宇宙最全系列 | C++面试题v1.0

03 了解析构函数吗？需要注意些什么？

04 指针和对象有何区别。

05 ❤虚函数的作用。什么是多态？

06 ❤为什么父类析构函数必须是虚函数？为什么C++默认析构函数不是虚函数。

07 ❤带有继承类的构造和析构顺序分别是怎么样的。

08 虚函数和静态函数的区别？

09 ❤纯虚函数。

10 抽象类（接口）是什么？

11 ❤虚表和虚指针的原理？

12 ❤构造函数和析构函数可以是虚函数吗？

13 ❤说一下重载与重写。

14 ❤C++中拷贝/赋值函数的形参能否进行值传递。

15 拷贝构造函数、赋值构造函数的定义？

16 ❤请回答什么叫左值引用，什么叫右值引用。

17 什么是将亡值，什么是纯右值。

18 ❤移动语义与完美转发了解吗。

19 什么是引用折叠?forward函数的原理。

20 ❤什么是移动构造和移动赋值？

21 什么是浅拷贝和深拷贝？

22 你介绍一下C++类访问权限。

23 你了解友元吗？

24 讲一下struct和class有什么区别？

25 类中可以定义引用数据成员吗？

STL

01 ❤STL由哪些组件组成。

02 ❤介绍一下C++ 容器。

03 ❤map和unordered_map有何区别？

04 map和multimap的区别？

05 ❤讲一下STL分配器。内部原理是什么？

06 ❤STL的两级分配器了解吗？

07 你刚才提到了C++的内存池技术，能介绍一下吗。

08 请你说一下STL迭代器删除元素是怎么做的。

作者：迹寒

作为一个后端人🤖，是无论如何要对C++有一定了解滴(TㅇT)。很多同学都对C++有一定的抵触情绪，因为C++知识点繁杂全面，深度与广度俱在，准备面试需要很长的时间，这可以理解！包括我也是这样的。

So，本篇的主要目的是梳理知识脉络，挑选最精华的面试题，以飨读者，事半功倍！

准备面试一定要有**侧重点**，标为❤属于高频考点，需要**理解**记忆。

赛道已铺好，只待尔努力。加油，你就是offer收割机！

基础

基础部分，考察较多。

01 ❤ 说一下用户空间内存分区?

栈区：向下生长，存储局部变量，系统自动管理

内存映射区(file mapping segment)：向下生长，存储文件映射（包括动态链接库）以及匿名映射

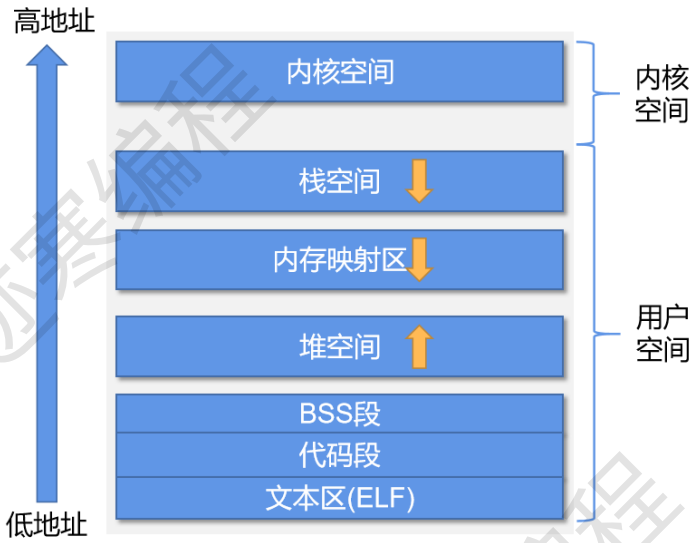
堆区：向上生长，存储动态分配对象，需要手动管理

.BSS段：存储未初始化的全局变量

代码段(.data)：存储已初始化的全局变量

文本区(.text)：存储程序代码和常量

注意：全局静态变量、常量、代码均在编译时分配，位于运行期。



02 说一下static关键字的作用?

全局静态变量：位于静态存储区，程序运行期间一直存在，对外部文件不可见。

局部静态变量：位于静态存储区，在局部作用域可以访问，离开局部作用域之后static变量仍存在，但无法访问。

静态函数：即在函数定义前加static，函数默认情况下为extern，即可导出的。加了static就不能为外部类访问。注意不要在头文件声明static函数，因为static只对本文件有效。

类的静态成员：可以实现多个不同的类实例之间的数据共享，且不破坏隐藏规则，不需要类名就可以访问。类的静态存储变量是可以修改的。

类的静态函数：不能调用非静态成员，只可以通过对象名调用<对象名>::<静态成员函数>

static 不需要初始化，默认为0值。

03 ❤讲一讲C++里面四种强制类型转换?

static_cast, const_cast, reinterpret_cast, dynamic_cast

static_cast：用于各种隐式转换，比如void*转ptr*,例如：

```C++
double a = 1.0f;
int b = static_cast<double>(a);
double a = 1.999;
void * vptr = & a;
double * dptr = static_cast<double*>(vptr);
cout<<*dptr<<endl;//输出1.999
```

const_cast: 用来移除变量的const或volatile限定符。

```C++
const int constant = 21;
const int* const_p = &constant; // *const_p = 7
int* modifier = const_cast<int*>(const_p); // *modifier = 7
```

dynamic_cast：安全的向下进行类型转换。只能用于含有虚函数的类，只能转指针或引用。

reinterpret_cast：允许将任何指针转换为任何其他指针类型，并不安全。

向上转换：从子类到父类（基类）；

向下转换：相反。

参考：强制转换运算符

04 static_cast和interpret_cast它们的区别知道吗?

static_cast 指向和来自 void* 的指针保留地址。 也就是说，在下面，a、b 和 c 都指向同一个地址：

```C++
int* a = new int();
void* b = static_cast<void*>(a);
int* c = static_cast<int*>(b);
```

reinterpret_cast 保证只有当指针转换为**不同**的类型，然后将其 reinterpret_cast 恢复为原始类型，您将获得原始值。 所以在下面：

```cpp
int* a = new int();
void* b = reinterpret_cast<void*>(a);
int* c = reinterpret_cast<int*>(b);
```

a 和 c 包含相同的值，但未指定 b 的值。（实际上，它通常包含与 a 和 c 相同的地址，但标准中没有指定，并且在具有更复杂内存系统的机器上可能不是这样。）

总结：

- 对于 void* 的转换，应该首选 static_cast。
- 对于模糊类型的转换，应该使用reinterpret_cast。

05 ❤说一下C++指针和引用的区别?

1. 指针有自己的内存地址，占四个字节（32位系统），而引用只是一个别名，没有专门的内存地址。
2. 指针可以被初始化为指向nullptr，而引用必须指向一个已有的对象。
3. 作为参数传递是，指针需要解引用(*)，而直接修改引用会改变原对象。
4. 指针可以多级，而引用最多一级。
5. 如果返回动态内存分配对象，必须用指针，否则可能引起内存泄漏。

06 volatile关键字有什么作用?

C++中的volatile和const对应。表示**变量可以被编译器未知的因素所更改**，比如操作系统，硬件或者其它线程。遇到volatile，编译器就不再进行优化，从而提供对特殊地址的稳定访问。

参考资料：C/C++ 中 volatile 关键字详解

07 ❤ C++的智能指针用过吗？怎么样?

