# 1 《C++基础语法》

## 1 常用语法

(1) namespace

① 命名空间定义: namespace SpaceA {.....}　作用域 (防止函数，变量，类的名字冲突)
使用方法：SpaceA::函数或变量;

② 匿名空间 namespace{.....}只能在所在文件里使用，如同 static
命名空间里函数或变量如果是声明，可以在外部定义:
<data_type> SpaceA:: fun(arguments) {.....}

可以结合匿名空间一起使用，内部程序不暴露，只暴露接口

(2) using

① 引用名称到所使用的作用域 (函数名，变量名，空间类名…)
using namespace SpaceA; 把整个命名空间引入所在作用域，但一般不提倡，由于可能不同空间有同名成员
using SpaceA::member; 同样在继承类里可以引入基类的功能 (using Base::func;)

② 给类型起别名 (替代 typedef 功能)
using uint = unsigned int;　如同 typedef unsigned int uint;
using cout = std::cout;

(3) auto

编译器会根据初始化表达式自动推导出变量类型 auto x = value;（可以结合引用）不能用于函数参数
for (auto x : y) x 为变量名首元素的拷贝，遍历有 begin() 和 end() 的 y，用于标准容器、数组、自己写的类。由于该方法一开始会记录这个数据的结束位置，所以不能在遍历的时候修改数据(添加或删除)，因为会导致结束位置混乱

(4) 类型转换

① 隐式类型转换：系统自动转换(不安全，通常当数据类型不一样时，且**"="赋值时会出现隐式转换**)
② 显示类型转换：通过 explicit 避免系统隐式转换，因此要显示转换 (下述方法)

| | |
|---|---|
| static_cast | 用于大多数安全的转换，编译器在编译期检查合法性。 |
| dynamic_cast | RIIT 对象指针的转换，转换要明确原本的类型避免无法转换，基类里必须有虚函数(可以多态)，因为需要虚函数表指针 (如 Base b; 里面没有子类信息所以无法转换成子类，而 Base* ap = &d; 此处 ap 可以动态转换为子类，因为 ap 动态类型原本就是子类)，在虚继承且有虚函数表时要用该转换才会更安全, (满足条件：继承且有虚函数表就用动态转换) |
| const_cast | 用于去除 const 或 volatile 限定的类型 (原本不是常量才安全) |
| reinterpret_cast | 几乎不进行任何类型检查，指针强转 (直接按内存解释) |

## 2 C++函数

① 函数的形参可以设置默认值，默认值后面的参数也都要有默认值 (在函数声明里有默认值，但在函数定义里不能再出现默认值)
② 函数的占位形参: func(int) 或 func(int=5)，要给占位参数实值除非有默认值
(注：有些 operator 函数使用时可以不用给占位参数值)
③ 函数的重载:
满足条件：同一个作用域 + 相同函数名 + 不同参数列表(参数类型，参数个数，参数顺序)
(注意不要出现二义性)
④ 尾返回形式：auto fun(arguments) -> return_data_type {.....}

## 3 动态内存

int* ptr = new int(5); 或 int* ptr = new int; 没有初始化
int* arr = new int[5]; 5 个元素的数组, (此处 arr 不是数组名，只是一个连续动态首地址，没有初始化数据)

delete ptr;　如果 ptr = nullptr 可以释放多次，如果 ptr ≠ nullptr 则不能重复释放 (否则报错)
delete[] arr;　释放数组

## 4 引用

① 引用的使用方法

引用是给一个变量起别名，从而使他们共同指向同一个内存空间 (无论左值还是右值引用) 引用绑定后不能再使其引用到另一个变量 (只能一直绑定该变量)
引用的好处：减少传参(形参, return 值)出现的临时变量拷贝的现象 (函数栈帧)

Tip: 作为函数的返回值 (可以让返回值作为左值使用) 或是函数形参 (代替指针，从而避免临时拷贝)

② 左值和右值引用

左值引用:
- 可以修改的左值的引用，从而减少临时拷贝
- 普通引用: int a = 10; int& b = a;
- 数组引用: int (&arr_2)[7] = arr_1;
- 函数指针引用: void (*& fun_2)(arguments) = fun_1;

