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
如果命名空间里的函数或变量是声明，则可以在外部定义:
<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)

3 动态内存

int* ptr = new int(5); 或 int* ptr = new int; 调用默认构造或内建类型值不确定
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，调用默认构造)
::operator new(内存大小); 全局的 new 不是自己定义的重载的函数　**Note that: 只分配空间不调用构造**
::operator new[](num*sizeof(int)); 通常: static_cast<T*>(::operator new(sizeof(T)))　(::operator new→void*)

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）会调用析构
delete[] arr;　释放数组 与 new 匹配使用　**Note that ::operator delete(ptr) 只释放空间不析构对象**
:: operator delete(ptr); 全局的 delete 不是自己定义的重载函数 或 ::operator delete[] (ptr); 不调用析构

placement new: new(ptr) CN(…); 在空间 ptr 上构造一个对象，**返回这个对象的地址，不再是 void***
Note that: 创建的对象大小要匹配空间的大小，**::operator new/delete 需要手动析构，两者匹配使用**

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）
函数里的参数是数组引用，则不会退化为首元素地址，而是一个数组且 sizeof(arr) = 数组长度而不再是指针长度。

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

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)

(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的构造函数

(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 的对象做为成员时，拷贝构造 B 对象时也要调用 A 的拷贝构造
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)；会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，则使用默认的移动构造函数(默认的移动构造函数只是浅拷贝，不安全)
转移构造函数的格式: CN(CN&& other){数据转移，原来堆空间的成员 = 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(普通，拷贝，移动) 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，再构造这个对象
- 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
- 匿名对象创建后在下一行代码时就会被析构
- CN arr[2]; 调用无参构造函数 2 次从而构造 2 个对象，而 CN* arr_ptr[2]; 只是创建了 2 个 nullptr，没有调用构造函数
- 初始化列表里不能使用 this

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); 返回是引用避免出现拷贝构造现象（函数栈帧）

类里的转移函数: CN& MovFunc(const CN&& obj_1) {.....; return *this;}
使用: obj_2. std::move(obj_1); 返回是引用避免出现拷贝构造现象（函数栈帧）

2 析构函数

~CN(){.....} (默认) 析构函数不能重载（类创建多少个不同的对象就会在最后析构几次)
析构与构造的顺序相反

注意当类里有成员开辟空间或打开文件时，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，对象之间的拷贝过程是深拷贝，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	类内部访问权限
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）可以是: **类 B 和 函数(外部函数或类 B 的成员 fun)** 它们通过类 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 指针

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。(inline 放在函数开头)

(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。

Note that: 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，静态变量的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和没有 this 指针的成员，由于 static 成员 fun ∉ 任何对象
从而没有 this，因此无法确定非 static 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

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 {          没有返回类型
    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 (Safe a =5)
};
```

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 表示子类访问父类成员的最高权限)

(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 Base 的成员的访问权限，如 virtual public Base 是 public 虚继承

继承的基类里的 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 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{
   Class Base{
      Base 成员
      vFunTablePtr
      (被 Derive 修改)
   Derived 成员;
};
嵌套形式
```