C++的智能指针均位于<memory>库内，有四种：**shared_ptr、unique_ptr、weak_ptr、auto_ptr**。

1. shared_ptr

共享式指针，只有共享的最后一个引用释放资源销毁。

原理：无非是利用一个计数器，当发生使用赋值拷贝构造函数，运算符重载，作为函数返回值，或者作为一个参数传递给另外一个参数，计数+1，当shared_ptr赋新值或者销毁，计数-1.直到计数为0，调用析构函数释放对象。

```C++
std::shared_ptr<T> sptr = std::make_shared<T>(...);  // 初始化方式1
std::shared_ptr<T> sptr(new T(...));  // 初始化方式2
```

2. unique_ptr

```C++
std::unique_ptr<T> uptr = std::make_unique<T>(...); // 方式1
std::unique_ptr<T> uptr(new T(...)); // 方式2
```

独占式的指针，离开 unique_ptr 对象的作用域时，会自动释放资源。

3. weak_ptr

与shared_ptr一起使用，作为资源的观察者，不影响对象的引用计数。

4. auto_ptr

已被弃用。

08 ❤unique_ptr是如何实现独占式指针?

由于指针或引用在离开作用域是不会调用析构函数的，但对象在离开作用域会调用析构函数。unique_ptr本质是一个类，将**复制构造函数**和**赋值构造函数**声明为delete就可以实现独占式，只允许移动构造和移动赋值。

具体的实现可以参考类-unique_ptr实现原理

09 ❤shared_ptr是如何实现共享式指针?

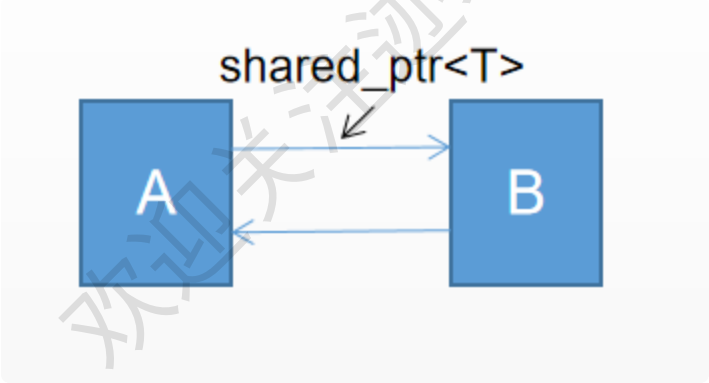
shared_ptr通过引用计数，使得多个shared_ptr对象共享一份资源。

如果对象被引用，则计数加1，如果对象被销毁，则计数减1。如果计数为0，表示对象没有被销毁，可以释放该资源。shared_ptr的缺点是存在**循环引用**的问题。

实现可以参考：面试题：简单实现一个shared_ptr智能指针 – 腾讯云开发者社区

10 ❤什么是shared_ptr的循环引用问题? 为什么要有weak_ptr?

一个最简单的情况是，某对象存在一个shared_ptr类型的指针ptr，A的ptr指向B，B的ptr指向A。两个智能指针对象指向A，B，再加上他们的ptr分别指向B，A，所以引用计数均为2，造成了循环引用，谁也不会被释放。



循环引用示意图

循环引用的代码 C++ 复制代码

```cpp
struct ListNode
{
    int _data;
    shared_ptr<ListNode> ptr;
        ListNode(int data):_data(data){}
    ~ListNode(){ cout << "~ListNode()" << endl; }
};
int main()
{
    shared_ptr<ListNode> node1(new ListNode(1));
    shared_ptr<ListNode> node2(new ListNode(2));
    cout << node1.use_count() << endl;  // 1
    cout << node2.use_count() << endl;  // 1
    node1->ptr = node2;
    node2->ptr = node1;
    cout << node1.use_count() << endl;  // 2
    cout << node2.use_count() << endl;  // 2
    return 0;
}
```

一般有三种解决方法：

1. 当剩下最后一个引用时，需要手动打破循环引用释放对象；
2. 当A的生存周期超过B的生存周期，B改为一个普通指针指向A；
3. 将共享指针改为弱指针weak_ptr

一般采用第三者办法，原理是弱指针的指针_prev和_next不会增加node1和node2的引用计数。

```C++
struct ListNode
{
    int _data;
    weak_ptr<ListNode> ptr;
        ListNode(int data):_data(data){}
    ~ListNode(){ cout << "~ListNode()" << endl; }
};
int main()
{
    shared_ptr<ListNode> node1(new ListNode());
    shared_ptr<ListNode> node2(new ListNode);
    cout << node1.use_count() << endl;  // 1
    cout << node2.use_count() << endl;  // 1
    node1->_next = node2;
    node2->_prev = node1;
    cout << node1.use_count() << endl;  // 1
    cout << node2.use_count() << endl;  // 1
        // ~ListNode()
    return 0;
}
```

11 数组与指针的区别？指针数组和数组指针？

数组存放一组元素，而指针指向某一个对象。从底层实现上看，数组也是由base指针和各维度长度等组成，数组元素存放在连续地址上。

指针数组是保存指针的数组，比如int* a[10]，而数组指针是 指向数组的指针，比如：

```C++
int var[10];
int *ptr = var;
int *ptr = &var[0];//与上面等价
```

在C++中，数组名代表数组中第一个元素（即序号为0的元素）的地址。如果是二维数组，则可以通过*(*(arr+i)+j)来访问arr[i][j]；

12 你知道函数指针吗？讲一讲。

函数指针是指指向函数的指针，在早期C的项目经常能看到。这里是指向函数的**入口地址**。作用是调用函数作为入口参数，比如回调函数：

C++ | 复制代码

```cpp
int foo(){return -1;}

int (*ptrfoo) () = foo; // 不要写成foo()

// 作为回调函数
void func(int (*foo)());
```

有入口参数的情况:

C++ | 复制代码

```cpp
int foo(int x);
void func(int (*foo)(int)){

}
```

13 什么是注册函数？什么是回调函数?

回调函数无非是对函数指针的应用，用函数指针来**调用一个函数**，而**注册函数**则是将**函数指针**作为参数传进去，便于其它函数调用。

14 ❤C++从源文件到可执行文件需要经历哪些步骤?

首先是**预处理阶段(preprocessing)->编译阶段(compilation)->汇编阶段(assembly)->链接阶段(linking)**。

预处理阶段，编译器对文件包含关系进行检查（头文件和宏），将其作相应替换，生成.i文件；

编译阶段，将预处理的生成文件转化为汇编文件.s;

汇编阶段，将汇编文件见转化为二进制机器码，对应后缀是.o(Linux), .obj(Windows);

链接阶段，将多个目标文件及所需要的库链接成可执行文件，.out(Linux), .exe(Windows);

语法

基本的语法。

01 ❤以下四行代码中"123"是否可以修改?

C++ | 复制代码

```
1  const char* a = "123";
2  char *b = "123";
3  const char c[] = "123";
4  char d[] = "123";
```

第1，2行，"123"位于常量区，加不加const效果一样，都无法修改。而第三行，"123"本来在栈上，但是由于const关键字编译器可能将其优化到常量区，第四行：“123”位于栈区。总结：只有第四行可以修改。

02 C++是怎么定义常量的?

C++有两个关键字const和constexpr(C++11)可以定义常量，常量必须被初始化。

对于局部常量，通常位于栈区，而对于全局常量，编译器一般不分配内存，放在符号表以提高效率。字面量一般位于常量区。

03 ❤const和constexpr有什么区别?

传统const的问题在于“双重语义”，既有“只读”的含义，又有“常量”（不可改变）的含义，而**constexpr严格定义了常量**。

只读一定不可改变吗？这还真不一定！比如下面这个例子：

C++ | 复制代码

```cpp
int main()
{
    int a = 10;
    const int & con_b = a;
    cout << con_b << endl; // 10
    a = 20;
    cout << con_b << endl; // 20
}
```

可以看到，程序中用 const 修饰了 con_b 变量，表示该变量“只读”，即无法通过变量自身去修改自己的值。但这并不意味着 con_b 的值不能借助其它变量间接改变，通过改变 a 的值就可以使 con_b 的值发生变化。

参考资料：C++11 constexpr和const的区别详解

04 ❤const放在类型/函数前和后有区别吗?

