C++常见面试题总结

本文主要总结了一些常见的C++面试题。链接现在不让发，所以如果需要**整理好的文档**的话，请关注本篇文章底部的**推广订阅公众号**获取：

Cpp编程小茶馆

进入正题，下面是自己整理的文档目录截图，目前只整理了41条常见面试题，也非常欢迎大家留言补充和讨论。

目录如下：

1、C和C++的区别

1）C是面向过程的语言，是一个结构化的语言，考虑如何通过一个过程对输入进行处理得到输出；C++是面向对象的语言，主要特征是“封装、继承和多态”。封装隐藏了实现细节，使得代码模块化；派生类可以继承父类的数据和方法，扩展了已经存在的模块，实现了代码重用；多态则是“一个接口，多种实现”，通过派生类重写父类的虚函数，实现了接口的重用。

2）C和C++动态管理内存的方法不一样，C是使用malloc/free，而C++除此之外还有new/delete关键字。

3）C++中有引用，C中不存在引用的概念

2、C++中指针和引用的区别

1）指针是一个新的变量，存储了另一个变量的地址，我们可以通过访问这个地址来修改另一个变量；

引用只是一个别名，还是变量本身，对引用的任何操作就是对变量本身进行操作，以达到修改变量的目的

2）引用只有一级，而指针可以有多级

3）指针传参的时候，还是值传递，指针本身的值不可以修改，需要通过解引用才能对指向的对象进行操作

引用传参的时候，传进来的就是变量本身，因此变量可以被修改

3、结构体struct和共同体union（联合）的区别

结构体：将不同类型的数据组合成一个整体，是自定义类型

共同体：不同类型的几个变量共同占用一段内存

1）结构体中的每个成员都有自己独立的地址，它们是同时存在的；

共同体中的所有成员占用同一段内存，它们不能同时存在；

2）sizeof(struct)是内存对齐后所有成员长度的总和，sizeof(union)是内存对齐后最长数据成员的长度、

结构体为什么要内存对齐呢?

1.平台原因（移植原因）：不是所有的硬件平台都能访问任意地址上的任意数据，某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常

2.硬件原因：经过内存对齐之后，CPU的内存访问速度大大提升。

4、#define和const的区别

1）#define定义的常量没有类型，所给出的是一个立即数；const定义的常量有类型名字，存放在静态区域

2）处理阶段不同，#define定义的宏变量在预处理时进行替换，可能有多个拷贝，const所定义的变量在编译时确定其值，只有一个拷贝。

3）#define定义的常量是不可以用指针去指向，const定义的常量可以用指针去指向该常量的地址

4）#define可以定义简单的函数，const不可以定义函数

5、重载overload，覆盖（重写）override，隐藏（重定义）overwrite，这三者之间的区别

1）overload，将语义相近的几个函数用同一个名字表示，但是参数列表（参数的类型，个数，顺序不同）不同，这就是函数重载，返回值类型可以不同

特征：相同范围（同一个类中）、函数名字相同、参数不同、virtual关键字可有可无

2）override，派生类覆盖基类的虚函数，实现接口的重用，返回值类型必须相同

特征：不同范围（基类和派生类）、函数名字相同、参数相同、基类中必须有virtual关键字（必须是虚函数）

3）overwrite，派生类屏蔽了其同名的基类函数，返回值类型可以不同

特征：不同范围（基类和派生类）、函数名字相同、参数不同或者参数相同且无virtual关键字

6、new、delete、malloc、free之间的关系

new/delete,malloc/free都是动态分配内存的方式

1）malloc对开辟的空间大小严格指定，而new只需要对象名

2）new为对象分配空间时，调用对象的构造函数，delete调用对象的析构函数

既然有了malloc/free，C++中为什么还需要new/delete呢?

运算符是语言自身的特性，有固定的语义，编译器知道意味着什么，由编译器解释语义，生成相应的代码。

库函数是依赖于库的，一定程度上独立于语言的。编译器不关心库函数的作用，只保证编译，调用函数参数和返回值符合语法，生成call函数的代码。

malloc/free是库函数，new/delete是C++运算符。对于非内部数据类型而言，光用malloc/free无法满足动态对象都要求。new/delete是运算符，编译器保证调用构造和析构函数对对象进行初始化/析构。但是库函数malloc/free是库函数，不会执行构造/析构。

7、delete和delete[]的区别

delete只会调用一次析构函数，而delete[]会调用每个成员的析构函数

用new分配的内存用delete释放，用new[]分配的内存用delete[]释放

一.构造函数

构造函数是和类名相同的一个函数，它的作用是实现对象的初始化。当对象被创建时，构造函数自动被调用。

特点：

1. 没有类型
2. 没有返回值（也不用写void）
3. 名字与类名相同
4. 可重载！

作用：完成类的对象的初始化

```
Cdate d; //定义对象d
```

注意：当对象d被创建时，会自动调用构造函数 d.Cdate()。

当类中未定义构造函数时，编译器会自动假设存在以下两个默认构造函数：(此构造函数什么都不做，就是个形式)。如果作者自己定义了构造函数，则默认的构造函数不会存在。

```
//默认构造函数一

Cdate::Cdate()

{

}

//默认构造函数二

Cdate::Cdate(const Cdate& a)

{

}
```

