**高性能C++编程**

**前言**

[高德纳](https://www-cs-faculty.stanford.edu/~knuth/)曾经曰过：“过早优化是万恶之源”。

**正确实现**

学会跑之前，至少要先学会走。请先确保程序的逻辑实现正确，能满足最初的设计需求。推荐采用TDD（[测试驱动开发](https://en.wikipedia.org/wiki/Test-driven_development)）的软件开发流程。

**什么时候做优化？**

很多代码实际上没有性能优化的必要，比如说只跑一次的临时性工具。此类程序甚至都没有用C++编写的必要。

做优化前，先明确程序的瓶颈在哪里，才能行之有效，而不是徒做无用功。

如果仔细评估后，仍有性能需求，请优先考虑让程序有横向扩展能力。换句话就是：多线程、多进程、分布式、微服务。

如果成本问题比较显著了，比如说线上已经跑了上万的核心，可能确实才有性能优化的必要。

常用的性能剖析工具：

**Perf**

**Bytedog**

**优化方法**

**Cache优化**

相比于CPU的运算速度，内存访问是相当耗时的操作。因此，写缓存友好的程序是十分必要的。

比如：遍历二维数据

|  |
| --- |
| C++ #include <chrono> #include <iostream>  **constexpr** **int** kSize = 1000; **int** data[kSize][kSize];  **int** test\_sum\_v0() {  **int** sum = 0;  for (**int** i = 0; i < kSize; ++i) {  for (**int** j = 0; j < kSize; ++j) {  sum += data[i][j]; // 注意这里  }  }  return sum; }  **int** test\_sum\_v1() {  **int** sum = 0;  for (**int** i = 0; i < kSize; ++i) {  for (**int** j = 0; j < kSize; ++j) {  sum += data[j][i]; // 注意这里  }  }  return sum; }  **int** main() {  std::srand(std::time(nullptr));  for (**int** i = 0; i < kSize; ++i) {  for (**int** j = 0; j < kSize; ++j) {  data[i][j] = std::rand();  }  }   **auto** start = std::chrono::steady\_clock::now();  **int** sum0 = test\_sum\_v0();  **auto** cost = std::chrono::steady\_clock::now() - start;    start = std::chrono::steady\_clock::now();  **int** sum1 = test\_sum\_v1();  cost = std::chrono::steady\_clock::now() - start;    return 0; } |

|  |
| --- |
| Plain Text > ./a.out  sum0=1991495305, cost=1977845 sum1=1991495305, cost=2770181  > ./a.out sum0=-1789705319, cost=1967020 sum1=-1789705319, cost=3313097  > ./a.out sum0=682011256, cost=1968688 sum1=682011256, cost=3208746 |

**CPU优化**

**避免拷贝**

用emplace簇函数

定义移动构造函数和移动复制函数

在确认安全的前提下，用std::string\_view替代std::string

**向量化指令优化SSE\AVX等**

**内存利用率优化**

**磁盘IO优化**

**网络优化**

**高性能库**

**Json解析**

推荐rapidjson，强烈不推荐boost::property\_tree

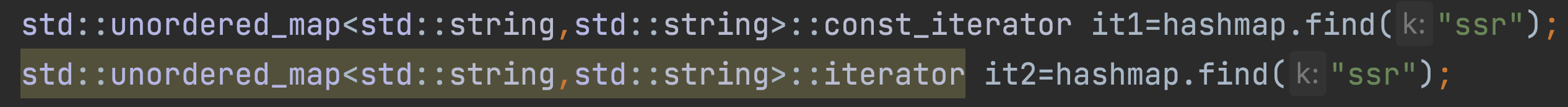
**CPP小技巧**

1. **c++函数签名**

函数名+参数个数+参数类型+参数顺序+const cv

**const\_iterator和iterator**

* 尽可能地使用const\_iterator，而不是iterator
* const\_iterator和const iterator的区别？
* 在unorder\_map和map中
* ~~find能够返回const\_iterator或iterator（根据const重载的两个方法），神奇的是，编译器能够根据返回值类型判断调用哪个find方法~~



|  |
| --- |
| C++ **void** test() {  std::unordered\_map<std::string, std::string> hashmap { {"a", "ssr"}, {"b", "s"} };   std::unordered\_map<std::string, std::string>::iterator it = hashmap.find("a");  std::unordered\_map<std::string, std::string>::const\_iterator cit = it;   } |

* auto接容器的find返回值，默认是iterator，如果map声明为const，则返回const\_iterator

**用emplace\_back代替push\_back**

* 往emplace\_back中传入大对象的raw ptr并且自定义析构器
* 危险⚠️可能造成堆内存大量泄露

|  |
| --- |
| C++ // 大对象BigObj class BigObj { public:  BigObj();  int getBigObjNu() const; private:  int bigObjNu; }   std::vector<BigObj> bigObjVec; auto myDtor = [](BigObj\* bigObj) {delete bigObj;} // 自定义大对象的析构器 bigObjVec.emplace\_back(new BigObj,myDtor); // 此时，要是内存不足导致new操作失败，new BigObj返回的raw ptr无人接管，将造成内存泄露 |

* emplave\_back会匹配带有explicit的构造方法

**编译头文件引用**

* ""头文件： 如下所示，会引用当前目录和g++编译"-I"后的目录，一般用于本项目的头文件。

|  |
| --- |
| C++ #include "constants.h" |

* <>头文件：如下所示，会引用系统目录和g++编译"-I"后的目录，一般用于系统的头文件

|  |
| --- |
| C++ #include <vector> |

const：

函数后置const修饰所有整个这个类的变量，不能改变这个类的变量

package和namespace一样，在C++里面都是namespace

所有变量都需要在当前模块的namespace下面

局部变量在离开作用域之后就会析构，比如for函数的局部变量在break之后就会析构。

**std::bind**

* bind普通函数：第一个入参是函数地址，后面的参数全是这个函数的入参
* bind成员函数：第一个入参是函数地址，需要加上类名，第二个参数是这个类的对象，后面的参数全是这个函数的入参
* 例子如下：

|  |
| --- |
| C++ // 普通函数 auto m\_fn = std::bind(&func, A, B, C); // 成员函数 auto m\_fn = std::bind(&Test::func, this, A, B, C); |

**C++11鼓励用using，不要用typedef**

**拷贝构造函数**

* 针对容器的拷贝，容器会针对每个value执行拷贝构造函数，如下所示，vector会遍历vec2调用N个A的拷贝构造函数

|  |
| --- |
| C++ std::vector<A> vec1 = vec2;  // A的拷贝构造函数入参是固定的const A& A(const A& a); |

* 默认拷贝构造函数：默认的拷贝构造函数会把自动调用类中的所有变量的拷贝构造函数。
* 注意：如果参数中含有指针，拷贝构造之后依然会指向同一份内存，不是深拷贝！！！

|  |
| --- |
| C++ class Test {  public:  TestA\* tmp; }  Test a; // 这里的a和b的tmp变量指针不同，但都是指向同一个内存块的，不是深拷贝！ Test b = a; |

* CRTP：奇异递归模板模式，用于深拷贝多态的子类，比如B是A的子类，Vector<A\*>中的元素是B的多态。如何深拷贝Vector<A\*>？
* CRTP实现：在A中实现clone函数，make\_shared用于深拷贝，返回\*this用于识别多态。