右值引用:
- 不可以 int& b = 10; 但可以 const int& b = 10; (此处建立 tmp = 10; b 是一个不可修改的左值即引用 tmp)
- int&& b =10; 是一个右值引用，引用的是 10 这个右值。此时 b 是左值可以修改
- 右值一般用在临时变量 (移动语义)　std::move()将变量转换为右值

# 2《封装 Encapsulation》

## 1 构造函数

(1) 默认构造函数 (普通构造函数):

系统有自己默认的构造函数，如果自己定义了就不会调用默认的，需要自己定义无参的构造函数

(2) 有参构造函数 (普通构造函数):

系统不会调用默认的构造函数，除非自己再定义一个无参构造函数

**必须用初始化列表的情况:**
1 const 成员
2 引用成员
3 无默认构造函数的其他类的对象作为成员 (is-a 和 has-a)

(3) 拷贝构造函数:

① 浅拷贝(默认): 对象的指针成员都指向同一个堆空间，析构时候会释放多次而导致错误
② 深拷贝: 每个对象的指针成员都指向不同的堆空间，不会导致析构时释放多次
**Note that: 当类 B 里有类 A 的对象最为成员，拷贝构造 B 时也要调用 A 的同理: B 的拷贝赋值也要调用 A 的拷贝赋值**

拷贝构造的格式: CN(const CN& other){ data = new int (*(other.data)); }

**Note that: 堆空间上创建对象 CN* obj_ptr = new CN(obj)；调用拷贝构造函数在堆区创建对象**

(4) 转移构造函数:
把原来的对象的数据传递给新的对象，但原来的对象数据=0 和 nullptr
如果没有自己定义的移动构造函数，也没有自己定义的拷贝构造函数，则使用默认移动构造 (只是浅拷贝，不安全)
转移构造的格式: CN(CN&& other){数据转移，堆空间的成员 = nullptr (避免重复释放); }

std::move()是强行把变量转换为右值，移动构造/移动函数的触发条件是 obj 是右值

**拷贝/转移构造的参数是引用，否则就是会无限递归由于函数的参数是一个临时变量 (函数栈帧)**

(5) 重用构造函数:
构造函数 A 里使用另一个构造函数 B，在构造函数 A 的初始化列表里调用其他构造函数可以是本类也可以是其他类的 (对于 is-a 和 has-a 情况要构造基类或是其他类的对象).

Note that:

- 构造函数先构造内部的成员在构造这个对象
- 类的指针 CN* obj = new CN(other 或...) 或 new CN[num]; (要有无参构造函数) 在堆空间上创建对象时会调用拷贝构造或普通构造
- CN arr[2]; 调用无参构造函数 2 次构造 2 个对象，而 CN* arr_ptr[2]; 没有调用构造函数只是 nullptr
- 匿名对象创建完在下一行代码时就会被析构

**Tip 拷贝函数 & 移动函数:**

类里的拷贝函数:　CN& CpyFunc(const CN& obj_1) {.....; return *this;}
使用：obj_2 . cpy_func(obj_1);

类里的转移函数:　CN& MovFunc(const CN&& obj_1) {.....; return *this;}
使用：obj_2 . std::move(obj_1);

## 2 析构函数

~CN(){.....} (默认) 析构函数不能重载 (类有多少个不同的对象就会在最后析构几次) 析构的顺序与构造相反

注意当有类里有成员开辟空间时，析构函数里要 delete 释放成员的堆空间，或关闭文件
(成员在堆上开辟空间，类里要用深拷贝，避免析构时重复释放该堆空间)

堆区上的对象 CN* obj = new CN(…); 需要 delete obj; 或 delete[] arr; 当 delete 释放堆空间上的对象时程序会自动调用析构函数析构该对象

## 3 访问权限

| 修饰符 | 对象外部访问权限 | 子类访问权限 | 本类访问权限 |
|---|---|---|---|
| public | 可以访问 | 可以访问 | 可以访问 |
| protected | 不能访问 | 可以访问 | 可以访问 |
| private | 不能访问 | 不能访问 | 可以访问 |

## 4 友元

类 A 的友元 (friend 修饰) **类 B** 和 **函数(外部函数或类 B 的成员 fun)** 可以通过类 A 的对象访问 A 的所有成员，不受类 A 访问权限的影响 (private, public, protected)
(因此：友元只能通过类 A 的对象去访问类 A 的信息)