三.析构函数

我们已经知道构造函数是在创建对象时，对其进行初始化。而析构函数与其相反，是在对象被删除前象由系统自动执行它做清理工作。

作为一个类，可能有多个对象，每个对象生命结束时都要调用析构函数，且每个对象调用一次。

特点：

1. 无类型
2. 无返回值
3. 名字与类名相同
4. 不带参数，不可重载，析构函数只有一个！
5. 析构函数前“~” (取反符，表示逆构造函数)

作用：在对象被删除前做清理工作。

注意：对象的析构函数在对象被销毁前被调用，对象何时销毁也与其作用域相关。

例如，全局对象是在程序运行结束时销毁;

自动对象是在离开其作用域时销毁；

而动态对象是在使用delete运算符时销毁。

析构函数特别适用于当一个对象被动态分配内存空间，而在对象被销毁前希望释放它所占用的内存空间的时候。我们不会忽略初始化的重要性，却常常忽略清除的重要性，然而对销毁变量的内存清理是非常重要的。

例如，我们在堆中申请了一些内存，如果没有用完就释放，会造成内存泄露，会导致应用程序运行效率降低，甚至崩溃，不可掉以轻心。

而在c++中提供有析构函数，可以保证对象清除工作自动执行。

析构与构造的调用次序相反，即最先构造的最后被析构，最后构造的最先被析构。

7.1、虚函数、纯虚函数

虚函数：虚函数是C++中用于实现多态(polymorphism)的机制。核心理念就是通过基类访问派生类定义的函数,是C++中多态性的一个重要体现。利用基类指针访问派生类中的虚函数，这种情况下采用的是动态绑定技术。

纯虚函数：纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”.纯虚函数不能实例化对象。

抽象类的介绍

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构的较上层。

（1）抽象类的定义： 称带有纯虚函数的类为抽象类。

（2）抽象类的作用： 抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

（3）使用抽象类时注意：

- 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

抽象类是不能定义对象的。

总结：

- 1、纯虚函数声明如下： virtual void funtion1()=0; 纯虚函数一定没有定义，纯虚函数用来规范派生类的行为，即接口。包含纯虚函数的类是抽象类，抽象类不能定义实例，但可以声明指向实现该抽象类的具体类的指针或引用。
- 2、虚函数声明如下：virtual ReturnType FunctionName(Parameter) 虚函数必须实现，如果不实现，编译器将报错，错误提示为：
- 3、对于虚函数来说，父类和子类都有各自的版本。由多态方式调用的时候动态绑定。
- 4、实现了纯虚函数的子类，该纯虚函数在子类中就编程了虚函数，子类的子类即孙子类可以覆盖该虚函数，由多态方式调用的时候动态绑定。
- 5、虚函数是C++中用于实现多态(polymorphism)的机制。核心理念就是通过基类访问派生类定义的函数。
- 6、在有动态分配堆上内存的时候，析构函数必须是虚函数，但没有必要是纯虚的。

纯虚函数的引入，是出于两个目的：

- 1、为了安全，因为避免任何需要明确但是因为不小心而导致的未知的结果，提醒子类去做应做的实现。
- 2、为了效率，不是程序执行的效率，而是为了编码的效率。

动态绑定：基类指针是调用派生类的中的成员函数还是调用基类中的成员函数要到程序运行时确定。主要看此时基类指针所指向的对象。 这里要涉及一些很重要的概念，也是我最近看完Effective C++才明白的东西，记录下来。这些概念就是静态类型和动态类型，静态绑定和动态绑定。静态绑定和动态绑定。静态绑定是说前期绑定。 所谓对象的静态类型，就是它在程序中被声明的时候采用的类型。 考虑下面的class继承体系：

```
1 class Shape{
```

```
virtual void draw(color = Red) const=0;

...

...

};
class Rectangle:public Shape{
virtual void draw(color = Red) const;

...

...

};
class Circle:public Shape
{
virtual void draw(color = Red) const;



...

...

};
现在考虑以下这些指针：
Shape* ps;//静态类型为Shape*
Shape*pc =new Circle;//静态类型Shape*
Shape*pr = new Rectangle;//静态类型Shape*
```

在本例中，ps,pc,pr都被声明为Shape*类型的，所以它们的静态类型都是Shape*。注意：无论它们真正指向什么，它们的静态类型都是Shape*. 所谓的对象的动态类型是指“当前所指对象的类型”。也就是说，动态类型可以表现出一个对象将会有什么行为。根据上面的例子，pc的动态类型是Circle*，pr的动态类型是Rectangle*。ps没有动态类型，因为它没有指向任何对象。 动态类型一如其名所示，可以在执行过程中改变（通常是经过赋值运算）：

```
ps=pc; \\ps的动态类型如今是Circle*

ps=pr; \\ps的动态类型如今是Rectangle*
```