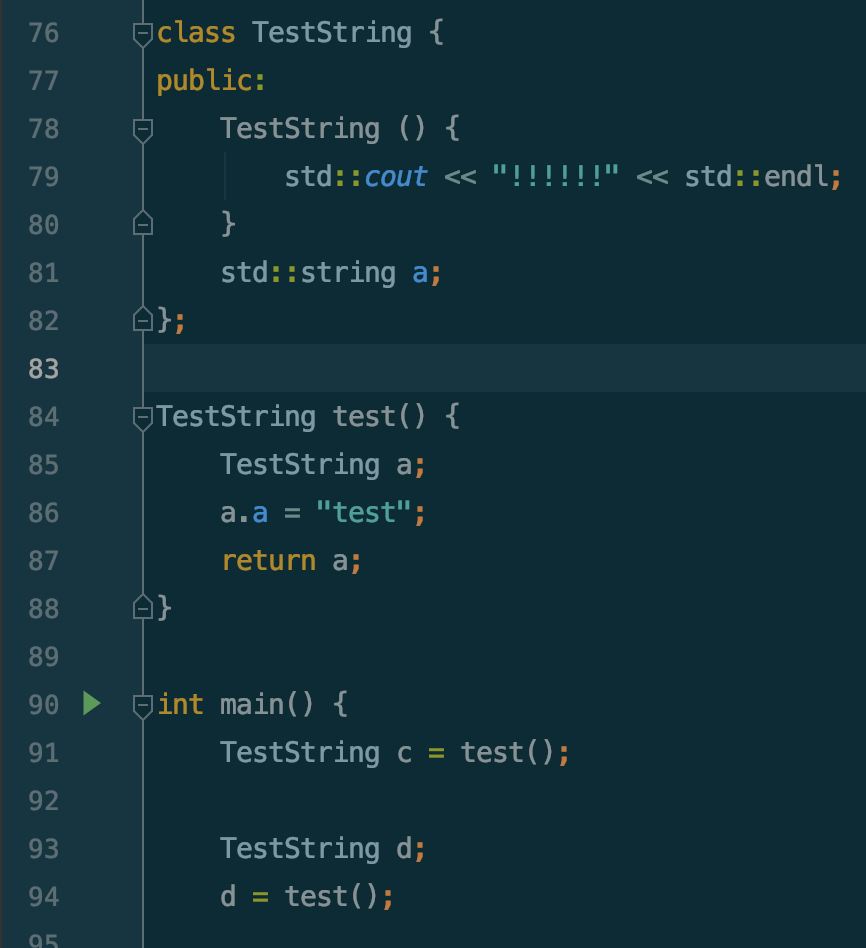
|  |
| --- |
| C++ *// Base class has a pure virtual function for cloning* class VirtualA { public:  virtual ~VirtualA() = default;  virtual std::shared\_ptr<VirtualA> clone() const = 0;  virtual void show() = 0; };  *// This CRTP class implements clone() for Derived* template <typename Derived> class A: public VirtualA{ public:  std::shared\_ptr<VirtualA> clone() const override {  return std::make\_shared<Derived>(static\_cast<Derived const&>(\*this));  }  void show() {  std::*cout* << "A" << std::endl;  }  protected:  *// We make clear Shape class needs to be inherited* A() = default;  A(const A&) = default;  A(A&&) = default; };  *// Every derived class inherits from CRTP class instead of abstract class*  class B: public A<B> { public:  void show() {  std::*cout* << "B" << std::endl;  } };  class C: public A<C> { public:  void show() {  std::*cout* << "C" << std::endl;  }  };  int main() {  std::vector<std::shared\_ptr<VirtualA>> vec;  B b;  vec.push\_back(std::make\_shared<B>(b));  C c;  vec.push\_back(std::make\_shared<C>(c));   vec[0]->show();  vec[1]->show();   std::vector<std::shared\_ptr<VirtualA>> new\_vec;  new\_vec.push\_back(vec[0]->clone());  new\_vec.push\_back(vec[1]->clone());  new\_vec[0]->show();  std::*cout* << new\_vec[0].use\_count() << std::endl;  new\_vec[1]->show();  std::*cout* << new\_vec[1].use\_count() << std::endl; } |

**函数引用**

* 函数从返回值到外面一共发生三次构造函数。第一次是函数内部变量的创建，第二次是函数return的时候复制到外面的拷贝构造函数，第三次是=新变量后的拷贝构造函数。一个&可以减少一次构造函数，因此只有当变量是&以及函数的返回值也是&，变量才会直接指向函数内部的元素。而这种情况下外面的a指向里面生成的类型T，如果这个T是栈变量，直接就core了！很危险！正确的做法是不要返回T，用T&作为函数入参

|  |
| --- |
| JavaScript // 1，a是拷贝构造函数生成的，不指向funcA内部变量 T& funcA() {...} a = funcA(); // 2，a指向funcA的返回值，而函数返回值也是引用，因此a->临时变量->函数内部变量，a指向函数内部变量 T& funcA() {...} a& = funcA(); // 3，编译不通过，a指向临时变量没有意义 T funcB() {...} a& = funcB(); // 最安全，函数入参会直接修改外面的变量 void funcC(T& t){...} funcC(a) |

* guaranteed copy elision**：**上面说到从函数内部到最外层一共需要3次构造函数，但是实际情况不是这样的，因为C++编译器会有RVO机制（return value optimization.）优化。即，会把从return值->外面临时变量，临时变量->外部变量这两个变量的拷贝给move了。真实情况是，下图91行的调用只执行了1次构造函数。
* 其他：94行也是完美转发，但因为93行的构造函数无法避免，85行的构造函数也无法避免，因此需要两次构造
* 实际应用：函数返回值慎用&，不要考虑性能问题，因为完美转发会规避这个问题。



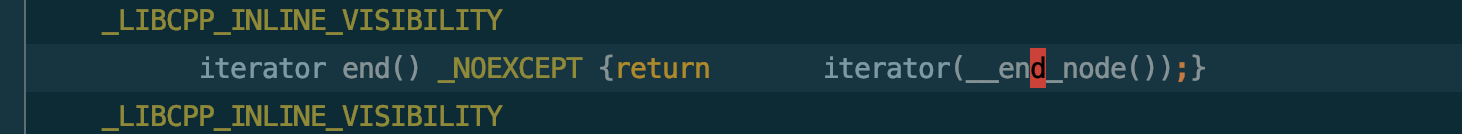
* guaranteed const生命周期：C++保证会把const的栈生命周期延长（C++ guarantee之一），如下图，87行返回了个栈变量。正常情况下91行调用必报错，但是“C++ guarantee”会把const生命周期延长，因此直到91行这个变量失效前，85行这个栈变量也不会失效。



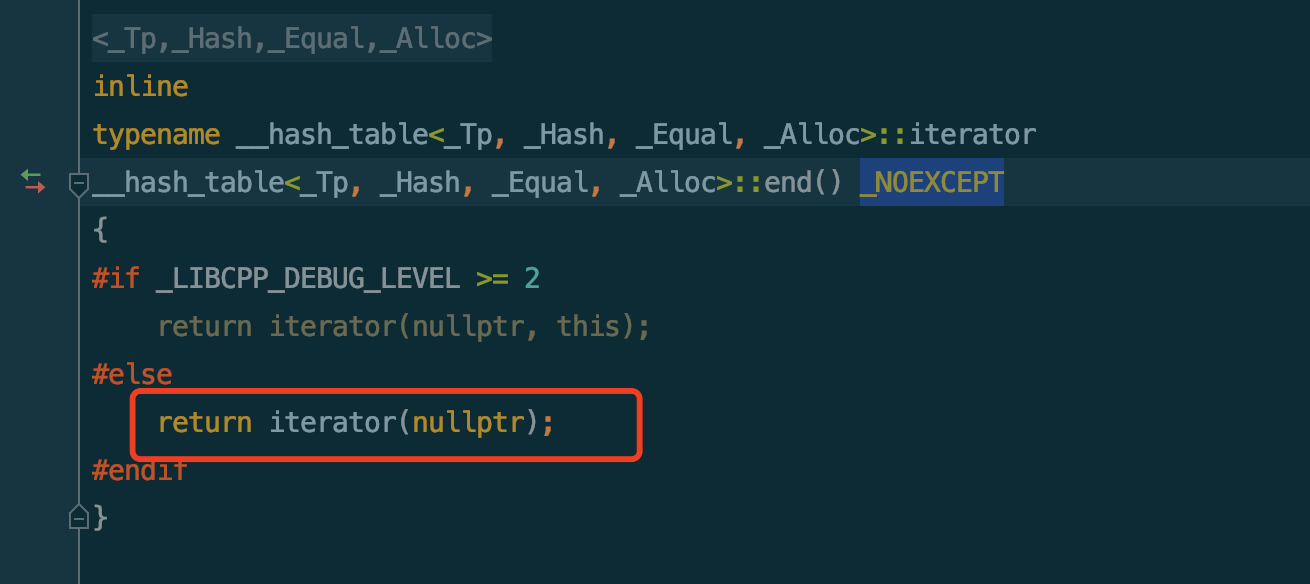
* 右值引用std::move：用于改变函数的性质，提高迭代效率，入参swap到出参后入参直接置为空。
* 注意事项：用move之后入参会置为空，因此禁止move之后的对入参的任何操作
* 建议只采用栈变量作为入参，且move之后不能做任何操作
* T = std::move(T)，输入和输出都是原值，不是指针
* 目的：C++11用于把左值转换为xrvalue，避免数据拷贝。
* 原始左值函数：push\_back(const std::string&)，要发生一次数据拷贝，把外面传入的string拷贝到容器内，数据变成两份。
* 右值函数：push\_back(std::string&&)，右值函数其实与左值函数功能一样，但是又两点不同，一，使用move意味着用户放弃原值的使用权，二，右值与左值是不同的参数，在不影响左值的情况下可以重新实现右值函数，进行性能优化。
* 因此，在某些特定变量如数据量较大的std::string（其内部变量是一个指向真实数据的指针），C++11对其右值构造函数进行优化，不同于const std::string&构造函数的数据拷贝。在用户放弃原值的使用权后，std::string&&被优化为“用新指针指向老内存，并把老指针置为空。”
* 参考：https://www.codenong.com/43528928/
* 总结：右值函数的任何效率优化都不是move本身的效率，而是用户放弃对原值的使用权的情况下，C++11在原有左值函数基础上优化出新增的右值函数产生的性能效率。而这个效率是嵌套的，右值函数才有资格调用更底层的右值函数（高效率）。因此，构造函数的调用次数是不会减少的，只会从拷贝构造函数变为move构造函数。
* 其它：move内部的实现其实也是完美转发，因此针对const变量其实也能move，只不过跟直接拷贝没区别。（const变量是左值）

