- C++ 面经整理

C++ 面经整理

1. const与define区别

（1）就起作用的阶段而言： #define是在编译的**预处理阶段**起作用，而const是在 **编译、运行**的时候起作用

（2）就起作用的方式而言： #define只是**简单的字符替换，没有类型检查,存在边界的错误；const对应数据类型，进行类型检查；** （3）就存储方式而言：#define只是进行展开，有多少地方使用，就替换多少次，**它定义的宏常量在内存中有若干个备份,占用代码段空间；const定义的只读变量在程序运行过程中只有一份备份，占用数据段空间。**

（4）从代码调试的方便程度而言： **const常量可以进行调试的，define是不能进行调试的，**因为在预编译阶段就已经替换掉了。 （5）从是否可以再定义的角度而言： **const不足的地方，是与生俱来的，const不能重定义，而#define可以通过#undef取消某个符号的定义，再重新定义。**

（6）从某些特殊功能而言： **define可以用来防止头文件重复引用，而const不能;**

（7）从用于类中来看：**const用于类成员变量的定义，只要一定义，不可修改。define不可用于类成员变量的定义，但是可以用于全局变量。**

（8）const采用一个普通的常量名称，define可以采用表达式作为名称;

2. Unordered_map实现原理?

   哈希表（数组）+链表/红黑树。

   底层使用hashtable+buket的实现原理，hashtable可以看作是一个数组 或者vector之类的连续内存存储结构（可以通过下标来快速定位时间复杂度为O(1)）处理hash冲突的方法就是在相同hash值的元素位置下面挂buket（桶），当数据量在8以内使用链表来实现桶，当数据量大于8 则自动转换为红黑树结构 也就是有序map的实现结构。

3. Hash_map与map

   hash_map的用法和map是一样的，提供了 insert，size，count等操作，并且里面的元素也是以pair类型来存贮的。虽然对外部提供的函数和数据类型是一致的，但

是其底层实现是完全不同的，map底层的数据结构是rb_tree(红黑树)而，hash_map却是哈希表来实现的。

4. 当哈希表容量很大甚至超限时，你会怎么处理呢?

   扩容。影响哈希表扩容的因素有两个，本身的容量和负载因子，当前的哈希表大小大于容量乘负载因子的时候，哈希表就需要扩容。

5. c++11新特性，比如你代码里出现的auto以及智能指针这些?

- 函数参数类型不能是 auto 类型;
- 类的成员变量不可以是 auto 类型。类的静态成员变量可以是 auto 类型的，但是需要使用 const 修饰，而且该变量的值在类内初始化

2. AVL树的插入、删除、查询的时间复杂度。

   AVL树是一种平衡的二叉搜索树。空二叉树是AVL树，如果T是AVL树，那么T的左右儿子也是AVL树，并且高度差小于等于1，AVL树的高度为O(logn)

3. Epoll的实现原理。

4. 讲述一下内存的结构。

   c++

   栈区:

   1. 作用：保存局部变量、函数调用参数等
   2. 生命周期：只在函数范围内存在，当函数运行结束时自动销毁

   堆区:

   1. 作用：保存new的对象
   2. 生命周期：遇见delete结束，如果没有主动调用delete则在程序结束后由OS自动回收

   数据区:

   通常又分为全局区（静态区）和 常量区

   全局区（静态区）

   1. 作用：存放全局变量和static静态变量，已经初始化的在.data区域，未初始化的在.bss区域

2. 生命周期：程序结束后由OS释放

常量区

1. 作用：存放不可修改的常量
2. 生命周期：程序结束后由OS释放

代码区：

存放程序执行代码

5. 多线程下，它们使用了内存中哪些区域?