Virtual函数系动态绑定而来，意思是调用一个virtual函数的时候，究竟调用的是哪一个函数代码，取决于发出调用的那个对象的动态类型。

```
ps->draw(); \\调用的是Rectangle::draw(Red)
```

8、STL库用过吗？常见的STL容器有哪些？算法用过几个？

STL包括两部分内容：容器和算法

容器即存放数据的地方，比如array, vector，分为两类，序列式容器和关联式容器

序列式容器，其中的元素不一定有序，但是都可以被排序，比如vector,list,queue,stack，heap, priority-queue, slist

关联式容器，内部结构是一个平衡二叉树，每个元素都有一个键值和一个实值，比如map, set, hashtable, hash_set

算法有排序，复制等，以及各个容器特定的算法

迭代器是STL的精髓，迭代器提供了一种方法，使得它能够按照顺序访问某个容器所含的各个元素，但无需暴露该容器的内部结构，它将容器和算法分开，让二者独立设计。

Vector是顺序容器，是一个动态数组，支持随机存取、插入、删除、查找等操作，在内存中是一块连续的空间。在原有空间不够情况下自动分配空间，增加为原来的两倍。vector随机存取效率高，但是在vector插入元素，需要移动的数目多，效率低下。

注意：vector动态增加大小时，并不是在原空间之后持续新空间（因为无法保证原空间之后尚有可供配置的空间），而是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。因此，对vector的任何操作，一旦引起空间重新配置，指向原vector的所有迭代器就都失效了。

二叉查找树

要想了解二叉查找树，我们首先看下二叉查找树有哪些特性呢？

1， 左子树上所有的节点的值均小于或等于他的根节点的值

2， 右子数上所有的节点的值均大于或等于他的根节点的值

3， 左右子树也一定分别为二叉排序树

红黑树

红黑树就是一种平衡的二叉查找树，说他平衡的意思是他不会变成“瘸子”，左腿特别长或者右腿特别长。除了符合二叉查找树的特性之外，还具体下列的特性：

1. 节点是红色或者黑色

2. 根节点是黑色

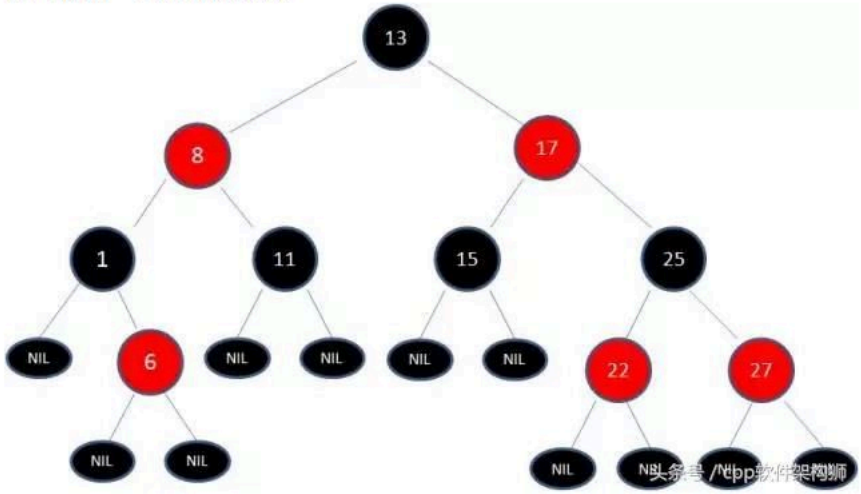
3. 每个叶子的节点都是黑色的空节点（NULL）

4. 每个红色节点的两个子节点都是黑色的。

5. 从任意节点到其每个叶子的所有路径都包含相同的黑色节点。

看下图就是一个典型的红黑树：

看下图就是一个典型的红黑树：



很多童鞋又会惊讶了，天啊这个条条框框也太多了吧。没错，正式因为这些规则，才能保证红黑树的自平衡。最长路径不超过最短路径的2倍。

当插入和删除节点，就会对平衡造成破坏，这时候需要对树进行调整，从而重新达到平衡。那什么情况下会破坏红黑树的规则呢？

有两种方式：变色和旋转。

9、const知道吗？解释一下其作用

const修饰类的成员变量，表示常量不可能被修改

const修饰类的成员函数，表示该函数不会修改类中的数据成员，不会调用其他非const的成员函数

const函数只能调用const函数，非const函数可以调用const函数

10、虚函数是怎么实现的

每一个含有虚函数的类都至少有有一个与之对应的虚函数表，其中存放着该类所有虚函数对应的函数指针（地址），

类的示例对象不包含虚函数表，只有虚指针；

派生类会生成一个兼容基类的虚函数表。

11、堆和栈的区别

1）栈 stack 存放函数的参数值、局部变量，由编译器自动分配释放

堆heap，是由new分配的内存块，由应用程序控制，需要程序员手动利用delete释放，如果没有，程序结束后，操作系统自动回收

2）因为堆的分配需要使用频繁的new/delete，造成内存空间的不连续，会有大量的碎片

3）对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方式是向下的，是向着内存地址减小的方向增长。

C++内存区域分为5个区域。分别是堆，栈，自由存储区，全局/静态存储区和常量存储区。

栈：由编译器在需要的时候分配，在不需要的时候自动清除的变量存储区。里面通常是局部变量，函数参数等。

堆：由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

自由存储区：由malloc等分配的内存块，和堆十分相似，不过它使用free来结束自己的生命。

全局/静态存储区：全局变量和静态变量被分配到同一块内存中，在以前的c语言中。全局变量又分为初始化的和未初始化的，在c++里面没有这个区分了，他们共同占用同一块内存。

