代码随想录知识星球精华（最强八股文）第五版（C++篇）

代码随想录知识星球精华（最强八股文）第五版为九份PDF，分别是：

- 代码随想录知识星球八股文概述
- C++篇
- go篇
- Java篇
- 前端篇
- 算法题篇
- 计算机基础篇
- 问答精华篇
- 面经篇

本篇为最强八股文之C++篇。

C++ 基础

指针和引用的区别

指针存放某个对象的地址，其本身就是变量（命了名的对象），本身就有地址，所以可以有指向指针的指针；可变，包括其所指向的地址的改变和其指向的地址中所存放的数据的改变

引用就是**变量的别名，从一而终，不可变**，必须初始化

不存在指向空值的引用，但是存在指向空值的指针

1. **定义和声明：**

指针是一个变量，其值是另一个变量的地址。声明指针时，使用`*`符号。

```
int x = 10;
int *ptr = &x;
```

引用是一个别名，它是在已存在的变量上创建的。在声明引用时，使用`&`符号。

```
int y = 20;
int &ref = y;
```

2. **使用和操作：**

指针： 可以通过解引用操作符`*`来访问指针指向的变量的值，还可以通过地址运算符`&`获取变量的地址。

引用： 引用在声明时被初始化，并在整个生命周期中一直引用同一个变量。不需要使用解引用操作符，因为引用本身就是变量的别名

```
int value = *ptr;  // 获取指针指向的值
int address = &x;  // 获取变量 x 的地址
```

```
int newValue = ref;  // 获取引用的值
```

3. **空值和空引用：**

指针可以为空（nullptr）表示不指向任何有效的地址。

引用必须在声明时初始化，并且不能在后续改变引用的绑定对象。因此，没有空引用的概念

4. **可变性：**

指针： 可以改变指针的指向，使其指向不同的内存地址。

引用： 一旦引用被初始化，它将一直引用同一个对象，不能改变绑定。

5. **用途：**

指针： 通常用于动态内存分配、数组操作以及函数参数传递。

引用： 通常用于函数参数传递、操作符重载以及创建别名。

数据类型

整型 short int long 和 long long

C++ 整型数据长度标准

short 至少 16 位

int 至少与 short 一样长

long 至少 32 位，且至少与 int 一样长

long long 至少 64 位，且至少与 long 一样长

在使用8位字节的系统中，1 byte = 8 bit

很多系统都使用最小长度，short 为 16 位即 2 个字节，long 为 32 位即 4 个字节，long long 为 64 位即 8 个字节，int 的长度较为灵活，一般认为 int 的长度为 4 个字节，与 long 等长。

可以通过运算符 sizeof 判断数据类型的长度。例如：

```
cout << "int is " << sizeof (int) << " bytes. \n";
cout << "short is " << sizeof (short) << " bytes. \n";
```

头文件climits定义了符号常量：例如：INT_MAX 表示 int 的最大值，INT_MIN 表示 int 的最小值。

无符号类型

即为不存储负数值的整型，可以增大变量能够存储的最大值，数据长度不变。

int 被设置为自然长度，即为计算机处理起来效率最高的长度，所以选择类型时一般选用 int 类型。

关键字

const 关键字

const的作用：

被它修饰的值不能改变，是只读变量。必须在定义的时候就给它赋初值。

1、常量指针（底层const ）

常量指针：

是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。常量指针强调的是指针对其所指对象的不可改变性。

特点：

靠近变量名。

形式：

（1）const 数据类型 *指针变量 = 变量名

（2）数据类型 const *指针变量 = 变量名

示例：

```
int temp = 10;

const int* a = &temp;
int const *a = &temp;

// 更改:
*a = 9; // 错误：只读对象
temp = 9; // 正确
```