**容器**

* std::map的end代码如下：因此用end访问second不会core



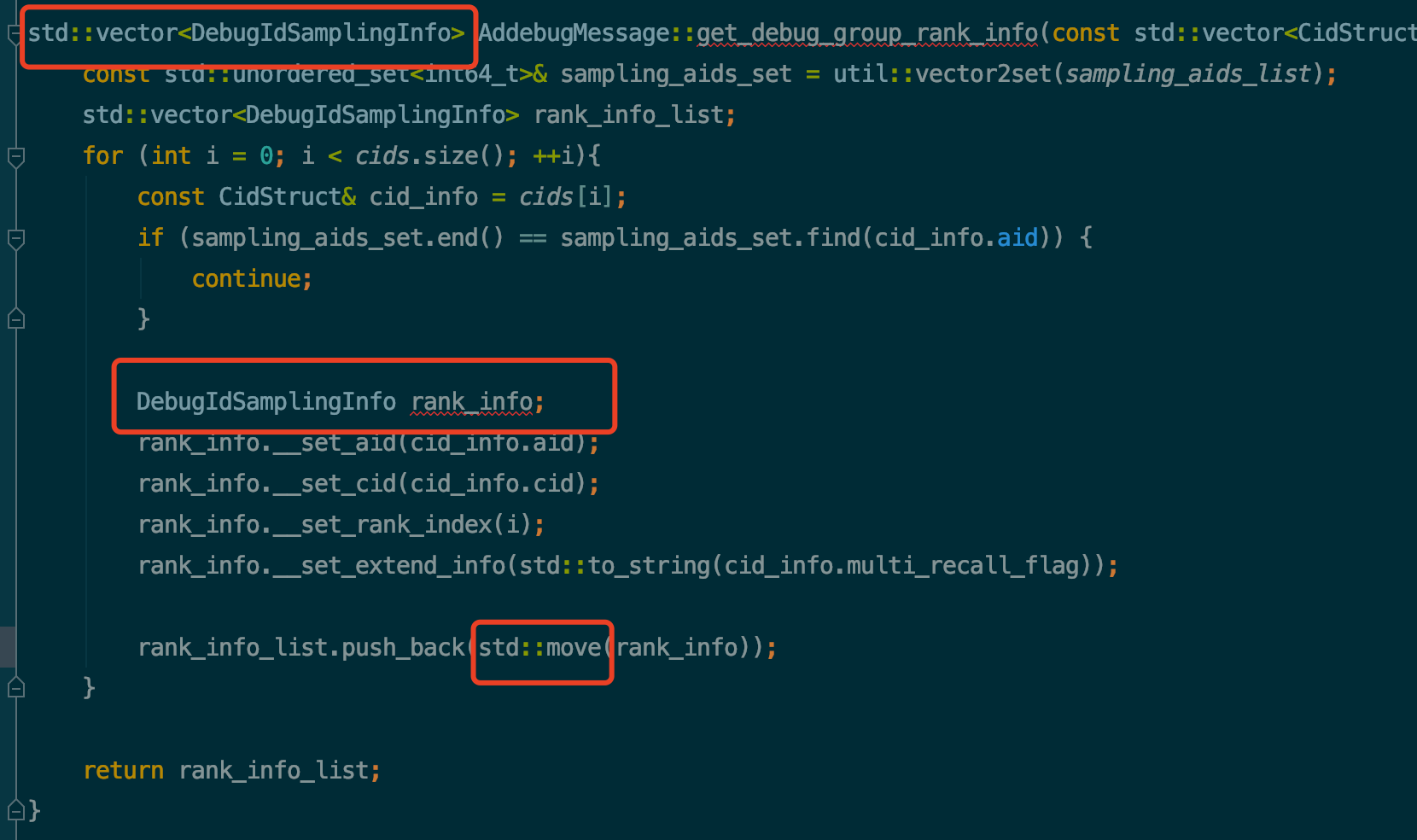
* std::unordered\_map的end代码如下：用end访问second会core



* 析构map：调用erase或者clear后map会自动调用key和value的析构函数，因此只要析构函数写了，就不用担心你的map内存泄漏。前提是你的key和value没有指针。
* 当value为指针：指针的析构函数不等于对象的析构函数，因此需要额外析构。操作办法为先把map中的value记录下来，erase key和value\*，然后根据刚才的记录delete value。erase仅仅是释放了map中的内存，delete操作才是真正调用value的析构函数。
* 优化建议：考虑到map如果存储大对象一般都会存指针（提升性能，指针的拷贝比对象拷贝代价少），因此建议存储为map<key, std::shard\_ptr<value>>，这样既能保证指针高性能，又不需要担心erase的析构的问题。
* std::map的核心结构是红黑树，红黑树是**有序的**！！！即用for each访问的结果是key由小到大排序

**堆和栈内存**

* 栈内存：在离开作用域之后就会被销毁。
* 示例：如下图所示（不建议vector作为返回值，建议用&函数入参，这里只是举个例子），这个move是没问题的，在使用move之后没有对原值进行任何操作，这时rank\_info已经指向空了，离开for函数后rank\_info被销毁。vector的push\_back内部使用的是拷贝构造函数（入参是move的话用的是move构造函数，但效率是否提高看该类型对move构造函数是否有优化），因此数据是两份（move类型有优化是一份，没优化是两份，但外面的数据作废）。无论是哪种情况，析构rank\_info都不会影响到list内部。list本身是栈变量，严禁使用&作为返回值，因此这个函数的返回值会被无效的拷贝2次（返回值拷贝到右值一次，外面右值拷左值一遍）



**智能指针**

* auto\_ptr：禁止使用，用unique\_ptr代替，因为auto\_ptr有拷贝语义，原对象会变得无效。
* unique\_ptr：只有一个unique\_ptr可以指向value
* 支持T[]

|  |
| --- |
| C++ std::unique\_ptr<T> 没有operator[]，对应delete  std::unique\_ptr<T[]> 没有operator\* 和 operator->，对应 delete[] |

* 删除：默认使用delete，可用模板传入delete函数，模板传入（即编译函数存储）对比参数传入（即对象成员存储）
* 模板传入优势：不额外占内存，少一次跳转，性能更好
* 数据转移：

|  |
| --- |
| C++ // unique to unique unique\_ptr<int> p1 = make\_unique<int>(1); unique\_ptr<int> p2(move(p1));  // unique to shared unique\_ptr<int> p1 = make\_unique<int>(1); shared\_ptr<int> p2(move(p1));  // shared to shared shared\_ptr<int> p1 = make\_shared<int>(1); shared\_ptr<int> p2 = p1;  // shared to unique // forbidden, even if shared’ counter is 1 |

* shared\_ptr：
* 特点：大小是普通指针两倍，引用计数的内存是动态分配的，原子操作，性能差一些。
* make\_shared：对比new的区别有两个，1，提高性能，shared\_ptr本身的控制器也占内存，make\_shared会把控制器的内存和value的内存开辟在同一片地方，这样访问速度快。2，异常安全。可参考：https://www.jianshu.com/p/03eea8262c11
* reset：可用于让shared\_ptr指向一个对象
* 删除：默认使用delete，成员对象存储，用参数传入。
* 成员对象存储：优点是不同的变量可以不同的析构器，缺点是有额外的空间占用

|  |
| --- |
| C++  auto loggingDel = [](Widget \*pw)  {   makeLogEntry(pw);  delete pw; }; // unique\_ptr析构器：模板 std::unique\_ptr<Widget, decltype(loggingDel)> upw(new Widget, loggingDel); // shared\_ptr析构器：成员对象存储 std::shared\_ptr<Widget> spw1(new Widget, loggingDel1); std::shared\_ptr<Widget> spw2(new Widget, loggingDel2); |