6. 多线程有什么优势呢，可和多进程以及单进程相比。

一个进程可以包含多个线程，线程在进程的内部。(包含关系)

进程之间的资源是独立的，线程之间的资源则是共享的。（资源是否独立共享）

多个进程同时执行时，如果一个进程崩溃，一般不会影响其他进程，而同一进程内的多个线程之间，如果一个线程崩溃，很可能使得整个进程崩溃。一个进程崩溃一般不会影响另一个，但是线程崩溃可能造成整个进程崩溃

进程的上下文切换速度比较慢，而线程的上下文切换速度比较快。（上下文切换速度）

进程的创建/销毁/调度开销大，线程的创建/销毁/调度开销相对少很多。（开销）

7. C++ explicit关键字

1. explicit关键字**只能用于修饰只有一个参数的类构造函数**, 它的作用是表明该构造函数是显示的, 而非隐式的, 跟它相对应的另一个关键字是implicit, 意思是隐藏的,类构造函数默认情况下即声明为implicit(隐式).
2. **如果类构造函数参数大于或等于两个时, 是不会产生隐式转换的, 所以explicit关键字也就无效了.**
3. **除了第一个参数以外的其他参数都有默认值的时候, explicit关键字依然有效**, 此时, 当调用构造函数时只传入一个参数, 等效于只有一个参数的类构造函数

8. c++智能指针

智能指针是**RAII(Resource Acquisition Is Initialization，资源获取即初始化)**机制对普通指针进行的一层封装。这样使得智能指针的行为动作像一个指针，本质上却是一个对象，这样可以方便管理一个对象的生命周期。auto_ptr、unique_ptr、

shared_ptr 和 weak_ptr。其中，auto_ptr 在 C++11已被摒弃，在C++17中已经移除不可用。

unique_ptr**没有复制构造函数，不支持普通的拷贝和赋值操作**。因为unique_ptr独享被管理对象指针所有权。unique_ptr虽然不支持普通的拷贝和赋值操作，但却可以**将所有权进行转移，使用std::move方法**即可。

unique**最常见的使用场景，就是替代原始指针，为动态申请的资源提供异常安全保证**。

shared_ptr**使用引用计数实现对同一块内存的多个引用**。在最后一个引用被释放时，指向的内存才释放。

当对象的所有权需要共享(share)时，share_ptr**可以进行赋值拷贝**。shared_ptr使用引用计数，每一个shared_ptr的拷贝都指向相同的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，删除所指向的堆内存。

** 不能将一个原始指针初始化多个shared_ptr**，因为p1,p2都要进行析构删除，这样会造成原始指针p0被删除两次，自然要报错。

```
void f2() {
    int *p0 = new int(1);
    shared_ptr<int> p1(p0);
    shared_ptr<int> p2(p0);
    cout<<*p1<<endl;
}
```

** shared_ptr最大的坑就是循环引用**，为避免循环引用导致的内存泄露，就需要使用weak_ptr。**weak_ptr 并不拥有其指向的对象**，也就是说，让 **weak_ptr 指向 shared_ptr 所指向对象，对象的引用计数并不会增加**。**因此不能保证所管理的对象一定存在。**用于检查所管理的资源是否已经被释放（**expired()**函数），以及协助 shared_ptr 防止循环引用。

```
shared_ptr 的**引用计数是通过原子操作**实现的，即**多线程环境下可以保证线程安全**。在 std::shared_ptr 中，使用 **std::atomic 类型来实现引用计数的原子操作**。该类型是 **C++11 标准库**中提供的，支持原子类型加、减、读、写等操作
```

** 智能指针指向的对象的线程安全问题，智能指针没有做任何保障**

** 1）同一个shared_ptr被多个线程“读”是安全的；**

** 2）同一个shared_ptr被多个线程“写”是不安全的；**

** 3）共享引用计数的不同的shared_ptr被多个线程”写“ 是安全的；**

> 读的时候，比如把智能指针赋值给另一个智能指针，这个不会修改原始指针的指向，只会修改引用计数，而修改引用计数这个操作是原子的，所以读操作就是线程安全的；写的时候，需要做两个事情，先修改原始指针然后再引用计数，这两个步骤是没有加锁保护的，所以这两个操作加起来就不是原子的。比如`swap`函数：

9. 左值引用，右值引用，万能引用。
10. c++转型: **const_cast**， **dynamic_cast**，**static_cast**，**reinterpret_cast**
11. c++面向对象封装，继承，多态

- 封装：类，属性和方法都在类里
- 继承：支持多继承
- 多态：相同的方法调用，不同的实现方式，一个接口，多种方法
    - 虚函数，方法重写（动态多态）通过基类类型的引用或指针调用虚函数
    - 静态多态：编译期间的多态，编译器根据函数实参的类型推断出要调用那个函数（函数重载，函数模板（泛型编程））