C++ | 复制代码

```cpp
int b = 1;
const int *a = &b;
int const *a = &b;
int* const a = &b;
const int* const a = &b;
```

C++规定const在类前和类后是一样的。并且按照“**从右向左读**”进行理解。2，3行相同。

2/3：一个int*型指向常量的指针；该指针可以指向其它的变量但无法修改它们的值。

4：一个常量的指向int*型的指针；它无法指向别的地址。

5：既不能指向其它变量的地址，也不能修改值。

对于函数而言

```cpp
const int func(){};
int const func(){};
void func() const{};
```

1和2作用相同，表示函数返回const int类型；

3通常是在类中，表示该函数不修改成员变量。

```cpp
class A{
    int a;
    const int b;
    public:
    void test(int x) const{
        this->a = 1; // 报错，表达式必须是可修改的左值
    };
};
```

对类而言

```cpp
class A{
    public:
    void test1() const;
    void test2();
};

const A classA;
classA.test1();//正确
classA.test2();//错误，对象含有与成员 函数 "A::test2" 不兼容的类型限定符 -- 对象类型是:  const A
```

该变量只能调用const成员函数。

05 C++如何处理函数返回值?

生成一个临时变量，将它的引用作为函数输入参数。

06 如何在C++引用C头文件?

采用extern关键字。如果定义了宏 `#define __cpluscplus` 就表示使用了C++的编译器

C++ | 复制代码

```C++
#ifdef _cplusplus
    extern "C"{
#endif _cplusplus
//头文件内容
...

#ifdef _cplusplus
    }
#endif _cplusplus
```

07 `#pragma once` 这句话有什么用?

`#pragma once` 一般由编译器提供保证：**同一个文件不会被包含多次**。它的作用和 `#ifndef` 定义头文件相同。操作简单，效率较高，缺点是它只能针对整个文件。

C++ | 复制代码

```C++
// 方式一定义头文件
#ifndef _CODE_BLOCK
#define _CODE_BLOCK

// code

#endif// _CODE_BLOCK
//方式二定义头文件
#pragma once
```

08 形参和实参有什么不同.

形参，是定义函数时的参数，比如void func(int x)这里的x就是形参。

实参，调用函数实际填入的参数，比如func(1)。

09 inline关键字有什么作用。

inline内联函数，它可以**避免相同函数重写多次的麻烦**，它**减少了运行时间但是增加了空间开销**。

具体而言，当编译器遇到内联函数，它不会编译成指令，而是整体的插入到调用处，增加代码可复用性。

使用inline关键字只是用户希望它成为内联函数，但如果此函数体太大，则编译器**不会把它当作内联函数**。

类内的成员函数，默认都为inline。一般小于等于10行的函数都可以加inline修饰。

面向对象

C++的一大特点是面向对象编程。

01 ❤面向对象编程的基本特性。

封装、继承和多态。

02 什么是基类，父类，超类和派生类?

基类就是父类，任何一个类都可以通过继承派生一个新类，称之为派生类。父类又称为“超类”。

03 了解析构函数吗？需要注意些什么?

析构函数和构造函数相对应，在对象生命周期结束，自动完成对象回收与销毁。用[~类名]表示，它**没有参数，返回值，也无法被重载**。

如果类中动态分配了空间，就需要在析构函数中释放指针。

04 指针和对象有何区别。

指针**指向**内存中存放的类对象(包括一些成员变量所赋的值). 在堆中赋值。

对象是利用类的**构造函数**在内存中分配一块内存(包括一些成员变量所赋的值). 用的是内存栈,是个局部的临时变量.

在应用时:

1.引用成员: 对象用" . "操作符; 指针用" –> "操作符.

2.生命期: 若是成员变量,则是类的析构函数来释放空间;若是函数中的临时变量,则作用域是该函数体内.而指针,则需利用delete 在相应的地方释放分配的内存块.

注意:用new ,一定要delete.. **如果要实现多态，或者离开作用域还要继续使用变量，只能用指针实现**。

参考链接：类里面对象和指针的区别

05 ❤虚函数的作用。什么是多态?

虚函数的主要作用是实现**多态**。子类继承父类函数后，如果需要重写其功能的，一般应该将父类函数声明为虚函数。

C++多态最大特点是**运行时虚表指针动态绑定函数地址**。虚表由编译器维护，每个类都生成虚表，有一定开销。

06 ❤为什么父类析构函数必须是虚函数？为什么C++默认析构函数不是虚函数。

通常将父类的析构函数设为虚函数。如果不将父类析构函数声明为虚函数，那么就不能实现**多态**（动态绑定），释放父类的指针的时候，子类对象得不到释放，就会造成**内存泄漏**的问题。

C++默认析构函数不是虚函数，是因为虚函数需要**虚函数表**和**虚表指针**，**会占用额外内存**。如果一个类没有子类，就没有必要将析构函数设为虚函数。

参考资料：为什么父类析构函数必须为虚函数。

07 ❤带有继承类的构造和析构顺序分别是怎么样的。

析构顺序一般是这样的：创建子类对象时，**先调用父类构造函数，再调用子类构造函数**。而子对象析构时则是先调用子类析构函数，再调用父类析构函数，**顺序刚好相反，先调用子类析构函数，再调用父类析构函数**。

08 虚函数和静态函数的区别?

区别：**静态函数在编译时就已经确定，而虚函数在运行时动态绑定**。虚函数是实现多态重要手段，在函数前加virtual关键字即可。

由于虚函数采用虚表，会增加额外内存开销。

09 ❤纯虚函数。

两者的区别在于纯函数尚未被实现，定义纯虚函数是为了实现一个接口。在基类中定义纯虚函数的方法是在函数原型后加 `=0`

```
virtual void function() = 0;
```

10 抽象类（接口）是什么?

抽象类（**接口**）是一种特殊的类，**不能定义对象，**需要满足以下条件：

- 类中没有定义任何的成员变量
- 所有的成员函数都是公有的
- 所有的成员函数都是纯虚函数

子类继承接口，需要实现接口的全部的方法。

11 ❤虚表和虚指针的原理?

这涉及到C++内存模型。虚表本质上是一个数组，存放着所有虚函数的指针。如果父类的虚函数没有被子类改写，那么子类虚函数表的指针就是父类对应的虚函数的指针；否则，虚表的指针是子类虚函数的指针。这个过程在程序运行过程中执行，被称为“**动态绑定**”；

12 ❤构造函数和析构函数可以是虚函数吗?

构造函数**不可以是虚函数**。析构函数**可以是虚函数**。

虚函数表是由编译器自动生成和维护的，virtual成员函数会被编译器放入虚函数表中，当存在虚函数时，每个对象都有一个指向虚函数的指针（vptr）。在定义子类对象时，vptr先指向父类的虚函数表，在父类构造完之后，子类的vptr才指向自己的虚函数表。**因此构造函数不可以是虚函数。**

与构造函数不同，vptr已经完成初始化，析构函数可以声明为虚函数，且类有继承时，析构函数常常必须为虚函数。

13 ❤说一下重载与重写。

重载（overload）是指重名的两个函数或方法，参数列表不同，这个时候编译器自动根据上下文判断最合适的函数。此外还有运算符重载，用以实现类的运算。

```cpp
class A
{
    void fun() {};
    void fun(int i) {};
    void fun(int i, int j) {};
};
```

重写（override）是指基类的虚函数，在子类更改了功能，这个叫重写。

```cpp
class A
{
public:
    virtual void fun()
    {
        cout << "A";
    }
};
class B :public A
{
public:
    virtual void fun()
    {
        cout << "B";
    }
};
```

14 ❤C++中拷贝/赋值函数的形参能否进行值传递。

不能。在默认情况下，编译器会自动生成一个**拷贝构造函数**和**赋值运算符**，用户可以用delete来不生成。

如果采用值传递，调用拷贝构造函数，先将实参传递给形参，这个传递又要调用拷贝函数，会导致**不断循环直到调用栈满**。

15 拷贝构造函数、赋值构造函数的定义?