* 使用教程

|  |
| --- |
| C++ auto p = new Widget();  std::shared\_ptr<Widget> p1 = std::make\_shared<Widget>(); //新增对象 std::unique\_ptr<Widget> up(new Widget()); //新增对象，性能差于上面  std::shared\_ptr<Widget> p2(std::move(up)); //拷贝对象，创建一个新的控制块  std::shared\_ptr<Widget> p3(p); //拷贝对象，创建一个新的控制块   std::shared\_ptr<Widget> p4(p1); //指针指向原有对象，不新增控制块，谨慎使用 ~~std::shared\_ptr<Widget> p5；~~ ~~p5.reset(p1); //指针指向原有对象，不新增控制块，谨慎使用~~ |

* 注意事项：shared\_ptr循环引用会导致引用计数内存泄漏，可用std::weak\_ptr解决如下：

|  |
| --- |
| C++ struct A; struct B; struct A {   std::shared\_ptr<B> pointer;   ~A() {   std::cout << "A 被销毁" << std::endl;   } }; struct B {   std::weak\_ptr<A> pointer; //如果是std::shared\_ptr<A> 则会内存泄漏  ~B() {   std::cout << "B 被销毁" << std::endl;   } };  int main() {   // 实现循环引用，但weak\_ptr不会内存泄漏  auto a = std::make\_shared<A>();   auto b = std::make\_shared<B>();   a->pointer = b;  b->pointer = a;    // 如何使用weak\_ptr  // std::weak\_ptr 没有 \* 运算符和 -> 运算符，所以不能够对资源进行操作，它的唯一作用就是用于检查 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  if (b->pointer.expired()) {  // weak\_ptr没有\*和->,需要强转为shared\_ptr才能用  ((std::shared\_ptr)b->pointer)->get\_value();  } } |

**Nullptr**

* shared\_ptr的默认值是nullptr，但普通指针\*的默认值不是nullptr，注意，如果用普通指针要手动设置为nullptr

允许

* 函数返回值
* const &限定条件：
* 函数入参
* std::move限制条件：
* 作为push\_back和emplace的入参
* 入参是栈元素
* 入参在move之后不能再用

不允许：

* 函数返回值如果是栈变量，禁止使用&

**左右值**

* 定义：
* 左值（lvalue）是非函数、非void的对象，一般用于**定义存储位置**，比如下标[]、间接运算符\*，指针成员访问->
* 另外，引用其实就是左值传递，把这个值的地址进行传递，因此不会发生拷贝本身。
* 右值（rvalue）是非左值的对象，一般用于**定义运算后的结果本身**，比如算术运算符+-\*/，取地址符&
* 三种基本转换：左值到右值、数组到指针、函数到指针。这三种转换都是自动转换的。
* Copy elision：
* NRVO (named return value optimization)：起初RVO技术仅支持匿名变量的优化，后期才加入对具名变量的优化。
* RVO：RVO即Return Value Optimization，是一种编译器优化技术，可以把通过函数返回创建的临时对象给”去掉”，然后可以达到少调用拷贝构造的操作。
* ~~说人话：函数return一个大对象，如果对象有move构造函数，会自动优化成move，不会重复调用拷贝构造函数。即下面这种重复调用的用法只会拷贝一次。（如果对象是自建的，没有move构造函数，则会退化成拷贝构造函数，std等第三方容器肯定会优化）~~
* 限制：在C++中没有限制，C++17 guarantee这种优化。低于C++17的编译器基本100%会优化，企业应用级别可以认为100%会进行优化。
* 应用场景：函数return一个vector，map这种大的std对象，性能也会很好。

|  |
| --- |
| C++ T f() {  return T(); } // 只调用一次构造函数，另外两次拷贝编译器会自动优化成move T x = T(T(f())); |

* 万能引用：模板函数的入参一般会用&&，因为&&是万能引用，能分别接受左右值。

|  |
| --- |
| C++ template<typename T> void func(T& param) {  cout << "只能接受传入左值" << endl; }  int main() {  func(2019); // 会报错，func只能接受左值，2019没有name，是右值  return 0; }  // 万能引用例子 template<typename T> void func2(T&& param) {  cout << "传入的值可以是左右值" << endl; } |

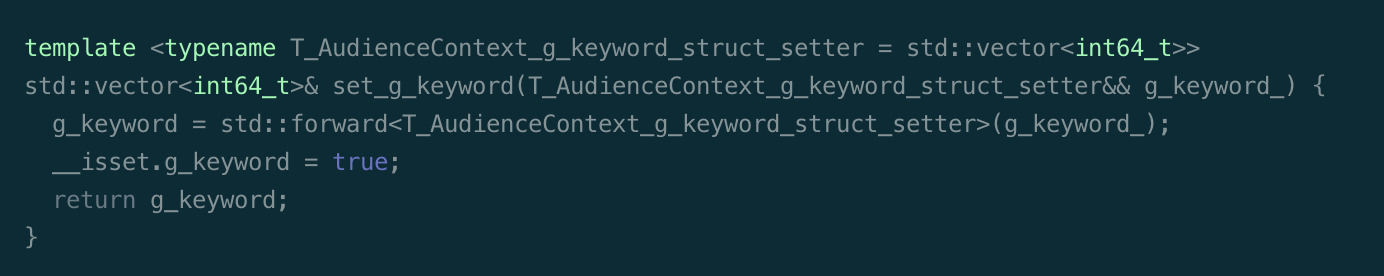
* 引用折叠：只有传入的参数为右值且函数形参也是右值，最终结果才是右值，否则都是左值，也就是左值有传染性。
* 例子：

|  |
| --- |
| C++ template<typename T> // 右值形参 void warp(T&& param) {  // 这里的param是右值，但是再往下传递给func就是左值了，因为param有名字了！  func(param);  // 那怎么办呢？  // 1. 可以用std::move解决，但这样做如果原本是左值这里会变成右值，有点不太合理  func2(std::move(param));  // 2. 完美转发，见下方 }  int main() {  // 右值实参  warp(2019);  return 0; } |

* 完美转发：用std::forword保留参数左右值属性
* 例子：

|  |
| --- |
| C++ template<typename T> // 右值形参 void warp(T&& param) {  // 2. 完美转发，如下所示，原本是左值就是左值，原本是右值就是右值  func(std::forward<T>(param)); }  int main() {  // 右值实参  warp(2019);  return 0; } |

* 应用场景：thrift的set\_XXX函数，当XXX是容器的时候就是用完美转发，如下所示



**unordered\_map和unordered\_set**

* unordered\_map：通过[]访问，如果不存在则调用默认构造函数创建
* at、[]和find：
* []：用[key]访问容器，返回value的引用。若不存在key则直接创建，不能用于const容器，建议用于写key。
* find：用find(key)访问容器，返回value对应的iter。如果不存在则返回map.end()，建议用于读key。
* at：用at(key)访问容器，返回value的引用。不存在key会抛异常（慎用）

**Initialize：初始化列表，避免写多行代码**

* 示例如下：一行代码搞定初始化 + 赋值

|  |
| --- |
| C++ std::map<std::string, std::vector<int>> mapOfOccur = {{"Riti", {3,4,5,6}}, {"Jack", {1,2,3,5}}}; |

* ⚠️警告：initilize\_list里的成员默认被const&修饰，也就是无法支持移动语义，传入大对象可能导致惊人的消耗

**请在某些情况下考虑使用前置声明代替头文件**

https://blog.csdn.net/EmSoftEn/article/details/50374232

* 背景：如果在.h文件中include过多的其他h文件会导致所有include当前h文件的cpp文件过大，编译效率差。前置声明就是解决这种过大h文件问题的存在
* 解决办法：如下图所示，左侧的就是前置声明，无需额外include Data.h。中间的是include写法，会导致h文件很大

|  |  |  |
| --- | --- | --- |
|  |  |  |

* 注意：上图最右侧的写法会编译失败，因为前置声明有限制。图三中因为Date d定义了一个Date类型变量，编译器为d分配内存空间的时候必须知道d的大小，必须包含定义Date类的Date.h文件。（任何Date类型均不可以，包括复合类型如map<int, Date>等）
* 包含变量的解决办法：用指针代替，定义成Data\* d即可。因为指针大小是固定的。
* 前置声明的有效范围：
* 成员变量、成员函数的入参和出参等：在.h中的所有类型都需要编译器评估大小
* 引用和指针都可以用前置声明，但对象不可以（引用本质就是指针）

**boost::optional<T>**

* 定义：容器语义，包含了无效值的定义。类似于包了一层pb和thrift中的is\_set函数，用于判断到底是真实的0还是not\_set。
* 例子：

|  |
| --- |
| C++ boost::optional<int> a; if (FLAG\_enable\_XXX) {  a = 100;  } if (a) {  // been seted } else {  // not seted } |

