C++八股文

叶茂林 2024年1月12日星期五

**C程序的编译过程**

预处理、编译、汇编、链接。

**C和C++有什么区别？**

C和C++在一般语句上没有什么区别，但是基本上属于两种编程语言，C++有新增的语法和关键字，增加了命名空间，管理内存方面增加了new和delete，在指针的基础上增加了引用，C++的不同版本还增加了一些新的关键字，C++还增加了函数重载

相对于C的结构体，C++增加了用于实现面向对象的类，出现了成员函数，并且提供了访问权限的控制，并在此基础上通过继承和虚函数实现了多态

最后就是C++中增加了模板，提供了强大的STL标准容器库

**面向对象的三大特性：封装、继承、多态**

**C++和Java有什么区别？**

Java没有指针，而C++允许通过指针操纵内存

Java不支持多重继承，但Java有接口，可以通过一个类继承多个接口来达到C++多重继承的效果

Java是完全面向对象的语言，所有的函数和变量都必须是某个类的成员

Java有垃圾回收机制可以自动回收无用内存，C++则要通过智能指针才能实现内存自动回收

**C++的内存分区有哪些？**

堆栈、全局静态存储区、常量区、代码区

栈空间用来存放局部变量和函数参数，由编译器分配和回收

堆空间手动分配和回收

全局静态存储区，分为初始化和未初始化两个相邻区域，用来存储全局变量和静态变量

常量存储区用来存储常量

代码区用来放代码

**堆栈有什么区别？**

内存分区：

栈空间由编译器分配和回收，用来存放局部变量和函数参数，是一块连续的空间，由高地址向低地址扩展

堆空间需要手动分配和回收，不连续空间，链表管理，从低地址向高地址扩展

数据结构：

栈是后进先出的数据结构，可以入栈和出栈，可以用数组或者链表实现

堆有最大堆和最小堆，是一种完全二叉树，用于堆排序和优先队列

**C++多态的实现**

C++多态分成静态多态和动态多态

静态多态（编译时多态，通过重载和模板实现）

动态多态（运行时多态，通过虚函数和继承实现）

**虚函数实现原理**

虚函数是通过虚函数指针和虚函数表实现的，当一个包含虚函数的类被编译的时候，编译器就会为这个类生成一个虚函数表（全局数据区），这个虚函数表是一个指针数组，存储了指向各个虚函数的指针，当类继承的时候会同时继承这个虚函数表，如果重写虚函数那么将会覆盖这个虚函数表中对应的虚函数指针，当这个类生成对象实例的时候，会在对象内存布局的起始位置添加一个虚表指针指向类的虚函数表，在通过基类指针调用虚函数的时候就会根据这个虚函数指针调用对应的虚函数。

**构造函数为什么不能写成虚函数**

1. 从内存空间实现来说，如果构造函数是虚函数，那么就需要通过虚表指针指向的虚函数表来调用，但是此时对象还没有实例化，还没有虚表指针
2. 从使用角度，虚函数的作用是可以通过基类指针来调用，而构造函数只能是在创建对象的时候主动调用的，无法使用虚函数机制来调用

**析构函数为什么要写成虚函数**

这是因为当使用指向派生类对象的基类指针来销毁对象的时候，如果析构函数不是虚函数，那么就会调用基类的析构函数，而不是派生类的，那这就不对了

**New、delete和malloc、free的区别？**

New和delete是C++的运算符，malloc和free是C的标准库里面的函数，malloc和free用来分配和释放内存，new和delete除了完成分配和释放内存还会自动调用对象的构造函数和析构函数。

从使用上来说，malloc需要指定分配的空间大小，以及返回的是void指针，而new不需要指定空间大小，返回的是具体类型指针

**为什么有了 malloc／free 还需要 new／delete？**

Malloc和free是库函数不是运算符，new和delete可以被重载实现自定义内存分配

New可以自动计算所需要的内存空间，并自动调用构造函数，delete会自动调用析构函数

**New的可以free吗**

没有语法错误，但是有逻辑错误，因为free不会调用析构函数，会导致未定义行为

**Malloc可以delete吗**

没有语法错误，但是有严重的逻辑错误，因为delete会尝试调用对象的析构函数，而malloc分配的内存并没有构造对象，会出现未定义行为

**多次delete同一个对象会怎么样？**

Delete对象会调用对象的析构函数进行内存的释放，指向该对象的指针会变成悬空指针，如果不将悬空指针赋值为空指针的话，再次delete会出现未定义的行为。

**野指针和悬空指针**

野指针是未初始化的指针，悬空指针是指指向已释放或超出作用域的对象的指针，使用野指针和悬空指针会导致未定义的行为。

**Delete空指针会怎么样？**

Delete空指针是安全的，delete内部有对指针判空，不会怎么样。

**C++什么时候会出现内存泄漏？**

申请的内存用完没有释放，发生异常跳过释放内存的代码，容器用完没有释放，delete与delete[]用混，还有就是share指针相互引用导致析构的时候引用无法降为0出现的内存泄露。