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{
   (Base 的内容)
   Base 成员;
   vFunTablePtr;
   (Derive 的内容)
   vFunTablePtr;
   vInhriTablePtr;
   Derived 成员;
};
子对象模块，唯一性
```

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 Base 作用域让 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 (子对象调用时如同函数重载)。如果重定义的函数是同名同参，则子对象调用的还是子类的。d.Base::fun(); 可以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
- **不能**用 using Base::fun; (不能用该方法让 Base 里重写的虚函数在子类里使用)
  只能通过子对象指定作用域 d.Base::fun(); 来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) 或 dynamic_cast<Base*>(DerivePtr) 将 DerivedPtr 转换为指向子类 Derive 里的 Base 的子对象模块的指针

5《模板 Template》

1 模板参数的设置

(1) template<typename T, data_type par, template<typename> class BT>

① 类型模板参数: typename T
T 是数据类型，也可以是类

② 非类型模板参数: data_type par
非类型模板参数在实例化时必须在编译阶段就是已知的
如: 整型常量(int, char, bool, enum)，指针常量(函数指针，对象指针, nullptr), ...

③ 类模板模板参数 template<typename CT1, ...> class CNT
实例化时不能有<参数>，它的参数通过模板里其他参数赋予，如果不是类模板模板参数，而是用类模板作为类型模板参数，则 CNT<参数>

Note that: (1) 参数可以有默认值，但默认值后面的参数都要有默认值
(2) 非类型模板参数 par 的数据类型可以是类型模板参数 T，如: template<typename T, T par>)
(3) typename 修饰的变量是数据类型(包括类) (当依赖于模板时，即模板上下文中)
template 修饰的变量是一个模板 (当依赖于模板时，即模板上下文中)
在非依赖上下文里不能使用 typename 和 template

类或模板类里嵌套数据类型: typename CN::data_type 从而区分这个不是一个成员，而是数据类型
类或模板类里嵌套的模板类: typename CN_A<...>::template CN_B<...> B_obj，如果没有模板上下文，则不能用 template 和 typename Note that: CN_A<...>::template func<…>(…))

(2) func 和 class 模板

① func 模板
函数模板定义
template<typename T, data_type par, template<typename, …> class BT>
RT func(arguments of T, ...){关于 T 和 par.....}

函数模板使用
func<Real_Parameters>(Real_arguments);
(Note that: func 的返回类型可以是具体的数据类型也可以是模板参数，此处 RT 是返回类型)

② class 类模板
类模板定义
template<typename T, data_type par, template<typename, …> class BT>
class CN{
RT func(.....); 类内的普通成员 func. , 如果是友元函数(非模版)，类外定义是具体化的

template<typename U, ...> 成员函数/类模板
RT func_T(.....); / class CN2; friend class CN2<T>; 类外引入只允许类型 T
template <typename U> friend class CN2; 类内声明自定义 U
template<typename U, ...> 友元函数/类模板(内部定义/声明)要有该行，如果类外引入不需要
friend void func(…); 友元函数模板外部引入有<空>。友元类模板不需要<空> 但需要指定<U, …>
友元函数/类模板在类内声明/定义不需要<空>和<U, ...>
};

类外定义的普通成员 func
template<typename T, data_type par, template<typename, …> class BT>
RT CN<T, par, …>:: func(arguments of T){关于 T 和 par.....} 友元函数(非模版)没有 CN<T, par, ..>

类外定义的成员 func/类模板
template<typename T, data_type par, template<typename, …> class BT>
template<typename U, ...>
RT CN<T, par, …>:: func_T(arguments of T){关于 T 和 par.....} / class CN<T, par, …>::CN2{…};

类外定义友元 func 或友元类模版
template<typename T, data_type par, template<typename, …> class BT> 类模板声明在类模板定义前
class CN;
template<typename U, data_type par, template<typename, …> class BT> 友元模版所用的模版参数
void func(CN<T, par, BT> obj){.....} 或是先声明 void func(CN<T, par, BT> obj); 在类之后定义

使用过程:
class<Real_Parameters> obj();
obj.func(...);
obj.func<实例化参数>(...);

Note that:
(1) 调用优先级: 同名 func 或类 > 模板 (如果模板能更好的匹配则优先调用模板)
(2) 模板分文件写时，调用模板定义的文件#include<template.cpp>而不是声明的 template.h 因为模板是编译期间定义的，或者把模板都写到.hpp 文件里 (表示这个文件里是模板) 模板在头文件里被多个文件调用不会出现重复定义现象 (函数模板也可以用 inline)
(3) **模版的显示实例化:** template class CNT<具体参数>;
template Real_RT func<具体参数> (Real_DateType1, …); (RT 是返回类型);
(一般在 A.cpp 里显式化实例模版，在.h 里 extern 声明这些显示化实例的模版，在 Other.cpp 里通过.h 直接使用)

2 模版的全特化和偏特化

全特化: 所有模板参数都被指定具体类型
偏特化: 只对部分模板参数进行特化 (只用于类模板，不能用于函数模版，一般用函数模版重载(再写一个函数模版))

template<typename T, data_size par, template<.....> class CNT …>
泛型模板

template<没有特化的模版参数 typename T, data_size par, template<typename> class BT>
func<特化全部模版参数> 或 class CNT<特化某些或全部模版参数 + 非特化参数>
(Note that: 按照模版参数顺序)

模版隐式实例化时会剥夺 const 和引用等信息，或出现引用折叠
引用折叠: 根据实参类型自动推导引用类型
数组参数 → 退化为指针 (除非用引用&)
函数参数 → 退化为函数指针

3 CRTP (Curiously Recurring Template Pattern)

子类将自己作为类型模板参数传给基类

RTP 特点:

- 不使用虚函数
- 在编译期完成子类的行为替换 (静态多态)
- 零运行时开销比虚函数快

```
template<typename Derived>
class Base {
  public:
    void interface() {
        static_cast<Derived*>(this)->impl();
    }
};

class Derived : public Base<Derived> {
  public:
    void impl() {
        std::cout << "Derived::impl()被调用\n";
    }
};
```