2、指针常量（顶层const）

指针常量：

指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。指针常量强调的是指针的不可改变性。

特点：

靠近变量类型。

形式：

数据类型 * const 指针变量=变量名

示例：

```
int temp = 10;
int temp1 = 12;
int* const p = &temp;

// 更改:
p = &temp2; // 错误
*p = 9; // 正确
```

```
#include<iostream>
int main(){
    // 1.常量指针
    int tmp = 10;
    int tmp2 = 11;
    const int* a = &tmp; // 指针常量
    // *a =9; // 报错：更改对象值
    tmp = 9; // 正确:在原对象上修改
    a = &tmp2; // 正确：可以指向别的对象

    // 2.指针常量
    int tmp3 = 12;
    int tmp4 = 13;
    int * const p = &tmp3; // 常量指针
    *p = 9; // 正确：可以更改指向对象的值
    // p = &tmp4; // 错误：不可以指向新的对象
    return -1;
}
```


static关键字的作用

`static`关键字主要用于控制变量和函数的生命周期、作用域以及访问权限。

1. 静态变量

- 在函数内部使用`static`关键字修饰的变量称为静态变量。
- 静态变量在程序的整个生命周期内存在，不会因为离开作用域而被销毁。
- 静态变量默认初始化为零（对于基本数据类型）。

```
void exampleFunction() {
    static int count = 0;  // 静态变量
    count++;
    cout << "Count: " << count << endl;
}
```

2. 静态函数

- 在类内部使用`static`关键字修饰的函数是静态函数。
- 静态函数属于类而不是类的实例，可以通过类名直接调用，而无需创建对象。
- 静态函数不能直接访问非静态成员变量或非静态成员函数。

```
class ExampleClass {
public:
    static void staticFunction() {
        cout << "Static function" << endl;
    }
};
```

3. 静态成员变量

- 在类中使用`static`关键字修饰的成员变量是静态成员变量。
- 所有类的对象共享同一个静态成员变量的副本。
- 静态成员变量必须在类外部单独定义，以便为其分配存储空间。

```
class ExampleClass {
public:
    static int staticVar;  // 静态成员变量声明
};

// 静态成员变量定义
int ExampleClass::staticVar = 0;
```

4. 静态成员函数

- 在类中使用`static`关键字修饰的成员函数是静态成员函数。
- 静态成员函数不能直接访问非静态成员变量或非静态成员函数。
- 静态成员函数可以通过类名调用，而不需要创建类的实例。

```
class ExampleClass {
public:
    static void staticMethod() {
        cout << "Static method" << endl;
    }
};
```

5. 静态局部变量

- 在函数内部使用`static`关键字修饰的局部变量是静态局部变量。
- 静态局部变量的生命周期延长到整个程序的执行过程，但只在声明它的函数内可见。

```
void exampleFunction() {
    static int localVar = 0;  // 静态局部变量
    localVar++;
    cout << "LocalVar: " << localVar << endl;
}
```

const 关键字的作用

`const`关键字主要用于指定变量、指针、引用、成员函数等的性质

1. 常量变量：声明常量，使变量的值不能被修改。
2. 指针和引用：声明指向常量的指针，表示指针所指向的值是常量，不能通过指针修改。声明常量引用，表示引用的值是常量，不能通过引用修改。

```
const int* ptr = &constantValue;  // 指向常量的指针
const int& ref = constantValue;  // 常量引用
```

3. 成员函数：用于声明常量成员函数，表示该函数不会修改对象的成员变量（对于成员变量是非静态的情况）。
4. 常量对象：声明对象为常量，使得对象的成员变量不能被修改。
5. 常引用参数：声明函数参数为常量引用，表示函数不会修改传入的参数。
6. 常量指针参数：声明函数参数为指向常量的指针，表示函数不会通过指针修改传入的数据。

define 和 typedef 的区别

define

1. 只是简单的字符串替换，没有类型检查
2. 是在编译的预处理阶段起作用
3. 可以用来防止头文件重复引用
4. 不分配内存，给出的是立即数，有多少次使用就进行多少次替换

typedef

1. 有对应的数据类型，是要进行判断的
2. 是在编译、运行的时候起作用
3. 在静态存储区中分配空间，在程序运行过程中内存中只有一个拷贝

define 和 inline 的区别

1、define：

定义预编译时处理的宏，只是简单的字符串替换，无类型检查，不安全。

2、inline：

inline是先将内联函数编译完成生成了函数体直接插入被调用的地方，减少了压栈，跳转和返回的操作。没有普通函数调用时的额外开销；

内联函数是一种特殊的函数，会进行类型检查；

对编译器的一种请求，编译器有可能拒绝这种请求；

C++中inline编译限制：

1. 不能存在任何形式的循环语句
2. 不能存在过多的条件判断语句
3. 函数体不能过于庞大
4. 内联函数声明必须在调用语句之前

const和define的区别

const用于定义常量；而define用于定义宏，而宏也可以用于定义常量。都用于常量定义时，它们的区别有：

1. const生效于编译的阶段；define生效于预处理阶段。
2. const定义的常量，在C语言中是存储在内存中、需要额外的内存空间的；define定义的常量，运行时是直接的操作数，并不会存放在内存中。
3. const定义的常量是带类型的；define定义的常量不带类型。因此define定义的常量不利于类型检查。