常量存储区：这是一块比较特殊的存储区，里面存放的是常量，不允许修改。

C++内存区域中堆和栈的区别：

管理方式不同：栈是由编译器自动管理，无需我们手工控制；对于堆来说，释放由程序员完成，容易产生内存泄漏。

空间大小不同：一般来讲，在32为系统下面，堆内存可达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定空间大小的，例如，在vc6下面，默认的栈大小好像是1M。当然，也可以自己修改：打开工程。 project-->setting-->link，在category中选中output，然后再reserve中设定堆栈的最大值和 commit。

能否产生碎片：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题。

生长方向不同：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方式是向下的，是向着内存地址减小的方向增长。

分配方式不同：堆都是动态分配的；栈有静态和动态两种分配方式。静态分配由编译器完成，比如局部变量的分配。动态分配由malloca函数进行、但栈的动态分配和堆是不同的，它的动态分配由编译器进行释放，无需我们手工实现。

分配效率不同：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是c/c++库函数提供的，机制很复杂。库函数会按照一定的算法进行分配。显然，堆的效率比栈要低得多。

进程内存中的映像，主要有代码区，堆（动态存储区，new/delete的动态数据），栈，静态存储区

12、关键字static的作用

1）函数体内： static 修饰的局部变量作用范围为该函数体，不同于auto变量，其内存只被分配一次，因此其值在下次调用的时候维持了上次的值

2）模块内：static修饰全局变量或全局函数，可以被模块内的所有函数访问，但是不能被模块外的其他函数访问，使用范围限制在声明它的模块内

3）类中：修饰成员变量，表示该变量属于整个类所有，对类的所有对象只有一份拷贝

4）类中：修饰成员函数，表示该函数属于整个类所有，不接受this指针，只能访问类中的static成员变量

注意和const的区别！！！const强调值不能被修改，而static强调唯一的拷贝，对所有类的对象

13、STL中map和set的原理（关联式容器）

map和set的底层实现主要通过红黑树来实现

14、#include<file.h> #include "file.h" 的区别

前者是从标准库路径寻找

后者是从当前工作路径

15、什么是内存泄漏？面对内存泄漏和指针越界，你有哪些方法？

动态分配内存所开辟的空间，在使用完毕后未手动释放，导致一直占据该内存，即为内存泄漏。

方法：malloc/free要配套，对指针赋值的时候应该注意被赋值的指针是否需要释放；使用的时候记得指针的长度，防止越界

16、定义和声明的区别

声明是告诉编译器变量的类型和名字，不会为变量分配空间

定义需要分配空间，同一个变量可以被声明多次，但是只能被定义一次

17、C++文件编译与执行的四个阶段

1）预处理：根据文件中的预处理指令来修改源文件的内容

2）编译：编译成汇编代码

3）汇编：把汇编代码翻译成目标机器指令

4）链接：链接目标代码生成可执行程序

18、STL中的vector的实现，是怎么扩容的？

vector使用的注意点及其原因，频繁对vector调用push_back()对性能的影响和原因。

vector就是一个动态增长的数组，里面有一个指针指向一片连续的空间，当空间装不下的时候，会申请一片更大的空间，将原来的数据拷贝过去，并释放原来的旧空间。当删除的时候空间并不会被释放，只是清空了里面的数据。对比array是静态空间一旦配置了就不能改变大小。

vector的动态增加大小的时候，并不是在原有的空间上持续新的空间（无法保证原空间的后面还有可供配置的空间），而是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，并释放原空间。在VS下是1.5倍扩容，在GCC下是2倍扩容。

19、STL中unordered_map和map的区别

map是STL中的一个关联容器，提供键值对的数据管理。底层通过红黑树来实现，实际上是二叉排序树和非严格意义上的二叉平衡树。所以在map内部所有的数据都是有序的，且map的查询、插入、删除操作的时间复杂度都是O(logN)。

unordered_map和map类似，都是存储key-value对，可以通过key快速索引到value，不同的是unordered_map不会根据key进行排序。unordered_map底层是一个防冗余的哈希表，存储时根据key的hash值判断元素是否相同，即unoredered_map内部是无序的。