**说一下C++的四种智能指针？**

C++的四种智能指针包括C++11新增的unique\_ptr、share\_ptr、weak\_ptr和C++11抛弃使用的auto\_ptr。

auto和unique\_ptr一样是独占所有权的，保证同一时刻只有一个智能指针可以指向该对象，auto在转换资源所有权的时候是通过reset和release来重载赋值运算符实现的，先是release将auto本指针赋值为了0，然后reset将资源转换给新的指针，这个时候如果再通过原来的指针进行资源的访问就会出现访问空指针的问题，会触发未定义行为，而unique不允许直接复制，需要move移动赋值。

Share\_ptr是共享式的，允许多个智能指针指向相同的资源，每次构造和拷贝和赋值的时候引用计数会加一，当析构的时候会引用减一，引用为0的时候会释放资源。

然而当share指针相互引用的时候会出现析构的时候引用无法降为0的问题，weak\_ptr就是用来解决这个问题的，weak指针是弱引用，不会增加资源的引用计数。

**指针和引用有什么区别？**

指针本质是一个变量，这个变量里面存储的是某个变量的内存地址，而引用是一个已经存在的变量的别名，具体反映在编译的时候它们两个的符号表的值是不一样的，指针符号对应的值是这个指针变量的地址，而引用符号对应的是引用变量的地址，所以这个指针是可以为空的，因为它本质是一个变量，但是引用不能为空，它的存在必须依赖于已经存在的变量。

**指针传参和引用传参的区别？**

先解释指针和引用的区别，虽然两种传参都可以修改实参，但是指针传参实际还是值传递，将变量的地址传递到参数上，而引用传参直接传递参数本身，并且这个指针传参可以传空指针，但是引用传参不能传空引用，而且在函数内部修改指针参数不会对外部指针造成影响。

**C++的模板确定类型在编译期**

**C++左值引用和右值引用**

左值：可以取地址，有名字的

纯右值：非引用返回的临时变量、运算表达式、常量、lambda表达式

将亡值：被移动的对象、返回右值引用的函数返回值、move返回值

右值引用可以减少拷贝构造和析构

[从4行代码看右值引用 - qicosmos(江南) - 博客园 (cnblogs.com)](https://www.cnblogs.com/qicosmos/p/4283455.html)

**右值引用**

解决临时对象的非必要拷贝（移动语义），解决模板函数中按照参数类型进行转发（完美转发）

**移动构造、移动语义**

当用一个对象给另一个对象赋值的时候匹配的是拷贝构造函数，为了安全和不改变其他对象会使用深复制，但是如果是用临时对象（函数返回值）构造新对象的时候会进行不必要的拷贝，

移动构造函数可以解决不必要的拷贝，移动构造函数通过右值引用来匹配临时值，用浅拷贝，复制指针的值，并将原对象的指针置为空，将资源转移过来

为了优化性能，也可以用move将左值转换为右值引用，然后匹配上移动构造函数

**完美转发**

模板右值引用（T&&）是万能引用可以接受左值和右值，但是右值引用本身是左值，在函数传参过程中会被引用折叠为左值引用，通过C++11新引入的forward函数可以将参数原封不动的传递到下一个函数中，即保留了引用折叠前的参数的左右值属性

**引用折叠**

右值引用叠加还是右值引用

其他引用叠加变成左值引用

**编译中的静态链接和动态链接**

静态链接：编译时所有被调用到的函数和库的代码全部被复制到可执行文件中。

动态链接：编译时可执行文件只包含被调用的库函数的引用，实际的库函数代码存放在共享库中。

**容器快速删除内部对象**

交换要删除的对象和尾部对象，然后直接pop。

**空类的大小**

1字节，因为类实例的内存地址不能相同。

**it++和++it**

it++返回临时对象，++it返回引用

**结构体对齐**

默认按最大成员对齐

C++11后可以用alignof得到对齐方式，alignas指定对齐方式，也可以用预处理命令#pragma pack(push,n)指定对齐方式，用pack(pop)恢复默认对齐方式

**参数初始化列表**

与在函数体初始化相比，参数初始化列表是初始化，函数体内是赋值，可以初始化const和引用成员

**迭代器无法取引用**

函数返回值是右值，普通的左值引用不能接受右值，常量左值引用（const T &）海纳百川

**静态成员变量只能在定义类的时候初始化或者在类外部初始化**