版本号：v1.0 阶段：初稿

# C++

MRL Liu

（未经许可，不得传播）

2022年02月02日

如今，计算机行业已经诞生了无数优秀的现代编程语言，但是现在C++却仍然具有强盛的生命力。显然，C++语言具备独特的优势：**和C兼容，可以和硬件设备密切交互，完成各种稀奇古怪的底层功能**。系统级软件是数字世界的基础设施，C++被誉为系统级编程皇冠上的明珠。现代C++版本一般指的是国际C++标准委员会发布的4个版本：C++11/14/17/20。

# C++程序篇

C++是一种静态类型的、编译式的通用的、大小写敏感的、不规则的编程语言，支持**面向过程编程**、**面向对象编程**和**泛型编程**。（注意：静态类型是指C++在编译时执行类型检查，而不是在运行时执行类型检查）。

C++具备面向对象程序设计的四大特性：**封装**、抽象、**继承**和**多态**。（也有的说面向对象的三大特性，则将**抽象**删除）。

标准的C++由三个重要部分组成：

1. 核心语言。提供了C++的变量、数据类型和常量等。
2. C++标准库。提供了大量的类和函数，用于操作文件、字符串string等。
3. 标准模板库（STL）。提供了常见数据结构的模板和算法等，属于泛型编程。

## 一、C++的文件

C++大型程序由多个源代码文件（.cpp）组成，不同的文件可能共享一些数据，每个文件都可以独立修改后由C++编译器（g++）单独编译，最终链接成可执行程序文件（.exe）。

C/C++中的变量和函数等都必须先声明后使用。而且C++还有著名的单定义规则（One Definition Rule，ODR），即C++的变量或非内联函数只能有一次定义。C++提供了2种声明：定义声明（简称为定义）和引用声明（简称为声明）。定义声明一般要显示地进行初始化，它会在程序运行时给该变量分配存储空间，引用声明则不会在程序运行时分配存储空间，因为它引用已有的变量。

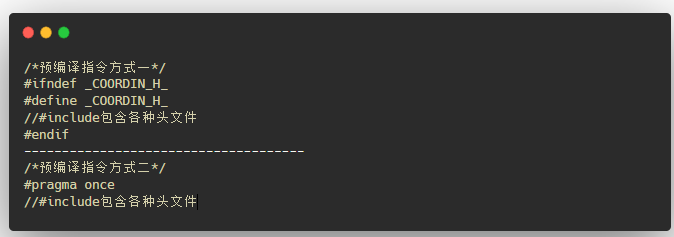
由于不同的源代码文件可能使用同一个声明，为了维护这种声明一致性，C++鼓励程序员将结构声明都放在头文件（.h）中，然后在每个源文件（.cpp）中包含该头文件，这样修改结构声明只需要在头文件修改一次即可。由此，整个C++程序也可以划分为三个部分：

|  |  |
| --- | --- |
| **三类文件** | **文件内容** |
| 头文件（.h） | 函数原型、结构声明、类声明、模板声明、内联函数、使用#define或const定义的符号常量 |
| 主程序文件（.cpp） | 包含main()的文件，程序入口，调用模块文件实现的方法。 |
| 模块文件（.cpp） | 放置一些函数定义，也称为功能模块。 |

### 1、头文件的预编译

头文件中包含的各种声明都不会让编译器分配内存，结构声明、类声明、模板声明、函数原型等不会创建变量，只是告诉编译器如何创建该变量；被声明为const的数据和内联函数有特殊的链接属性。

同一个源文件（.cpp）只能将同一个头文件（.h）包含一次，但是很可能在不知情的情况下将头文件包含多次。为了避免一个头文件在多个.cpp文件中被包含而被多次重复编译，现代C++提供了针对头文件的预编译机制，以尽可能减少反复编译同一个头文件。这种预编译机制如下：



这种预编译指令告知编译器只编译一次包含该头文件的指令，这样头文件就不会被多个源文件重复编译，也可以加快大型C++程序的编译速度。C++中的头文件会被预处理器进行预编译，包含在不同的源文件中。一个源文件（.cpp）和其对应的头文件（.h）会组成一个编译单元。

### 源文件的预处理和主函数

C++程序源文件的后缀是.cpp，一个最简单的.cpp文件如下：



这个最简单的.cpp文件中也包含了三部分：预编译指令、主函数main()。

#### 预处理指令

预处理指令经常用于C++程序文件的开头，用来指定C++预处理器的任务。预处理指令以#开头，常见的预处理指令如#include、#define、#if、#else、#line。这里介绍两种常见的预处理指令。预处理指令不属于C++语句，因为预处理指令都不需要‘;’结尾。

##### #include指令

#include指令有两种使用方法：

#include<>：一般用于包含系统头文件，预处理程序默认在系统默认目录查找文件。

#include””：一般用于包含用户自定义头文件，预处理程序会现在程序源文件中查找，找不到再找系统默认目录。

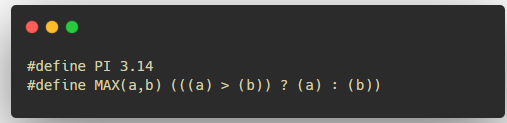
由于历史原因，系统头文件存在几种不同的写法，例如：

#include<math.h>是是C风格旧式写法

#include<iostream>是C++新式风格

##### #define指令

#define经常用于定义一个宏，其为一个标识符定义了宏名，源程序中宏名出现的地方都会用其定义的标识符进行替换，称为宏替换。宏名一般使用大写字母定义。宏经常用于定义一个常量或者简单函数，如下：

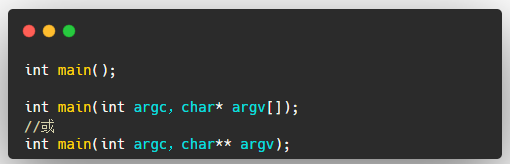


上述示范中用宏定义了一个常量PI，但现代C++中建议使用const进行常量定义，因为宏替换并不会进行类型匹配之类的安全性检查。

上述示范中也用宏定义了一个MAX函数，其好处是没有函数调用的额外开销，运行速度较快，但容易出错。

#### main函数

main函数是C++程序的入口函数，所有C++程序的执行从main函数开始，并由main的返回结束（Windows编程中的DLL除外）。main函数的返回值为一个整数，返回0表示程序正常退出，返回非0表示程序出现异常。main函数的原型如下：

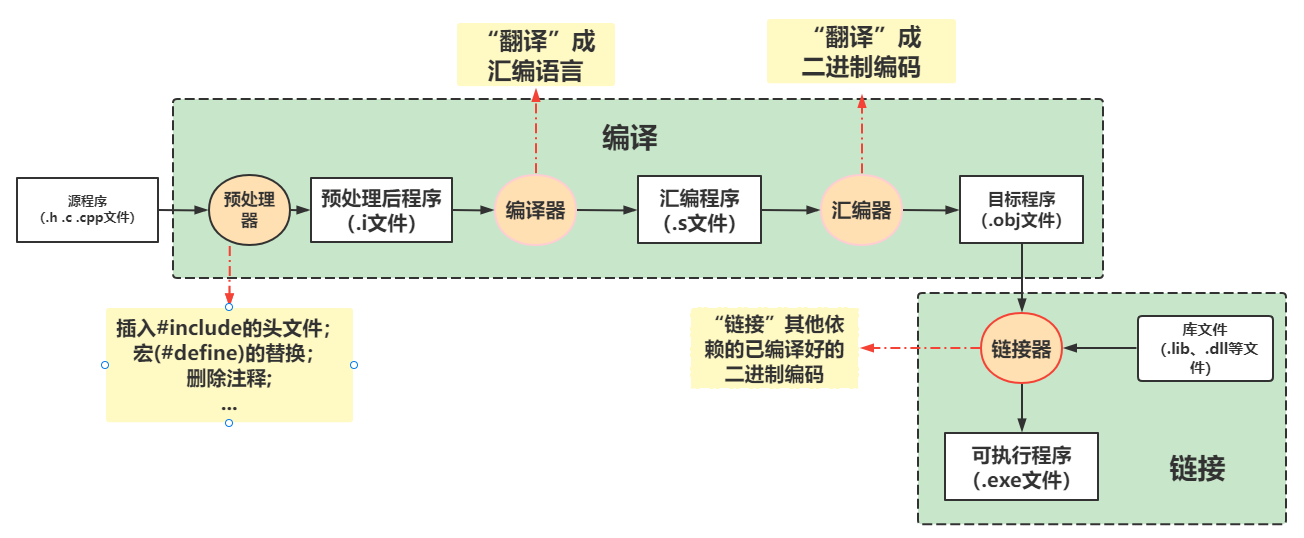


main()函数是C++程序经过特殊处理的函数，其他的返回值类型不是void的函数，如果没有使用return语句，编译器将报错，当main()函数内没有出现return语句时，同样可以通过编译并正常运行，这是因为编译器在main()函数的末尾自动添加了return 0;语句。

main()函数的带参原型可以接收用户向程序输入的参数，其中argc代表参数个数，argv数组中的每一个元素保存命令行参数内容的字符串。

## C++的编译流程

在开发过程中，程序员写的C++源程序文件（.cpp）都要被IDE中集成的C++编译器（g++）编译生成.exe文件，C++编译器就完成了C++的编译流程。C++的编译流程总体可以分为两个步骤：**编译**和**链接**，其过程如下：



### 编译过程

C++的编译过程首先读取各类源程序文件，头文件（.h）会被预处理器包含进源文件中（.cpp），然后进行宏替换，删除注释等就得到了可以用于编译的预处理后程序文件。预处理后的程序文件会被依次翻译为二进制编码语言，从而得到目标程序文件，此时翻译过程便结束了。

### 链接过程

编译过程产生的目标程序文件不能直接执行，因为每个目标程序文件都是由一个源文件（.cpp）单独编译过来的，不同的源文件可能相互引用，所以还需要进行C++的链接过程将各个目标程序文件相互链接最终生成可执行程序文件（.exe）。

链接过程有静态链接和动态链接2种方式。从图中也可以看出链接过程有.lib和.dll两种文件。

静态链接就是将各种目标程序文件和静态链接库（.lib文件）链接起来，实际静态链接库也是一组目标程序文件的集合，所有的目标程序文件会被以复制的形式拷贝到最终的可执行程序文件（.exe）中。

动态链接是指一些目标程序文件不会被复制到可执行程序文件（.exe）中，而是存放在动态链接库（.dll文件）中，作为一种可执行程序组件，其不可以单独执行，其必须依赖于对应的可执行程序（.exe）执行。

## C++的内存模型

C++的内存模型决定了C++程序在执行过程中各种数据所在的内存分区。C++的内存分区总体可以划分为栈区、堆区和固定存储区。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **内存分区** | **内存特点** | **变量类型** | **声明方式** | **生命周期** | **作用域** | **链接性** |
| 栈 | 系统自动维护 | 局部变量  函数参数  局部常量 | 在代码块中 | 自动存储持续性 | 所在代码块 | 无链接性 |
| 寄存器 | 系统自动维护 | 寄存器变量 | 在代码块中，使用关键字register | 自动存储持续性 | 所在代码块 | 无链接性 |
| （静态存储区）  固定存储 | 系统自动维护 | 局部静态变量 | 在代码块中，使用static关键字 | 静态存储持续性 | 所在代码块 | 无链接性 |
| 全局静态变量 | 不在任何函数内，使用static关键字 | 所在文件 | 内部链接性 |
| 全局变量 | 不在任何函数内 | 所在文件 | 外部链接性 |
| （常量存储区）  固定存储 | 全局常量  函数指针 |  |  |  |  |
| 堆 | 程序员手动维护 | 动态变量（new分配的指针变量） | 使用C++运算符new声明 | 动态存储持续性 |  |  |
| 代码区 | 系统自动维护 | 函数体的二进制代码 |  | 静态存储持续性 |  | 外部链接性 |