20、C++的内存管理

在C++中，内存被分成五个区：栈、堆、自由存储区、静态存储区、常量区

栈：存放函数的参数和局部变量，编译器自动分配和释放

堆：new关键字动态分配的内存，由程序员手动进行释放，否则程序结束后，由操作系统自动进行回收

自由存储区：由malloc分配的内存，和堆十分相似，由对应的free进行释放

全局/静态存储区：存放全局变量和静态变量

常量区：存放常量，不允许被修改

21、 构造函数为什么一般不定义为虚函数？而析构函数一般写成虚函数的原因？

1、构造函数不能声明为虚函数

1）因为创建一个对象时需要确定对象的类型，而虚函数是在运行时确定其类型的。而在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型，是类本身还是类的派生类等等

2）虚函数的调用需要虚函数表指针，而该指针存放在对象的内存空间中；若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表地址用来调用虚函数即构造函数了

2、析构函数最好声明为虚函数

首先析构函数可以为虚函数，当析构一个指向派生类的基类指针时，最好将基类的析构函数声明为虚函数，否则可以存在内存泄露的问题。

如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除指向派生类的基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全。

子类析构时，要调用父类的析构函数吗?

析构函数调用的次序时先派生类后基类的。和构造函数的执行顺序相反。并且析构函数要是virtual的，否则如果用父类的指针指向子类对象的时候，析构函数静态绑定，不会调用子类的析构。

不用显式调用，会自动调用

22、静态绑定和动态绑定的介绍

静态绑定和动态绑定是C++多态性的一种特性

1）对象的静态类型和动态类型

静态类型：对象在声明时采用的类型，在编译时确定

动态类型：当前对象所指的类型，在运行期决定，对象的动态类型可变，静态类型无法更改

2）静态绑定和动态绑定

静态绑定：绑定的是对象的静态类型，函数依赖于对象的静态类型，在编译期确定

动态绑定：绑定的是对象的动态类型，函数依赖于对象的动态类型，在运行期确定

只有虚函数才使用的是动态绑定，其他的全部是静态绑定

23、 引用是否能实现动态绑定，为什么引用可以实现

可以。因为引用（或指针）既可以指向基类对象也可以指向派生类对象，这一事实是动态绑定的关键。用引用（或指针）调用的虚函数在运行时确定，被调用的函数是引用（或指针）所指的对象的实际类型所定义的。

24、深拷贝和浅拷贝的区别

深拷贝和浅拷贝可以简单的理解为：如果一个类拥有资源，当这个类的对象发生复制过程的时候，如果资源重新分配了就是深拷贝；反之没有重新分配资源，就是浅拷贝。

25、 什么情况下会调用拷贝构造函数（三种情况）

系统自动生成的构造函数：普通构造函数和拷贝构造函数 （在没有定义对应的构造函数的时候）

生成一个实例化的对象会调用一次普通构造函数，而用一个对象去实例化一个新的对象所调用的就是拷贝构造函数

调用拷贝构造函数的情形：

1）用类的一个对象去初始化另一个对象的时候

2）当函数的参数是类的对象时，就是值传递的时候，如果是引用传递则不会调用

3）当函数的返回值是类的对象或者引用的时候

26、 C++的四种强制转换

类型转化机制可以分为隐式类型转换和显示类型转化（强制类型转换）

(new-type) expression

new-type (expression)

隐式类型转换比较常见，在混合类型表达式中经常发生；四种强制类型转换操作符：

static_cast、dynamic_cast、const_cast、reinterpret_cast

1）static_cast：编译时期的静态类型检查

static_cast < type-id > (expression)

该运算符把expression转换成type-id类型，在编译时使用类型信息执行转换，在转换时执行必要的检测（指针越界、类型检查），其操作数相对是安全的

2）dynamic_cast：运行时的检查

用于在集成体系中进行安全的向下转换downcast，即基类指针/引用->派生类指针/引用

dynamic_cast是4个转换中唯一的RTTI操作符，提供运行时类型检查。

dynamic_cast如果不能转换返回NULL

dynamic_cast转为引用类型的时候转型失败会抛bad_cast

源类中必须要有虚函数，保证多态，才能使用dynamic_cast<source>(expression)

3）const_cast

去除const常量属性，使其可以修改 ; volatile属性的转换

4）reinterpret_cast

通常为了将一种数据类型转换成另一种数据类型

27、调试程序的方法

windows下直接使用vs的debug功能

linux下直接使用gdb，我们可以在其过程中给程序添加断点，监视等辅助手段，监控其行为是否与我们设计相符

28、extern“C”作用

extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言的进行编译，而不是C++的。

29、typdef和define区别

#define是预处理命令，在预处理是执行简单的替换，不做正确性的检查

typedef是在编译时处理的，它是在自己的作用域内给已经存在的类型一个别名

typedef (int*) pINT;

#define pINT2 int*

效果相同？实则不同！实践中见差别：pINT a,b;的效果同int *a; int *b;表示定义了两个整型指针变量。而pINT2 a,b;的效果同int *a, b;表示定义了一个整型指针变量a和整型变量b。