由于友元 ∉ 类 A，所以友元不是类 A 的成员从而没有对应的类 A 的 this

## 5 inline 函数(内联函数)

(1) 内联函数是**建议编译器**在调用函数的地方用函数体替换函数调用语句，从而减小调用函数的开销，提高执行效率。

(2) 类内定义的成员 fun 默认是 inline，在类外定义类内声明的成员 fun 要手动设置 inline 在函数定义处

(3) 在 .h 文件里**定义的函数如果**在多个.cpp 里使用这个.h 文件会出现该函数的多次定义问题(链接过程)
因此要用 inline 修饰该函数。

通常 inline 函数用于小而频繁调用的代码

### 6 static 成员

类的不同对象对应的成员信息都是各自的，只有 static 成员是该类的所有对象共享的
static 成员属于整个类，不是属于某个对象，所有对象共享同一份数据，static 成员没有 this，静态变量生命周期从程序开始到结束

static 成员的访问方法：类名::变量名；每个对象也都可以访问但不推荐

(1) 静态成员变量在类内声明，要在类外定义(初始化) 如: int CN::a = 10; (因为 static 变量只初始化一次，类内不允许它分配内存，除非是不分配空间的 static 类型的成员可以初始化，如 static const, ……)
(不直接属于类的 static 变量可以直接初始化，如类内成员 fun 的局部 static 变量不需要类外初始化)

(2) 静态成员 fun 只能直接访问静态成员变量，由于 static 成员 fun ∉ 任何对象从而没有 this，因此无法确定非成员变量的具体值，但可以通过传入对象（参数）访问类里的非 static 成员

### 7 const 成员

(1) const 对象只能读取：**成员变量(不能修改)** 和 **const 成员 fun** (因为它的 this 是 const CN* const) 但可以修改 mutable 成员变量和没有 this 指针的成员（如 static 成员）

(2) const 成员 fun (const 放在函数后面)只能读取：**成员变量(不能修改)** 和 **const 成员 fun** (因为它的 this 是 const CN* const) 但可以修改 mutable 成员变量和没有 this 指针的成员（如 static 成员）。可以通过传入对象（参数）访问和修改其他的成员

Note: mutable 修饰的**成员变量**可以被 const 修饰的对象或成员 fun 修改（mutable 只用于修饰成员变量）。

### 8 this 指针

this 是 CN* const this 是当前正在使用的对象的自身指针（常在成员 fun 里使用）
(在 const 成员 fun 或 const 对象里 this 是 const CN* const 类型，友元和 static 没有 this)

使用情况:
(1) 区分成员变量和函数参数名重复 this->a = a; (左 a 是类的成员，右 a 是参数)
(2) 实现链式调用 (return *this 当前对象的引用，对应函数的返回值 CN&)
(3) 判断拷贝里的自赋值，如 if (this == &other)

const 成员 fun 和 const 对象的 this 是 const CN* const (因此不能直接修改成员内容，除非是 mutable 和没有 this 指针的成员)

### 9 operator 运算符重载

运算符重载格式: <data_type> operator 符号(arguments) {.....} 根据需要设定返回类型和形参类型

类内定义:
(1) 当只有一个参数：对象(*this) → 符号 → 参数
(2) 当没有参数：符号 → 对象(*this)
Note that: 仿函数 <data_type> operator()(arguments){……} 参数个数不固定 使用方法: Obj()(实参);

类外定义:
(1) 当只有两个参数：1st 参数 → 符号 → 2nd 参数
(2) 当只有一个参数：符号→参数 (一般通过 friend 函数访问类内部成员)

```
ostream& operator<<(ostream& os, const Person& p) {
    os << "姓名: " << p.name << ", 年龄: " << p.age;
    return os;
}
(Tip: cout 是类 ostream 的对象)
```

```
类型转换函数 operator <type>(){.....} 通常结合
static_cast<Type>(expression)
operator int() const {
    return val;
}
MyInt a(42);
int b = a;   // 隐式转换：调用 operator int()
int c = static_cast<int>(a);   // 显式转换：也是调用 operator int()
```