不同内存分区的数据具有不同的生命周期、作用域和链接性，下面依次讲解。

### 1、数据的生命周期

数据的生命周期指的是数据在内存中的保留时间，即分配内存到回收内存的时间。数据生命周期的不同是因为其定义生命的起始时间可能不一样。

C语言和C++都不允许在一个函数中定义另一个函数，因此函数的生命周期都是静态存储持续期，即从程序开始执行到程序结束。

变量可以在各种代码块中定义，存在不同的生命周期，总体分为如下4种：

|  |  |  |
| --- | --- | --- |
| **生命周期** | **内存分配** | **相关变量** |
| 自动存储持续性 | 函数被执行时分配内存；函数执行完毕，内存释放 | 函数定义内部声明的变量和函数参数 |
| 静态存储持续性 | 程序被执行时分配内存；程序执行完毕，内存释放 | 函数定义外定义的变量和使用关键字static声明的变量 |
| 动态存储持续性 | new运算符被执行时分配内存；delete运算符被执行或程序结束内存释放 | 由new运算符分配的变量 |
| 线程存储持续性 | 生命周期和所属线程一样长（属于并行编程内容） | 使用关键字thread\_local声明的变量 |

### 2、名称的作用域

名称（函数或变量的名字）的作用域描述了名称在文件的多大范围内可见。C++函数的作用域可以是整个类或全局的，不能是局部的，因为函数不能在代码块内定义。

变量的作用域从大到小如下：

命名空间内声明的变量作用域为整个命名空间；源文件内代码块外声明的变量为全局作用域（文件作用域），作用于整个文件；类成员的作用域为整个类；局部变量有局部作用域，作用于所在代码块；函数原型作用域中使用的名称只在包含参数列表的括号中可用。

请注意，因为C++要求函数和变量必须先声明后使用，所以实际作用域都是默认从声明开始到其理论作用域结束。代码块内部的变量的作用域会隐藏代码块外部的相同名称的全局变量的作用域。

### 3、名称的链接性

名称（函数或变量的名字）的链接性描述了名称如何在不同源文件间共享。名称的链接性有三种：

|  |  |  |
| --- | --- | --- |
| **链接性** | **相关含义** | **相关代表** |
| 外部链接性 | 可以在其他源文件访问 | 全局变量（也叫外部变量）、非静态函数 |
| 内部链接性 | 只能在当前文件访问 | 全局静态变量、静态函数 |
| 无链接性 | 只能在当前函数或代码块中访问 | 局部静态变量、局部变量 |

全局变量（外部变量）和函数由于具有外部链接性，可以用来在多文件程序的不同部分之间共享数据。如果要在多个文件中使用全局变量/函数，只需要在一个源文件中包含该变量/函数的定义声明，在其他使用该变量/函数的源文件中使用extern引用声明它即可。

全局静态变量/全局静态函数由于具有内部链接性，可以用来在同一个文件中的多个函数之间共享数据。

局部静态变量由于具有无链接性，其只能用于所在代码块，但由于其静态的生命周期，所以其可以用于再生，即同一函数调用2次，该值第二次调用不会再次初始化而是使用上次计算的值进行计算。

### 4、C++的命名空间

C++11提供了命名空间（namespace），在命名空间中声明的函数和变量都默认具有外部链接性，所以它允许程序的其他部分使用该命名空间中声明的东西，同时，一个命名空间中的名称不会与另一个命名空间的相同名称发生冲突。使用命名空间内部的东西需要添加作用域解析运算符::或者using声明。

命名空间是一个逻辑分组，其用来作为附加信息区分不同库中的相同名称的函数、类和变量等。命名空间的定义需要使用namespace关键字，C++中的关键字都在命名空间std中。

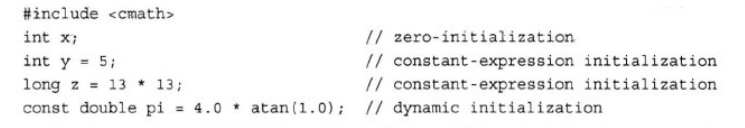
使用其他命名空间的对象时需要在前面加上命名空间的名称，例如使用std::cout。但是为了避免多次添加命名空间前缀的繁琐，C++提供了using namespace指令来告诉编译器，后续代码的关键字将使用指定的命名空间中的名称。

### 5、变量的初始化方式

变量的初始化方式分为静态初始化和动态初始化，其中静态初始化又分为零初始化和常量初始化。静态初始化是值变量在编译时进行初始化，动态初始化是指变量在程序运行时进行初始化。

普通变量必须由程序员显示地初始化，否则会编译报错，即必须进行静态初始化。

静态变量不必须由程序员显示地初始化，所有静态变量在编译时都会自动进行零初始化，如果静态变量还有显示初始化，则看编译是是否可以计算出来，可以则继续进行常量初始化，否则由程序执行时动态初始化。



## C++的说明符和限定符

### 1、常见说明符

关键字auto在C++11之前用于将变量声明为自动变量，在C++11之后用于自动类型推断。

关键字register用于在声明中知名函数用寄存器存储，提高访问速度，在C++11之后只是显示地指出该变量是自动的。

关键字static用于代码块外部声明时表示该变量具有内部链接性，用于代码块内部声明时表示该局部变量的存储持续性为静态的。

关键字extern表明该声明是引用声明，即引用其他源文件中定义的外部变量。

关键字thread\_local用于表明该变量的生命周期与所属线程相同，可以与static或extern结合使用。

### 2、常见限定符

关键字const用于表明变量的内存空间被初始化后程序不能再次修改它，如果const修饰全局变量，则将会将全局变量的链接性转换为内部，表明这是该文件独享的一组常量，但是其他文件可以继续使用extern来覆盖它的内部链接性。



关键字volatile用于表明硬件可以修改该变量的内存空间，即使程序不修改它。例如程序访问同一个变量2次时编译器一般会提前检查其值是否被程序更改过，如果程序没有更改它而自己发生了变化则认为需要优化，而该关键字避免编译器做这样的优化。

关键字mutable用来表明即使结构或类变量被整体声明为const，其某个使用了mutable的变量成员可以不受const限制继续修改。

# C++语法篇

## 一、数据类型

C++中有常量、数组、字符串、指针、引用、类、结构体、联合体、枚举等各种概念，但是它们**本质上都是变量**，是的，你没听错，**编程语言中的常量也是一种特殊的变量（使用const关键字）**。

变量是什么？**变量本质是程序可操作的内存空间的名称，每个变量概念都由type和value组成**，其中type就是其**数据类型**，其决定了数据在内存中的存储方式和运算方式，**常量本质是定义后不可更改value的变量**。所以准确理解了变量概念，就相当于拿到了C++中的各种变量类型的总把手。

变量的value没什么好说的，但是变量的type可是五花八门，不同的type在内存中的大小也不一样。**C++中的变量必须先声明后使用**。

这里先简单说一些基本变量类型、指针和引用。

### 基础变量

基础变量就是变量的type为编程语言内置数据类型的变量，这些类型很常见，但是我们要**记住每种变量所占的内存字节数**。可以使用一个或多个类型修饰符进行修饰：signed、unsigned、short和long。

#### （1）整型变量

布尔型（bool）、字符型（char）和整数型（int）一起合称为整型，他们可以方便地进行转换。

bool变量与整数变量可以进行隐式转换：true具有值1，false具有值0；非零整数转换为true，而0转换为false。

char变量占据一个字节数，因此它可以保存2^8=256种不同的值，其默认和ASCII字符集进行对应，因此int（c）可以将一个char变量c显示转换为ASCII值。char变量默认是无符号（unsigned），unsigned char表示的值是0~255，signed char表示的值是-128~127。wchar\_t用来保存更大的字符集里的字符。

int默认是有符号的（signed），这样默认的int变量就可以表示正数和负数，unsigned int是无符号整数，其只能用来表示整数，一般不常用。int一共有3种大小的用法，short int（简写为short）；int；long int（简写为long）。

#### （2）浮点型变量

浮点数也有3种精度大小的：float（单精度）、double（双精度）和long double（扩展精度）。

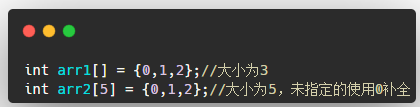
#### 数据类型的大小

C++对象的大小是用char大小的倍数表示的，所以char的大小一般为1，一个C++对象的字节数可以根据运算符sizeof得到。每一种数据类型实际的内存大小与其系统位数有关，下表列出的是目前为主的64位系统，一字节为8位。

|  |  |  |
| --- | --- | --- |
| **关键字** | **数据类型** | **内存大小** |
| void | 无类型 | —（没有void对象，只有void指针） |
| bool | 布尔型 | 1个字节 |
| char | 字符型 | 1个字节 |
| wchar\_t | 宽字符型 | 2或4个字节 |
| int | 整型 | 4个字节 |
| float | 浮点型 | 4个字节 |
| double | 双浮点型 | 8个字节 |
|  | 指针类型 | 8个字节 |

### 数组

C++中的数组必须在定义时就指定大小，C++中的数组初始化方式如下：



### 字符串

这里着重介绍下C++中的字符串，首先介绍下**字符串常量**。在C++中一个**单引号**用于单个**字符常量**（‘a’），一个**双引号**用于单个**字符串常量**（“A”）。

字符常量本质上就是一个固定value大小的字符变量（‘a’的类型是const char[1]），字符串常量本质上是末尾加了一个空字符’\0’的cost字符数组（“A”的类型是const char[2]）。特别地，一个字符串常量可以给一个char指针赋值：

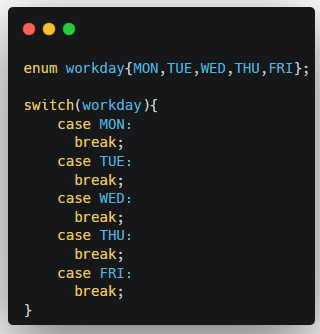


C++中有两种表示字符串的形式：C风格的字符数组和C++引入的string类。string类本质上也是封装了C风格的字符数组，但是其也集成了许多字符串的常见操作，是C++推荐的表示字符串的方式。string实际上就是使用null字符‘\0’终止的字符数组

数组类型

### 3、枚举类型

枚举类型可以保存一组定义的整数常量，其用法一般如下：



数据结构和类

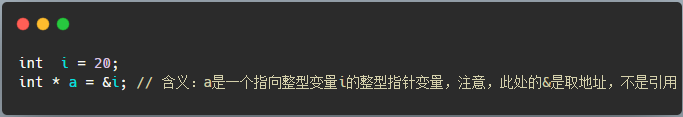
### 指针和引用变量

接下来我们介绍下**指针**和**引用**，这是面试的高频知识，这里我们只强调一些关键信息。

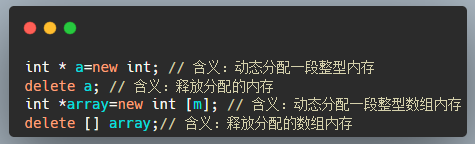
#### （1）指针

**指针本质上是一种特殊的变量，特殊在其value是另一个变量的地址，所以无论何种类型的指针，其内存大小都是固定的**（例如在X86中为4个字节，在X64中为8个字节）；**指针可以在任何时候初始化或者更改；存在空的指针**。

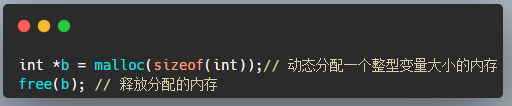
指针的定义如下：



但是实际编程中程序员不会这么无聊地用，基本都是new一个类对象，然后访问其属性和方法。new是一种操作符，和delete结合使用：



当然除了new和delete，也可以使用标准库函数malloc()和free()：



new和malloc()的主要区别：

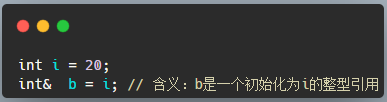
|  |  |  |
| --- | --- | --- |
|  | **new/delete** | **malloc()/free()** |
| **本质** | 运算符 | 标准库函数 |
| **分配大小** | 自动计算 | 作为参数手动输入 |
| **主要操作** | 分配/释放内存时调用对象的构造和析构函数 | 分配/释放内存时不调用对象的构造和析构函数 |
| **安全性** | **内存分配失败**时抛出bad\_alloc异常 | **内存分配失败**时返回null |
| **进行类型识别检测**，返回定义时具体类型指针，如果为int指针分配float变量时报错 | **不进行类型识别检测**，返回void指针，需要手动进行类型转换，如果为int指针分配float大小的字节数不报错。 |