30、引用作为函数参数以及返回值的好处

对比值传递，引用传参的好处：

1）在函数内部可以对此参数进行修改

2）提高函数调用和运行的效率（所以没有了传值和生成副本的时间和空间消耗）

值传递：

形参是实参的拷贝，改变形参的值并不会影响外部实参的值。从被调用函数的角度来说，值传递是单向的（实参->形参），参数的值只能传入，

不能传出。当函数内部需要修改参数，并且不希望这个改变影响调用者时，采用值传递。

指针传递：

形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作

引用传递：

形参相当于是实参的“别名”，对形参的操作其实就是对实参的操作，在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

用引用作为返回值最大的好处就是在内存中不产生被返回值的副本。

但是有以下的限制：

1）不能返回局部变量的引用。因为函数返回以后局部变量就会被销毁

2）不能返回函数内部new分配的内存的引用。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一 个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak

3）可以返回类成员的引用，但是最好是const。因为如果其他对象可以获得该属性的非常量的引用，那么对该属性的单纯赋值就会破坏业务规则的完整性。

31、纯虚函数

纯虚函数是只有声明没有实现的虚函数，是对子类的约束，是接口继承

包含纯虚函数的类是抽象类，它不能被实例化，只有实现了这个纯虚函数的子类才能生成对象

32、什么是野指针

野指针不是NULL指针，是未初始化或者未清零的指针，它指向的内存地址不是程序员所期望的，可能指向了受限的内存。

成因：

1）指针变量没有被初始化

2）指针指向的内存被释放了，但是指针没有置NULL

3）指针超过了变量了的作用范围，比如b[10]，指针b+11

33、线程安全和线程不安全

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可以使用，不会出现数据不一致或者数据污染。

线程不安全就是不提供数据访问保护，有可能多个线程先后更改数据所得到的数据就是脏数据。

34、C++中内存泄漏的几种情况

内存泄漏是指动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

1）类的构造函数和析构函数中new和delete没有配套

2）在释放对象数组时没有使用delete[]，使用了delete

3）没有将基类的析构函数定义为虚函数，当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确释放，因此造成内存泄露

4）没有正确的清楚嵌套的对象指针

35、栈溢出的原因以及解决方法

栈溢出是指函数中的局部变量造成的溢出（注：函数中形参和函数中的局部变量存放在栈上）

栈的大小通常是1M-2M,所以栈溢出包含两种情况，一是分配的的大小超过栈的最大值，二是分配的大小没有超过最大值，但是接收的buf比原buf小。

1）函数调用层次过深,每调用一次,函数的参数、局部变量等信息就压一次栈

2）局部变量体积太大。

解决办法大致说来也有两种：

1> 增加栈内存的数目；如果是不超过栈大小但是分配值小的，就增大分配的大小

2> 使用堆内存；具体实现由很多种方法可以直接把数组定义改成指针,然后动态申请内存;也可以把局部变量变成全局变量,一个偷懒的办法是直接在定义前边加个static,呵呵,直接变成静态变量(实质就是全局变量)

36、C++标准库vector以及迭代器

每种容器类型都定义了自己的迭代器类型，每种容器都定义了一对命名为begin和end的函数，用于返回迭代器。

迭代器是容器的精髓，它提供了一种方法使得它能够按照顺序访问某个容器所含的各个元素，但无需暴露该容器的内部结构，它将容器和算法分开，让二者独立设计。

38、C++中vector和list的区别

vector和数组类似，拥有一段连续的内存空间。vector申请的是一段连续的内存，当插入新的元素内存不够时，通常以2倍重新申请更大的一块内存，将原来的元素拷贝过去，释放旧空间。因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为o(n)。

list是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以list的随机存取非常没有效率，时间复杂度为o(n);但由于链表的特点，能高效地进行插入和删除。

vector拥有一段连续的内存空间，能很好的支持随机存取，因此vector<int>::iterator支持“+”，“+=”，“<”等操作符。

list的内存空间可以是不连续，它不支持随机访问，因此list<int>::iterator则不支持“+”、“+=”、“<”等

vector<int>::iterator和list<int>::iterator都重载了“++”运算符。

总之，如果需要高效的随机存取，而不在乎插入和删除的效率，使用vector;

如果需要大量的插入和删除，而不关心随机存取，则应使用list。

39、C++中的基本数据类型及派生类型

1）整型 int

2）浮点型 单精度float，双精度double

3）字符型 char

4）逻辑型 bool

5）控制型 void

基本类型的字长及其取值范围可以放大和缩小，改变后的类型就叫做基本类型的派生类型。派生类型声明符由基本类型关键字char、int、float、double前面加上类型修饰符组成。

类型修饰符包括：

>short 短类型，缩短字长

>long 长类型，加长字长

>signed 有符号类型，取值范围包括正负值

>unsigned 无符号类型，取值范围只包括正值

40、友元函数和友元类

友元提供了不同类的成员函数之间、类的成员函数和一般函数之间进行数据共享的机制。

通过友元，另一个类中的成员函数可以访问类中的私有成员和保护成员。

友元的正确使用能提高程序的运行效率，但同时也破坏了类的封装性和数据的隐藏性，导致程序可维护性变差。

1）友元函数

友元函数是可以访问类的私有成员的非成员函数。它是定义在类外的普通函数，不属于任何类，但是需要在类的定义中加以声明。

friend 类型 函数名(形式参数);