拷贝构造函数是一种构造函数，和类同名，参数通过类的对象引用传递，无返回值。

赋值构造函数是通过重载=运算符实现的，也是通过类的对象引用传递。

C++ | 复制代码

```cpp
//不生成拷贝构造函数的例子
class Person {
public:
    Person(const Person& p) = delete;
    Person& operator=(const Person& p) = delete;
private:
    int age;
    string name;
};
//生成拷贝构造函数
class A {
    public:
    //拷贝构造函数
    explicit A(A& a) : x(a.x)
    {
        cout << "Copy Constructor" << endl;
    }
    //赋值函数
    A& operator=(A& a)
    {
        x = a.x;
        cout << "Copy Assignment operator" << endl;
        return *this;
    }
    private:
    int x;
}
```

16 ❤请回答什么叫左值引用，什么叫右值引用。

右值引用是C++11引入的，与之对应C++98中的引用统称为左引用。左引用的一个最大问题就是，它不能对不能取地址的量（比如字面量常量）取引用。比如int &a = 1;就不可以。

为此专门定义了左值和右值，**能取地址的都是左值，反之是右值**。通过右值引用，可以增长变量的生命周期,避免分配新的内存空间.

并用&&来表示右值引用，这样就可以int &&a = 1;并用&来表示左值引用。

总结：左值引用只能绑定左值；右值引用只能绑右值，但常量左值引用可以绑字面量，比如const int &b = 10;已命名的右值引用，编译器会认为是一个左值；临时对象是左值。

17 什么是将亡值，什么是纯右值。

所谓纯右值就是**临时变量或者字面值**，将亡值是C++11新定义的**将要被“移动”的变量，**比如move返回的变量。

18 ❤移动语义与完美转发了解吗。

移动语义(move semantic)：**某对象持有的资源或内容转移给另一个对象**。为了保证移动语义, 必须记得用std::move 转化左值对象为右值,以避免调用复制构造函数.

C++ | 复制代码

```
vector<int> a{1,2,3};
vector<int> b = std::move(a);//我们不希望为了b拷贝新的内存空间，采用移动语义C++
// a的元素变为{}，b的元素变为{1,2,3}
```

完美转发(perfect forwarding): **为了解决引用折叠问题,必须写一个任意参数的函数模板,并转发到其他函数**. 为了保证完美转发,必须使用std::forward, 我们希望左值转发之后还是左值,右值转发后还是右值.

19 什么是引用折叠?forward函数的原理。

引用折叠就是，如果间接创建一个引用的引用，那么这些引用就会折叠。规则:

```C++
&& + &&->&& : 右值的右值引用是右值
&& + &->& : 右值的左值引用是左值
& + &&->& : 左值的右值引用是左值
& + &->& : 左值的左值引用是左值
```

为此引入了forward函数:

```C++
// 精简了标准库的代码，在细节上可能不完全正确，但是足以让我们了解转发函数 forward 的了
template<typename T>
T&& forward(T &param){
    return static_cast<T&&>(param);
}
```

1.传入 PrintType 实参是右值类型： 根据以上的分析，可以知道T将被推导为值类型，也就是不带有引用属性，假设为 int 。那么，将T = int 带入forward。

```C++
int&& forward(int &param){
    return static_cast<int&&>(param);
}
```

param在forward内被强制类型转换为 int &&，还是右值引用。最终保持了实参的右值属性，转发正确。

2.传入 PrintType 实参是左值类型:

根据以上的分析，可以知道T将被推导为左值引用类型，假设为int&。那么，将T = int& 带入forward。

```C++
int& && forward(int& &param){
    return static_cast<int& &&>(param);
}
```

引用折叠一下就是 int &类型，转发正确。

20 ❤什么是移动构造和移动赋值?

移动构造函数能直接使用临时对象已经申请的资源，它以**右值引用**为参数，拷贝以左值。

由于临时对象是右值，这里就需要使用一个move函数，它的作用的将左值强制转换为右值。

移动赋值是在赋值运算符重载的基础上，将对象右值引用作为形参进行拷贝或者赋值，从而避免创建新对象。

下面的例子展示了拷贝构造函数、赋值运算符重载、移动拷贝和移动赋值运算符重载，请仔细区别：

C++ | 复制代码

```cpp
class A{
    public:
    //拷贝构造函数
    A(A& a) : x(a.x)
    {
        cout << "Copy Constructor" << endl;
    }
    //赋值运算符
    A& operator=(A& a)
    {
        x = a.x;
        cout << "Copy Assignment operator" << endl;
        return *this;
    }
    //移动拷贝
    A(A&& a) : x(a.x)
    {
        cout << "Move Constructor" << endl;
    }
    //移动赋值
    A& operator=(A&& a)
    {
        x = a.x;
        cout << "Move Assignment operator" << endl;
        return *this;
    }
    private:
    int x;
}
```

21 什么是浅拷贝和深拷贝?

浅拷贝就是增加了一个新指针指向原来的地址，那么改变原有对象也会改变新对象。而深拷贝则是开辟了新的内存空间，并增加一个指向该空间的指针。

22 你介绍一下C++类访问权限。

C++有三个关键字public, protected, private.

public: 完全公开，任何类都可以访问。

protected，当前**类和子类**可以访问。

private，仅当前类可以访问。

23 你了解友元吗?

类的友元函数以关键字friend修饰，它可以让外部函数访问类成员。具体用法是在类内部用friend声明外部函数。也可以声明友元类。

C++ | 复制代码

```cpp
class Box
{
    double width;
public:
    friend void printWidth(Box box);
    friend class BigBox;
    void setWidth(double wid);
};

class BigBox
{
public :
    void Print(int width, Box &box)
    {
        // BigBox是Box的友元类，它可以直接访问Box类的任何成员
        box.setWidth(width);
        cout << "Width of box : " << box.width << endl;
    }
};

// 请注意：printWidth() 不是任何类的成员函数
void printWidth(Box box)
{
    /* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 */
    cout << "Width of box : " << box.width << endl;
}
```

24 讲一下struct和class有什么区别?

struct默认的访问权限是public而class默认的访问权限是private，除此以外都一样。

25 类中可以定义引用数据成员吗?

可以，但必须使用外部引用变量初始化，也就是说构造函数的形参必须是引用形式

```cpp
class A
{
public:
    A(int &target) :a(target)
    {
            ...
    }
private:
    int &a;
};
```

STL

C++的官方库。

01 ❤STL由哪些组件组成。

STL由6个组件和13个头文件组成。

容器：一些封装数据结构的模板类，比如vector,list等。

算法：它们被设计为一个个模板函数，大部分位于<algorithm> ，小部分位于<numeric>。

函数对象：如果一个类将()重载为成员函数，那么这个类称为函数对象类，类的对象称为仿函数。

迭代器：容器对数据的读写是通过迭代器完成的，它充当容器和算法之间的胶合剂。

适配器：将一个类的接口设计成用户指定形式。

内存分配器：为容器类模板提供内存分配和释放功能。

13个头文件：

<vector> <map> <list> <queue> <stack> <deque> <set> <iterator> <functional> <algorithm> <numeric> <memory> <utility>

02 ❤介绍一下C++ 容器。

C++容器有三种，**顺序容器**，**关联容器**和**容器适配器**。

顺序容器是各元素有顺序关系的顺序表，比如vector,deque,list；

而关联容器分为**有序关联容器map, multimap, set, multiset**是非线性表，底层实现是二叉树;**无序关联容器unordered_map/unordered_multimap和unordered_set/unordered_multiset**。

容器适配器：可以理解为**容器的模板**，比如stack, queue 和priority_queue。

03 ❤map和unordered_map有何区别?