```
int main() {
  Derived d;
  d.interface();  // 输出 Derived::impl()
}
friend class Tree<T>;  只允许同类型
template <typename U> friend class Tree;  所有类型
```

4 模版参数包(了解) 可变参数模版

X... args 表示数据形式是 X 的参数包 args (args 里是 X 类型的各种参数), X 也可以是一个数据类型包(里面是各种数据类型) 如:

```
template<typename... T>   typename... 表示 T 是类型模版参数包
void func(T... args){.....}        T... 表示 args 是形参包
```

sizeof...(args)表示形参包里的参数个数　sizeof...(T)表示类型模版参数包里的类型模版参数个数

参数包的展开方式: 一般使用递归, tuple 元组, …展开方法

例如: 一个模版参数, 和模版参数包的形式最适合参数包展开

```
void func(){
    递归结束…….
}
template<typename T, typename... U>
void func(T& first_arg, U&... other_args){
    std::cout<<first_arg<<std::endl;
    func(other_args...);        此处 other_args 就是一个形参包
}
```

6 《Standard Template Library》

1 CONTAINERS

(1) 顺序容器

① vector

vector 数据结构: 动态数组 (连续内存), 容量满时分配更大的内存(通常是当前容量的 2 倍)把旧数据移动/拷贝过去并且释放旧内存。重新分配内存后, 之前所有使用的迭代器, 所有元素的指针, 所有元素的引用都会失效 (由于重新分配了内存, 导致之前元素的指针和引用没有指向元素的新地址, 所以就无效了)。因此每次重新分配内存后, 之前使用的迭代器, 元素的指针, 元素的引用都要重新获取

vector 特点:

1 尾插速度快, 中间插入速度慢
2 能预分配内存就预分配 (reserve(num_capacity)避免频繁开辟空间)
3 如果只是遍历, 查找, 排序, 则用 vector

Note that: vector 有 bool 类型, 是个特殊化形式 (位操作 + 代理封装, 会比普通 vector<char>慢)

② list

list 数据结构: 双链表, 不是连续内存(因此不能迭代器+n), 元素之间的链接是通过指针, 因此不能通过迭代器随机访问, 只能按顺序一个个访问。插入/删除后所有其他元素的迭代器仍然有效, 被删除元素的迭代器失效, 非连续内存的数据没有 capacity

list 特点:

1 任意位置增删速度快
2 遍历性能比 vector 慢 (不是连续内存, 相邻元素之间通过指针链接查找)
3 迭代器不能随机访问 (没有[ ]访问)
4 如果频繁在中间插入/删除, 则用 list

③ forward_list

forward_list 数据结构: 单链表, 不是连续内存, (不能迭代器+n, 也不能向前访问--it), 元素之间的链接是通过指针 (因此不能通过迭代器随机访问, 只能按顺序一个个访问)。插入/删除后所有其他元素的迭代器仍然有效, 被删除元素的迭代器失效, 非连续内存的数据没有 capacity

forward_list 特点:

1 forward_list 是一个轻量级的链表结构, 没有 push_back(要从开头节点才能找到最后的节点位置)
2 适合于在开头插入数据
3 单链表只有"前插"和"删除某个位置之后的节点" (有一个前置迭代器表示第一个元素之前位置)

④ deque

deque 数据结构: 是分段连续存储 (连续的小内存块+中央控制表(map/索引表)), 扩容时只需增加新块, 原来的块完全不动, 支持两端快速插入和删除迭代器会失效, 由于数据有连续存储结构(与 vector 相同的情况)

deque 特点:

1 每个数据块都是连续内存结构如同 vector, 不同的数据块都是链表结构如同 list
2 适合于两端操作 + 偶尔随机访问 (随机访问速度比 vector 慢)

⑤ array

array 数据结构: 静态数组 (连续内存, 创建后不能扩容, 固定长度) 与 C 语言里的数组类似

array 特点:

1 可以使用 data()成员函数获取 array 的首元素地址用于与 C 语言 API 交互
2 在动态内存里 capacity() 和 size() 表示容量和目前元素个数, 而静态内存里没有 capacity() 由于没有动态分配

⑥ string

string 数据结构: 动态数组(连续内存), 类似 std::vector<char> 的结构

string 特点:

1 内部会在末尾放一个'\0' (方便与 C 字符串兼容)
2 成员函数 c_str()获取 C 风格字符串指针返回 const char*
3 元素类型: char 或 wchar_t / char16_t / char32_t 取决于具体类型别名

(2) 关联容器

① map & unordered_map

(1) map 数据结构: 红黑树(平衡二叉树)存储键值对, 默认按 key 升序排列
(2) unordered_map 数据结构: Hash Table 无序排列