总结，new封装了malloc()方法，new一般用来返回一个类对象，malloc(）一般用在基本变量（int、float），前者更加常用。

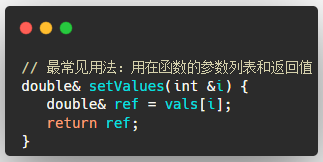
#### （2）引用

**引用**可能初学者不经常使用，但是没关系，**引用本质就是一个已存在的变量的别名，所以引用的内存分配区域和大小由其指向的变量类型决定**。需要注意的是**引用必须在创建时初始化，之后不可更改或再次初始化；不存在空的引用；这两点与指针很不同**。

引用的定义：



引用一般用在函数的参数和返回值上：



## **常量**

前文我们已经知道，常量本质上就是使用了关键字const的变量，使用了const的变量在其作用域内都不能更改其value，const的最常见用途是作为数组的最大界和分情况标号。

## 二、输入输出

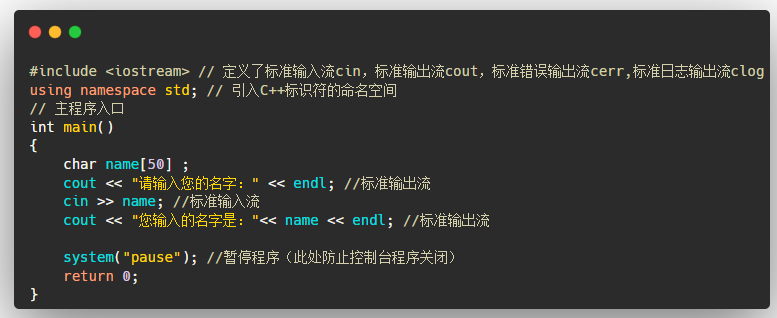
C++标准库提供了丰富的**I/O**（输入/输出）功能。首先看下C++编程中最基本和最常见的**I/O操作**。C++的I/O发生在**流**中，**流**是**字节序列**。

**输入操作**被C++看做**字节流**从**设备**（如键盘、磁盘驱动器、网络连接等）流向**内存**；

**输出操作**被C++看做**字节流**是从**内存**流向**设备**（如显示屏、打印机、磁盘驱动器、网络连接等）。

### 标准输入输出流

下面给出标准输入输出流的演示代码：



首先要用#include语句包含头文件<iostream>；using namespace std语句负责引入C++标识符的命名空间，没有该语句的话，则以下cout、cin和endl都得换成std::cout、std::cin和std::endl。cout可以直接输出字符串或字符数组，只在输出endl时换行；cin以阻塞的方式从控制台读取数据到字符数组中。

# 第3章 C++抽象篇

C++语言是面向对象语言，面向对象程序设计的四大特性：抽象、封装、继承和多态。

### 一、类的成员

C++语言通过类的概念来实现抽象和封装。一个类（class）中包含了成员变量和成员函数，一般C++的结构体（struct）中只有成员变量，不包含大量成员函数。

#### （1）类的访问权限

class通过private、public、protect三个关键字来实现类成员的访问控制，class的成员默认都是private类型，struct的成员默认都是public类型，这也是两者最大的不同。

class的private成员只准类本身内部的成员函数访问；

class的public成员可供类本身、派生类及外部访问；

class的protect成员可以被类本身以及派生类访问，外部不可访问；

#### （2）类的构造函数和析构函数

类的构造函数和析构函数一直被其开发者认为是C++的核心。

C++中类的构造函数具有与类相同的名字，输入参数可以自定义也可以不定义，其由C++程序自动调用，不需要程序员显示调用。一个类通常具有多个构造函数，从而为用户提供多种不同的初始化方式。在所有构造函数中，有一类特殊的复制构造函数，其特别之处在于传入的参数是一个自身类型的参数。包含const成员的类不能自动调用默认构造函数。

析构函数

#### （3）类的静态成员

一个类可以实例化出多个对象，不同的对象占据不同的内存。在类成员的声明中使用关键字static就可以定义类的静态成员，不同对象的所有static成员占据同一份内存。

#### （3）类的常量成员函数

在类的成员函数声明的参数表后面使用关键字const就变成了常量成员函数，常量成员函数不允许其修改类本身的变量。

成员函数

### 二、类的继承

继承是面向对象程序设计中最重要的一个概念，代表了is-a关系。从一个类中派生出另一个类中，原始类称为基类（或父类），继承类称为派生类（或子类）。

子类可以访问父类中所有的非私有成员，并继承所有的父类方法，以下情况除外：

1. 父类的构造函数、析构函数和复制构造函数。
2. 父类的重载运算符。
3. 父类的友元函数。

类的继承有三种方式（在大部分情况下几乎都不使用protected和private）：

1. 公有继承（public）。父类的公有成员成为子类的公有成员，父类的保护成员成为子类的保护成员，子类无法直接访问父类的私有成员，只能通过父类的公有和保护成员函数来间接访问。
2. 保护继承（protected）。父类的公有成员和保护成员都会成为子类的保护成员。
3. 私有继承（private）。父类的公有成员和保护成员都将成为子类的私有成员。

### 三、类的重载

C++允许同一个作用域中的某个函数或运算符具有多个定义，这也叫做函数重载和运算法重载。重载声明是指该作用域内声明的函数或方法具有相同名称，但是参数列表和实现不相同。C++编译器会自行选用最合适的定义，即重载决策。

### 三、类的多态

类的多态指的是调用类的成员函数时会根据调用函数的对象的类型来执行不同的函数。虚函数是在父类中使用关键字virtual声明的函数。假设父类有一个方法area()，子类也有一个相同的方法area()。

假设父类的方法area()没有声明为虚函数，则调用子类的area()时会默认执行父类的area()，这就是所谓的静态多态（或静态链接），函数调用在程序执行前就准备好了。有时候这也被称为早绑定，因为area()函数在程序编译期间就已经设置好了。

假设父类的方法area()声明为虚函数，则调用子类的area()时会默认执行子类的area()，这就是所谓的动态链接，或后期绑定。虚函数的声明会告诉编译器不要静态链接到该函数。

虚函数必须进行实现，且子类不一定需要覆盖该函数；纯虚函数（也叫抽象函数）则不需要实现，且子类一定需要覆盖该函数。

在上述举例说明中，子类通过将父类方法声明为虚函数就可以在子类覆盖其实现，总结起来，子类方法想要覆盖父类方法需要满足如下要求：

一虚：父类中被覆盖的成员函数必须声明为虚函数。

二容：父类和子类的成员函数的返回类型和异常规格必须兼容

四同：父类和子类的成员函数的名程、形参类型、常量属性和引用限定符必须完全相同。

override是C++11中添加的安全声明的关键字，子类要覆盖的方法可以添加override关键字来声明，编译器将会对该函数和父类的对应函数进行一虚二容四同的比较，错误的地方会报错。注意，不添加override关键字也是可以的，但是出现问题时编译器可能不会报错。

在子类中调用父类的方法：父类名::方法名

# 第4章 STL与泛型编程

C++标准库（Starndard Library）可以看做是STL+其他API函数，其中STL是系统性的一个开源库。C++标准库中提供的容器和函数都以头文件的形式提供，例如#include<vector>

同时头文件内的组件封装于命名空间“std”。

STL（Standard Templates Library）是C++的标准模板库，泛型编程就是使用template（模板）为主要工具来编写程序，在C++中泛型编程就是使用STL进行编程。泛型编程与面向对象编程在设计思想上不一样。面向对象将数据和对其操作封装为一个对象，泛化编程将数据和操作分离。

## STL的六大部件

### 1、六大部件的关系

STL中主要有六大部件：

容器（Containers）是存储数据的数据结构，容器封装了内存分配的机制。

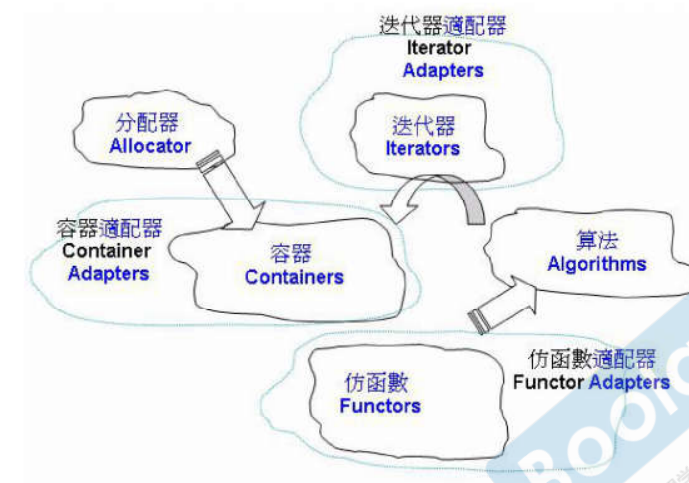
分配器（Allocators）为容器的内存分配机制由提供支持。

算法（Algorithms）封装了对于容器内数据的常见操作。

迭代器（Iterators）是一种泛化的指针，可以看做是泛型编程中的指针。算法通过迭代器来操作容器内的数据。

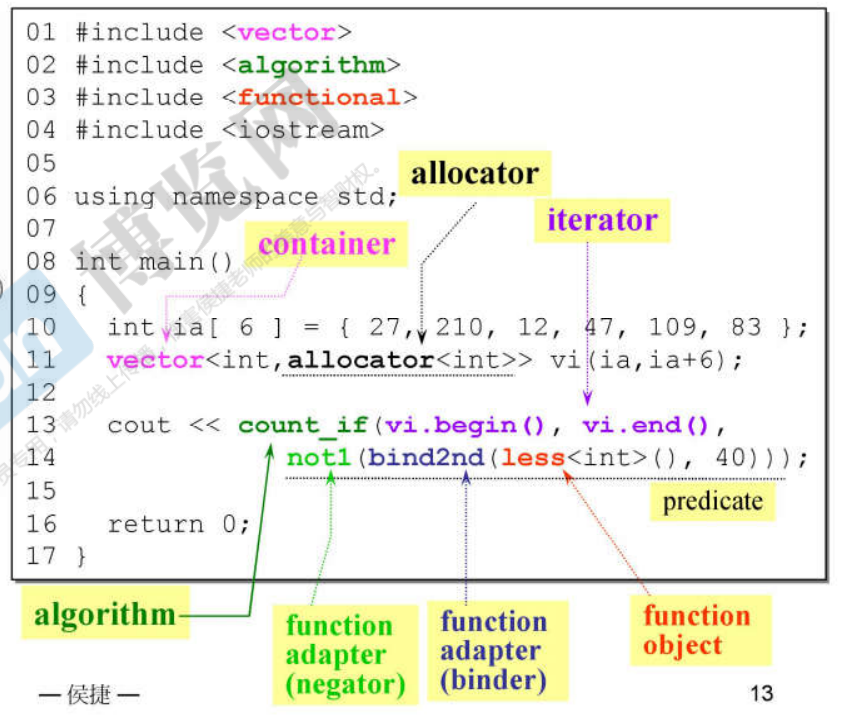
仿函数（Functors）的作用类似于函数。

适配器（Adapters）可以对容器、迭代器、仿函数进行转换。



### 2、六大部件的使用案例

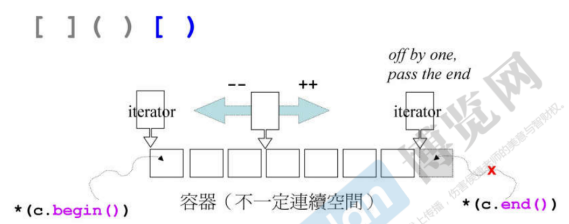
如下代码使用了六大部件，其中常见的是容器、算法和迭代器。



## 容器的遍历

### 1、前闭后开区间的通用遍历

**STL中的任意一种容器都包含两个迭代器c.begin()和c.end()**。容器内的c.begin()和c.end()指定的区间是一个前闭后开的区间，即c.begin()指向容器内的第一个元素，c.end()指向容器内的最后一个元素的后一个空位：



