模板的声明
    - 类模板 template <模板参数表> class 类名 {类成员声明};如果需要在类模板以外定义其成员函数，则要采用以下的形式： template <模板参数表> 类型名 类名<模板参数标识符列表>::函数名（参数表）

- 线性群体
- 数组
- 链表
- 栈
- 队列
- 排序
- 查找

C++标准模板库与泛型程序设计 Templates and Generic Programming

- 泛型程序设计及STL的结构
- 迭代器
- 容器的基本功能与分类
- 顺序容器
- 关联容器
- 函数对象
- 算法

流类库与输入\输出 Inheritance and Derivative

异常处理