new 和 malloc的区别

1、new内存分配失败时，会抛出bac_alloc异常，它不会返回NULL；malloc分配内存失败时返回NULL。

2、使用new操作符申请内存分配时无须指定内存块的大小，而malloc则需要显式地指出所需内存的尺寸。

3、opeartor new /operator delete可以被重载，而malloc/free并不允许重载。

4、new/delete会调用对象的构造函数/析构函数以完成对象的构造/析构。而malloc则不会

5、malloc与free是C++/C语言的标准库函数,new/delete是C++的运算符

6、new操作符从自由存储区上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。

表格

	New/delete	Malloc/free 本人
本质属性	运算符	CRT 函数
内存分配大小	自动计算	手工计算
类型安全	是 (一个 int 类型指针指向 float 会报错)	不是 (malloc 类型转换成 int，分配 double 数据类型大小的内存空间不会报错)
两者关系	new 封装了 malloc	
其他特点	除了分配和释放内存还会调用构造和析构函数	只分配和释放内存
	内存分配失败时抛出 bad_alloc 异常	内存分配失败时返回 null
	返回定义时具体类型的指针	返回的是 void 类型的指针，使用时需要进行类型转换

constexpr 和 const

const 表示“只读”的语义，constexpr 表示“常量”的语义

constexpr 只能定义编译期常量，而 const 可以定义编译期常量，也可以定义运行期常量。

你将一个成员函数标记为constexpr，则顺带也将它标记为了const。如果你将一个变量标记为constexpr，则同样它是const的。但相反并不成立，一个const的变量或函数，并不是constexpr的。

constexpr变量

复杂系统中很难分辨一个初始值是不是常量表达式，可以将变量声明为constexpr类型，由编译器来验证变量的值是否是一个常量表达式。

必须使用常量初始化：

```
constexpr int n = 20;
constexpr int m = n + 1;
static constexpr int MOD = 1000000007;
```

如果constexpr声明中定义了一个指针，constexpr仅对指针有效，和所指对象无关。

```
constexpr int *p = nullptr; //常量指针 顶层const
const int *q = nullptr;  //指向常量的指针，  底层const
int *const q = nullptr;  //顶层const
```

constexpr函数：

constexpr函数是指能用于常量表达式的函数。

函数的返回类型和所有形参类型都是字面值类型，函数体有且只有一条return语句。

```
constexpr int new() {return 42;}
```

为了可以在编译过程展开，constexpr函数被隐式转换成了内联函数。
constexpr和内联函数可以在程序中多次定义，一般定义在头文件。

constexpr 构造函数：

构造函数不能说const，但字面值常量类的构造函数可以是constexpr。

constexpr构造函数必须有一个空的函数体，即所有成员变量的初始化都放到初始化列表中。对象调用的成员函数必须使用 constexpr 修饰

const：

指针常量: `const int* d = new int(2);`
常量指针: `int *const e = new int(2);`

区别方法：

左定值，右定向：指的是const在*的左还是右边

拓展：

顶层const：指针本身是常量；
底层const：指针所指的对象是常量；

若要修改const修饰的变量的值，需要加上关键字volatile；

若想要修改const成员函数中某些与类状态无关的数据成员，可以使用mutable关键字来修饰这个数据成员；

『const和static的区别』

关键字	修饰常量【非类中】	修饰成员变量	修饰成员函数
const	超出其作用域后空间会被释放； 在**定义时必须初始化**，之后无法更改； const形参可以接受const和非const类型的实参；	只在**某个对象**的生命周期内是常量；而对整个对象而言是可变的； 不能赋值，不能在类外定义；只能通过构造函数的参数初始化列表初始化【原因：因为不同的对象对其const数据成员的值可以不同，所以不能在类中声明时初始化】	防止成员函数修改对象的内容【不能修改成员变量的值，但是可以访问】 const对象不可以调用非const的函数；但是非const对象可以调用；
static	在函数执行后不会释放其存储空间	只能用在类定义体内部的声明，外部初始化，且不加static	①作为类作用域的全局函数【不能访问非静态数据成员和调用非静态成员函数】 ②没有this指针【不能直接存取非类的非静态成员，调用非静态成员函数】 ③不能声明为virtual
const 和static不能同时修饰成员函数，原因：静态成员函数不含有this指针，即不能实例化，而const成员函数必须具体到某一实例			