### 10 常用关键字

=default 使用默认情况 (也可以有其他重载函数) CN() = default;
=delete 禁止使用某种情况 CN(const A&) = delete;
noexcept (函数末尾) 不抛出异常, (一般是移动构造和移动赋值函数) void func() noexcept{……}
explicit (函数开头) 防止隐式类型转换 (防止=的隐式转换) explicit CN(int x){……}

```
class Safe {
public:
    Safe() = default;
    Safe(const Safe&) = delete;          // 禁拷贝
    Safe(Safe&&) noexcept = default; // 支持移动构造，不抛异常
    explicit Safe(int x);                    // 禁止隐式转换
};
```

## 3《继承 Inheritance》

### 1 继承的概念

(1) 类与类的嵌套

嵌套关系彼此相互独立只是作用域划分的关系
类 A 里封装类 B (类 B 不受类 A 访问权限的影响)，封装的类 B ∉ 类 A 的成员只是作用域关系

(2) has-a 组合
类 B 里有类 A 的对象作为成员，此时类 B 的基础函数(构造，拷贝，相关的符号重载)都需要涉及到类 A 对象的生成 (否则会调用类 A 的默认基础函数)

使用 has-a 的场景
- 类的行为是由若干功能模块组合而成
- 希望低耦合，高可替换，运行时灵活绑定
- 想要使用已有模块，而不是继承它

(3) is-a 继承
① 继承方式 (public, protected, private)

(a) 继承的权限

```
class Base {
   父类成员
};
class Derive: (virtual) 继承方式(public, protect, private) Base {
   子类的成员
};
```

不同的继承方式决定继承的 Base 成员的访问权限，如 virtual public Base 是 public 虚继承

**对应基类里的 static 成员可以 obj.Der::Base::member; 或 Der::Base::member 调用**

如果两个父类中有同名函数只能通过作用域来明确调用哪个父类 (避免二义性)
如：d.Base_A::show(); d.Base_B::show(); 或 在子类里 using Base_A::show;

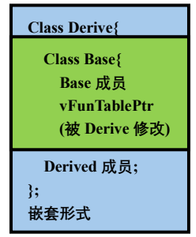
(b) 切片 slicing:

子类转换为父类时会丢失自己的数据只保留父类的数据，因为 Base b = d (子类对象) 这个过程是调用 Base 的拷贝构造 Base(const Base&)，此时隐式转换把子类转换为基类(切片 slicing)，可以用显示转换 Base b = static_cast< Base>(d); (用 explicit 的避免隐式转换)
同理，子对象 d2.Base::operator=(d1); 调用了 Base 的 operator=函数，此时 d2 只保留父类的数据

is-a 和 has-a 一起使用：构造函数：先 Base → 其他类的对象成员 → 最后子类对象

② 普通继承和虚继承

(a) 普通继承:

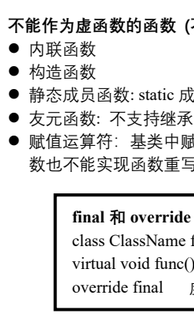



class Derive: 继承访问权限(public, protected, private) Base{……};

子类的基础函数要涉及到直接继承的基类的基础函数(构造，拷贝，赋值，…)，因为要使用基类的成员所以要创建基类的对象

**直接继承里的虚函数表在每个嵌套里只有一个**

(b) 虚继承:




class Derive: **virtual** 继承访问权限(public, protected, private) Base{……};

**子类的基础函数要涉及到直接继承的基类 和 全部虚继承的基类的基础函数** (构造，拷贝，赋值，…)，因为要使用基类的成员所以要创建基类的对象
**最底层子类(此刻创建对象的子类)的基础函数对于它虚继承的全部基类的操作不受中间继承类的影响**

**虚继承里每个继承的基类模块都是唯一的，不会重复，避免使用基类成员时路径导致的二义性 (如菱形继承)**

### 2 函数重定义(hide) & 函数重写(cover)

(1) 函数重定义

子类中定义了与父类同名的函数 (参数列表和返回值可以不同)，此时所有基类的该同名的函数都会被隐藏，是静态绑定

- 函数重定义不支持多态，只是作用域的改变
- 可以用 using Base::fun; 引入这个作用域从而让 Base 里重定义的函数在子类里可以使用，必须在子类的同名函数前面引入。如果函数重定义是同名同参，则还是子类的或指定作用域 d.Base::fun(); 调用基类里重定义的函数