静态链接库&动态链接库

* 静态链接库：常见的.a和.o文件，可以理解为代码的打包。静态链接会函数链接到可执行文件中，这样可执行文件就包含了所有代码。仅在编译时生效。
* .a和.o的区别：.a其实就是.o的打包，内部是.o的list
* 静态链接库vs源码：装载速度比源码快，但代码不可读且不能debug。
* 优点：可执行程序包含所有代码，换个机器照样执行。
* 缺点：每个进程都有自己独立的静态链接库且不共享，因此如果静态链接库相同则会存在多个拷贝。
* 动态链接库：常见的.so文件，可以理解为本机提供的外部接口。动态链接并不会把代码拷贝到可执行文件中，仅仅是加入函数的描述信息，当进程开始运行后需要调用函数时，动态链接根据描述信息执行.so内部的代码。主要在运行阶段生效。
* 优点：所有进程共享一份.so的内存。.so升级不需要升级可执行文件，换一个.so进程照样跑起来。
* 缺点：可执行程序不完整，不能单独执行。

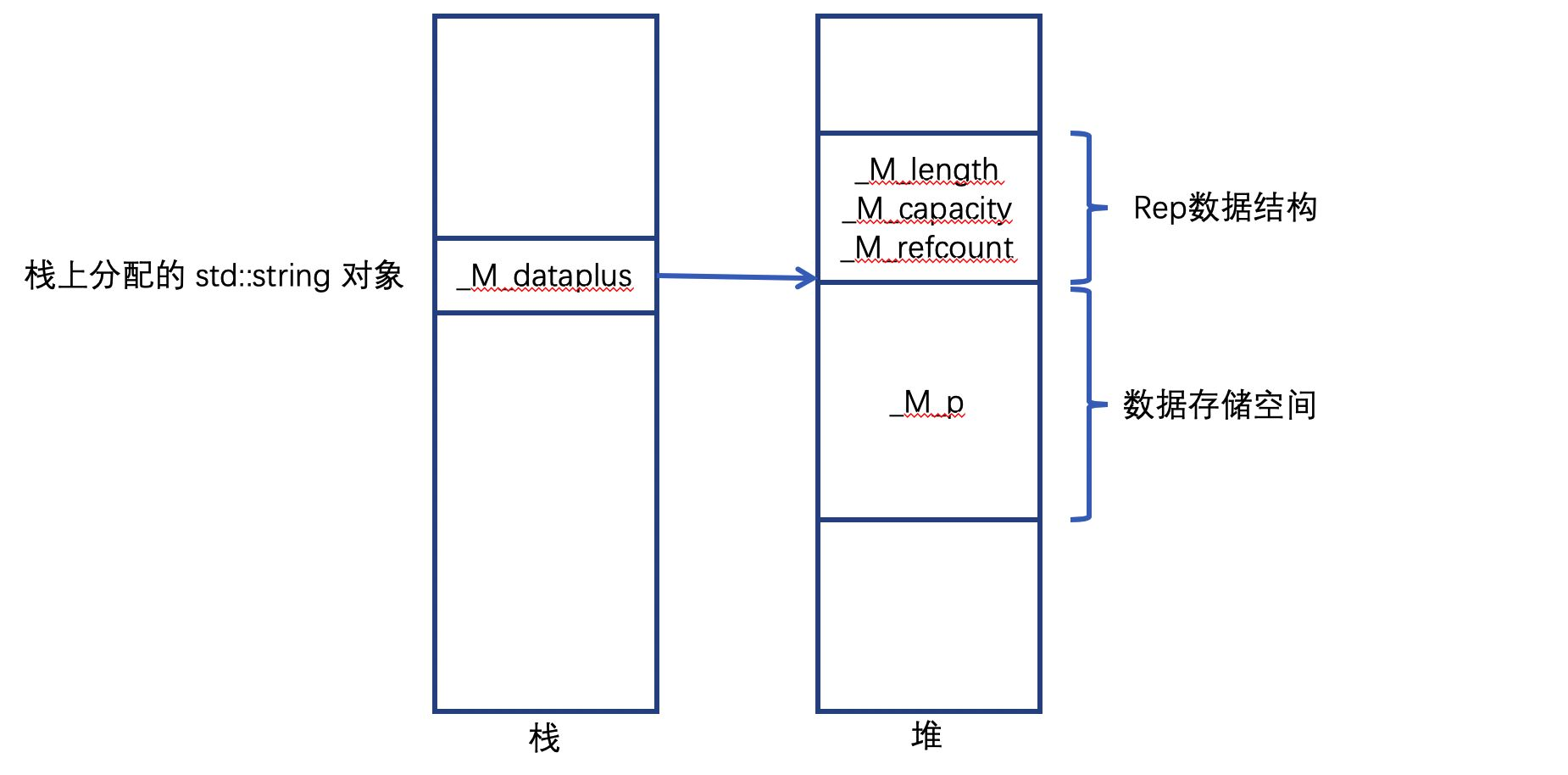
**禁用引用的情景**

* 函数的返回值：禁用&，如果返回值是局部变量，外面收到的直接是空（被销毁）
* ~~接函数返回值：禁用&，如果直接获取函数内部的原变量可能出事（该变量可能会被函数内部销毁或pop）~~

**std::string底层**

[C++小分享：string](https://bytedance.feishu.cn/docs/doccnDdfoSe3F0TWeHIl9PqLDWd#)

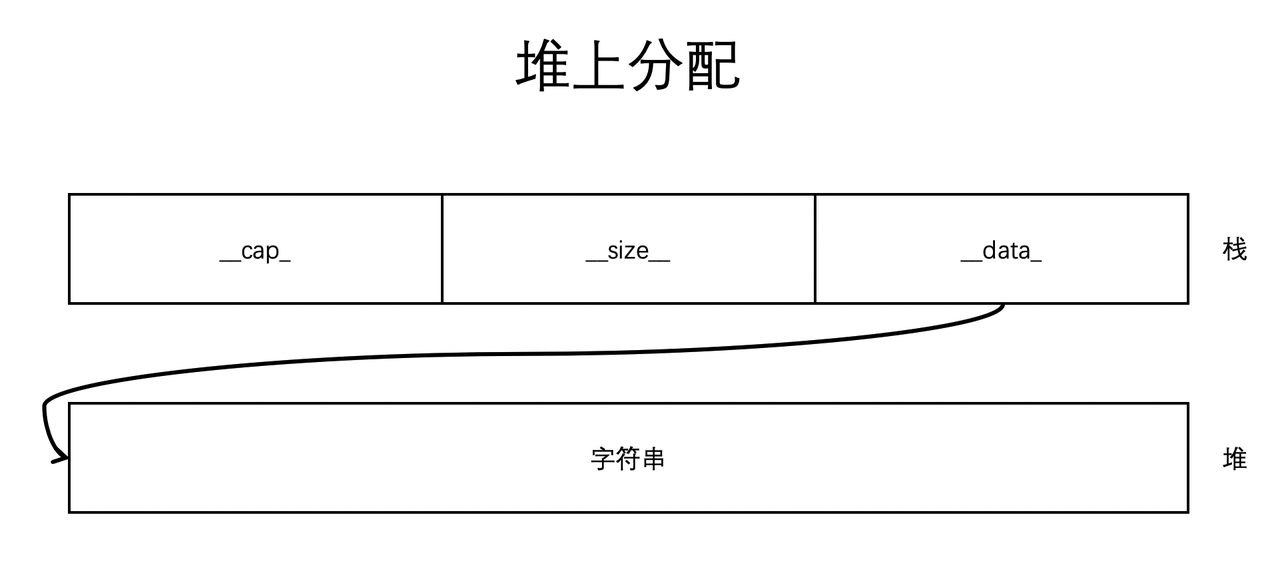
* 小知识：C++的string结构是在栈上保存字符串容量、大小和堆上申请的字符串实体的地址。
* COW(copy on write)：
* 写时复制策略：当string产生复制行为如a = b，不会直接进行字符串拷贝，而是会把指针指向之前那个string并给引用计数加一。当string需要write操作的时候才会进行真正的copy行为。析构的时候也只需要把引用计数减一就行。