1. map的底层实现都是红黑树，插入查询删除的时间复杂度是O(logn)，unordered_map底层实现是哈希表，里面元素是乱序排序的，元素插入，删除，搜索的时间复杂度都是O(1)；

2. map内部元素默认按照key 进行排序，所以支持upper_bound和lower_bound 这样的二分查找算法进行**范围查询**。

04 map和multimap的区别?

multimap 保存多个多个相同的key，而map不可以。

multimap是基于键的查找/插入，而不是值。因此，在插入元素时，重复键上的值是否相同并不起作用。比如一个multimap插入('A',1),('A',1),('A',3)之后，multimap的大小为3。因为没有值的比较，multimap被使用的并不多。

05 ❤讲一下STL分配器。内部原理是什么?

STL分配器用于容器内存管理。主要职责是：new申请空间；delete释放空间。

为了精密分工，分配器要将两阶段分开：1. **内存配置先由allocate()（operator new()）完成，然后对象构造由构造函数负责**；2. **对象析构先由析构函数完成，内存释放由deallocate()（operator delete()）完成**。注意顺序不要弄错。

参考资料：C++STL学习笔记(4) 分配器(Allocator)

06 ❤STL的两级分配器了解吗?

为了提升内存管理效率，STL采用两级分配器：**对于大于128B的内存申请，采用第一级分配器**，用malloc(), realloc(), free()进行空间分配；**对于小于128B的内存申请，采用内存池技术**，采用链表进行管理。

07 你刚才提到了C++的内存池技术，能介绍一下吗。

C++默认的内存管理采用malloc(), free() 等，会频繁的在堆动态分配和回收内存，内存空间碎片化严重，导致空间利用率低。内存池很好的解决了这个问题，它是针对小对象而言的，首先申请一定数量，指定大小（通常8B）的内存块，当有新的内存申请就拿出一个块，如果不够再申请。

算法：

1. 预申请一个内存区chunk，将内存中按照对象大小划分成多个内存块block
2. 维持一个空闲内存块链表，通过指针相连，标记头指针为第一个空闲块
3. 每次新申请一个对象的空间，则将该内存块从空闲链表中去除，更新空闲链表头指针
4. 每次释放一个对象的空间，则重新将该内存块加到空闲链表头
5. 如果一个内存区占满了，则新开辟一个内存区，维持一个内存区的链表，同指针相连，头指针指向最新的内存区，新的内存块从该区内重新划分和申请

参考资料：C++内存池的简单原理及实现

08 请你说一下STL迭代器删除元素是怎么做的。

对于顺序容器而言，vector,deque使用erase删除元素的迭代器后，**会使后面所有的迭代器会失效，后面每个迭代器都会向前移动一个位置**，erase返回**下一个有效的迭代器**；

对于有序关联容器而言，set/multiset, map/multimap，删除元素并不会导致后面迭代器失效，因为他们底层实现是红黑树，所以**只需要递增迭代器**即可，对于无序关联容器，底层实现是哈希表，删除元素会导致迭代器失效，erase会返回下一个有效的迭代器。

对于list而言，它使用了不连续分配的内存，因此erase会返回下一个有效的迭代器，上面两种方式都可以使用。

09 deque和list用过吗，有什么心得。

两者都属于顺序容器。

deque是双向队列，它底层实现是一个**双端队列**，可用在头部和尾部添加或删除元素(push_front, push_back, pop_front, pop_back)。

- deque内部采用分段连续的内存空间来存储元素，在插入元素的时候随时都可以重新增加一段新的空间并链接起来，因此虽然提供了随机访问操作，但访问速度和vector相比要慢。
- deque并没有data函数，因为deque的元素并没有放在数组中。
- deque不提供capacity和reserve操作。
- deque内部的插入和删除元素要比list慢。

list是链表，它在插入删除元素的时间复杂度都是O(1)比deque更好。不支持按下标访问(随机访问)。

10 emplace_back()和push_back()哪个更好，为什么?

emplace_back()更好，因为它调用的是**移动构造函数**。而push_back()调用的是**拷贝构造函数**。移动构造函数不需要分配新的内存空间，所以更快一些。

11 ❤ vector::push_back()的时间复杂度是多少?

答案：O(1)。

当容器的大小达到容量后，为了保证内存的连续性，就会再开辟一个新的内存，把之前的数据复制过去。每次复制的时间复杂度是O(n)，但是因为**复制过程极少发生，所以均摊的时间复杂度还是O(1)。**

一开始vector的空间大小为1，那么每次从新分配空间的capacity变化为：

实际运行时， 会输出 `1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072`

很明显都是2的次幂，所以， 每次插入代价

$$c_i = \begin{cases} i & i-1\text{为}2\text{的幂} \\ 1 & \text{其他情况} \end{cases}$$

则

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

共有 n 次插入操作， 总时间复杂度为 $O(3n)$，单次均摊时间复杂度为 $O(3)$ 。

推导过程，感谢@tingtingli

12 迭代器是指针吗?

迭代器**不是指针**，而是**类模板**。它封装了指针并重载指针的一些运算符，如++，--，*等，所以能够遍历部分或全部访问容器元素的对象。**迭代器返回的是对象的引用，所以不能直接访问，需要用*解引用再访问。**

13 ❤讲一下capacity(), size(), reserve(), resize() 函数的区别。

size()用于返回容器当前的元素个数。而capacity()返回容器的容量。

reserve()是为容器预留空间，改变的是capacity，size保持不变。

resize()既改变了capacity，又改变了size。

reserve(x), 只有x>capacity才有用。

resize(x,val)，1. x > capacity，那么会在原容器内补充x−capacity个值为val的元素；2. x <= capacity，那么容器内前x个元素值变为为val，其余不变

14 ❤vector数组的底层原理?

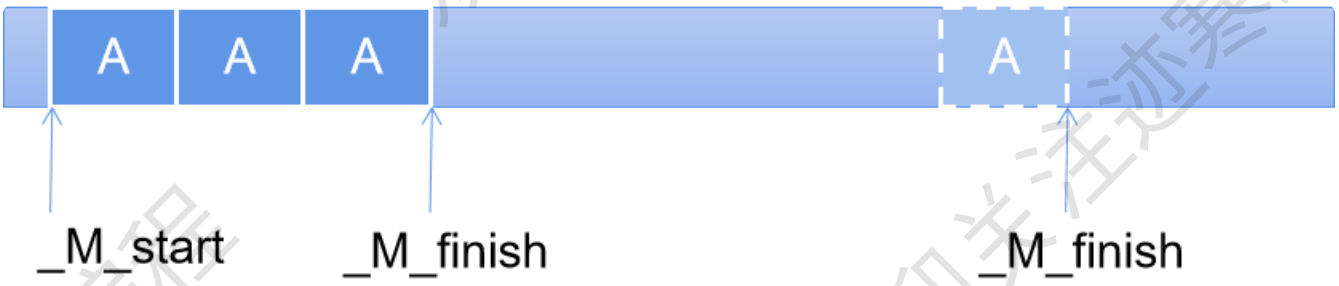
通过分析 vector 容器的源代码不难发现，它就是使用 3 个迭代器来表示的：

gcc\x86_64-w64-mingw32\8.1.0\include\c++\bits\stl_vector.h C++ 复制代码

```
//_Alloc 表示内存分配器，此参数几乎不需要我们关心
  template<typename _Tp, typename _Alloc>
    struct _Vector_base
    {
      struct _Vector_impl
      : public _Tp_alloc_type
      {
        pointer _M_start;
        pointer _M_finish;
        pointer _M_end_of_storage;
      }
    }
```

其中，`_M_start` 指向的是 vector 容器对象的起始字节位置；`_M_finish` 指向当前最后一个元素的末尾字节；`_M_end_of_storage` 指向整个 vector 容器所占用内存空间的末尾字节。

一图读懂：



参考链接：C++ vector（STL vector）底层实现机制（通俗易懂）

15 list底层实现原理

list底层是链表，通过查看 list 容器的源码实现，其对节点的定义如下：

```C++
template<typename T,...>
struct __List_node{
    //...
    __list_node<T>* prev;
    __list_node<T>* next;
    T myval;
    //...
}
```

可以看到，list 容器定义的每个节点中，都包含 *prev、*next 和 myval。其中，prev 指针用于指向前一个节点；next 指针用于指向后一个节点；myval 用于存储当前元素的值。下面是list的定义：

```C++
template <class T,...>
class list
{
    //...
    //指向链表的头节点，并不存放数据
    __list_node<T>* node;
    //...以下还有list 容器的构造函数以及很多操作函数
}
```

16 map的数组模式(operator[])插入和insert()插入的区别.

