**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

可以结合匿名空间一起使用，内部程序不暴露，只暴露接口

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) {......}

(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即多态情况, 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 = &c; ap可以动态转换为类C因为ap动态类型原本就是类C), 在虚继承且有虚函数表时要有该转换会更安全, (满足: 继承, 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

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

定后不能再令其引到另一个变量 (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):

系统有自己的默认构造函数, 如果自己定义了就不会调用默认的

(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

(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栈空间, 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栈空间, 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: 栈空间上创建对象CN\* obj\_ptr = new CN(obj); 调用拷贝构造函数在栈区创建对象**

(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, 栈空间的成员 = 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

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, 不受访问类A的访问权限的影响 (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的内容从而没有对应的类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, 从而以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成员是该类的所以对象共享的

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\*) 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\*) 但可以修改mutable成员变量和没有this指针的成员。但可以通过传入对象 (参数) 访

问和修改其他的成员 (如static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）

**9 operator运算符重载**

**10 常用关键字**

**3《继承Inheritance》**

**1 继承的语法**

**2 函数重定义(hide) & 函数重写(cover)**

**4《多态Polymorphism》**

1 多态的原理

2 多态的使用方法

3 抽象类

4 RIIT