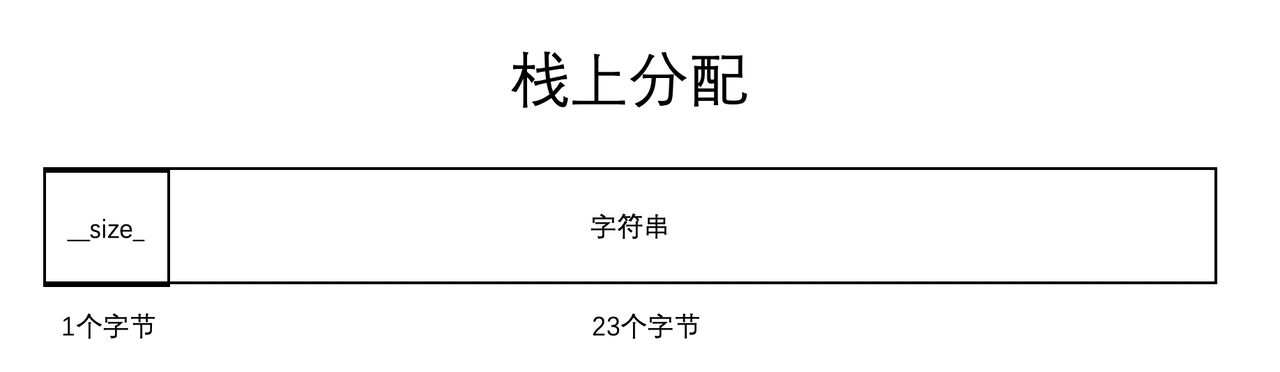


* 优点：拷贝效率高，避免了大量拷贝操作。
* 缺点：会引入悬挂指针问题，如下

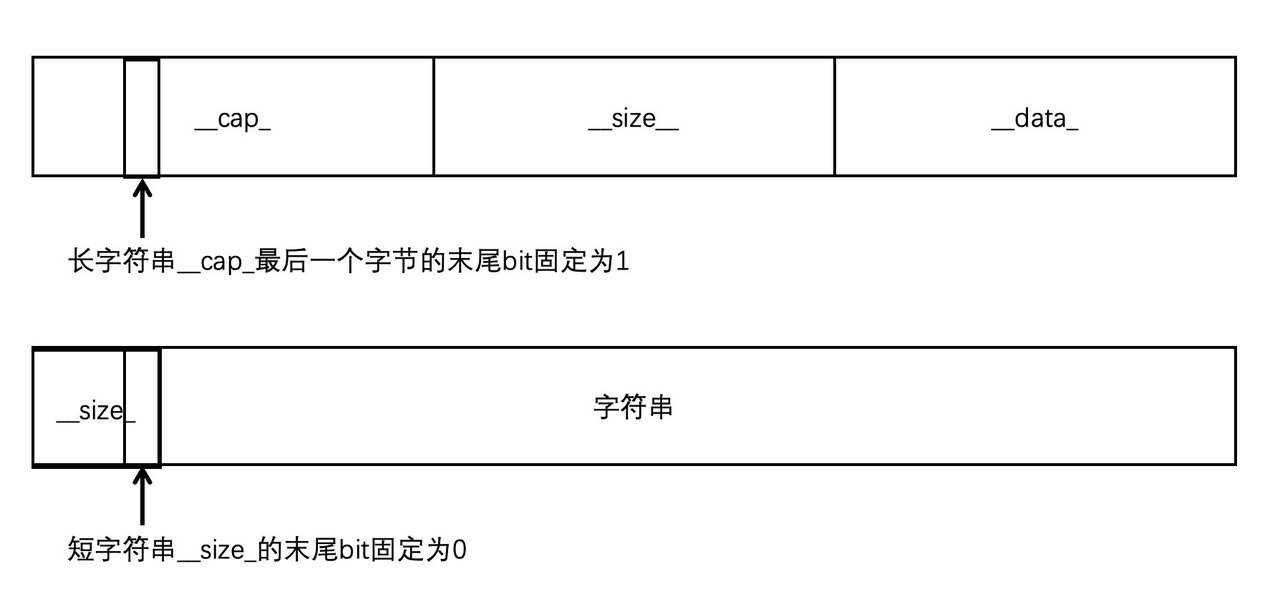
|  |
| --- |
| C++ **std::string s = "str";** **const char\* p = s.data(); // 指向s的空间** **{**  **std::string s2(s); // COW机制起作用**  std::cout<< **s[0] << std::endl; // []运算符会导致s需要复制一份新的数据**  **// s2析构掉，原先的p指向的内存空间被释放** **}**  **std::cout << \*p << '\n'; // Core dump，p是悬挂指针** |

* PS：为了保证COW实现的正确性，string只得统统认定operator[]和at()具有修改的“语义”。哪怕原本的目的是read。
* 应用场景：GCC5.x以下的版本，如GCC494
* SSO（small string optimization）：
* 小字符串优化策略：SSO策略下，彻底舍弃COW，任何string拷贝都是深拷贝。在这个基础上对短字符串进行优化，在于：短字符串模式下，直接将其数据存在栈中，而不去堆中动态申请空间，避免了申请堆空间所需的开销。用24字节作为区分，长、短字符串结构分别如下：



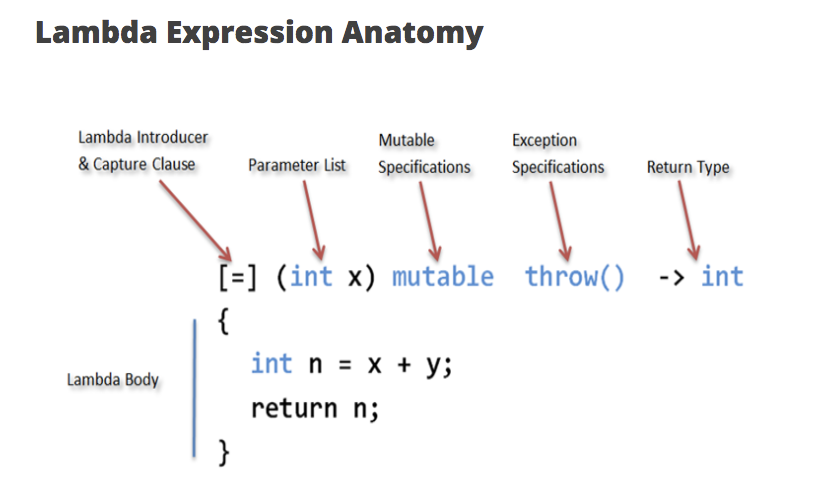


* 如何判断长、短字符串：
* 通过一个 bit 标志位来判断的。
* 长字符串 \_\_cap\_最后一个字节的末位 bit 固定为 1
* 短字符串 \_\_size\_ 的末位 bit 固定为 0



* 由于引入了这个标志位：
* 长字符串的容量就必须为偶数（末位只作为标志位，真实容量 = \_*cap* - 1）
* 短字符串的长度保存时需要左移一位，而取出是需要右移一位，用于保存末位的 0
* 优点：都是深拷贝，没有悬挂指针问题。
* 缺点：SSO本身设计上性能是不如COW的，但因为SSO基于C++11之后，因此像右值、完美转发等概念可以引入，因此性能并不差。短字符串拷贝会优于COW，但长字符串拷贝会差于COW。
* 应用场景：GCC5.x以上的版本，如GCC830
* folly::fbstring
* facebook设计的string，取COW和SSO的优点设计，思路如下：
* Small Strings (<= 23 chars) ，使用 SSO，即栈内存模式
* Medium strings (24 - 255 chars)，使用 eager copy，即完全堆内存深拷贝模式
* Large strings (> 255 chars)，使用 COW，即引用计数模式

**Lamdba表达式**



* 原理：本质就是一个匿名对象的函数对象，与直接构造一个匿名对象然后把里面的函数赋值给func A是一样的。
* 捕获：捕获是lambda的一个功能，add3就是用的捕获功能获取local的a和b。作为入参，a&b和c完全不是一回事，**a&b是通过捕获传入lambda的，而c是通过入参传入的**。在汇编层面，a&b的传入方式与下面A2中成员变量a&b的函数传入方式比较类似，都是在初始化阶段把数据压到自己的栈里面，然后使用的时候通过从栈上位置偏移获取，而c作为函数入参则是通过外部寄存器获取。两者在性能上无太大区别，基于可读性建议用后者。

|  |
| --- |
| C++ // a和b使用了lambda的捕获功能传入函数对象中 int a = 1, b = 2; auto add3 = [=] (int c) -> int {  return a + b + c; }; add3(1);  struct A2 {  A2(int a, int b) : a(a), b(b) {};  int operator()() const {  return a + b;  }  int a;  int b; }; |