constexpr的好处

1. 为一些不能修改数据提供保障，写成变量则就有被意外修改的风险。
2. 有些场景，编译器可以在编译期对constexpr的代码进行优化，提高效率。
3. 相比宏来说，没有额外的开销，但更安全可靠。

volatile

定义：

[与const绝对对立的，是类型修饰符]影响编译器编译的结果，用该关键字声明的变量表示该变量随时可能发生变化，与该变量有关的运算，不要进行编译优化；会从内存中重新装载内容，而不是直接从寄存器拷贝内容。

作用：

指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值，保证对特殊地址的稳定访问

使用场合：

在中断服务程序和cpu相关寄存器的定义

举例说明：

空循环：

```
for(volatile int i=0; i<100000; i++); // 它会执行，不会被优化掉
```

extern

定义：声明外部变量【在函数或者文件外部定义的全局变量】

static

作用：实现多个对象之间的数据共享 + 隐藏，并且使用静态成员还不会破坏隐藏原则；默认初始化为0

前置++与后置++

```
self &operator++() {
    node = (linktype)((node).next);
    return *this;
}

const self operator++(int) {
    self tmp = *this;
    ++*this;
    return tmp;
}
```

为了区分前后置，重载函数是以参数类型来区分，在调用的时候，编译器默默给int指定为一个0

1、为什么后置返回对象，而不是引用

因为后置为了返回旧值创建了一个临时对象，在函数结束的时候这个对象就会被销毁，如果返回引用，那么我请问你？你的对象对象都被销毁了，你引用啥呢?

2、为什么后置前面也要加const

其实也可以不加，但是为了防止你使用i++++,连续两次的调用后置++重载符，为什么呢?

原因：

它与内置类型行为不一致；你无法活得你所期望的结果，因为第一次返回的是旧值，而不是原对象，你调用两次后置++，结果只累加了一次，所以我们必须手动禁止其合法化，就要在前面加上const。

3、处理用户的自定义类型

最好使用前置++，因为他不会创建临时对象，进而不会带来构造和析构而造成的格外开销。

std::atomic

问题：a++ 和 int a = b 在C++中是否是线程安全的?

答案：不是

例1：

a++：从C/C++语法的级别来看，这是一条语句，应该是原子的；但从编译器得到的汇编指令来看，其实不是原子的。

其一般对应三条指令，首先将变量a对应的内存值搬运到某个寄存器（如eax）中，然后将该寄存器中的值自增1，再将该寄存器中的值搬运回a代表的内存中

```
mov eax, dword ptr [a] # (1)
inc eax     # (2)
mov dword ptr [a], eax # (3)
```

现在假设i的值是0，有两个线程，每个线程对变量a的值都递增1，预想一下，其结果应该是2，可实际运行结构可能是1！是不是很奇怪?

分析如下：

```
int a = 0;
// 线程1（执行过程对应上文汇编指令(1)(2)(3)）
void thread_func1() {
 a++;
}
// 线程2（执行过程对应上文汇编指令(4)(5)(6)）
void thread_func2() {
 a++;
}
```

我们预想的结果是线程1和线程2的三条指令各自执行，最终a的值变为2，但是由于操作系统线程调度的不确定性，线程1执行完指令(1)和(2)后，eax寄存器中的值变为1，此时操作系统切换到线程2执行，执行指令(3)(4)(5)，此时eax的值变为1；接着操作系统切回线程1继续执行，执行指令(6)，得到a的最终结果1。

例2：

`int a = b;` 从C/C++语法的级别来看，这是条语句应该是原子的；但从编译器得到的汇编指令来看，由于现在计算机CPU架构体系的限制，数据不能直接从内存某处搬运到内存另外一处，必须借助寄存器中转，因此这条语句一般对应两条计算机指令，即将变量b的值搬运到某个寄存器（如eax）中，再从该寄存器搬运到变量a的内存地址中：

```
mov eax, dword ptr [b]
mov dword prt [a], eax
```