根据这两个迭代器，遍历STL中任何一个容器都可以有以下写法：

|  |
| --- |
| Container<T> c;  for(auto iter=c.begin();iter!=c.end();++iter) ; |

其中auto关键字是让编译器自动推断iter的类型,iter的类型就是其容器对应的迭代器类型Container<T>::iterator。

容器内的c.begin()和c.end()指定的区间是一个前闭后开的区间，STL中的所有容器都包含这两个迭代器，所以遍历任何一个容器都可以有以下写法：

### 2、C++11新加的遍历写法

在现代C++11中添加了遍历容器的一种简便写法：

|  |
| --- |
| for(auto num:arr) cout<<num; |

请注意，这种遍历方式只是以只读的形式读取容器中的每个元素，如果想要遍历修改容器内的每个元素，可以采取**引用**的写法。

|  |
| --- |
| vector<int> arr={1,2,3,4,5,6};  方式一：for(auto& num:arr) {num=num\*3;} // 3,6,9,12,15,18  方式二：for(auto num:arr) {num=num\*3;} // 1,2,3,4,5,6 |

## 容器的结构与分类

STL中的容器可以分为**序列式容器**（Sequence Containers）和**关联式容器**（Associative Containers）。

**关联式容器**通过key来进行快速、大量地查找，set/multiset/map/mulitmap的底层基本都使用红黑树，multi意味着允许key重复。

其中C++11在**关联式容器**中添加了一类新的**不定序式容器**（Associative Containers），其底层使用哈希表的结构（HashTable），查找效率可以接近O(1)。

这些容器的头文件导入如下：



# 附录 C++常见面试题

## 一、C++的语言特性

1、C语言和C++有什么区别？

|  |
| --- |
| 两者最主要的区别是编程语言的设计思想。C语言是一种**面向过程的编程语言**，其使用大量的函数来处理数据，数据和函数一般是分离的。C++语言是一种**面向对象的编程语言**，其将数据和操作数据的函数封装为类，通过抽象出一个个的对象模型来解决问题。C语言目前适合于代码体积小、效率高的场景，例如嵌入式；C++多用于大型的底层系统场景。  其次是语法的不同，例如C语言中的struct不可以有函数，而C++中的struct可以有函数，**C++包含C语言，而且还支持泛型编程**。 |

C++和Java有什么区别？

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **编译流程** | **语法规则** | **效率** | **跨平台** | **学习难度** |
| C++ | **编译性语言**，其源代码经过C++编译器编译和链接后生成可执行代码。 | C++支持**面向对象**、**面向过程**和**泛型编程**，可以定义**全局变量**和**全局函数**。  C++**支持指针**，允许程序员动态分配内存 | 最高 | 不可以 | 最高 |
| **静态语言**，变量定义必须有类型声明 | **支持多继承**、**函数重载**、**运算符重载**等 |
| Java | **解释性语言**，其源代码经过Java编译器编译后需要由Java虚拟机（JVM）解释执行 | Java只支持**面向对象**，所有代码必须在类中实现，不存在全局变量或者全局函数。  Java**没有指针概念**，使用垃圾回收器实现内存的自动回收，对程序员来讲，Java内存管理更加安全 | 较高 | 可以 | 较高 |
| **静态语言**，变量定义必须有类型声明 | Java**支持函数重载**，**不支持多继承**、**运算符重载**等 |
| Python | **解释性的脚本语言**，其源代码直接由python解释器生成并输出 | Python支持**面向对象**和**面向过程**，可以定义全局变量和全局函数。  Python**没有指针概念**，使用垃圾回收器实现内存的自动回收，对程序员来讲，Python内存管理更加安全 | 较低 | 可以 | 较低 |
| **动态语言**，变量定义无类型声明 | Python**支持多继承**，不支持**函数重载**和**运算符重载**等 |

面向对象的三大特性/四大特性是什么？C++如何实现这些特性？/谈谈对面向对象的理解

|  |
| --- |
| 面向对象的四大特性分别是抽象、封装、继承和多态。  C++通过类来实现**抽象**和**封装**。一个C++类将数据和操作数据的方法封装在一起，数据变成了成员变量，方法变成了成员函数。类可以通过控制变量和方法的访问权限（public、protected、private）来达到对外隐藏变量和方法的目的，更加安全。  C++提供类的继承机制，而且允许单继承和多继承，类的继承是让某种类型的对象快速获得另一个类型对象的属性和方法。可以实现代码的重用，节省开发时间。  C++提供类的多态机制，当C++类存在继承关系时，就可能存在多态。类的多态指的是调用类的成员函数时会根据调用函数的对象的类型来执行不同的函数。 |

## 二、C++的编译机制

请简述一下C++为什么要使用头文件？/头文件的作用是什么？

|  |
| --- |
| （1）**头文件用来保存程序的声明**。C++是一种静态编译的编程语言，C++中的变量必须先声明后定义，C++需要将一些重复性使用的声明保存在头文件中。  （2）**头文件可以提升程序安全性**。C++中有些核心代码是不能对用户提供的，但是其代码又必须被用户引用。C++可以向用户提供头文件和编译后的二进制的函数库（例如.lib和.dll），用户只需要按照头文件中的接口声明就可以调用函数库功能。  （3）**头文件具备说明文档的作用**。C++中代码的使用必须和头文件中的声明一致，这就大大减轻了C++程序员调试程序的负担。 |

请简述以下C++的（文件）编译机制？

|  |
| --- |
| C++是一门编译性的编程语言，其源文件（.cpp）到编译为（.exe）基本需要以下4个阶段：  第一阶段：**预处理阶段**。主要是一些代码文本的替换工作，处理#开头的预处理指令，比如拷贝#include包含的文件代码，#define宏定义的替换，条件编译等。  第二阶段：**编译阶段**。将上一阶段的文件“翻译”成汇编语言。  第三阶段：**汇编阶段**。把汇编语言“翻译”成二进制语言。  第四阶段：**链接阶段**。一个.cpp文件编译汇编为一个.obj文件，但是最终的.exe执行文件只有一个，链接阶段将多个.obj文件和静态链接库（.lib）、动态链接库（.dll）组装成最终的执行文件。 |

请简述一下C++的预编译机制？

|  |
| --- |
| C++的预编译机制又称为C++的预处理，它是C++在正式编译源程序之前的预处理步骤，其主要是做些代码文本的替换工作。**删除注释**，处理#开头的指令,比如拷贝**#include包含的文件代码**，**#define宏定义的替换**，**条件编译**等。请注意，#开头的预处理指令不属于C++语句，因为其没有以;结尾。 |

请简述一下#include<>和#include””的区别。

|  |
| --- |
| #include<>一般用于包含C++自身的标准头文件，其从C++的默认头文件路径导入文件。  #include””一般用于包含程序员自定义或者第三方的头文件，其现在自定义目录寻找，找不到才会寻找C++的默认头文件路径。 |

请简述一下C++的条件编译？简述下#define #endif 和#ifndef的作用

|  |
| --- |
| C++的条件编译是C++预处理阶段的步骤之一，是根据定义宏来进行代码静态编译的手段，最常见的是在C++的头文件中引入和#ifndef、#define和#endif来防止文件编译中重复包含头文件的内容。 |

请简述一下静态链接和动态链接的区别。

|  |
| --- |
| 静态链接在C++的文件链接阶段就将各种目标程序文件（.obj文件）和第三方提供的静态链接库（.lib文件，本质也是一组.obj文件）链接起来，.lib文件会被以复制的形式直接拷贝到最终的可执行程序文件（.exe）中，所以会使.exe文件的体积大幅提升。  动态链接是在C++的.exe文件执行过程中发生的链接，.exe文件执行时，用到动态链接库（.dll文件）的代码就会去调用.dll文件中的代码，此时，.dll文件作为.exe文件的一种可执行程序组件，其不可以单独执行，其必须依赖于对应的可执行程序（.exe）执行。动态链接不会增加.exe的体积。 |

## 三、C++的内存模型

简述C++的内存模型/说下你对C++内存分配的了解。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C++程序在运行时会向操作系统申请一块内存空间，C++程序会将这块内存空间划分为如下5个区域，这种C++程序的内存分配机制就是C++的内存模型：   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **内存分区** | **内存特点** | **变量类型** | **声明方式** | **生命周期** | **作用域** | **链接性** | | **栈** | 系统自动维护 | **局部变量**  **函数参数**  **局部常量** | 在代码块中 | 自动存储持续性 | 所在代码块 | **无链接性** | | 寄存器 | 系统自动维护 | 寄存器变量 | 在代码块中，使用关键字register | 自动存储持续性 | 所在代码块 | 无链接性 | | （**静态存储区**）  固定存储 | 系统自动维护 | **局部静态变量** | 在代码块中，使用static关键字 | 静态存储持续性 | 所在代码块 | **无链接性** | | **全局静态变量** | 不在任何函数内，使用static关键字 | 所在文件 | **内部链接性** | | **全局变量** | 不在任何函数内 | 所在文件 | **外部链接性** | | （**常量存储区**）  固定存储 | **全局常量**  **函数指针** |  |  |  |  | | **堆** | 程序员手动维护 | **动态变量**（new分配的指针变量等） | 使用C++运算符new声明 | 动态存储持续性 |  |  | | **代码区** | 系统自动维护 | 函数体的二进制代码 |  | 静态存储持续性 |  | **外部链接性** | |

C++中变量的作用域、生命周期、链接性、可见性是什么意思？

|  |
| --- |
| 变量的作用域是描述了变量在文件的多大范围内可见，根据变量作用域的不同，可以将变量定义为全局变量、局部变量和形式参数。函数的作用域默认是全局的（因为C++函数中不能定义函数），可见性指的是变量只能在变量的作用域内可见和被引用。  变量的生命周期指的是变量在内存中的保留时间，即分配内存到回收内存的时间。  变量的的链接性指的是变量如何在不同源文件间共享，外部链接性指的是变量可以在其他文件中使用，内部链接性指的是变量只能在本文件中使用，无链接性指的是变量只能在当前代码块内使用。函数的作用域默认是外部链接性。 |

C++的全局变量可不可以定义在头文件中？如何引用其他文件的全局变量？

|  |
| --- |
| C++的全局变量不可以在头文件中定义。头文件基本只能存放变量、函数和类的声明，内联函数的定义。全局变量如果在当前文件，直接引用即可；如果不在当前文件，有两种方式：  （1）直接引用头文件。  （2）用extern关键字重新声明一下。 |

C++中堆和栈的区别是什么？堆和栈中数据的生命周期？

|  |
| --- |
| C++中的堆和栈是C++中数据存储的内存区域，它们有如下区别：  （1）内存分配不同。栈中的数据的内存由系统自动分配和自动回收，主要存放局部常量、局部变量和函数参数，其生命周期从声明处开始，到所在代码块结束；堆中的数据的内存由程序员手动分配和手动回收，一般是new操作符或者malloc等内存分配函数分配的变量指针，其生命周期从声明处开始，到程序员手动释放，如果程序员没有释放则到程序结束。  （2）数据结构不同。栈是一种先进后出的数据结构。堆是可以被看成是一棵树，如：堆排序。  （3）缓存方式不同。栈使用的是一级缓存，存储的数据用完毕立即释放，速度很快；堆是存放在二级缓存中，存储的数据程序员手动释放或者系统的内存回收算法来决定（智能指针），速度相对较低。 |

C++的静态变量的初始化时机分别是什么时候？

|  |
| --- |
| 文件的全局静态变量在程序运行之初就初始化。  函数的局部静态变量在首次执行此函数时进行初始化。  类的静态成员变量在类实例化之前就已初始化，即程序运行之初。 |