**Future**

* 异常捕获：std的future会把子线程的异常接住并且往外抛，因此在future.get()使用try catch可以捕获子线程的异常。
* 背景：线程之间是平级的，因此父子线程之间的异常也是隔离的，子线程异常，父线程是接不住的，可以用最基础的std::thread尝试，是会宕机的。
* std::futures是future\_error把exception\_ptr传给父线程，exception\_ptr：https://stackoverflow.com/questions/233127/how-can-i-propagate-exceptions-between-threads
* 伪代码：

|  |
| --- |
| C++ class Future {  std::thread thread;  static std::exception\_ptr teptr = nullptr;     void Future(std::function func) {  // 简单封装，对子线程的异常捕获  std::function tmp = []() {  try {  func();  }  catch (std::exception e) {  teptr = std::current\_exception();  }  }  thread = std::thread(tmp);  }    std::Future get() {  // 如果子线程抛异常，把异常抛给父线程  if (null != teptr) {  std::rethrow\_exception（teptr）;  }  thread.get();  } } |

**Promise**

* 定义：在future的基础上追加回调函数，并在合适的时机调度
* 用法：如下所示，建立一个promise后获取与之对应的future，然后建立一个回调函数。注意，这里的future是空的。然后调用setValue后会执行future和then的回调函数。如下面标准输出所示，future和回调函数会在调用setXXX后马上执行，注意f.isReady()的变化。

|  |
| --- |
| C++ void foo(int x) {  // do something with x  }  // ...   cout << "making Promise" << endl;  Promise<int> p;  Future<int> f = p.getFuture(); // 获取这个promise底层的future，不设置就是空  f.then(foo); // 只设置回调函数，不设置future  cout << "Future chain made" << endl;  // ... now perhaps in another event callback    p.setValue(42); // 执行future及其回调then     // 标准输出 making Promise Future chain made fulfilling Promise 0(f.isReady() == false) foo(42) Promise fulfilled 1(f.isReady() == true) |

* via & executor：
* 定义：via和executor一般结合使用，例如via(executor)用于限定某个回调函数的执行环境
* 用法：如下所示，下面的回调函数用exe1限制y的环境，用exe2限制了z的环境。例如y是cpu密集型，exe1是个大线程池，z是小任务，exe2是个小线程池。
* Folly提供的executor：ThreadPoolExecutor(线程池，支持调整大小)、ManuelExecutor（手动执行，一般用于测试）等，一般用静态变量传入。

|  |
| --- |
| C++  Future(x).via(exe1).then(y).via(exe2).then(z); |

* collectAll：
* collectAll()：就是Java的invokeAll函数，会等所有future.isReady()都为ture才会离开，哪怕中间有future有异常。入参是vector<future>
* collectAll() variadic：与collectAll类似，支持变长参数。如collectAll（a, b, c ....），每个Future的返回值可以不一样。
* shared\_promise：
* 定义：与promise几乎一样的接口，唯一不同在于可以支持跨线程调用，只执行一次。当A线程执行过这个promise之后，B、C线程调用会直接得到isReady的结果，避免重复调用。
* Effective C++建议：
* **不要用默认捕获：**
* 引用的默认捕获[&]风险：注意被引用变量的生命周期，如果被引用变量是local，离开作用域之后再调用lambda就会core。
* 值的默认捕获[=]风险：误把类成员变量捕获
* **捕获参数太大加上move**：如下，可以节省一次拷贝。（注意，虽然这里用了move，但基于lambda的模板，这里传入lambda还是左值，如果想用右值看下面）

|  |
| --- |
| JSON [a = std::move(a)] // (c++14) |

* **用auto&&来做多态或者右值：**如下，基于lambda的模板，变量x传入后变成右值。另外如果要做多态，也是用auto&&，然后用forword+delctype来反解多态。

|  |
| --- |
| Swift auto f = [](auto&& x) {  return func(normalize(std::forward<delctype(x)>(x))); }; |

* **用lambda代替std::bind：**无论是可读性还是性能都更优
* 注意事项：
* 在include中使用lambda，调用的不同cpp中是不同的匿名函数，比如cpp1是#1，cpp2是#2，两者不相等。
* 在template中使用lambda，针对不同的类型是不同的匿名函数，比如T=string是#1，T=int是#2，两者不相等

**尽量不要用全局变量**

|  |
| --- |
| C++ const char \*str = "HELLO ByteDance";  char foo() {  return str[2]; } |

|  |
| --- |
| Assembly language \_Z3foov: # @\_Z3foov  movq str(%rip), %rax # 读取str地址  movb 2(%rax), %al # 读取str[2] 内容  retq |

这边的类型是const char \*, 为什么编译器不能直接优化成 'L' 呢？

试想有另外一个cpp文件里这么写

|  |
| --- |
| C++ extern const char \*str; // 变量类型不属于签名的一部分 struct A {  A() { str = "WORLD"; } };  A a; |

foo() 的结果就不再是'L', 所以编译器是不能做这个优化的。

编译器很难对全局做优化！

可以把全局变量打包到一起，变成一个类似Context的对象，全文传递。

|  |
| --- |
| C++ struct Context {  int m; };  int foo(const Context &c) {  return c.m; }  int main() {  Context c = {2};  return foo(c); // 直接优化成2 } |

**如果一定要用全局变量，加上static修饰**

|  |
| --- |
| C++ static const char \*str = "HELLO ByteDance";  char foo() {  return str[2]; } |

|  |
| --- |
| Assembly language \_Z3foov: # @\_Z3foov  movb $76, %al # 76 是'L'的ASCII  retq |

少了两次内存访问

static修饰之后，编译器可以确定str不会被其他编译单元访问，就可以做很多静态分析。

**std::string vs std::string\_view**

|  |
| --- |
| C++ #include <string> static const std::string str = "HELLO ByteDance";  char foo() {  return str[2]; } |

|  |
| --- |
| Assembly language \_Z3foov: # @\_Z3foov  movq \_ZL3strB5cxx11(%rip), %rax # 读取str 地址  movb 2(%rax), %al # 读取str[2] 内容  retq |

但是多了\_GLOBAL\_\_sub\_I\_a.cpp。每个编译单元会生成一个类似的函数初始化全局对象。

|  |
| --- |
| Assembly language \_GLOBAL\_\_sub\_I\_a.cpp: # @\_GLOBAL\_\_sub\_I\_a.cpp  movq $\_ZL3strB5cxx11+16, \_ZL3strB5cxx11(%rip)  movabsq $8737581752006755656, %rax # imm = 0x7942204F4C4C4548  movq %rax, \_ZL3strB5cxx11+16(%rip)  movabsq $7305804384563459193, %rax # imm = 0x65636E6144657479  movq %rax, \_ZL3strB5cxx11+23(%rip)  movq $15, \_ZL3strB5cxx11+8(%rip)  movb $0, \_ZL3strB5cxx11+31(%rip)  movl $\_ZNSt7\_\_cxx1112basic\_stringIcSt11char\_traitsIcESaIcEED2Ev, %edi  movl $\_ZL3strB5cxx11, %esi  movl $\_\_dso\_handle, %edx  jmp **\_\_cxa\_atexit # 调用析构函数** |

由于std::string 本身要支持写的操作，所以它必须own这块内存，所以需要分配一块内存，然后进行一次拷贝。本例子全局对象的内存是由编译器直接分配在data段，如果是局部变量，则是分配到栈上。做的事情类似下面这段代码。

|  |
| --- |
| C++ struct string {  char buf[15];  int len; }; string str;  char foo() {  return str.buf[2]; }  void global\_ctor() {  memcpy(str.buf, "HELLO ByteDance", 15);  str.len = 15; } |

const修饰符在这边没有任何用处，因为C++是可以const\_cast的。所以std::string 本身是很低效的，相关的很多api效率也比较低（需要构建一个新的std::string 对象）。

如果改成：

|  |
| --- |
| C++ #include <string> static const std::string str = "HELLO ByteDance!"; // 多了一个 ！，切换成堆分配 char foo() {  return str[2]; } |

在字符串后面多了一个 '!', 这时会额外生成大量的指令, 这是因为std::string 的sso优化不再起作用。具体可以看STE文档[The implementations of std::string](https://bytedance.feishu.cn/docs/doccnIM51VQqhViKRp42SSwAkac) 简单说就是如果字符串长度超过15个字符，std::string 会开始用堆管理内存。由于需要调用new，所以就有可能会失败，也就意味着，编译器就再也无法优化这个对象的构造过程, 因为这改变了程序语意。绝大多数C++标准库的容器都是堆分配的，所以一旦构建出来，编译器是无法优化掉。

string的字符串长度的阈值不同的实现可以不一样，默认libstdc++ 的长度是15， clang实现的libc++会长一点。

C++17 引入string\_view 来优化std::string 的性能。string\_view 本身不own内存，它只维护了一个指针和长度。

|  |
| --- |
| C++ #include <string\_view> static const std::string\_view str = "HELLO ByteDance!"; // 没有任何开销 char foo() {  return str[2]; } |

等价

|  |
| --- |
| C++ #include <string\_view> char foo() {  return 'L'; } |

总结：

* std::string\_view 本质上就是一个const char\*的指针，所以字符串常量的初始化建议用std::string\_view.
* std::string\_view 可以转成std::string

**不要在头文件中定义容器对象**

xxx.h

|  |
| --- |
| C++ #include <string>  static const std::string MODEL\_X = "MODEL-X"; static const std::string MODEL\_Y = "MODEL-Y"; |

xxx.cpp

|  |
| --- |
| C++ #include "xxx.h"  void foo(const std::string &type) {  if (type == MODEL\_X)  do\_something(); } |

MODEL\_X 和MODEL\_Y 都会被创建出来，不管cpp里面有没有用到。而且任何include "xxx.h" 所编译出来的.o 文件里面都会创建这些对象。也就是链接出来的二进制中有很多份MODEL\_X和MODEL\_Y.