一个函数可以是多个类的友元函数，只需要在各个类中分别声明。

2）友元类

友元类的所有成员函数都是另一个类的友元函数，都可以访问另一个类中的隐藏信息（包括私有成员和保护成员）。

friend class 类名;

使用友元类时注意：

(1) 友元关系不能被继承。

(2) 友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。

(3) 友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明

41、c++函数库<algorithm>中一些实用的函数

1. __gcd(x, y)

求两个数的最大公约数，如__gcd(6, 8)就返回2。

2. reverse(a + 1, a + n + 1)

将数组中的元素反转。a 是数组名，n是长度，跟 sort 的用法一样。值得一提的是，对于字符型数组也同样适用。

3. unique(a + 1, a + n + 1)

去重函数。跟sort的用法一样。不过他返回的值是最后一个数的地址，所以要得到新的数组长度应该这么写： _n = unique(a + 1, a + n + 1) - a - 1.

4.lower_bound(a + 1, a + n + 1, x); upper_bound(a + 1, a + n + 1, x)

lower_bound是查找数组中第一个小于等于x的数，返回该地址，同理也是 pos = lower_bound(a + 1, a + n + 1, x) - a

upper_bound是查找第一个大于x的数，用法和lower_bound一样

复杂度是二分的复杂度，O(logn)。（其实就是代替了手写二分）

5.fill(a + 1, a + n + 1, x)

例如

int数组：fill(arr, arr + n, 要填入的内容);

vector也可以：fill(v.begin(), v.end(), 要填入的内容);

fill(vector.begin(), cnt, val); // 从当前起始点开始，将之后的cnt个元素赋值为val。

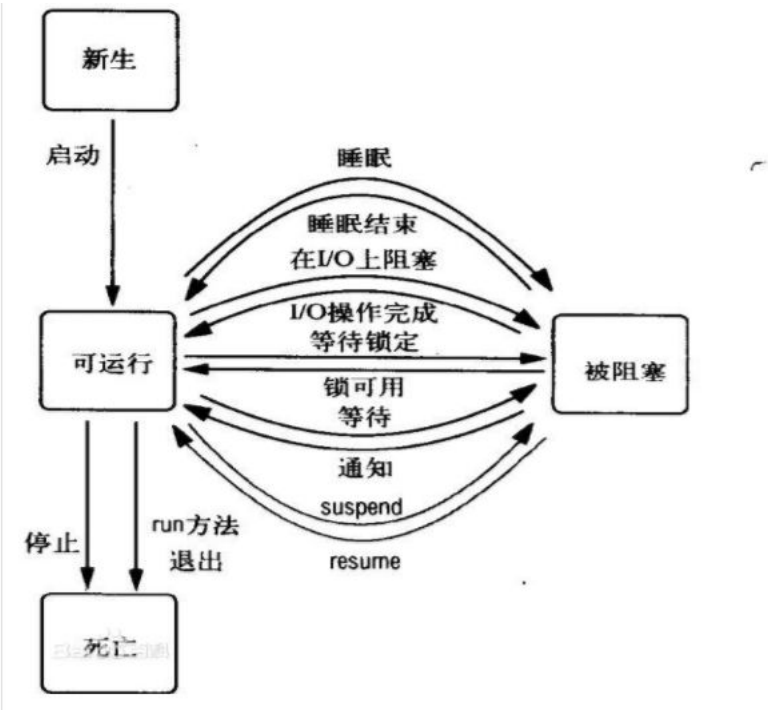
memset(arr, val, cnt); // 在头文件<cstring>里。

将数组a中的每一个元素都赋成x，跟memset的区别是，memset函数按照字节填充，所以一般memset只能用来填充char型数组，（因为只有char型占一个字节）如果填充int型数组，除了0和-1，其他的不能。

42、线程的基本概念、线程的基本状态及状态之间的关系?

线程，有时称为轻量级进程，是CPU使用的基本单元；它由线程ID、程序计数器、寄存器集合和堆栈组成。它与属于同一进程的其他线程共享其代码段、数据段和其他操作系统资源（如打开文件和信号）。

线程有四种状态：新生状态、可运行状态、被阻塞状态、死亡状态。状态之间的转换如下图所示：



43、线程与进程的区别?

