C++

* C++11新特性
  + auto自动类型推导
    - 用于初始化表达式中推断出变量的数据类型
      * auto a;//错误，应该是auto i=1;
      * 最常见用在迭代器vector<int>::iterator it = v.begin();auto it = v.begin();
      * 基于范围的for循环：for(auto x: str){...}
    - 使用限制
      * auto的使用必须马上初始化，否则无法推导类型
        + auto a;
      * auto在一行定义多个变量时，各变量的推导不能产生二义性
        + int i = 10;auto a = i, &b = i, \*c = &i; // a是int，b是i的引用，c是i的指针，auto就相当于int
        + auto d = 0, f = 1.0; // error，0和1.0类型不同，对于编译器有二义性，没法推导
      * 不能定义数组，可以定义指针
        + int a[10] = {0}; auto b = a; // ok auto c[10] = a; // error，auto不能定义数组，可以定义指针 vector<int> d; vector<auto> f = d; // error，auto无法推导出模板参数
      * 不能作为函数参数
    - auto实际是编译时候对变量进行类型推导，不会对程序的运行效率造成影响
    - auto并不会影响编译速度，编译的时候本来就是要从右侧推导，然后判断与左侧是否类型匹配
    - 原理
  + decltype类型指示符
    - auto可以声明一个变量，而decltype可以从一个变量或表达式中得到类型
    - decltype（exp）
      * exp是表达式，decltype(exp)和exp类型相同exp是函数调用，decltype(exp)和函数返回值类型相同其它情况，若exp是左值，decltype(exp)是exp类型的左值引用
    - auto x=1；auto y=2;decltype(x+y) z;
    - 实用之处是和auto用在函数模板的拖尾返回类型
    - 例如template <typedef T,typedef U> auto add(T x, U y)->decltpe(x+y){return x+y;}
  + nullptr常量
    - 引入nullptr专门用来区分空指针NULL和0
    - 传统的C++会把NULL、0视为同一个东西，例如在调用重载函数时就会出现问题，void foo(char \*); void foo(int i);foo(NULL);//两个函数都会调用
  + lambda表达式（匿名函数）
    - 定义并创建匿名的函数对象，简化编程工作
    - 使用场景
    - 原理
      * 定义完lambda表达式后，编译器自动生成一个匿名类，在运行的时候返回一个匿名实例
    - 基本语法
      * [捕获列表](参数表) mutable 函数选项 -> 返回值类型{函数体;};
        + 捕获列表

[ ]

空，不捕获外部变量

[=]

值传递，所有外部变量

[&]

引用传递，所有外部变量

* + - * + mutable

可修改标识符

可省略，加上后可修改按值传递进来的拷贝（是拷贝，不是值本身）

* + - * + 函数返回值

可省略

当返回值为void，或函数体只有一处return，可以自动判断类型

* + 智能指针
    - 智能指针主要是管理一个指针，管理堆上分配的内存智能指针是一个类，这个类的构造函数传入一个普通的指针，析构函数释放传入的指针避免了普通指针忘记释放内存而造成内存泄漏的问题智能指针会通过析构函数自动释放内存
    - 使用智能指针会出现什么问题，怎么解决
    - auto\_ptr
      * C++98方案，C++11已经抛弃，；不支持复制和赋值，拷贝或赋值时原指针会被置为NULL，但不会报错，存在潜在的内存崩溃问题
      * 也是所有权模式
    - unique\_ptr
      * 所有权模式，严格拥有，保证同一时间只有一个智能指针指向对象，避免了内存泄漏（多个指针指向同一个内存；忘记释放内存）
        + 一个uniquer如何赋值给另一个unique

利用 std::move 将其转移给其他的 unique\_ptr

使用内置函数release和reset

u.release()

释放所有权，返回指针，并将u置为空

u.reset(p)

