第四章

\*头文件的代码一般不应该使用using声明

左值：当对象被用作左值的时候，用的是对象的身份（在内存中的位置）。

右值：当对象被用作右值的时候，用的是对象的值（内容）。

\*赋值运算符需要一个（非常量）左值做的其左侧运算对象，得到的结果也仍然是一个左值。

\*取地址符作用于一个左值运算对象，返回一个指向该运算对象的指针，这个指针是右值。

\*内置解引用运算符，下标运算符，迭代器解引用运算符，string和vector的下标运算符的求值结果都是左值。

\*内置类型和迭代器的递增减运算符作用于左值运算对象，其前置版本所得的结果也是左值。

第七章

Mutable 关键字

Explicit 关键字

\*聚合类（aggregate class） 老版本的struct 类

第九章

\*顺序容器： forward\_list(单向链表), array(数组)

选择容器的基本原则

\*除非你由很好的理由选择其他容器，否则应使用vector

\*如果你的程序有很多小的元素，且空间的额外开销很重要，则不要使用list或forward\_list

\*随机访问元素， 使用vector或deque

\*需要在头尾位置插入或删除元素，但不会在中间位置进行插入或删除，则使用deque

\*读取输入时才需要在容器中间插入元素，随后需要随机访问元素，则：

---vector追加数据，然后调用sort函数

---如果必须在中间插入元素，考虑在输入阶段使用list， 输入完成后list中的内容拷贝到一个vector

第十章

\*lambda不能有默认的参数

\*bind函数，普通函数

第12章

\*Shared\_ptr: 允许多个指针指向同一个对象。

\*Unique\_Ptr: “独占”所指向的对象。

\*Weak\_ptr: 伴随类，弱引用，指向 shared\_ptr所管理的对象。

\*Shared\_ptr 独有的操作:

make\_shared, shared\_ptr, p = q, p.unique(), p.use\_count()

\*使用智能指针语法 shared\_ptr<A> a1(new A());

\*某个时刻只能有一个unique\_ptr指向一个给定对象。

\*unique\_ptr被摧毁时，它所指向的对象也被摧毁。

\*weak\_ptr指向由一个shared\_ptr管理的对象。weak\_ptr绑定到一个shared\_ptr不会改变shared\_ptr的引用计数。一旦最后一个指向对象的shared\_ptr被摧毁，对象就会被释放。

第十三章

\*拷贝构造函数不应该是explicit的

\*三/五法则：

拷贝构造好书，拷贝赋值运算符，析构函数，移动构造函数，移动赋值运算符

\*如果需要一个析构函数，我们几乎肯定也需要一个拷贝构造函数和一个拷贝赋值运算符

\*Sales\_data() = default;

\*Sales\_data() = delete; 阻止拷贝

\*不能delete 析构函数；~Sales\_data() = delete 错误

\*swap 函数 自定义

\*拷贝并交换 (copy and swap) ????

\*move函数

\*右值引用，&& 来获取右值引用

\*一个左值的表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值

\*左值持久： 右值短暂

\*右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。

\*移动构造函数不分配任何新内存

\*五个拷贝控制成员应该看成一个整体，定义任何一个拷贝操作，则应该定义所有五个操作。

第十四章

\*函数对象

\*lambda表达式 auto mod = [](int i, int j) { return i %j; } ;

\*function 关键字 可以放入 函数指针，函数对象，lambda表达式

Function<int(int,int)> f1 = add; //函数指针

Function<int(int,int)> f2 = divide(); //函数对象

Function<int(int,int)> f3 = [] (int i, int j) { return i%j; }; //lambda

第十五章

\*final 关键字

第十六章

\*std::move

其他 略

第十七章

\*tuple 类型

希望将一些数据组合成单一对象，但又不想定义一个新的结构来表示，tuple是非常有用的。

\*bitset

\*正则表达式： RE库

\*随机数

第十八章

\*内联命名空间 inline namespace FifthEd {}

\*虚继承

第十九章

略

//////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////

其他重要知识点：

\*std::mem\_fn

\*std::ref

\*多线程相关函数

\*std::future

\*std::packaged\_task

\*std::async

//////////////////////////////////////////////////////////////////////////////////////////////////////

\*\*std::async函数的三种调用方式

std::future<int> result = std::async(mythread);//创建一个线程并开始执行,绑定关系,流程并不卡在这里

std::future<int> result = std::async(std::launch::deferred,&A::mythread5,&a,tmppar); //第二参数是对象引用，才能保证线程里面是同一个对象

std::future<int> result = std::async(std::launch::async, &A::mythread5, &a, tmppar);

auto r = result.get();

\*\*\*\*packaged\_task 结合 std::thread 实现

std::packaged\_task<int(int)> mypt(mythread3);//我们把函数mythread通过packaged\_task包装起来

std::thread t1(std::ref(mypt), 1); //线程直接开始执行,第二个参数作为线程入口函数的参数

t1.join();

std::future<int> result1 = mypt.get\_future();//std::future对象里包含有线程入口函数的返回结果，这里result保存mythread返回的结果

cout << result1.get() << endl;