既然是两条指令，那么多个线程在执行这两条指令时，某个线程可能会在第一条指令执行完毕后被剥夺CPU时间片，切换到另一个线程而出现不确定的情况。

解决办法： C++11新标准发布后改变了这种困境，新标准提供了对整形变量原子操作的相关库，即std::atomic，这是一个模板类型：

```
template<class T>
struct atomic:
```

我们可以传入具体的整型类型对模板进行实例化，实际上stl库也提供了这些实例化的模板类型

```
// 初始化1
std::atomic<int> value;
value = 99;

// 初始化2
// 下面代码在Linux平台上无法编译通过（指在gcc编译器）
std::atomic<int> value = 99;
// 出错的原因是这行代码调用的是std::atomic的拷贝构造函数
// 而根据C++11语言规范，std::atomic的拷贝构造函数使用=delete标记禁止编译器自动生成
// g++在这条规则上遵循了C++11语言规范。
```

常量指针和指针常量的区别

- 常量指针是指针本身是常量，内容可变。
- 指针常量是指针所指向的内容是常量，指针本身可变。
- 在常量指针中，`const`关键字位于`*`之前。
- 在指针常量中，`const`关键字位于`*`之后。

常量指针是指指针本身是一个常量，即指针的值（地址）不能被修改，但它所指向的内容可以被修改。

```
int x = 10;
int y = 20;

const int *ptr = &x;  // 常量指针，不能修改指针的值，但可以修改指针所指向的内容
// ptr = &y;  // 错误，不能修改指针的值
// *ptr = 30;  // 错误，不能修改指针所指向的内容
```

指针常量是指指针所指向的内容是一个常量，即指针所指向的内容不能被修改，但指针本身的值（地址）可以被修改。

```
int x = 10;
int y = 20;

int *const ptr = &x;  // 指针常量，不能修改指针的值，但可以修改指针所指向的内容
// ptr = &y;  // 错误，不能修改指针的值
*ptr = 30;  // 正确，可以修改指针所指向的内容
```

什么是函数指针，如何定义和使用场景

函数指针是指向函数的指针变量。它可以用于存储函数的地址，允许在运行时动态选择要调用的函数。

```
// 返回类型 (*指针变量名)(参数列表)
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int main() {
    // 定义一个函数指针，指向一个接受两个int参数、返回int的函数
    int (*operationPtr)(int, int);

    // 初始化函数指针，使其指向 add 函数
    operationPtr = &add;

    // 通过函数指针调用函数
    int result = operationPtr(10, 5);
```

```
    cout << "Result: " << result << endl;

    // 将函数指针切换到 subtract 函数
    operationPtr = &subtract;

    // 再次通过函数指针调用函数
    result = operationPtr(10, 5);
    cout << "Result: " << result << endl;

    return 0;
}
```

使用场景：

1. **回调函数：** 函数指针常用于实现回调机制，允许将函数的地址传递给其他函数，以便在适当的时候调用。
2. **函数指针数组：** 可以使用函数指针数组实现类似于状态机的逻辑，根据不同的输入调用不同的函数。
3. **动态加载库：** 函数指针可用于在运行时动态加载库中的函数，实现动态链接库的调用。
4. **多态实现：** 在C++中，虚函数和函数指针结合使用，可以实现类似于多态的效果。
5. **函数指针作为参数：** 可以将函数指针作为参数传递给其他函数，实现一种可插拔的函数行为。
6. **实现函数映射表：** 在一些需要根据某些条件调用不同函数的情况下，可以使用函数指针来实现函数映射表。

函数指针和指针函数的区别

函数指针是指向函数的指针变量。可以存储特定函数的地址，并在运行时动态选择要调用的函数。通常用于回调函数、动态加载库时的函数调用等场景。

```
int add(int a, int b) {
    return a + b;
}

int (*ptr)(int, int) = &add;  // 函数指针指向 add 函数
int result = (*ptr)(3, 4);    // 通过函数指针调用函数
```