重置所有权，令u指向普通指针p，如果u不为空，则将原来指向的对象释放。p为空时，直接释放u所指向的内存对象

* + - * + 它不允许其它智能指针共享其内部指针，也不允许unique\_ptr的拷贝和赋值
      * 常用函数
        + unique\_ptr<T> u1 空unique\_ptr，可以指向类型为T的对象。u1会使用delete来释放它的指针；unique\_ptr<T,D> u2 使用一个类型为D的可调用对象来释放它的指针unique\_ptr<T,D> u(d) 空unique\_ptr，指向类型为T的对象，用类型为D的对象d代替deleteu=nullptr 释放u指向的对象，将u置为空u.release() u放弃对指针的控制权，返回指针，并将u置为空u.reset() 释放u指向的对象u.reset(q) 如果提供了内置指针q，令u指向这个对象；否则将u置为空u.reset(nullptr)
      * 通过delete修饰，禁止对象的拷贝
    - shared\_ptr
      * 循环引用问题
        + class B;class A{public: shared\_ptr<B> pb\_; ~A() { cout << "析构A" << endl; }};class B {public: shared\_ptr<A> pa\_; ~B() { cout << "析构B" << endl; }};void testShared\_ptr() { shared\_ptr<B> pb(new B); shared\_ptr<A> pa(new A); pb->pa\_ = pa; pa->pb\_ = pb; cout << "pb 的引用计数："<< pb.use\_count() << endl; cout << "pa 的引用计数："<< pa.use\_count() << endl;}int main(void) { //pb 的引用计数：2 //pa 的引用计数：2 // 且未调用析构函数，说明两块内存一直没被释放 testShared\_ptr(); //跳出函数后，二者的引用计数减一，但还是1，不为0，所以内存得不到释放}

解决办法

class B;class A{public: weak\_ptr<B> pb\_; ~A() { cout << "析构A" << endl; }};class B {public: shared\_ptr<A> pa\_; ~B() { cout << "析构B" << endl; }};void testShared\_ptr() { shared\_ptr<B> pb(new B); shared\_ptr<A> pa(new A); pb->pa\_ = pa; pa->pb\_ = pb;//shared\_ptr可以直接给weak\_ptr赋值 cout << "pb 的引用计数："<< pb.use\_count() << endl; cout << "pa 的引用计数："<< pa.use\_count() << endl;}int main(void) { // pb 的引用计数：1 // pa 的引用计数：2 // 析构B // 析构A testShared\_ptr();}

* + - * + 引入了weak\_ptr弱指针来辅助shared\_ptr，weak\_ptr指针是一种不控制所指向对象生存周期的指针，将weak\_ptr绑定到shared\_ptr，一旦最后一个指向对象的shared\_ptr被销毁，对象就是被销毁
      * 常用函数
        + shared\_ptr和unique\_ptr都支持的操作

shared\_ptr<T> sp 、unique\_ptr<int> up 空智能指针，可以指向类型为T的对象p 将p用作一个条件判断，若p指向一个对象，则为true\*p 解引用p，获得它指向的对象p->mem 等价于（\*p）.menp.get() 返回p中保存的指针，若智能指针释放了其对象，返回的指针所指向的对象也就消失了swap(p,q） 交换p和q中指针 ，p.swap(q)

* + - * + shared\_ptr独有的操作

make\_shared<T> (args) 返回一个shared\_ptr，指向一个动态分配的类型为T的对象。使用args初始化此对象。shared\_ptr<T>p(q) p是shared\_ptr q的拷贝：此操作会递增q中的计数器。q中的指针必须能转换为T\*p=q p和q都是shared\_ptr，所保存的指针必须能相互转换。此操作会递减p的引用计数，递增q的引用计数。若p的引用计数变为0，则将其管理的原内存释放p.unique() 若p.use\_count()为1，返回true，否则返回falsep.use\_count() 返回与p共享对象的智能指针数量：可能很慢，主要用于调试