如果一个key存在， operator[] 对这个key-value进行重写

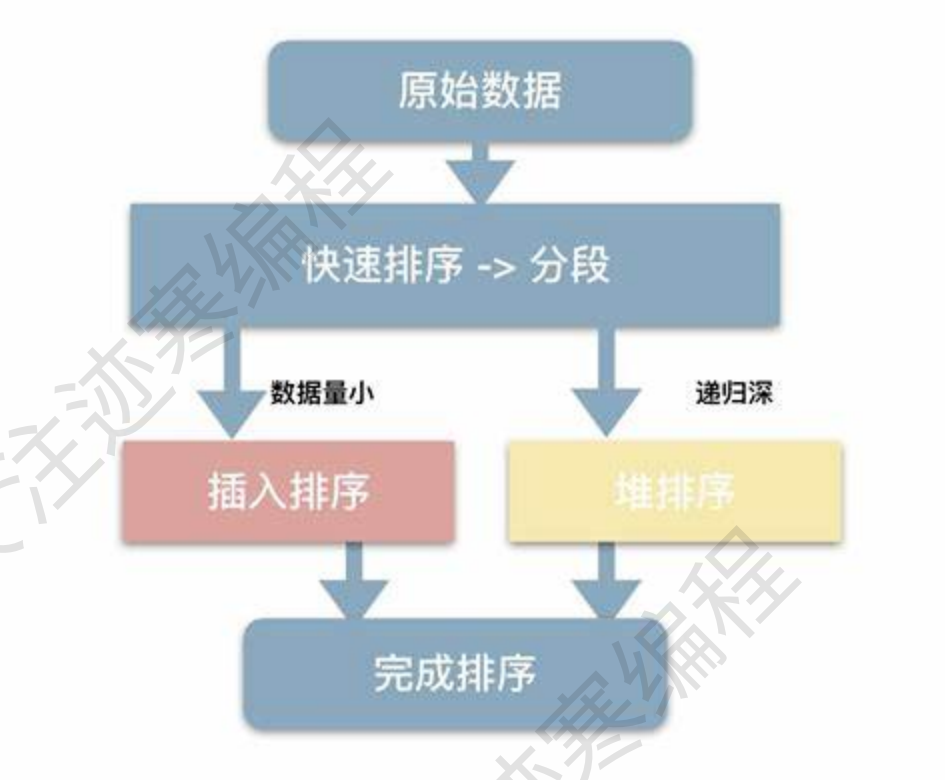
如果一个key存在， insert 不会对原来的key-value进行重写

17 ❤讲一下<algorithm>中的sort原理。

STL的sort采用了**快速排序、插入排序和堆排序**。根据数据量大小选择合适的算法：

- 当数据量较大，采用快速排序，分段递归；

- 一旦分段后的数据量小于一个阈值，改为插入排序。
- 为避免递归深度过深，达到一定递归深度采用堆排序。



18 什么是仿函数?

仿函数（Functor）又称为函数对象（Function Object）是一个能行使函数功能的类。

仿函数的语法几乎和我们普通的函数调用一样，不过作为仿函数的类，都必须重载 operator() 运算符。因为调用仿函数，实际上就是通过类对象调用重载后的 operator() 运算符。

```cpp
class StringAppend {
public:
    explicit StringAppend(const string& str) : ss(str){}
    void operator() (const string& str) const {
         cout << str << ' ' << ss << endl;
    }
private:
    const string ss;
};

int main() {
    StringAppend myFunctor2("and world!");
    myFunctor2("Hello");
}

```

C++11新特性

关注C++11新特性。

01 C++类型安全有什么特点。

C++比C有更高的安全性，这体现在：

1. 操作符new返回的对象类型严格与对象匹配，而不是void*;
2. C++模板支持类型检查
3. 引入了常量const来替代宏定义#define，#define只是简单的文本替换，不支持类型检查
4. 一些#define宏可以被改写为inline函数，可以在类型安全的前提下支持多种类型
5. C++提供dynamic_cast，它比static_cast有更多类型检查。

02 ❤C++泛型和模板了解吗。

泛型可以独立于任何特定参数类型进行编程，模板是泛型编程的基础。包括函数模板和类模板两种，比如：

1. 函数模板

```cpp
template<typename T>
void func(T a){};
```

2. 类模板

```cpp
template <typename Type>
class Queue
{
public:
    Queue();
    Type & front();
    const Type & front() const;
    void push(const Type &);
    void pop();
    bool empty() const;
private:
    // …
};
//指定默认参数
template<typename T = int, typename Y = char> // 此处指定了模板默认参数，部分指定必须从右到左指定
class Test {
public:
    Test(T t, Y y) : t(t), y(y) {
    }
    void tfunc();
private:
    T t;
    Y y;
};
```

03 模板可以传入形参吗?

可以。模板传入的参数被称为**非类型实参**。例如`template<typename T, int MAXSIZE>`，非类型实参在模板内部被定义为常量值。

04 ❤C++泛型的原理清楚吗?

泛型的核心是模板。模板是将一个定义里面的类型参数化出来，是宏的改进版本。宏不进行任何**变量类型检查**，仅仅进行文本替换，这样就可能造成那种难以发现的错误。

下面是两个例子，来描述泛型编程的好处：

C++ | 复制代码

```
//不用泛型
void qsort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));
//使用泛型
template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
        Compare comp);
```

1. **类型安全性**：如果你调用std::sort(arr, arr + n, comp);那么comp的类型就必须要和arr的数组元素类型一致，否则编译器就会帮你检查出来。而且comp的参数类型再也不用const void*这种不直观的表示了，而是可以直接声明为对应的数组元素的类型。

2. **通用性**：这个刚才已经说过了。泛型的核心目的之一就是通用性。std::sort可以用于一切迭代器，其compare函数可以是一切支持函数调用语法的对象。如果你想要将std::sort用在你自己的容器上的话，你只要定义一个自己的迭代器类（严格来说是一个随机访问迭代器，STL对迭代器的访问能力有一些分类，随机访问迭代器具有建模的内建指针的访问能力），如果需要的话，再定义一个自己的仿函数类即可。

3. **接口直观性**：跟qsort相比，std::sort的使用接口上没有多余的东西，也没有不直观的size参数。一个有待排序的区间，一个代表比较标准的仿函数，仅此而已。

4. **效率**：如果你传给std::sort的compare函数是一个自定义了operator()的仿函数。那么编译器就能够利用类型信息，将对该仿函数的operatpr()调用直接内联。消除函数调用开销。

关于模板更细致的讨论参见：C++ 模板详解 | 菜鸟教程

05 auto 和 decltype 区别。

auto可以自动类型推导，但无法定义变量类型，可以用于返回值和形参。

decltype返回变量类型，和auto一样在编译器起作用 。大部分情况下auto更好用一些，但是在比如一些容器的比较函数上，只能用decltype。

auto不能用于含有递归的匿名函数。

编译与内存

01 ❤malloc原理。

Malloc函数用于**动态分配内存**。malloc其采用**内存池**的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块。当用户申请内存时，直接从堆区分配一块合适的空闲块。

malloc采用隐式链表结构将堆区分成连续的、大小不一的块；同时malloc采用显示链表结构来管理所有的空闲块，每一个空闲块记录了一个连续的、未分配的地址。

搜索空闲块最常见的算法有：首次适配，下一次适配，最佳适配。（其实就是操作系统中动态分区分配的算法）

02 `brk()` 和 `sbrk()` 及 `mmap` 的区别和联系?