什么是内存泄露？一般什么情况下会出现内存泄漏？怎么用编程中尽量避免它？

|  |
| --- |
| 内存泄露是指程序员在用new运算符或者malloc等内存分配函数时没有如期回收内存，结果导致一直占据该内存单元，这种内存管理失误就是内存泄露。内存泄露通常发生的情况有：   1. 类的**构造函数**为成员指针分配了内存，但是**析构函数**没有释放成员指针。 2. **释放和分配的运算符不匹配**，例如用new分配指针数组，释放时没有使用delete[]。 3. 类含有成员指针，但是**没有自定义拷贝构造函数或者重载赋值运算符（=）**，复制后的对象指向同一块内存。 4. **没有将析构函数定义为虚函数**，如果父类析构函数不是虚函数，那么子类的析构函数将会调用默认的空的析构函数。   对于含有成员指针的类的定义和继承要格外小心，减少指针的多次赋值操作。 |

new和malloc的区别是什么？

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  | **new/delete** | **malloc()/free()** | | **本质** | C++的运算符 | C的标准库函数 | | **分配大小** | 自动计算 | 作为参数手动输入 | | **主要操作** | 分配/释放内存时自动调用对象的构造和析构函数，所以new用来动态分配对象指针 | 分配/释放内存时不自动调用对象的构造和析构函数，所以malloc无法动态分配对象指针 | | **安全性** | **内存分配失败**时抛出bad\_alloc异常 | **内存分配失败**时返回null | | **进行类型识别检测**，返回定义时具体类型指针，如果为int指针分配float变量时报错 | **不进行类型识别检测**，返回void指针，需要手动进行类型转换，如果为int指针分配float大小的字节数不报错。 | |

三个内存操作函数memset、memcpy和strcpy的区别

|  |
| --- |
| memset用来对一段内存空间全部设置为某个字符，例如:char a[100];memset(a, '', sizeof(a))。  memcpy是内存拷贝函数，可以拷贝任何数据类型的对象，例如memcpy(b,a,sizeof(b))。  strcpy只能拷贝字符串，遇到’\0′结束拷贝。 |

C++的不同变量占据多少字节？16位机、32位机、64位机是什么含义？

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16位、32位、64位指的是计算机的CPU一次可以处理数据的字节数。C++可以根据运算符**sizeof()**得到变量类型的字节数。每一种数据类型实际的内存大小与其系统位数有关。一般同一个变量在32位机上是64位机字节数的一半，例如指针类型在16位机--2字节、32位机--4字节、64位机--8字节。  下表列出的是目前为主的64位系统，一字节为8位。   |  |  |  | | --- | --- | --- | | **关键字** | **数据类型** | **内存大小** | | void | 无类型 | —（没有void对象，只有void指针） | | bool | 布尔型 | 1个字节 | | char | 字符型 | 1个字节 | | wchar\_t | 宽字符型 | 2或4个字节 | | int | 整数型 | 4个字节 | | float | 浮点型 | 4个字节 | | double | 双浮点型 | 8个字节 | |  | 指针 | 8个字节 | | string | 字符串 | 字符数+1(最后一个是空字符) | |  | 对象 | 较复杂 | | struct | 结构体 | 较复杂（和对象一致） | | union | 联合体 | 联合体的各个成员变量的内存最大值 | | enum | 枚举 | 枚举中单个成员的内存（一般为int，4个字节） | |

## 三、C++的常见术语辨析

命名空间（namespace）的作用是什么？

|  |
| --- |
| 命名空间是C++为了避免不同的程序员在引用C++代码时出现名称相同的问题而设计的机制。C++标准库的所有标识符都被定义于一个名为std的namespace中。命名空间有如下三种用法：  1、直接使用命名空间指定标识符：std::ostream  2、使用using关键字引入命名空间的单个标识符：using std::cout;  3、使用using关键字引入命名空间的全部标识符：using namespace std; |

变量/函数的声明和定义的区别是什么？

|  |
| --- |
| 简单说，变量不进行赋值就是声明，一旦赋值就是定义；函数不定义函数体（带{}）就是声明，一旦定义函数体就是定义。C++中的变量可以有多个声明，但是只能有一次定义。变量的声明代表了该标识符可见性的开始，但是不会分配内存空间，变量的定义会分配内存空间，表示了变量生命周期的开始。 |

指针和引用有什么区别？什么情况下用指针，什么情况下用引用？

|  |
| --- |
| 指针本质上是一种特殊的变量，特殊在其value是另一个变量的地址，所以无论何种类型的指针，其内存大小都是固定的（例如在X86中为4个字节，在X64中为8个字节）；指针可以在任何时候初始化或者更改；存在空的指针。  引用本质就是一个已存在的变量的别名，所以引用的内存分配区域和大小由其指向的变量类型决定。需要注意的是引用必须在创建时初始化，之后不可更改或再次初始化；不存在空的引用；这两点与指针很不同。引用一般用在函数的参数和返回值上 |

struct和class的区别？

|  |
| --- |
| （1）C语言和C++中的struct。struct是从C语言就有的自定义类型，C语言默认struct中只能定义变量，不能定义函数。C++中struct中可以定义函数，和class的用法基本一致。  （2）struct的成员默认是public的，默认按照public继承；class的成员默认是private的，默认按照private继承。 |

const关键字的作用是什么？什么时候使用const关键字？

|  |
| --- |
| const是C++的一个限定符，用来限定某个变量不被改变，目的是提升程序的安全性是可靠性。const可以修饰**变量**（包括引用、指针、数组、对象、全局变量、局部变量、类成员变量），**类成员函数**、**函数参数**和**函数返回值**，被const修饰的对象都是只读的，不可以被修改。使用场景如下：   1. 定义常量。const修饰的变量本质上就是一个常量，其值无法再改变。 2. 修饰函数参数、函数返回值，防止被函数无意间改动，减少bug。 3. const修饰类成员函数在本函数内不会修改类的数据成员，并且不能调用其他非const成员函数。如果const构成函数重载，const对象只能调用const函数，非const对象优先调用非const函数。   总结：const不是程序编译必须，但是可以提升程序安全性和效率，const可以修饰几乎所有的变量类型，但是只能修饰类成员函数。 |

constexpr关键字的作用是什么？[什么时候需要使用constexpr关键字](http://mp.weixin.qq.com/s?__biz=MzkyODE5NjU2Mw==&mid=2247484767&idx=1&sn=0b45be91a1968c2d564395b6797b81c1&chksm=c21d37e3f56abef5e3e0b9fd8b720bad451e5562aaf0b7d2a1501696f1db7dafe350150aab29&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/s/_blank)？

|  |
| --- |
| constexpr是C++的一个限定符，可以看做是const的进阶版本，constexpr可以修饰const修饰的变量和函数。constexpr可以判断变量的值是否是一个常量表达式（复杂系统中可能很难判断一个初始值是否是常量表达式），在编译过程中如果不是则会报错。constexpr和const的不同如下：  constexpr表达的语义是“常量”，const表达的语义是“只读”；constexpr只能定义编译期常量，const可以定义编译期常量和运行期常量。如果一个变量被constexpr修饰，那么它一定具备被const修饰的特点，反之不然。 |

static关键字的作用是什么？什么时候使用static关键字？

|  |
| --- |
| const是C++的一个限定符，用来控制某个变量的存储方式和可见性。const可以修饰**变量**（全局变量、局部变量、类成员变量等），**类成员函数**、**普通函数**。所有的静态变量都存储在C++的全局数据区，在声明处被初始化，如果没指明初始值就自动初始化为0，一旦初始化就直至程序结束才释放内存。使用场景如下：  （1）**静态全局变量**只能在本文件中访问，不能在其它文件中访问，即便是extern外部声明也不可以（**外部可见性缩小**）。  （2）**静态局部变量**在首次执行时初始化，直至程序运行结束后才释放（**生命周期延长**）。  （3）**类的静态成员变量**必须在类外声明，且**类的多个对象共享同一个静态成员变量**。  （4）**类的静态成员函数**只能调用静态成员变量（函数内没有this指针）。 |

extern关键字的作用是什么？什么时候使用extern关键字？

|  |
| --- |
| extern是C++的一个限定符，一般用来修饰全局变量的声明，也可以修饰全局函数。全局变量一般的可见性一般只在本文件中，如果其他文件也想使用该全局变量就要使用extern进行修饰，注意extern只能修饰声明，不能修饰定义。 |

explicit/implicit关键字的作用是什么？[什么时候需要使用explicit/implicit关键字](http://mp.weixin.qq.com/s?__biz=MzkyODE5NjU2Mw==&mid=2247484767&idx=1&sn=0b45be91a1968c2d564395b6797b81c1&chksm=c21d37e3f56abef5e3e0b9fd8b720bad451e5562aaf0b7d2a1501696f1db7dafe350150aab29&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/s/_blank)？

|  |
| --- |
| explicit是C++的一个限定符，只能用于修饰类构造函数，它表明该构造函数只可以显示调用,不能由编译器隐式调用,跟它相对应的另一个关键字implicit,意思是表明该构造函数可以隐式调用，该操作常用来避免一些歧义。 |

volatile关键字的作用是什么？[什么时候需要使用volatile关键字](http://mp.weixin.qq.com/s?__biz=MzkyODE5NjU2Mw==&mid=2247484767&idx=1&sn=0b45be91a1968c2d564395b6797b81c1&chksm=c21d37e3f56abef5e3e0b9fd8b720bad451e5562aaf0b7d2a1501696f1db7dafe350150aab29&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/s/_blank)？

|  |
| --- |
| volatile是C++的一个限定符，用来声明某个变量为容易变动的，编译器不要进行优化。C++编译器为了提升执行效率，经常会将一些变量的存储到寄存器中，这样很可能在关键时刻无法更改变量的值，volatile则不允许编译器自行将变量存储到寄存器中。Volatile一般用在：   1. 多任务程序中需要多个任务共享的一些标志变量。 2. 实时嵌入式程序中的中断处理程序中修改的供其他程序检测的变量。 |

enum、const和#define定义的常量有什么不同？

|  |
| --- |
| enum定义的是变量，该变量的值只能是预先定义的枚举常量，**需要分配内存空间**。  const是修饰已有的变量，其值初始化后不可以再更改，**需要分配内存空间**。  #define定义的是宏，**不分配内存空间**，**不进行类型检查**，在C++的预编译阶段进行简单的本文替换。#define虽然也可以定义宏函数，但它不进行类型检查，传参时很容易出错，使用时很不安全，C++不推荐频繁使用。 |

inline关键字的作用是什么？[什么时候需要使用inline关键字](http://mp.weixin.qq.com/s?__biz=MzkyODE5NjU2Mw==&mid=2247484767&idx=1&sn=0b45be91a1968c2d564395b6797b81c1&chksm=c21d37e3f56abef5e3e0b9fd8b720bad451e5562aaf0b7d2a1501696f1db7dafe350150aab29&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/s/_blank)？

|  |
| --- |
| inline是C++的一个限定符，用来显示声明一个函数为内联函数，类中定义的函数都默认为内联函数。  **内联函数可以减少函数调用的开销，提高程序执行效率，是一种优化效率的手段**。编译器处理内联函数时，不会单独进行函数调用，而是直接将整个函数体的代码插入调用语句处，就像整个函数体在调用处被重写了一遍一样，**这个过程发生在编译期间**。很显然，使用内联函数会使最终可执行程序的体积增加，因为内联函数是以空间换取时间。  **内联函数适用于小而简单、执行很快的函数**，如果一个函数非常庞大或者需要消耗大量时间，那么将其声明为内联函数虽然节省了函数调用的时间，但是却让程序体积增加了更多，这样程序执行速度很可能反而会下降。现代C++编译器提供了内联函数的保护机制，**一般程序员声明的内联函数只是给编译器的建议**，具体编译器是否真的按照内联函数的方式处理可能由内部算法决定。 |

typedef的用法

|  |
| --- |
| typedef是C++中用来给复杂的数据类型或者声明**定义别名**，可以提升程序员的开发效率、程序的可读性和兼容性 |

i++和++i的区别

|  |
| --- |
| i++是先使用i的值然后再让i自增1，该表达式需要先返回一个旧值所以编译器会自动创建一个临时对象返回。  ++i直接执行i的自增，不会创建临时对象，所以++i比i++的执行效率更高。 |