* + - * 计数模式，共享式拥有，多个指针可以指向同一个对象，使用计数机制来表明被几个对象共享当计数为0时，释放内存
      * shared\_ptr的行为最接近原始指针，是否可以在任何地方代替原始指针并消灭内存泄漏
    - weak\_ptr
      * 解决shared\_ptr指针两个对象互相指向对方，造成引用死锁，使计数失效，而造成内存泄漏的问题
      * weak\_ptr协助shared\_ptr来工作，只能从一个shared\_ptr或weak\_ptr对象构造，弱指针是当指针指向原来空间时，引用计数不再加一，释放时不等待，解决循环引用问题
      * 弱引用
        + 它指向一个由shared\_ptr管理的对象。将一个weak\_ptr绑定到一个shared\_ptr，**不会改变shared\_ptr的引用计数**
      * 常用函数
        + weak\_ptr<T> w 空weak\_ptr可以指向类型为T的对象weak\_ptr<T> w(sp) 与shared\_ptr sp 指向相同对象的weak\_ptr。T必须能转换为sp指向的类型w = p ，p可以是一个shared\_ptr或者weak\_ptr，赋值后w与p共享对象w.reset() 将w置为空w.use\_count() 与w共享对象的的shared\_ptr的数量w.expired() 若w.use\_count()为0，返回true，否则返回falsew.lock() 如果expired为true，返回一个空shared\_ptr；否则返回一个指向w的对象的shared\_ptr
      * 由于对象可能不存在，我们不能使用weak\_ptr直接访问对象，而必须调用lock
  + 强制类型转换
    - cosnt\_cast
      * 解除常量性，去除指向常量对象的指针或引用的常量性（传入的类型必须是指针或引用）
    - static\_cast
      * 完成基本数据类型之间的转换；
      * 将空指针转换成目标类型的空指针
      * 把任何类型的表达式转换成void类型
      * 无RTTI，无安全检查，有安全隐患
        + 能用于多态向上转换，向下转换能成功，但是不安全
    - dynamic\_cast
      * RTTI，运行时类型检查，其他三个是编译时完成的
      * 不能用于内置数据类型，用于类继承层次之间的指针、引用转换
      * 要使用多态，基类要有虚函数
        + 向上转换（子转父）
        + 向下转换（父转子）
      * 转换失败返回NULL
    - reinterpret\_cast
      * 改变指针或引用的类型；将指针或引用转换成一个足够长度的整型；将整型转换为指针或引用类型
    - 为什么不用C的强制转换
      * 不安全
        + C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错
  + std::move()
    - 将一个左值强制转化为右值引用
      * static\_cast<T&&>(t);
    - C++标准库使用比如push\_back等这类函数时，会对参数的对象进行复制，连数据也会复制这就会造成对象内存的额外创建，使用std::move可以避免不必要的拷贝操作
    - std::move是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝所以可以提高利用效率，改善性能
    - 实现原理
  + 左值右值
    - 左值
      * 有名字的，可以取地址的就是左值
        + int a=b+c;//a为左值，b+c为右值
    - 右值
      * 没有名字，不能取地址的就是右值
        + 纯右值

临时变量

非引用返回的函数返回值，表达式（如b+c）

不跟对象关联的字面量值，如true，2，“Hello”

* + - * + 将亡值

C++11新增的跟右值引用相关的表达式，要被移动的对象（移位他用）

右值引用T&&的返回值

std::move的返回值

转换为T&&的类型转换函数的返回值

可以理解为盗取其他变量内存空间的方式获取到的值，在确保其他变量不再使用、或即将被销毁时，通过“盗取”的方式可以避免内存空间的释放和分配，能够延长变量的生命期

* + 可变参数模板
  + final、override
    - final
      * 修饰一个类，表示禁止该类进一步派生和虚函数的进一步重载
    - override
      * 修饰派生类的成员函数，标明该函数重写了基类函数如果一个函数声明了override但父类却没有这个虚函数，编译报错
        + 避免开发者在重写基类函数时无意产生的错误
  + delete
    - 禁止对象的拷贝和赋值
      * struct A { A() = default; A(const A&) = delete; A& operator=(const A&) = delete; int a; A(int i) { a = i; }};int main() { A a1; A a2 = a1; // 错误，拷贝构造函数被禁用 A a3; a3 = a1; // 错误，拷贝赋值操作符被禁用}
  + default
  + constexpr
    - 编译时的常量和常量函数
      * const只表示read only的语义，只保证了运行时不可以被修改，但它修饰的仍然有可能是个动态变量
      * constexpr修饰的才是真正的常量，它会在编译期间就会被计算出来，整个运行过程中都不可以被改变
      * constexpr可以用于修饰函数，这个函数的返回值会尽可能在编译期间被计算出来当作一个常量，但是如果编译期间此函数不能被计算出来，那它就会当作一个普通函数被处理
  + static\_assert
    - static\_assert(true/false, message);
    - 用于在编译期间检查，如果第一个参数值为false，则打印message，编译失败
  + C++最新标准
    - C++20