三者都是**系统调用函数**。

- brk() 和 sbrk()都是扩展堆的上界。

C++ | 复制代码

```cpp
#include <unistd.h>
int brk( const void *addr )//参数设置为新的brk上界地址，成功返回1，失败返回0;
void* sbrk ( intptr_t incr );//申请内存的大小，返回heap新的上界brk的地址;
```

- mmap采用的是**匿名映射**。

```C++
#include <sys/mman.h>
//mmap的第一种用法是映射此盘文件到内存中；
//第二种用法是匿名映射，不映射磁盘文件，而向映射区申请一块内存。
void *mmap(void *addr, size\_t length, int prot, int flags, int fd, off\_t offset);
int munmap(void *addr, size_t length);//释放内存。
```

1）当开辟的空间小于 128K 时，调用 **brk函数**，malloc 的底层实现是系统调用函数 brk，其主要移动指针 _enddata(此时的 _enddata 指的是 Linux 地址空间中堆段的末尾地址，不是数据段的末尾地址)。

- malloc分配了这块内存，然后如果从不去访问它，那么物理页是不会被分配的。
- 当最高地址空间的空闲内存超过128K（可由M_TRIM_THRESHOLD选项调节）时，执行内存紧缩操作。

2）当开辟的空间大于 128K 时，**mmap**系统调用函数来在虚拟地址空间中（堆和栈中间，称为“文件映射区域”的地方）找一块空间来开辟。

其实，很多人开始诟病 glibc 内存管理的实现，特别是高并发性能低下和内存碎片化问题都比较严重，因此，陆续出现一些第三方工具来替换 glibc 的实现，最著名的当属 google 的tcmalloc和 facebook 的jemalloc 。

参考链接： linux环境内存分配原理--虚拟内存 mallocinfo

03 ❤说一说malloc, realloc, calloc的区别。怎么使用。

malloc是最基本的内存分配函数，典型的用法是

```C++
int *arr = (int*) malloc(10 * sizeof(int))
```

calloc除了分配内存，还会进行**初始化**，有两个参数：元素个数和元素字节大小。

```C++
int* arr = (int*)calloc(10, sizeof(int));
```

realloc是给已经分配过空间的变量**重新分配空间**。它有两个参数：原指针和新的空间大小。

```C++
int *arr = (int*)realloc(arr, 20*sizeof(int));
```

04 ❤说一下new/delete和malloc/free的区别。

new/delete是C++的关键字，会自动调用对象的构造函数和析构函数。具体而言：**new先调用operator new()函数申请空间，之后调用构造函数；delete先在空间上执行析构函数，再调用operator delete()来释放空间。**

malloc/free是C函数，malloc申请一段空间并随机填充，并不安全，可以使用calloc（初始化为0）代替。

05 ❤你了解哪些new方法?

有四种：**plain new, no_throw new, placement new和new[]**

new最常见的用法是先用**operator new()**分配空间再调用构造函数。调用构造函数是采用**placement new()**来完成的。这种new允许在一块已经分配成功的内存上重新构造对象或对象数组。placement new不用担心内存分配失败，因为它根本不分配内存，它做的**唯一**一件事情就是调用对象的构造函数。定义如下：

```C++
void* operator new(size_t, void*);
void operator delete(void*, void*);
```

no_throw new是new**不抛出异常**的形式，传统的new默认抛出bad_alloc异常。

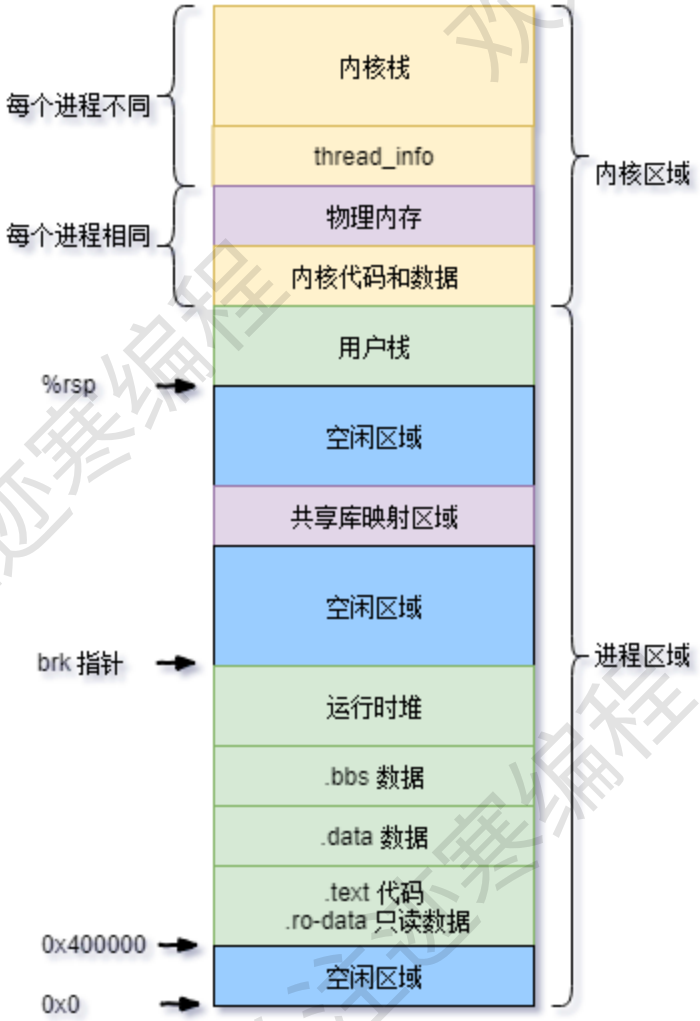
```C++
//plain new
void* operator new(std::size_t) throw(std::bad_alloc);
void operator delete( void *) throw();
//no_throw new
void * operator new(std::size_t, const std::nothrow_t&) throw();
void operator delete(void*) throw();
```

new[] 是在堆上分配数组对象的方式。

06 ❤介绍一下C++虚拟内存分区。

地址从低到高：

- 代码段(.text)：保存程序二进制机器码，以及文本常量。
- bss段(.bss)：又称全局静态变量区，存储全局变量和静态变量。
- 堆区：动态分配的对象，手动申请和释放。
- 文件映射区：存储动态链接库及mmap函数进行文件映射。
- 栈区：存储临时变量和局部变量，系统自动管理内存。
- 内核区：受保护的的内存区域。包括每个进程不同的内核栈和thread_info和每个进程相同的物理内存和内核代码和数据。



07 include头文件""和<>有何区别?

""会优先查找当前文件目录。<>则是查找编译器设置的头文件目录。

08 namespace有什么作用?

为了解决变量和函数等的作用范围，在C++语言中引入了名空间的概念，并增加了关键字namespace和using

通过命名空间，可以在**同一个文件中使用相同的变量名或函数名**，只要它们属于不同的名空间。另外，名空间可以使得代码操作具有相同名字但属于不同库的变量。而且，名空间也可以提高C语言

与C++语言的兼容性。

09 什么时候会发生段错误?