指针函数是一个返回指针类型的函数，用于返回指向某种类型的数据的指针。

```
int* getPointer() {
    int x = 10;
    return &x;  // 返回局部变量地址，不建议这样做
}
```

struct和Class的区别

- 通常，`struct`用于表示一组相关的数据，而`class`用于表示一个封装了数据和操作的对象,在实际使用中，可以根据具体的需求选择使用`struct`或`class`。如果只是用来组织一些数据，而不涉及复杂的封装和继承关系，`struct`可能更直观；如果需要进行封装、继承等面向对象编程的特性，可以选择使用`class`。
- struct结构体中的成员默认是公有的（public）。类中的成员默认是私有的（private）。
- **struct** 继承时默认使用公有继承。**class** 继承时默认使用私有继承。
- 如果结构体没有定义任何构造函数，编译器会生成默认的无参数构造函数。如果类没有定义任何构造函数，编

译器也会生成默认的无参数构造函数。

```
// 使用 struct 定义
struct MyStruct {
    int x;  // 默认是 public
    void print() {
        cout << "Struct method" << endl;
    }
};

// 使用 class 定义
class MyClass {
public:  // 如果省略，默认是 private
    int y;
    void display() {
        cout << "Class method" << endl;
    }
};
```

静态局部变量\全局变量\局部变量的区别和使用场景

1. 静态局部变量

- **作用域：** 限定在定义它的函数内。
- **生命周期：** 与程序的生命周期相同，但只能在定义它的函数内部访问。
- **关键字：** 使用 `static` 关键字修饰。
- **初始化：** 仅在第一次调用函数时初始化，之后保持其值。

当希望在函数调用之间保留变量的值，并且不希望其他函数访问这个变量时，可以使用静态局部变量

```
void exampleFunction() {
    static int count = 0;  // 静态局部变量
    count++;
    cout << "Count: " << count << endl;
}
```

2. 全局变量

- **作用域：** 整个程序。
- **生命周期：** 与程序的生命周期相同。
- **关键字：** 定义在全局作用域，不使用特定关键字。

当多个函数需要共享相同的数据时，可以使用全局变量。

```
int globalVar = 10;  // 全局变量

void function1() {
    globalVar++;
}

void function2() {
    globalVar--;
}
```