(2) 函数重写

基类里的 virtual 函数，在子类里重新定义 (函数名，参数列表，返回类型完全相同)，子类函数后面建议有 override (避免函数格式不一致)，此时子类的虚函数表会把基类子对象(继承的基类模块)的该虚函数重写为自己的形式，多态是动态绑定，非多态是静态绑定

- 函数重写支持多态
- **不可以**用 using Base::fun; 引入这个作用域让 Base 里重写的虚函数在子类里可以使用，只能指定作用域 d.Base::fun(); 调用基类里重写的虚函数

**不能作为虚函数的函数 (不能重写的函数)**
- 内联函数
- 构造函数
- 静态成员函数: static 成员函数是属于类的，不属于任何对象
- 友元函数：不支持继承，没有作为虚函数的必要
- 赋值运算符：基类中赋值操作运算符的形参很多时候是基类类型，即使声明成虚函数也不能实现函数重写

```
final 和 override 使用
class ClassName final{.....}   禁止这个类被继承
virtual void func() final   禁止该虚函数被子类重写 (只用于虚函数)
override final   虚函数声明处重写父类函数并禁止再次重写
```

# 4《多态 Polymorphism》

## 1 多态的原理和使用方法

(1) 多态原理

多态是通过子类继承基类的虚函数，通过重写或不重写子类虚函数表里继承的虚函数(基类的)，当调用子类里基类的子对象模块里的虚函数时是被重写后的版本，而不是基类原有的版本
子类对象都有自己的虚函数表指针 vFunTablePtr，而继承的虚函数表 vFunTable∈子类 (∉每个单独的对象)，基类指针或引用通过动态绑定到子类对象的指针或引用才能触发多态

**多态满足条件:**

① 存在继承关系
② 基类有虚函数 (有虚函数才会有虚函数表，子类才会继承基类的虚函数表)
③ 基类指针动态绑定到子类对象

(2) 多态的使用方法 (动态绑定)

Base* bptr = &d; 或 Base* bptr = new Derive(…); (此处 Derive d; 子类对象)
Base& bptr = d; (此处 Derive d; 子类对象)

**Note that: bptr 只能调用它在子类里的子对象模块里的内容（虚函数是被重写的版本)**

通常: Base* bptr = dynamic_cast<Base*>(&d); 或 Base& bptr = dynamic_cast<Base&>(d);

如果中间有其他子类，要避免二义性，需要指定路径 (如下所示):
Base_B* bBptr = dynamic_cast<Base_B*>(&d); 指明过渡的路径
Base_A* bAptr = dynamic_cast<Base_A*>(bBptr); 此处 bAptr 的虚函数表是 C 的

Note that: dynamic_cast 的使用条件: ①继承，②虚函数表(基类)

**Note that:**

- Base* bptr = &d; 或 Base& bptr = d; 或 Base* bptr = new Derive(…); 只是在已有对象上建立一个指针或引用，此处不会发生切片，不会构造 Base 对象 (new Derive(…) 只是创建一个子类对象)
  同样: dynamic_cast 也是如此
  **但是 Base b = d; 发生切片，并创建一个 Base 对象通过拷贝构造 (此时不是动态绑定)**

- 构造期间虚函数调用是无效的 (此时 vFunTablePtr 尚未完成构造) 在构造函数中调用虚函数，最终调用的也只是所在类的版本，而不会通过多态调用子类重写的版本

- **多态情况时，基类的析构函数必须是 virtual，**否则只是静态绑定基类的析构函数，导致不能触发多态情况下子类的析构函数。(如果 Base 的析构是纯虚析构函数，子类要提供定义)

## 2 抽象类

包含至少一个纯虚函数的类 (纯虚函数: virtual void f() = 0;)

抽象类的用处:

- 抽象类不能实例化，只能作为接口(interface) 或 基类存在
- 让子类强制实现某些函数。如果子类不重写这些纯虚函数，则子类也是抽象类

## 3 RIIT

C++的特性用于在程序运行时识别对象的真实类型，尤其在多态情况下

typeid 获取对象的实际类型: typeid(*ptr).name()

dynamic_cast 让基类和子类对象之间进行安全的类型转换: dynamic_cast<Derived*>(BasePtr) 前提是该 BasePtr 是子对象模块∈子类 (即含有子类内容)