- 访问了**不存在**的地址，比如试图修改null指针的值。

```C++
int *p = NULL;
(*p) = 0;
```

- 访问了**受保护**的地址：

```C++
int *p = 0;
(*p) = 1;
```

- 试图修改**只读区**，比如修改字面值常量

```C++
const int a = 1;
a = 2;
```

- 栈溢出，无限递归
- new一次但是delete多次。

10 C++如何抛出一个错误?

C++异常处理流程一般为：抛出（Throw）--> 检测（Try） --> 捕获（Catch）。

使用 `throw` 关键字来显式抛出：

```C++
throw exceptionData;
```

通常结合try catch来抛出异常，下面是一个例子：

C++ 复制代码

```cpp

double divideBy(double x, double y)                    //定义函数
{
    if(y==0)
    {
            throw y;                                   //除数为0，抛出异常
        }
        return x/y;                                    //否则返回两个数的商
    }
}

int main()
{
    double res;
    try                                                //定义异常
    {
        res=divideBy(2,3);
        cout<<"The result of x/y is : "<<res<<endl;
        res=fuc(4,0);                                  //出现异常
    }
    catch(double)                                      //捕获并处理异常
    {
        cerr<<"error of dividing zero.\n";
        exit(1);                                       //异常退出程序
    }
    return 0;
}
```

如果一个函数不会抛出异常可以加上 `noexcept` 来修饰。

11 C++断言是什么，如何使用?

assert用在那些你知道**绝对不会发生**的事情上。

什么是不应该发的的状况呢？这要区分数据的来源：１、数据来源于系统内部（子程序、子模块间的调用）２、数据来源于系统外部（外部设备如键盘的输入、串口数据的读取、网络数据的读取）。对内部来源的数据，我们没法去通过常规的测试手段去验证，此时断言就用上了。

当然你也可也用if，这样会导致代码臃肿。

通常有两种断言用法：

```cpp
assert(condition); // 我明白，condition一定为true
static_assert(condition, message); // 在condition为false时打印消息
// 关闭断言
#define NDEBUG
#include <assert.h>
```

操作系统

C++与操作系统。

01 ❤C++如何避免死锁?

操作系统里面讲到，破坏死锁产生的四个条件中的一个就可以（互斥、不可剥夺、循环等待、请求和保持），这里需要展开来说：

- 加锁的时候使用try_lock，如果获取不到锁则释放自身的所有的锁；
- 使用mutex加锁的时候按照地址从小到大进行顺序加锁；
- 将线程锁设置为 `PTHREAD_MUTEX_ERRCHECK` ，死锁会返回错误，不过效率较低。

02 说一说你知道哪一些操作线程的函数?

`pthread_create`: 创建一个线程，返回0表示线程创建成功。例子

`pthread_t pthread_self()`：获取进程id

`int pthread_join(pthread_t tid, void** retval)`：等待线程结束

`void pthread_exit(void *retval)`：结束线程

`int pthread_detach(pthread_t tid)`：主线程、子线程均可调用。主线程中pthread_detach(tid)，子线程中pthread_detach(pthread_self())，调用后和主线程分离，子线程结束时自己立即回收资源。

03 说一说你知道哪些进程有关的函数?

进程结构由以下几个部分组成：**代码段，堆栈段，数据段**。代码段是静态的二进制码，多个程序可以共享，父进程与子进程除了pid不一样，其它都一样。父进程通过fork产生一个子进程。

父进程与子进程通过**写时复制(Copy on Write)**技术共享页面，只有当子进程需要写入页面才进行复制。如果子进程想要运行自己的代码段，就需要execv().

`pid_t fork(void)`: 创建进程，返回一个非负整数，父进程返回子进程的pid，子进程返回0；

`void exit(int status)`: 结束进程；

`pid_t getpid(void)`: 获取进程pid；

`pid_t getppid(void)`: 获取父进程pid。

04 关于程序退出方式，你知道哪些?

正常退出方式有：return, _exit(), exit()

exit()其实是对_exit() 的一个封装，都会终止进程并做相关的首尾工作，最主要的区别是exit()会调用终止处理程序和清除I/O缓存。

return和exit的区别，exit是函数，有参数，执行完后控制权交还给OS，return 可以在函数中，调用后控制权返回给上一级函数，若是main函数，则返还给OS。

还有一些其它退出方式：

abort()，异常程序终止，同时发送SIGABRT给调用进程。

接能导致进程终止的信号，比如cltr+c就是SIGINT信号.

测试

01 什么是黑盒测试和白盒测试。

黑盒测试

黑盒，其实从字面意思上来理解就是将测试对象看作是一个不公开透明的黑色盒子。黑盒测试简单来说就是在测试的时候，不考虑盒子里面的逻辑结果跟程序运行，只是根据程序的需求规格书来检查程序的功能是否符合它的功能说明，检验输出结果正不正确。比如**性能测试，压力测试**。

白盒测试

与黑盒恰恰相反，这种方法是把测试对象看作一个打开的透明盒子。测试时，测试人员会利用程序内部的逻辑结构及有关信息，通过在不同点检查程序状态，检验程序中的每条通路是否都能按预定要求进行正确工作。比如**单元测试，集成测试等**。

有点东西

不太容易考的知识点（但其实考过，如果你碰到一个狡黠的家伙的话）。

01 如何实现++i与i++?

重写int的++运算符；

C++ | 复制代码

```cpp
//++i
int& int::operator++(){
    *this = *this + 1;
    return *this;
}
//i++;
const int int::operator++(int){
    int old = *this;
    *this = *this+1;
    return old;
}
```

02 写一个函数在main函数之前运行。

```C++
__attribute((constructor)) void before(){

}
```

如果是在之后运行呢?

```C++
__attribute((deconstructor)) void after(){


}
```

GNU C的一大特色（却不被初学者所知）就是**attribute**机制。**attribute**可以设置函数属性（Function Attribute）、变量属性（Variable Attribute）和类型属性（Type Attribute）。

03 两个几乎完全相同的函数，第二个函数仅仅多了const，问这种情况会报错吗?

不会，这相当于函数重载。

04 函数参数压栈顺序?

从右到左。

05 下面的输出是多少?为什么?

```cpp
int main()
{
    int i = 5;
    void* pInt = &i;
    double d = (*(double*)pInt);
    cout << d << endl;

    return 0;
}
```

输出不是5，用到了空类型指针void*，类型不安全。

06 C++有什么优化方法。

宏优化。也就是编译时加上 `-O` ：

- -O0： 不做任何优化，这是默认的编译选项。
- -O1优化会消耗少多的编译时间，它主要对代码的分支，常量以及表达式等进行优化。
- -O2会尝试更多的寄存器级的优化以及指令级的优化，它会在编译期间占用更多的内存和编译时间。
- -O3在O2的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。
- -Os主要是对代码大小的优化，我们基本不用做更多的关心。 通常各种优化都会打乱程序的结构，让调试工作变得无从着手。并且会打乱执行顺序，依赖内存操作顺序的程序需要做相关处理才能确保程序的正确性。

O2优化能使程序的编译效率大大提升。从而减少程序的运行时间，达到优化的效果。

在C++程序中的O2开关：

```cpp
#pragma GCC optimize(2)
```

同理O1、O3优化只需修改括号中的数即可。 只需将这句话放到程序的开头即可打开O2优化开关。

开启O3优化：

```C++
#pragma GCC optimize(3,"Ofast","inline")
```

总结

C++博大精深，曾有人断言”100%精通C++不存在，包括C++之父Bjarne Stroustrup“。C++面试可以分为5个部分：

基础：C++语法，关键字（尤其是static和const），指针，数组，函数，类型强制转换。

面向对象：封装继承多态，构造函数，析构函数，基类，子类，虚函数，重载重写。

C++11新特性：左值，右值，右值引用，移动语义和完美转发。

STL：容器，算法，函数对象，迭代器，适配器，内存分配器。

编译与内存：malloc,calloc,realloc原理，new/delete，内存池技术，虚拟内存，编译器参数等。

如果对你有帮助的话，欢迎关注我的公众号”**迹寒编程**“：