如: map<int, string> m2{{1,"A"},{2,"B"}};　//列表初始化 key=int, value=string

map 和 unordered_map 特点:

1 insert 不会修改已存在的键 (对于已存在的键值对不做操作)
2 [ ] 会创建或更新键值对, 即便是访问不存在键值对也会创建。
3 同一个 key 只能出现一次, unordered 的键类型必须可哈希 (默认支持基本类型, 如: string, 或自定义哈希函数)

② multimap & unordered_multimap

(1) multimap 数据结构: 红黑树(平衡二叉树)存储键值对, 默认按 key 升序排列
(2) unordered_multimap 数据结构: Hash Table 无序排列

multimap 和 unordered_multimap 特点:

1 insert 可以插入重复的键, 一个 key 对应多个值
2 不能用 [ ] 访问
3 同一个 key 可以出现多次, unordered 的键类型必须可哈希 (默认支持基本类型, 如: string, 或自定义哈希函数)

③ set & unordered_set

(1) set 数据结构: 红黑树 (平衡二叉树) 默认升序排序
(2) unordered_set 数据结构: Hash Table 无序排列

set 和 unordered_set 特点: 元素唯一, 自动去除重复元素

④ multiset & unordered_multiset

(1) multiset 数据结构: 红黑树 (平衡二叉搜索树) 默认升序
(2) unordered_multiset 数据结构: Hash Table 无序排列

set 和 unordered_set 特点: 可以出现重复元素

Note that: 容器里面有相关的增删查改成员函数, insert, erase, push, find, ...等
不同的容器都有自己独特的成员函数实现所需的数据操作

容器的使用:

- 随机访问多: vector, deque, string (文本存储)
- 频繁中间插入/删除: list, forward_list
- 有序快速查找: 有序关联容器
- 无序快速查找: 无序关联容器
- 固定容量, 小数据块: array

2 ITERATORS

对容器来说: 迭代器是访问内部元素的唯一通用方式 (不暴露底层结构)
对算法来说: 迭代器提供了统一接口, 不用关心容器类型

container<data_type>::iterator itr = Real_ContainerFunIter();　　容器成员函数返回值是迭代器
(此处 Real_Container 是实例化的容器对象)
如: .begin(); .end(); ……

不同容器的迭代器支持的操作形式也不同 (指针也被视为一种迭代器)

- vector, deque, array 可以支持随机访问
- list, set, map, multiset, multimap, 双向访问
- forward_list, unordered_* 向后访问

迭代器失效指的是容器结构发生变化后, 原有迭代器不再指向原来的元素由于扩容, 插入新元素(导致后面元素位置变化), 元素被删除, Hash Table 重建, ... 等原因
本质就是迭代器所指向的元素的地址不再是此时该元素的实际地址

3 ALGORITHMS

(1) 非修改性算法 (只读取, 不改元素)

常用的算法:
for_each;　find;　count;　equal;　mismatch; ……

(2) 修改性算法(会改容器元素)

常用的算法:
copy;　fill;　replace;　remove;　transform; ……

(3) 排序与相关算法

常用的算法:
sort;　stable_sort;　partial_sort;　nth_element; ……

(4) 集合与数值算法

常用的算法:
set_union;　set_intersection;　accumulate;　inner_product; ……

Note that: 算法不依赖容器在, 只依赖迭代器类型, 迭代器类型决定了能用哪些算法

4 FUNCTORS

(1) 函数对象 functor

functor 是有仿函数的类(类或对象可以直接使用( )功能)，通常使用 CN( )调用仿函数（此处是 CN 创建的临时对象调用仿函数)

Lambda 表达式本质是一个有仿函数的类，通过临时对象调用仿函数（Lambda 的名字就是该类的对象）
STL 里很多功能是通过可调用对象（成员函数要通过 bind/mem_fn 包装转换才行）
STL 里提供了一些常用的 functor：算数类，比较类，逻辑类

可调用对象：可以直接使用( )的变量（通常：普通函数，函数指针，函数对象, lambda，适配器转换后的）

(2) Lambda 表达式

```
auto func = [capture_list] (parameter_list) -> return_type {
            function_body };
```

函数名和返回类型都可以省略由编译器自动推导（Note that：仅限简单情况时）

Tip：在 lambda 表达式里通过对象调用成员函数，则 lambda 表达式可以将成员函数转换为可调用对象

Note that：普通的值捕获默认是 const，要修改它们需要 mutable