\*\*std::promise 类模板，我们能够在某个线程中给它赋值，然后我们可以在其他线程中把这个值取出来用;

void mythread(std::promise<int>& tmpp, int calc)

{

//做一系列复杂的操作

calc++;

calc \*= 10;

//做其他运算，比如整整花费了5秒钟

std::chrono::milliseconds dura(5000); //定一个5秒的时间

std::this\_thread::sleep\_for(dura); //休息一定时常

int result = calc; //保存结果

tmpp.set\_value(result); //结果保存到了tmpp这个对象中

}

void mythread2(std::future<int>& tmpf)

{

auto result = tmpf.get();

cout << "mythread result = " << result << endl;

}

int main()

{

std::promise<int> myprom; //声明一个std::promise对象myprom,保存的值类型为int;

std::thread t1(mythread, std::ref(myprom), 180);

t1.join();

//获取结果值

std::future<int> fu1 = myprom.get\_future();//promise与future绑定，用于获取线程返回值

std::thread t2(mythread2, std::ref(fu1));

t2.join(); //等mythread2执行完毕

cout << "I love China!" << endl;

return 0;

}

//////////////////////////////////////////////////////////////////////////////////////////////////////

现代C++教程：高速上手C++11/14/17/20

//////////////////////////////////////////////////////////////////////////////////////////////////////

\*nullptr 替换 NULL

\* constexpr int len\_foo\_constexpr() {

return 5;

}

char arr\_6[len\_foo\_constexpr() + 1]; // 合法

\*std::initializer\_list

MagicFoo(std::initializer\_list<int> list) {

for (std::initializer\_list<int>::iterator it = list.begin();

it != list.end(); ++it)

vec.push\_back(\*it);

}

MagicFoo magicFoo = { 1, 2, 3, 4, 5 };

\*多个返回值

std::tuple<int, double, std::string> f()

{

return std::make\_tuple(1, 2.3, "456");

}

\*decltype

+用法和typeof相似

auto x = 1;

auto y = 2;

decltype(x + y) z;

+委托构造

class Base {

public:

int value1;

int value2;

Base() {

value1 = 1;

}

Base(int value) : Base() { // 委托 Base() 构造函数

value2 = value;

}

};

+继承构造

class Base {

public:

int value1;

int value2;

Base() {

value1 = 1;

}

Base(int value) : Base() { // 委托 Base() 构造函数

value2 = value;

}

};

class Subclass : public Base {

public:

using Base::Base; // 继承构造

};

\* 显示禁用默认构造好书

+ Magic() = default;

+Magic& operator=(const Magic&) = delete;

\*强类型枚举

enum class new\_enum : unsigned int {

value1,

value2,

value3 = 100,

value4 = 100

};

\*左值：表达式后依然存在的持久对象。

\*右值：表达式结束后就不再存在的临时对象。

\*右值分为：纯右值，将亡值

\*纯右值(pure rvalue)：要么是纯粹的字面值，要么是求值结果相当于字面量或匿名临时对象。

+例如：1, 2， 1+2， 非引用返回的临时变量，运算表达式产生的临时变量，原始字面量，Lambda表达式。

+字符串字面量只有在类中才是右值， 其位于普通函数中是左值。

\*将亡值(expiring value)：即将被摧毁，却能够被移动的值。

\*std::array: 固定大小的数组

\*std::forward\_list: 单向链表

\*无序容器： 使用 hash表来实现， 插入搜索复杂度为 O(constant)

+std::unordered\_map/std::unordered\_multimap , std::unodered\_set/std::unordered\_multiset;

\*std::shared\_ptr: 记录共同指向一个对象。Std::make\_shared来分配创建传入参数中的对象。

+auto pt = std::make\_shared<int> (10);

+****unique\_ptr 不支持复制和赋值，unique\_ptr 只支持移动****

****+****shared\_ptr 通常使用在共享权不明的场景。有可能多个对象同时管理同一个内存时。

+对象的延迟销毁。陈硕在《Linux 多线程服务器端编程》中提到，当一个对象的析构非常耗时，甚至影响到了关键线程的速度。可以使用 BlockingQueue<std::shared\_ptr<void>> 将对象转移到另外一个线程中释放，从而解放关键线程。

+避免使用裸指针型别的变量来创建 std::shared\_ptr 指针

\*std::unique\_ptr: 独占的智能指针，它禁止其他智能指针与其共享一个对象。

+unique\_ptr 代表的是专属所有权，即由 unique\_ptr 管理的内存，只能被一个对象持有。

\*std::weak\_ptr: 不会引起引用计数增加。它的唯一作用就是坚持std::shared\_ptr是否存在,其expired()方法能在资源未被释放时会返回false，否则返回true。

+weak\_ptr是一种不控制所指向对象生存期的智能指针，指向shared\_ptr管理的对象，但是不影响shared\_ptr的引用计数。它像shared\_ptr的助手，一旦最后一个shared\_ptr被销毁,对象就被释放，weak\_ptr不影响这个过程。

+weak\_ptr 是为了解决 shared\_ptr 双向引用的问题。