函数的覆盖（重写）和重载有什么区别？

|  |
| --- |
| 函数覆盖发生在类的继承中，父类用virtual关键字修饰的函数可以被子类覆盖，子类要覆盖的函数用override关键字修饰，子类的函数覆盖时参数列表、返回值、抛出的异常要和父类的函数一致。  函数重载和类的多态紧密相关，C++允许定义同一个函数名，但是参数列表、返回值、修饰符等可以不同。 |

do……while和while……do有什么区别？

|  |
| --- |
| do...while先执行循环再判断条件，while...do先判断条件再执行循环。 |

简述下C++中的强制类型转换机制（不常用，但是得知道）。

|  |
| --- |
| C语言中，代码可以使用小括号来进行强制类型转换，C++新增了4个关键字来强制类型转换：  static\_cast<double>(a)，将int的a强制转换为double  const\_cast<int>(a)，将const int的a强制转换为int  reinterpret\_cast<double \*>(a)，将int\*的a强制转换为double \*  dynamic\_cast<derived \*>(new base)，将对象base\*的强制转换为对象derived\* |

## 五、C++的类

请简单介绍下C++类的访问权限控制。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C++类的访问权限（也叫访问级别）就是指类外和子类对类内成员的访问权限。C++成员的默认访问权限是private，其中private一般用来存放私有的成员变量，   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **访问权限** | **类外（实例化对象）** | **类内成员** | **子类成员** | **友元函数** | **友元类** | | public | 可访问 | 可访问 | 可访问 | 可访问 | 可访问 | | protected | **不可访问** | 可访问 | 可访问 | 可访问 | 可访问 | | private | **不可访问** | 可访问 | **不可访问** | 可访问 | 可访问 |   在C++中，鼓励将所有数据声明为private，想要访问数据则单独定义public的成员函数来访问，public为公有的成员函数。 |

请简单介绍下C++类的构造函数和析构函数。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 构造函数和析构函数是C++类中2个非常特殊的成员函数。C++创建类的对象时**自动调用**的创建函数就是构造函数，C++销毁类的对象时**自动调用**的析构函数就是析构函数。C++类的定义会提供默认的构造函数和析构函数，但是**构造函数一般要自定义**，如果**类中没有指针成员变量时一般可以不写析构函数或者使用默认的析构函数**。  构造函数和析构函数的对比如下：   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | **形式** | **访问权限** | **参数** | **返回值** | **重载方式** | **虚函数** | | 构造函数 | 类名(参数列表){}; | 一般为public，  单例模式中设为private | 自定义 | 无 | 可以重载，一般有多个构造函数 | 不推荐声明为虚函数 | | 析构函数 | ~类名(){}; | public | 无 | 无 | 不可以重载，只能有一个析构函数 | 推荐声明为虚函数 |   构造函数中有一种特殊的**初始化列表**，用其来初始化类的成员变量**执行效率会更高**：   |  | | --- | | // 构造函数(2种写法实现的功能一致，但是第一种写法效率更高)  complex(double r=0,double i=0):re(r),im(i){}  complex(double r=0,double i=0){re=r;im=i;} | |

请简单介绍下C++类中的this指针。

|  |
| --- |
| **this指针是C++中类隐藏的一个const的自身类型的指针，它指向当前对象**，用来在普通的成员函数中标识当前调用对象。例如stu.show()；对成员函数show()来讲，其内部可以访问this指针，该指针指向当前调用它的对象stu。  **this指针只能用于类的普通成员函数内部**，可以用来调用该类的其他普通成员函数和成员变量。  **类的静态成员函数内部无法访问this指针**，因为静态成员函数不需要调用对象。 |

请介绍下C++中的友元类和友元函数

|  |
| --- |
| 类A中如果添加一个用friend修饰的全局函数的声明，则该函数被视为类A的友元函数，友元函数可以访问类A的所有权限的成员。  类A中如果添加一个用friend修饰的类B的声明，则类B被视为类A的友元类，类B的所有成员函数可以访问类A的所有权限的成员。**友元关系是单向性的，不能被继承，也不具备传递性**。  **友元函数和友元类可以提供程序运行效率**（让编译器放弃类型检查和安全性检查），**但是它破坏了类的封装性和安全性**，推荐不使用或者少用。 |

请简单介绍下C++中的临时对象？什么时候会产生临时对象？

|  |
| --- |
| C++中的临时对象（也叫无名对象）通过“类型名()”的形式来创建，其特点就是不需要声明对象名称，编译器在很多情况下都会自动创建很多临时对象，常见的有以下几种情况：   1. **按值传参**。在调用函数F(A a)时需要传入一个对象a，这时编译器会自动创建一个和a一样的临时对象去传入函数中。 2. **按值返回**。return A();这种情况下编译器会创建一个A()的临时对象，然后将临时对象返回，原有的对象销毁。 3. **类型转换**。假设类A有一个可输入一个int的构造函数，那么在执行语句A a=10;时，10会被编译器隐式调用构造函数构建一个A的临时对象赋给a。   由于临时对象也需要调用构造函数和析构函数，所以也会增加程序的开销，**在实际编程中要尽可能避免临时对象的产生，例如多使用按引用传递和按引用返回**。 |

请简单介绍下C++类的继承机制？

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 类的继承是让某种类型的对象快速获得另一个类型对象的属性和方法的机制，目的是提高代码的复用性。常见的继承有3种形式：  **实现继承**：父类提供属性和方法，子类不需要添加额外代码（对应C++的普通函数）。  **可视继承**：父类提供属性和方法，子类可以重写父类的方法（对应C++的虚函数）  **接口继承**：父类提供属性和方法名，子类必须实现父类的方法（对应C++的纯虚函数）  C++中按照继承权限可以分为：   |  |  |  |  | | --- | --- | --- | --- | | **继承方式** | **公有成员** | **保护成员** | **私有成员** | | public | 公有成员->公有成员 | 保护成员->保护成员 | 父类私有成员，子类不可访问 | | protected | 公有成员->保护成员 | 保护成员->保护成员 | 父类私有成员，子类不可访问 | | private | 公有成员->私有成员 | 保护成员->私有成员 | 父类私有成员，子类不可访问 |   C++一般最常用的是公有继承（public）,显然保护继承和私有继承会让子类变成一个外界不可访问的黑盒。  C++的继承中，子类会继承父类的所有成员变量和部分成员函数，**父类的成员变量会在子类中复制一份**，**子类会继承父类成员函数的调用权**，而不会复制函数代码。父类的如下方法无法被子类继承：  （1）**父类的构造函数、析构函数和拷贝构造函数**。  （2）**父类的重载运算符**。  （3）**父类的友元函数** |

请简单介绍下C++类的多态机制/谈一谈你对多态的理解

|  |
| --- |
| 多态是指调用对象的同一个方法时会根据调用对象的类型来执行不同的实现（“看人下菜”）。C++中类的多态有2种形式：  **重载（overload）**:C++类允许定义具有相同方法名的类成员函数，但是其参数列表不同。重载是**编译式多态（也叫静态多态，静态链接，早绑定）**，即C++程序在编译阶段就确定了对象的行为。C++中的重载分为**函数重载**和**运算法重载**2种形式。  **覆盖（override）**:子类重新定义父类的虚函数的做法。覆盖是**运行时多态（也叫动态多态，动态链接，后期绑定）**，即通过继承方式在程序运行时才能确定相应调用的方法。 |

请简单介绍下你对C++中运算符重载的理解。

|  |
| --- |
| C++中的重载分为函数重载和运算符重载，函数重载容易理解，运算符重载则是C++特有的机制。C++允许重写或者重载大部分C++内置的运算符，重载的运算符是带有operator+重载运算符号构成的函数，例如假设要为类A重载+运算符，使得它的对象可以进行加法运算，有2种实现方式：   1. 声明为类A的非成员函数（传递2个参数）：   A operator+(const A&,const A&);   1. 声明为类A的成员函数（只传递1个参数即可，另一个是自身this）：   A operator+(const A&); |

请简单介绍下你对C++中虚函数的理解。

|  |
| --- |
| 虚函数是C++中用virtual关键字声明的成员函数，允许子类覆盖该函数，属于C++中类的**运行时多态**的实现。  **运行时多态（实现条件）**要求：   1. 调用函数的对象必须是**指针**或**引用**; 2. 被调用的函数必须是**虚函数**且子类完成了虚函数的重写。   **运行时多态（内部原理）**为**每个具有虚函数的类对象**在内存中添加一个隐藏成员，即**虚指针**（vptr，也叫虚表指针）。虚指针指向**虚函数表**（vtbl），虚函数表像一个数组，存放了当前类和其所有父类的每个虚函数的地址。  **运行时多态（最终目的）**允许将**子类指针**赋值给**父类指针**，当程序员用**父类指针**指向**子类**时，虚指针指向子类的虚函数表，这样**父类指针可以调用到子类的虚函数实现**。 |

为什么说虚函数是运行时多态/动态多态/后期绑定？

|  |  |
| --- | --- |
| **运行时多态是根据调用对象的类型来确定应该调用哪个虚函数**，没法在编译期确定，只能在运行期确定，所以说是运行时多态。  例如，父类A有两个子类B和C，A中有一个虚函数method()，且B和C中也都有method()的实现。   |  | | --- | | A \*a;//定义父类指针  a=new B();  a->method();//执行B的method()  a=new C();  a->method();//执行C的method() | |

类中的所有成员函数都可以设为虚函数吗？

|  |
| --- |
| C++类中有3类函数不能声明为虚函数：  （1）**内联函数**。内联函数是在**编译期**用函数体来替换函数的调用指令，而虚函数要求**运行时**根据对象类型才能知道调用哪个虚函数。  （2）**构造函数**。类的构造函数不能声明为虚函数，即使声明了也会调用父类的函数，因为子类还没构造好，不会产生多态（子类构造好才有多态）；相反，**类的析构函数被推荐声明为虚函数**，这样子类就可以覆盖父类的析构函数。  （3）**静态函数**。静态成员函数中不存在this指针，无法知道自身的调用对象是谁，虚函数必须知道自己是被谁调用，所以矛盾。 |

请问C++中的纯虚函数的作用是什么？什么场景需要用到纯虚函数？

|  |
| --- |
| **纯虚函数**是在虚函数的声明中加“=0”，**含有一个即以上的纯虚函数的类就是抽象类**，**抽象类无法实例化对象**，只能作为C++的接口使用。**抽象类的子类必须实现其父类的所有纯虚函数**。  **纯虚函数的作用就是将类的声明和实现分离**，父类负责声明，子类负责实现，为其他程序提供一个适当的、通用的、标准化接口。  当不知道某个方法如何实现或者有多种实现方式时，可以声明为纯虚函数。 |

请介绍下C++中单继承、多继承和虚继承的区别。

|  |
| --- |
| C++允许单继承、多继承和虚继承。多继承往往会使代码和问题变得复杂，所以**在实际的开发中，不推荐使用多继承和虚继承**，java等语言不允许多继承。  单继承是子类只有一个父类；  多继承是子类有多个父类，**多继承存在一种特殊情况：菱形继承**，容易存在数据冗余和二义性问题。**虚继承是为了解决多继承时的命名冲突和冗余数据问题**，虚继承允许派生类中只保留一份间接基类的成员。虚基类不论被继承多少次，其子类中都只包含一份虚基类的成员。C++标准库中的iostream就是虚继承的实际应用案例。 |

C++中如何估计类对象的内存大小？

|  |
| --- |
| 不考虑内存对齐的话，C++中的类大小计算与**成员函数**和**静态成员变量**无关，影响类大小的有普通成员变量、虚函数。所以普通的类的对象的内存大小是普通成员变量的内存之和+虚指针大小（如果有类中有虚函数的话，32位机上是4个字节）。如果该类存在父类，则需要加上父类对象的大小，虚继承的父类认为其存在一个虚指针  特殊情况：空类（无函数无变量）也可以实例化，其对象的大小为1，这是因为类的实例化必须在内存中分配一块地址，所以编译器会为空类分配一个字节。 |