```
int x = 0;
auto f = [x]() mutable {
    x++;   //此处直接修改外部 x；如果 x+=5；则是 lambda 内部的局部变量 x 不是外部的 x
    return x;
};
cout << f() << endl; // 输出 1
cout << f() << endl; // 输出 2（修改的是内部副本）
```

批量捕获外部值（大量使用 lambda 外部变量)

- [=]：按值捕获所有外部变量，默认为 const
- [&]：按引用捕获所有外部变量
- [=, &x]：默认值捕获，但 x 用引用捕获
- [&, x]：默认引用捕获，但 x 用值捕获

Note that：简写只会捕获 lambda 里实际用到的变量，避免捕获的值悬空(Lambda 运行时外部值已不存在)

5 ADAPTORS

(1) Container Adaptors

① stack (后进先出)

默认用 deque 作为底层容器，也可以用 vector 或 list 作为底层实现，不能直接遍历元素
(底层容器必须支持 push_back, pop_back, back, empty, size)

使用场景:

- 括号匹配
- 深度优先搜索（DFS)
- 撤销（undo）功能
- 表达式求值（后缀表达式)

② queue (先进先出)

默认用 deque 作为底层容器，也可以用 list 作为底层实现
(底层容器必须支持 push_back, pop_back, back, empty, size)

使用场景:

- 任务调度（按顺序处理)
- 广度优先搜索（BFS)
- 消息队列
- 打印队列

③ priority_queue

按优先级出队的队列，默认是 vector<T> 默认是 less<T>大顶堆（最大值优先）内部通常用堆(heap)实现。greater<int>小顶堆（最小值优先)
(底层容器必须支持 front(), push_back(), pop_back(), size()，随机访问)

使用场景:

- 任务调度（优先级高的任务先执行)
- Dijkstra 最短路径算法
- Top-K 问题（找前 K 个最大/最小的数)
- 事件驱动模拟

Container Adaptors 常用的操作:

- push(); 入队
- pop(); 出队
- top(); 访问最高优先级元素或堆顶元素
- empty(); 判断队列是否为空
- size(); 队列中的实际存储的元素个数
- emplace(args, ……) 原地构造元素并插入队列（Cpp11)

(2) Iterator Adaptors

① reverse_iterator

让容器从尾到头遍历和普通迭代器一样，写入时会覆盖元素
容器常用接口：rbegin() / rend()

② insert_iterator

插入迭代器会把写操作转换为插入操作

常见的 3 种插入迭代器的适配器:

back_inserter(container)：把元素追加到容器末尾（调用 push_back)
适用于：vector, deque, list

front_inserter(container)：把元素插到容器开头（调用 push_front)
适用于：deque, list

inserter(container, pos)：把元素插入到指定迭代器位置（调用 insert)
适用于：所有支持 insert 的容器（如 set, map, vector, list)

③ istream_iterator 和 ostream_iterator

输入输出流适配器把输入输出流（cin/cout/文件流等）当作容器来处理

常见的 2 种输入输出流迭代器的适配器:

istream_iterator：把元素从外部（cin 或 file）逐个读到容器里
ostream_iterator：把元素从容器逐个写入输出流(cout, file)

它们能和算法（如 copy, for_each, …）配合使用

(3) Function Adaptors

把函数/仿函数转换或包装成能适应 STL 算法的要求

常见的函数适配器:

std::bind：把一个可调用对象（普通函数，成员函数(需要提供对象)，函数对象, lambda）转换成一个新的函数对象

std::function：把一个可调用对象（普通函数，成员函数，函数对象, lambda）转换成有统一调用接口的可调用对象，从而可以使不同形式的函数（lambda，普通函数，成员函数，仿函数）都能放进一个容器里

std::not_fn：把一个可调用对象（普通函数，成员函数，函数对象, lambda）转换成一个新的函数对象且返回结果逻辑取反（算法需要"条件为假"的情况)

mem_fn：把类的成员函数转换为函数对象（转换后的函数第一个参数是类的对象)，常用于 STL 算法里而不用写 lambda)
例如:

```
auto fn = mem_fn(&Foo::show); //  把成员函数转换成可调用对象
Foo f;
fn(&f, 42);   //  调用 f.show(42)   如同仿函数

结合 bind 使用时: std::bind(std::mem_fn(&CN::memFunc), &obj, other_pars);
                等价于: std::bind(&CN::memFunc, &obj, other_pars);
```

6 ALLOCATORS

负责容器内存的申请与释放（new/delete 的封装)
默认 std::allocator<T>，几乎所有情况下都够用

可以自己根据需要设计 allocator 此处涉及到内存相关的知识（如：内存池)