3. 局部变量

- **作用域：** 限定在定义它的块（大括号内）。
- **生命周期：** 在块结束时销毁。
- **关键字：** 定义在函数、语句块或类的成员函数中。

当变量只在某个特定作用域内有效，并且不需要其他作用域访问时，可以使用局部变量。

4. 总结

- 静态局部变量用于在函数调用之间保留变量的值。
- 全局变量适用于多个函数需要共享的数据。
- 局部变量适用于仅在特定作用域内有效的情况。

C++强制类型转换

关键字：static_cast、dynamic_cast、reinterpret_cast和 const_cast

1. static_cast

没有运行时类型检查来保证转换的安全性

进行上行转换（把派生类的指针或引用转换成基类表示）是安全的

进行下行转换（把基类的指针或引用转换为派生类表示），由于没有动态类型检查，所以是不安全的。

使用：

1. 用于基本数据类型之间的转换，如把int转换成char。
2. 把任何类型的表达式转换成void类型。

2. dynamic_cast

在进行下行转换时，dynamic_cast具有类型检查（信息在虚函数中）的功能，比static_cast更安全。

转换后必须是类的指针、引用或者void*，基类要有虚函数，可以交叉转换。

dynamic本身只能用于存在虚函数的父子关系的强制类型转换；对于指针，转换失败则返回nullptr，对于引用，转换失败会抛出异常。

3. reinterpret_cast

可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，平台移植性比价差。

4. const_cast

常量指针转换为非常量指针，并且仍然指向原来的对象。常量引用被转换为非常量引用，并且仍然指向原来的对象。去掉类型的const或volatile属性。

C++内存管理

堆和栈的区别

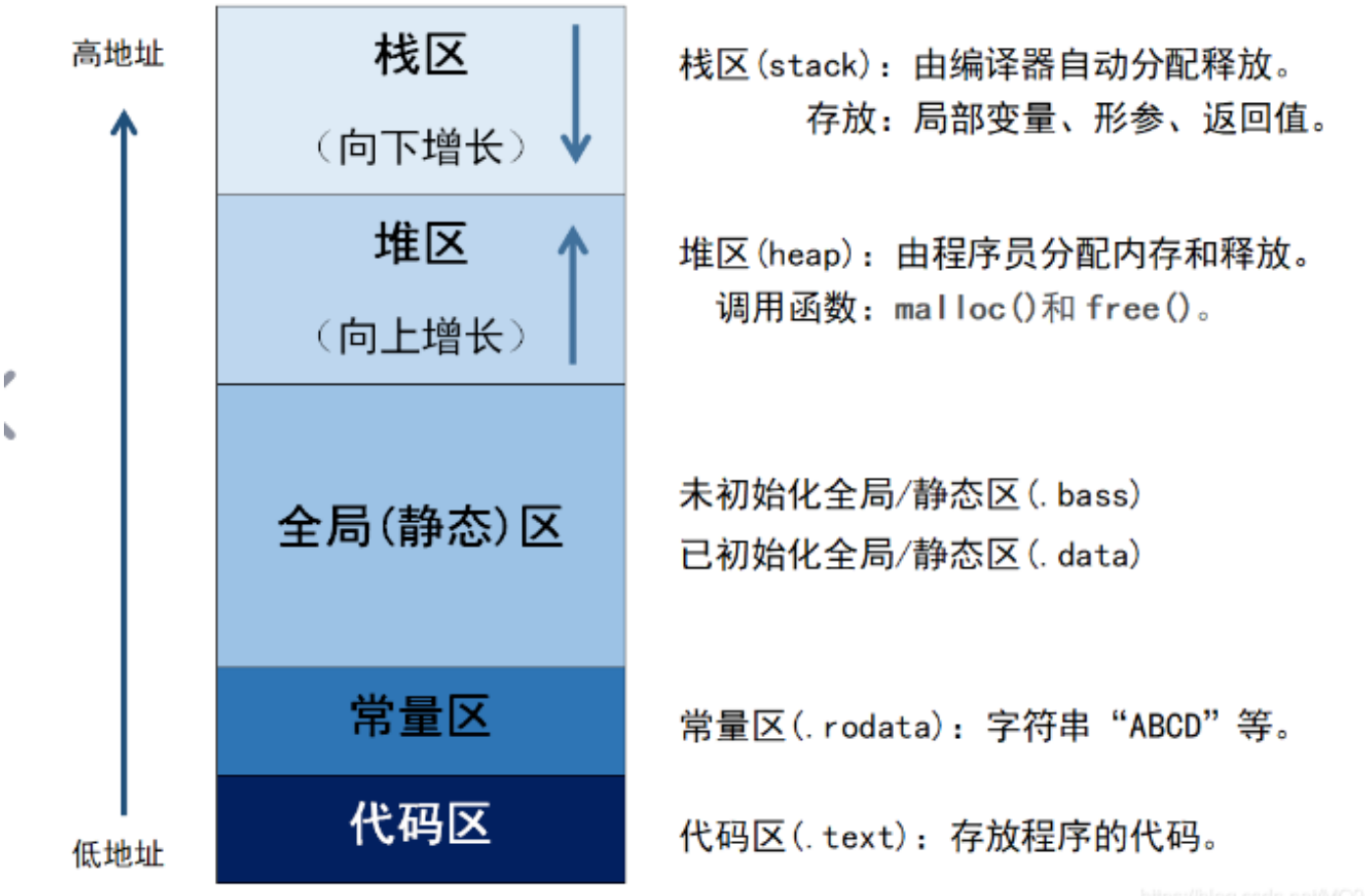
栈和堆都是用于存储程序数据的内存区域。栈是一种有限的内存区域，用于存储局部变量、函数调用信息等。堆是一种动态分配的内存区域，用于存储程序运行时动态分配的数据。

栈上的变量生命周期与其所在函数的执行周期相同，而堆上的变量生命周期由程序员显式控制，可以（使用 `new` 或 `malloc`）和释放（使用 `delete` 或 `free`）。

栈上的内存分配和释放是自动的，速度较快。而堆上的内存分配和释放需要手动操作，速度相对较慢。

C++内存分区

C++程序运行时，内存被分为几个不同的区域，每个区域负责不同的任务。



1. 栈

栈用于存储函数的局部变量、函数参数和函数调用信息的区域。函数的调用和返回通过栈来管理。

2. 堆

堆用于存储动态分配的内存的区域，由程序员手动分配和释放。使用 `new` 和 `delete` 或 `malloc` 和 `free` 来进行堆内存的分配和释放。

3. 全局/静态区

全局区存储全局变量和静态变量。生命周期是整个程序运行期间。在程序启动时分配，程序结束时释放。

4. 常量区

常量区也被称为只读区。存储常量数据，如字符串常量。

5. 代码区

存储程序的代码。

内存泄漏？如何避免？

1、什么是内存泄露?