具体见STE文档 [业务例子](https://bytedance.feishu.cn/docs/doccnTMRlBs6ICcrIz7RxYMn68d#6sF6vq)

但是有时候在头文件里定义会让代码更一致。比如希望在头文件定义一张表。

|  |
| --- |
| C++ static const std::unordered\_map<std::string, int> map = { {"HELLO", 4}, {"WORLD", 6}}; |

这是一种很低效的实现方式，因为所有include这个头文件的都会初始化好map这个对象，以及map里所用到的所有的string对象，在链接的时候，这些对象都会被链接进产物并做运行时初始化，但是绝大多数对象是完全没人用的。可以维护一个def文件来描述映射关系

|  |
| --- |
| C++ #ifndef MAP #define MAP(x, y) #endif MAP(HELLO, 4) MAP(WORLD, 6) ... #undef MAP |

在需要用这张表的地方，加上

|  |
| --- |
| C++ const std::unordered\_map<std::string, int> map = { #define MAP(x, y) {#x, y}, #include "xxx.def" }; |

也可以很灵活的选择出来其中的key组成一个容器，而且这个map不再是全局对象，提供了更多的优化可能。

当工程很复杂的时候，甚至可以自己定制一些DSL语法来描述关系，通过工具来生成这个def文件，再嵌入到代码。

**多用std::array 而不是std::vector**

std::vector 是变长数组，而std::array 是定长数组。std::vector 的内存是分配在堆上的，而std::array 是分配在栈上的。和std::string 类似，一旦分配在堆上，编译器就很难做优化。std::array 本身是一个constexpr容器，都是编译期可知的。

|  |
| --- |
| C++ #include <string> #include <vector> #include <array> #include <string\_view>  static const std::vector vec = {"HELLO", "ByteDance"}; std::string\_view foo() {  return vec[0]; } |

vec对于编译器来说是一个变量，所以在构造string\_view 的时候，会调用strlen去计算长度。而且编译器会把整个std::vector 构建出来。如果换成std::array, vec 是一个常量，编译器知道vec[0] 就是“HELLO”。最终出来的汇编如下：

|  |
| --- |
| Assembly language \_Z3foov: # @\_Z3foov  movl $5, %eax  movl $.L.str, %edx # .L.str 是在str段的位置  retq |

std::array vec = {"HELLO"} 这种在构造函数自动推到模版类型的写法是在C++17之后支持的。

由于这种constexpr容器对性能确实有很大的提升，C++20 开始支持更多的constexpr容器。[More constexpr containers](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html)

**尽量避免指针的转换**

|  |
| --- |
| C++ #include <iostream> int foo(short \*s, int \*i) {  \*i = 1;  \*s = 2;  return \*i; }  int main() {  int x = 0;  x = foo((short\*)&x, &x);  std::cout << x << "\n"; // Expect 2? } |

|  |
| --- |
| Shell g++ a.cpp ./a.out 2 g++ a.cpp -O2 ./a.out 1 #!!! What the fuck happens ? |

这段代码的执行结果是很反直觉的，第一反应是编译器优化的锅。其实，在C++中，对cast完的指针做dereference是个UB(Undefined Behavior)行为。代码中把x的地址转成short \*, 最后做dereference，这个时候是个未定义行为, 所以编译器做了优化。但这种代码在现实中太常见了，所以编译器提供了一个选项 **-fno-strict-aliasing** 来禁掉该优化。但如果关掉strict-aliasing, 对程序的性能有巨大的影响。所以不要做指针的cast, 如果做了，理论上开启编译器高优化会出问题。

* strict-aliasing

编译器在做别名分析（aliasing analysis）的时候，需要知道两个指针所指向的内存是否有交叉，这个信息特别关键，因为它直接决定了指令之间的依赖关系，也决定了一块内存是否重新load。

|  |
| --- |
| C++ int a = \*p; \*q = 4+a; return \*p; // 如果我们知道p和q指针所指向的内存不交叉，在这个位置，就不需要重新从内存读取p，而是直接用之前存在寄存器上的a |

|  |
| --- |
| C++ int a = \*p; \*q = 4; // 如果我们知道p和q不交叉，编译器就可以随便调换这两条指令的位置，暴露更多的优化机会。 |

但是这个别名分析太耗时间了，也很容易不精准。所以编译器基于语言标准里面规定的，对cast完的指针做解引用是UB行为，做了个假设，用户的代码不会对cast完的指针做解引用。那意味着s和i不可能指向一块内存，因为指针和类型映射在一起了。所以它会认为这两次的写是不相关的。这就是ANSI aliasing规则.

由编译器前端把所有的指针按照类型进行分组，不同类型的指针之间是不交叉(alias)的, 这个就是TBAA(type based alias analysis), 然后编译器后端会基于这个分组做进一步的分析(数据流分析等），把这个依赖关系弄的更精准。

|  |
| --- |
| C++ int foo(short \*s, int \*i) {  \*i = 1;  \*s = 2;  return \*i; }  x = foo((short\*)&x, &x); // x 和 \*(short\*)&x 是两块不同的内存（shadow） |

总结：

* 不要试图对cast过的指针进行解引用，优化开启的时候，有可能行为会达不到预期。完全取决于编译器实现。
* 如果发现优化开起来程序行为出问题，试试选项 -fno-strict-aliasing
* 关掉strict-aliasing会对性能有巨大影响

**慎用C++异常**

具体可以参考STE 文档 [Exception Handling.pdf](https://bytedance.feishu.cn/file/boxcn69vB1DD9Mwf5qD9vKiQclc) （比较细，有需要可以补一个简单化的版本）

C++异常的主要特点是

* 对于不抛异常的这条路，是零开销的
* 一旦抛异常，运行库会做两次调用栈回溯，这个过程是极其耗费时间的

所以，异常被设计出来做极端情况的容错处理，而不是用来处理代码逻辑的。X86架构的异常实现本身是不会对正常路径的程序性能有太多的影响，但它要求编译器在eh\_frame 里面生成一些数据，所以会影响代码的size，同时，因为它会在插入很多代码来处理异常发生的情况，所以也会影响icache，可以通过 -fno-exceptions 关掉，如果你确定代码中不会有异常。

**auto vs auto &**

|  |
| --- |
| C++ int\* h(); auto x = h(); //x : int\*  int &h(); auto x = h(); // x : int , 不是 int &, 所以会发生值拷贝！ |

只有当你需要一份值拷贝的时候，用 auto，不然如果是指针，用auto \*, 如果是值，用auto &.

|  |
| --- |
| C++ *// 这种情况一般不需要copy* **for** (**const** **auto** &**Val** : Container) observe(Val); **for** (**auto** &**Val** : Container) Val.change(); *// 如果确实需要copy，去掉引用* **for** (**auto** **Val** : Container) { Val.change(); saveSomewhere(Val); } *// 这么写可以让代码更清晰的知道是指针* **for** (**const** **auto** \***Ptr** : Container) observe(\*Ptr); **for** (**auto** \***Ptr** : Container) Ptr->change(); |

尽量减少auto的使用，不然会影响代码可读性。一般只在 清楚的可以从上下文判断出来类型采用auto，比如 auto x = static\_cast<...>(), 或者某些特别场景。

|  |
| --- |
| C++ class A; A foo(); auto x = foo(); A x = foo(); |

后者能让你更加警觉这边是有deep copy。具体可以参考STE文档 [C++ 最佳实践之 auto vs auto &](https://bytedance.feishu.cn/docs/doccn3piRKEHc80fQwWEBNhxQIb)

**短函数尽量在头文件实现**

这样子编译器在不开启LTO的情况下都可以做inline，做性能优化。如果放在cpp文件编译器在不开启LTO的情况下是无法做inline的。inline关键字对于编译器而言基本是透明。编译器有内部cost model来判断是否inline。

**循环条件尽量外提**

**std::move和move构造的使用**