请结合虚函数机制介绍类对象在单继承、多继承和虚继承的内存布局和虚函数表结构

|  |
| --- |
| 一个有1个普通函数FuncA和2个int变量a和b的类对象Base的布局  一个有1个普通函数FuncA、1个虚函数FuncB和2个int变量a和b的类对象Base的布局    一个单继承Base但是不覆盖虚函数FuncB的子类对象Derive的布局  一个单继承Base但是覆盖虚函数FuncB的子类对象Derive的布局    一个多继承BaseA和BaseB但是不覆盖其虚函数的子类对象Derive的布局（左侧）  一个多继承BaseA和BaseB但是覆盖其虚函数的子类对象Derive的布局（右侧）    一个虚继承Base但是不覆盖其虚函数FuncB的子类对象Derive的布局（左侧）  一个虚继承Base但是覆盖其虚函数FuncB的子类对象Derive的布局（右侧）    结论：  offset 0 表示当前虚函数表地址距离对象顶部地址的偏移量  RTTI指针 指向存储运行时类型信息的地址，用于运行时类型识别 |

C++类的内存对齐

|  |
| --- |
| C++在实际分配类的对象的内存时会存在一个内存对齐原则，并不是严格按照各个成员变量的内存大小顺序分配，内存对齐比较复杂，这里不详细介绍，只介绍内存对齐的意义：  内存对齐可以使CPU处理C++程序的效率大幅度提升；部分硬件平台的CPU无法访问其地址空间的任意数据，所以必须使用内存对齐。 |

## 七、C++的泛型编程

你了解C++的模板吗？请简单介绍下函数模板和类模板。

|  |
| --- |
| C++的模板是实现代码重用的工具，可以实现数据类型参数化，当模板使用时才会实例化。C++中的模板分为函数模板和类模板。  类模板：  template<class T> class 类名{};  函数模板：  template<class T> 返回值类型 函数名(T arg){};  template<class T1 class T2> 返回值类型 函数名(T1 arg1,T2 arg2){}; |

请简单介绍下C++的STL？STL的六大组件是什么？

|  |
| --- |
| STL是C++中的标准模板库，用于C++的泛型编程。STL具有高可重用性、高性能、高移植性和跨平台的优点，是C++程序员编程的重要函数库。STL的设计时将数据和操作分离（与类的封装相反），其中主要有六大组件：  **容器**（Containers）提供各种数据结构，封装了内存分配的机制，本质是C++的类模板。  分配器（Allocators）为容器的内存分配机制由提供支持，本质是C++的类模板。  **算法**（Algorithms）封装了对于容器内数据的常见操作，本质是C++的函数模板。  **迭代器**（Iterators）是一种泛化的智能指针，每个容器都有专属的迭代器。  仿函数（Functors）类似于函数，本质是重载了operator()的类模板。  适配器（Adapters）可以对容器、迭代器、仿函数进行转换。  六大组件的关系如下：容器通过分配器获得内存空间，算法通过迭代器操作容器中的数据，仿函数协助完成算法设计（传入函数参数），适配器可以修饰仿函数或者容器来适配不同的算法。 |

C++直接使用数组好还是使用STL中的array？它是怎么实现的？

|  |
| --- |
| array是STL中的序列式容器，它是一段无法扩容的固定长度的顺序存储的数组结构。它是在普通数组的基础上，添加一些常见的成员函数，它必普通数组更安全，并且效率接近。由于array是STL库中的，所以它可以直接使用其算法库的函数（如sort等），它使用at()的成员函数来获取数组内的元素，不用担心数组越界的问题。 |

C++的vector是怎么实现的？resize、reserve、clear是怎么实现的？

|  |
| --- |
| vector是STL中的序列式容器，它是一段可以单向扩容的顺序存储的数组结构。它内存存在三个迭代器：first（指向起始字节位置）、last（指向最后一个存储的元素）和end（指向末尾字节位置）。size()获取其容器中存储的元素个数，capacity()获取其容器此时的容量。vector每次插入新节点，不一定分配内存，只有超过容量时才会申请新内存。当size=capacity时，vector就会自动扩容一倍。reserve()是主动设置的vector容量，resize()是主动调整的vector的元素个数，不足补足，超过删除。clear()把size设置为0，capacity不变。 |

C++的deque的内部是怎么实现的？它和queue和stack的关系是什么？

|  |
| --- |
| deque是STL中的序列式容器，它是一段可以双向扩容的顺序存储的双向队列结构。由于其双端都可以添加或者删除元素，所以其本质已经不满足队列的“先进先出”的性质。deque运行效率不如vector。  【deque的内部原理】deque内部维护一个map作为主控，其中每个元素都是指针，指向一段连续线性的缓存区来存储数据；同时维护2个迭代器：start（指向第一缓冲区的第一个元素）和end（指向最后缓冲区的最后一个元素）；维护一个容量，会提前分配多个缓冲区，所以其插入元素可能不会分配内存。    stack和queue是deque的容器适配器，即在deque的基础上封装而来： |

C++的list的使用场景？它是怎么实现的？

|  |
| --- |
| list是STL中的序列式容器，它是一段可以双向扩容的链式存储的双向链表结构。它适合作为需要随机插入或删除元素的容器。list每次插入新节点都会新分配内存。 |

你了解pair容器吗？它经常怎么使用？

|  |
| --- |
| Pair是STL提供的用于关联式容器保存数据的类模板，它可以保存2个相关的值，通过它的两个公有的数据成员first或者second来访问，可以通过make\_pair()函数来创建一个pair对象。  它常见的用法是遍历关联式容器map和unordered\_map |

C++的map和unordered\_map有什么区别？分别在什么场景下使用?

|  |
| --- |
| map和unordered\_map是STL中的4个关联式容器的2个（另外2个是set和unordered\_set）。map和set底层都使用红黑树进行实现，并且会默认按照key进行字典序排序，查询、查找、删除效率都为O(logn)。适合用于数据规模较小、效率要求不高、元素有序的场景。  unordered\_map和unordered\_set底层都使用哈希表来实现，其内部元素是无序的，查询、查找、删除效率都为常数级O(1)。适合用于数据规模较大、效率要求较高、元素可以无序的场景。 |

C++的string内部是怎么实现的，它的常用API有哪些？

|  |
| --- |
| string不属于C++的STL库，但是使用频率也非常高。string本质上是一个保存字符的序列容器，其末尾字符是空字符’0’。 |

C++的heap和priority\_queue有什么区别？

|  |
| --- |
| heap不属于STL的容器，它是priority\_queue(优先级队列)的幕后助手。heap建立在完全二叉树上，分为大根堆和小根堆，它的特点是以任何顺序加入容器的元素，取出时一定从优先权最高的元素开始取。priority\_queue也不属于STL的容器，它是容器适配器。 |

什么是模板全特化和偏特化？

|  |
| --- |
| C++中模板分为类模板和函数模板，特化分为全特化和偏特化。模板的特化即模板的特殊化，针对某些特殊类型单独定义函数实现，可以提高编译器的效率。这种模板特化可以认为是模板的一种重载形式，模板全特化后就与普通的类或函数相同，例如因为某些函数对int和double型数据的处理策略不同，不能统一的类型处理。  函数模板只有全特化，没有偏特化，偏特化的功能可以靠函数重载实现。    类模板有全特化和偏特化。    对主版本模板类、全特化类、偏特化类的调⽤优先级从⾼到低进⾏排序是：全特化类>偏特化类>主版本模板类 |

简述一下C++的RAII机制？

|  |
| --- |
| RAII是Resource Acquisition Is Initialization（“资源获取就是初始化”）的简称，是C++的管理资源、避免泄漏的方法，实际上我们已经在不自觉的使用。RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命期内控制对资源的访问，使之始终保持有效，最后在对象析构的时候，释放构造时获取的资源。  简单说，就是将对资源的申请放在类的构造函数中，将对资源的释放放在类的析构函数中，这里的资源不单单指使用new或delete分配和释放内存资源，也可以指网络套接字、互斥锁、文件句柄等。这样当执行某些操作时就会自动执行资源的申请和释放操作。  假如我们使用RAII思想来计算函数执行用时，可以创建一个timer类，求构造函数记录此刻运行时间，析构函数记录此时已运行时间并且输出。然后让timer和函数fun放在一个代码块中执行即可：{timer t;fun()}。  ScopeExit就是基于RAII机制实现的一个资源管理类，当退出它的作用域时自动执行资源的销毁操作。 |

你了解智能指针吗？请简述下C++中的智能指针。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C++的指针是在堆上分配内存，裸指针（普通指针）使用很不安全，处理不当容易变成野指针（指向的内存已释放），造成内存泄露。C++的指针在开发场景中可能经常被多次复制、赋值、移动，例如函数传参、并行场景处理共享对象。  **为了更加安全地使用动态内存，C++引入了智能指针的概念**。智能指针（pointer-like classes，像指针的类）。智能指针类似普通的C++指针，其重要区别是它负责自动释放所指向的对象。  C++98提供了auto\_ptr；C++11舍弃了C++98的auto\_ptr，在头文件<memory>提供了3个新的智能指针类型，这些智能指针都是在裸指针的基础上封装而来的，对比如下：   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 对象所有权 | 执行效率 | 应用频率 | 安全性 | 特点 | | 裸指针 | — | 最高 | 最高 | 较低 | 手动delete、复制、拷贝、处理异常等 | | unique\_ptr<T> | 专属所有权 | 较高 | 最高 | 较高 | 不支持复制和赋值;支持移动; | | shared\_ptr<T> | 共享所有权 | 较低 | 较低 | 较高 | 支持复制、赋值和移动；循环引用时出错 | | weak\_ptr<T> | 共享所有权 | 较低 | 最低 | 较高 | 支持复制、赋值和移动；弥补循环引用 |   unique\_ptr是一种具有专属所有权的智能指针。unique\_ptr管理的内存只能被一个对象持有，该指针不支持复制和赋值，只支持移动。其内存占用和执行性能和裸指针接近。unique\_ptr作为类成员时，不需要在析构函数中delete；unique\_ptr在执行代码抛出异常时，离开作用域也能自行释放内存。  shared\_ptr是一种具有共享所有权的智能指针。shared\_ptr管理的内存可以被多个对象持有，内部使用引用计数来管理内存；该指针支持复制和赋值和移动。由于shared\_pt需要额外维护一个原子级别的引用计数，其内存占用和执行效率都远远不如裸指针。该指针适合用于共享权不明的场景，比如并发场景的多线程。shared\_ptr可以被多个线程同时读，同时写时需要加锁。  weak\_ptr是一种具有临时所有权的智能指针。weak\_ptr可以认为是shared\_ptr的补充，它可以解决shared\_ptr双向引用的问题，经常需要和 shared\_ptr 一起使用，例如用于类的继承场景：父类持有指向子类的shared\_ptr，子类持有指向父类的weak\_ptr。  智能指针本质是一个C++类模板，封装了C++的普通类指针，覆写了析构函数、拷贝构造函数、移动构造函数，重载了\*、->、=操作符函数。 |

unique\_ptr如何转换所有权？

|  |
| --- |
| unique\_ptr是一种具有专属所有权的智能指针。unique\_ptr管理的内存只能被一个对象持有，该指针不支持复制和赋值，只支持移动。unique\_ptr通过内部的**移动构造函数**来转移对象所有权，在**移动构造函数**中新对象会指向原来对象指针指向的内存，然后原有对象指针会被设为空。 |

## 七、C++11的新特性

你了解C++中的可调用对象？/bind、function和lambda表达式吗？

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C++11引入了一个重要的概念叫做**可调用对象**，简单理解，就是可以让一个函数调用作为其他函数的参数。C++中的可调用对象有以下几种情况：  (1)**普通函数**。bool result= cmp(a, b);//cmp是一个返回值为bool的普通函数  (2)**函数指针**。   |  | | --- | | bool (\*p)(const int &a, const int &b);//创建一个函数指针  p = cmp;//与函数进行绑定 |   (3)**仿函数**（重载了()运算符的类对象）   |  | | --- | | class MyPlus{  int operator()(const int &a , const int &b) const{ return a + b; }  };  MyPlus a;//定义一个仿函数  a(1,2);//调用仿函数 |   (4)**匿名函数**（lambda表达式）。支持lambda表达式是C++11引入的重要特性。lambda表达式可以让程序更加灵活简介。简单说，lambda表达式定义了一个匿名函数，可以快速插入程序中，不需要单独定义一个函数体，例如：   |  | | --- | | auto f = [](int a) **->** int { return a + 1; };//定义一个lambda表达，**->** int表示返回一个int  auto f = [](int a) { return a + 1; };//定义一个lambda表达，**省略返回值**  std::cout << f(1) << std::endl; // 输出: 2 |   (5)**std::function对象**。function可以认为就是C++11中的可调用对象的包装器。   |  | | --- | | std::function<int(int, int)> Func;//声明一个返回值为int，参数为两个int的可调用对象类型 |   **std::bind()**用来将参数和可调用对象一起绑定，绑定结果可以使用function保存。例如：   |  | | --- | | std::function<void()> f\_display= std::bind(print\_num, 31337);//print\_num是函数名，31337是为它输入的参数。 | |