1、 线程是进程的一部分，所以线程有的时候被称为是轻权进程或者轻量级进程。 2、 一个没有线程的进程是可以被看作单线程的，如果一个进程内拥有多个进程，进程的执行过程不是一条线（线程）的，而是多条线（线程）共同完成的。 3、 系统在运行的时候会为每个进程分配不同的内存区域，但是不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源。那就是说，出了CPU之外（线程在运行的时候要占用CPU资源），计算机内部的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源。 4、 与进程的控制表PCB相似，线程也有自己的控制表TCB，但是TCB中所保存的线程状态比PCB表中少多了。

5、 进程是系统所有资源分配时候的一个基本单位，拥有一个完整的虚拟空间地址，并不依赖线程而独立存在。

44、 C++多线程有几种实现方法，都是什么?

```
#include <thread>
#include <condition_variable>
#include <mutex>
```

1 std::thread

关键点

a. C++ 11中创建线程非常简单，使用std::thread类就可以，thread类定义于thread头文件，构造thread对象时传入一个可调用对象作为参数（如果可调用对象有参数，把参数同时传入），这样构造完成后，新的线程马上被创建，同时执行该可调用对象；

b. 用std::thread默认的构造函数构造的对象不关联任何线程；判断一个thread对象是否关联某个线程，使用joinable()接口，如果返回true，表明该对象关联着某个线程（即使该线程已经执行结束）；

c. "joinable"的对象析构前，必须调用join()接口等待线程结束，或者调用detach()接口解除与线程的关联，否则会抛异常；

d. 正在执行的线程从关联的对象detach后会自主执行直至结束，对应的对象变成不关联任何线程的对象，joinable()将返回false

e. std::thread没有拷贝构造函数和拷贝赋值操作符，因此不支持复制操作（但是可以move），也就是说，没有两个 std::thread对象会表示同一执行线程；

f. 容易知道，如下几种情况下，std::thread对象是不关联任何线程的（对这种对象调用join或detach接口会抛异常）：

默认构造的thread对象；

被移动后的thread对象；

detach 或 join 后的thread对象；

2 std::mutex (轻松实现互斥)

常做多线程编程的人一定对mutex（互斥）非常熟悉，C++ 11当然也支持mutex，通过mutex可以方便的对临界区域加锁，std::mutex类定义于mutex头文件，是用于保护共享数据避免从多个线程同时访问的同步原语。它提供了lock，try_lock,unlock等几个接口，功能如下：

调用方线程从成功调用lock()或try_lock()开始，到unlock()为止占有mutex对象

线程占有mutex时，所有其他线程若试图要求mutex的所有权，则将阻塞（对于 lock 的调用）或收到false返回值（对于 try_lock ）；

调用方线程在调用 lock 或 try_lock 前必须不占有mutex。

mutex和thread一样，不可复制（拷贝构造函数和拷贝赋值操作符都被删除），而且，mutex也不可移动；

备注

a.操作系统提供mutex可以设置属性，C++11根据mutext的属性提供四种的互斥量，分别是

std::mutex，最常用，普遍的互斥量（默认属性），

std::recursive_mutex ，允许同一线程使用recursive_mutext多次加锁，然后使用相同次数的解锁操作解锁。mutex多次加锁会造成死锁

std::timed_mutex，在mutex上增加了时间的属性。增加了两个成员函数try_lock_for()，try_lock_until()，分别接收一个时间范围，再给定的时间内如果互斥量被锁主了，线程阻塞，超过时间，返回false。

std::recursive_timed_mutex，增加递归和时间属性

b.mutex成员函数加锁解锁

lock()，互斥量加锁，如果互斥量已被加锁，线程阻塞

bool try_lock()，尝试加锁，如果互斥量未被加锁，则执行加锁操作，返回true；如果互斥量已被加锁，返回false，线程不阻塞。

void unlock()，解锁互斥量

c. mutex RAII式的加锁解锁

std::lock_guard，管理mutex的类。对象构建时传入mutex，会自动对mutex加入，直到离开类的作用域，析构时完成解锁。RAII式的栈对象能保证在异常情形下mutex可以在lock_guard对象析构被解锁。

std::unique_lock 与 lock_guard功能类似，但是比lock_guard的功能更强大。比如std::unique_lock维护了互斥量的状态，可通过bool owns_lock()访问，当locked时返回true，否则返回false

3 std::lock_guard （有作用域的mutex，让 程序更稳定，防止死锁）

很容易想到，mutex的lock和unlock必须成对调用，lock之后忘记调用unlock将是非常严重的错误，再次lock时会造成死锁。有时候一段程序中会有各种出口，如return，continue，break等等语句，在每个出口前记得unlock已经加锁的mutex是有一定负担的，而假如程序段中有抛异常的情况，就更为隐蔽棘手，C++ 11提供了更好的解决方案，对的，RAII，本系列文章多次提到RAII，想必大家应该不陌生。

类模板std::lock_guard是mutex封装器，通过便利的RAII机制在其作用域内占有mutex。

创建lock_guard对象时，它试图接收给定mutex的所有权。当程序流程离开创建lock_guard对象的作用域时，lock_guard对象被自动销毁并释放mutex，lock_guard类也是不可复制的。

一般，需要加锁的代码段，我们用{}括起来形成一个作用域，括号的开端创建lock_guard对象，把mutex对象作为参数传入lock_guard的构造函数即可，比如上面的例子加锁的部分，我们可以改写如下：