你了解C++11的左值和右值吗？谈论下左值引用和右值引用？

|  |
| --- |
| 一般来讲，等号左边的符号是**左值**，等号右边的符号是**右值**，如果表达式中没有等号，则能用取地址符&的皆为左值，否则就是右值。常见的左值一般是**函数名**、**变量名**、返回左值引用的**函数调用**、**前置自增自减表达式**（++i，--i）,a=b,a+=b,a-=b等。运算表达式产生的**临时变量**、**lambda表达式**等都是**纯右值**。  **左值引用**就是对左值的引用，**右值引用**就是对右值的引用。 |

你了解C++11的类型推导吗？请区分下C++11的auto和decltype

|  |
| --- |
| C++11中的auto和decltype都是让编译器在编译期间就推导出变量的类型。两者的区别如下:  用法不同。auto varname = value;auto根据右值来推导出变量的类型，  auto使用时必须立即初始化，不能推断数组、不能用作函数参数。  decltype(exp) varname = value;decltype根据exp表达式来推导出变量的类型。decltype使用时不需要立即初始化，其表达式可以是函数调用，是左值时则推导出左值引用。 |

你了解深拷贝和浅拷贝吗？那移动语义和移动构造函数呢？

|  |
| --- |
| 复制指针时，只复制指针变量，不复制指针指向的内存，那复制就是**浅复制**，反之就是**深复制**。一般来讲，浅复制在C++中不安全。  **移动语义**是C++11新引入的概念，即转移**对象所有权**。常见的就是移动指针指向的内存，指针移动后，旧指针变为NULL，新指针指向旧指针指向的内存，这个过程就移动了对内存的所有权。  一般**含有指针成员的类**可能会想支持**移动语义**，这时就要在类中重写**移动构造函数**：  A(A&& a){this->data\_=a.data\_;a.data\_=nullptr;}  然后通过**move函数**调用A c=std::move(a);// 调用移动构造函数。  **移动语义仅针对于那些实现了移动构造函数的类的对象**，对于那种基本类型int、float等没有任何优化作用，还是会拷贝，因为它们实现没有对应的移动构造函数，**C++中的STL基本都支持移动语义**。 |

请简述下C++的nullptr和NULL的区别

|  |
| --- |
| **nullptr是C++11引入的空指针，目的是解决NULL的二义性问题**。在C++中NULL即可以表示0也可以表示空指针，这样在函数重载时可能调用非目标函数，例如两个重载函数一个接收int，一个接收void\*指针，那么NULL参数就会引起歧义，它是表示指针还是数字0，C++11专门引入nullptr来表示空指针。 |

请详细介绍下C++的各种强制类型转换的原理及其使用？

|  |
| --- |
| C语言中，代码可以使用小括号来进行强制类型转换，C++新增了4个关键字来强制类型转换：  **static\_cast<double>(a)**的使用场景：基本数据类型之间的转换（如int转换为char），将任何类型的表达式转换为void类型；将子类指针转换为父类指针是安全的；将父类指针转换为子类指针是不安全的，没有运行时类型检查机制。  **const\_cast<int>(a)**的使用场景：将常量指针转换为非常量指针，并且仍然指向原来的对象，常引⽤被转换为⾮常引⽤，并且仍然指向原来的对象。去掉类型的const或volatile属性。  **reinterpret\_cast<double \*>(a)**的使用场景：以将整型转换为指针，也可以把指针转换为数组；可以在指针和引⽤⾥进⾏肆⽆忌惮的转换，平台移植性差。  **dynamic\_cast<derived \*>(new base)**的使用场景，将将父类指针转换为子类指针是安全的，只能用于存在虚函数的父子关系的强制类型转换。 |

malloc出来20字节内存，为什么free不需要传入20呢？不会产生内存泄露吗？

|  |
| --- |
| **malloc申请20字节的内存，实际上拿到的内存比20字节大**，多出来的部分用来管理内存，这些管理信息中记录了该内存块的长度。  free释放内存时，需要传入一个malloc返回的指针，这个指针标识了申请的内存块的管理信息的所在区，free内部读取管理信息就可以知道要如何释放了。  不过free并不会立即将内存块释放掉，甚至上边的数据都不会改写，只会做个标记标识该内存块的所有权已经归还，但是为了安全起见，free后建议将指针设为NULL来防止释放后有被访问。 |

new[]和delete[]为什么必须得配对使用？如果不配对使用呢？

|  |
| --- |
| new[]和delete[]申请C++内置数据类型（int,float等）的内存时，可以不配对使用，如果申请的是自定义对象的内存时，必须配对使用。**这是因为new[]在处理非内置数据类型时，其会在内存中多分配4个字节来存储数组的长度**。  如果new[]处理C++内置数据类型，delete[]知道int是内置类型，也知道其元素大小，只要按大小删除即可，不需要调用析构函数。  如果new[]处理自定义对象，new[]会在内存起始部分分配4个字节作为数组个数。如果用delete删除，delete不会访问前4个字节来获取元素大小，只知道要删除起始到结尾的一段内存，它将会只调用一次析构函数；delete[]则会访问前4个字节来获取元素大小，调用多次析构函数。 |

你了解返回值优化吗？

|  |
| --- |
| **返回值优化（RVO）是C++编译器的优化技术**，当函数需要返回一个局部对象实例时，如果返回值类型和局部对象实例一样则会直接返回局部对象实例，否则就会创建一个临时对象并通过复制构造函数将目标对象复制到临时对象，然后返回临时对象。 |

enum和enum class有什么区别？

|  |  |  |
| --- | --- | --- |
| **enum是C语言就存在数据类型**，C++11引入了**枚举类**（enum class）,目的是对枚举类型的作用域进行限定，避免枚举成员的重定义。例如：  传统C风格的枚举定义，编译错误：   |  | | --- | | enum Sex{Girl,Boy}; enum Student{Girl,Boy}; |   使用C++11的枚举类定义，编译通过：   |  | | --- | | enum class Sex{Girl,Boy}; enum class Student{Girl,Boy}; | |

你了解列表初始化吗？

|  |
| --- |
| C++11引入了一个新的语法：列表初始化，即A d{123};对于满足以下条件的类，可以直接使用列表初始化：  类型是一个普通数组，如int[5]，char[]，double[]等  类型是一个类，且满足以下条件：  没有用户声明的构造函数  没有用户提供的构造函数(允许显示预置或弃置的构造函数)  没有私有或保护的非静态数据成员  没有基类  没有虚函数  没有{}和=直接初始化的非静态数据成员  没有默认成员初始化器 |

范围for循环

|  |
| --- |
| C++11引入了一种范围for循环的语法，即for(char c:s)是**只读写法**，for(char&c:s)是可写写法。 |

你了解C++11中引入的多线程部分吗？

|  |
| --- |
| 关于并行编程，C++11引入了许多新设计，例如线程对象std::thread、用于线程同步的互斥锁std::mutex和条件变量std::condition\_variable、使用RAII机制封装的std::lock、原子类型std::atomic、调用一次的std::call\_once、异步操作的std::future、async等。 |

## 八、C++的设计模式和算法

请简述下编译性语言和解释性语言有什么区别？

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 编译性语言和解释性语言有各自的优缺点。需要说明的是，这些优缺点并不是不可以改变的，一门语言的优点或缺点并不完全取决于其先天的设计，也取决于后天的生态建设，例如C#也是半编译半解释语言，但是其跨平台能力和Java不可同日耳语。Python作为跨平台能力强的语言，不仅因为它是解释性语言，而且它的学习成本低，开源生态好，很多程序员贡献了十分丰富的代码库。   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | **代表语言** | **转换工具** | **原理** | **优点** | **缺点** | | 编译性语言 | C、C++、Go | 编译器 | 先编译后执行，执行时可脱离源码和编译器 | 一次编译，多次运行。执行效率高。 | 跨平台性差 | | 解释性语言 | Python、JavaScript、PHP | 解释器 | 边翻译后执行，执行时不可脱离源码和解释器 | 跨平台性高，一次编译，到处运行。 | 无法脱离开发环境。  执行效率低。 | | 半编译半解释语言 | Java | 虚拟机 | 先编译后执行，执行时可脱离源码，不可脱离Java虚拟机 | 兼顾跨平台性和执行效率 | 执行效率不如C/C++等  跨平台不如Python等 | |

什么场景下使用继承方式，什么场景下使用组合？

|  |
| --- |
| 继承体现的是一种专门化的概念而组合则是一种组装的概念  另外确定是组合还是继承，最清楚的方法之一就是询问是否需要新类向上映射，也就是说当我们想重用原类型作为新类型的内部实现的话，我们最好自己组合，如果我们不仅想重用内部实现而且还想重用接口的话，那就用继承。  一般对于大型程序来讲，尽量使用组合，继承应该尽可能少用，或者继承层数不超过2层，复杂的继承关系会降低程序的可靠性。 |

你了解单例模式吗？它的饿汉模式和懒汉模式以及懒汉线程安全是怎么实现的？

|  |
| --- |
|  |

说出观察者模式类关系和优点

|  |
| --- |
|  |

说出代理模式类关系和优点

|  |
| --- |
|  |

说出工厂模式概念和优点

|  |
| --- |
|  |

说出构造者模式概念

|  |
| --- |
|  |

说出适配器模式概念

|  |
| --- |
|  |

你了解设计模式吗？可以列举一些常见的设计模式？

|  |
| --- |
| 设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。  比如单例模式，保证一个类仅有一个实例，并提供一个访问它的全局访问点。  适用于：当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时；当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。  比如工厂模式，定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。  适用于：当一个类不知道它所必须创建的对象的类的时候；当一个类希望由它的子类来指定它所创建的对象的时候；当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。 |

常见的排序算法有哪些？简单描述下排序算法的优缺点？

|  |
| --- |
| 答：选择、冒泡、快速、希尔、归并、堆排等。  1.快排：是冒泡排序的一种改进。  优点：快，数据移动少  缺点：稳定性不足  2.归并：分治法排序，稳定的排序算法，一般用于对总体无序，但局部有序的数列。  优点：效率高O(n)，稳定  缺点：比较占用内存 |

如何实现一个线程池？[C++线程池的实现之格式修订版](http://mp.weixin.qq.com/s?__biz=MzkyODE5NjU2Mw==&mid=2247484800&idx=1&sn=402ac9441890f201a5eb46641307a832&chksm=c21d373cf56abe2aa51c2aa20314a395c86266ae72a0e53fdcbc00d07d9d180308b183a48be0&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/s/_blank)

|  |
| --- |
|  |

Debug模式和Release模式有什么区别？

|  |
| --- |
| Debug模式会多申请一部分空间，分布在内存块的前后，用于存放调试信息。对于未初始化的变量，Debug模式下会默认对其进行[初始化](https://www.zhihu.com/search?q=%E5%88%9D%E5%A7%8B%E5%8C%96&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":1720297063}" \t "https://www.zhihu.com/question/_blank)，而Release模式则不会，所以就有个常见的问题，局部变量未初始化时，Debug模式和Release模式表现有所不同。Debug模式下可能运行正常，但Release模式下可能会返回错误结果，因为found局部变量在Release模式下没有初始化。 |

你经常使用的数据结构有哪些？怎么使用的？

|  |
| --- |
|  |

你了解C++中的正则表达式吗？

你了解C++中的输入输出吗？

你了解C++中的IO模型吗？

你了解C++提供的进程通信方式吗？

你了解C++提供的多线程、多进程机制吗？