内存泄漏(memory leak)是指由于疏忽或错误造成了**程序未能释放掉不再使用的内存的情况**。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

可以使用Valgrind, mtrace进行内存泄漏检查。

2、内存泄漏的分类

（1）堆内存泄漏 （Heap leak）

对内存指的是程序运行中根据需要分配通过malloc,realloc new等从堆中分配的一块内存，再是完成后必须通过调用对应的 free或者 delete 删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生 Heap Leak.

（2）系统资源泄露（Resource Leak）

主要指程序使用系统分配的资源比如 Bitmap,handle ,SOCKET 等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。

（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是释放，因此造成内存泄露。

3、什么操作会导致内存泄露?

指针指向改变，未释放动态分配内存。

4、如何防止内存泄露?

将内存的分配封装在类中，构造函数分配内存，析构函数释放内存；使用智能指针

5、智能指针有了解哪些?

智能指针是为了解决动态分配内存导致内存泄露和多次释放同一内存所提出的，C11标准中放在< memory>头文件。包括:共享指针，独占指针，弱指针

6、构造函数，析构函数要设为虚函数吗，为什么?

（1）析构函数

析构函数需要。当派生类对象中有内存需要回收时，如果析构函数不是虚函数，不会触发动态绑定，只会调用基类析构函数，导致派生类资源无法释放，造成内存泄漏。

（2）构造函数

构造函数不需要，没有意义。虚函数调用是在部分信息下完成工作的机制，允许我们只知道接口而不知道对象的确切类型。 要创建一个对象，你需要知道对象的完整信息。 特别是，你需要知道你想要创建的确切类型。 因此，构造函数不应该被定义为虚函数。

什么是智能指针？有哪些种类?

智能指针用于管理动态内存的对象，其主要目的是在避免内存泄漏和方便资源管理。

1. **`std::unique_ptr`** 独占智能指针

`std::unique_ptr` 提供对动态分配的单一对象所有权的独占管理。通过独占所有权，确保只有一个 `std::unique_ptr` 可以拥有指定的内存资源。移动语义和右值引用允许 `std::unique_ptr` 在所有权转移时高效地进行转移。

```
#include <memory>

std::unique_ptr<int> ptr = std::make_unique<int>(42);
```

2. **`std::shared_ptr`（共享智能指针）**：

`std::shared_ptr` 允许多个智能指针共享同一块内存资源。内部使用引用计数来跟踪对象被共享的次数，当计数为零时，资源被释放。提供更灵活的内存共享，但可能存在循环引用的问题。

```
#include <memory>

std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
std::shared_ptr<int> ptr2 = ptr1;
```

3. `std::weak_ptr`（弱引用智能指针）：

- `std::weak_ptr` 用于解决 `std::shared_ptr` 可能导致的循环引用问题。
- `std::weak_ptr` 可以从 `std::shared_ptr` 创建，但不会增加引用计数，不会影响资源的释放。
- 通过 `std::weak_ptr::lock()` 可以获取一个 `std::shared_ptr` 来访问资源。

```
#include <memory>

std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
std::weak_ptr<int> weakPtr = sharedPtr;
```

new 和 malloc 有什么区别?

类型安全性：

- `new` 是C++的运算符，可以为对象分配内存并调用相应的构造函数。
- `malloc` 是C语言库函数，只分配指定大小的内存块，不会调用构造函数。

返回类型：

- `new` 返回的是具体类型的指针，而且不需要进行类型转换。
- `malloc` 返回的是`void*`，需要进行类型转换，因为它不知道所分配内存的用途。

内存分配失败时的行为：

- `new` 在内存分配失败时会抛出 `std::bad_alloc` 异常。
- `malloc` 在内存分配失败时返回 `NULL`。

内存块大小：

- `new` 可以用于动态分配数组，并知道数组大小。
- `malloc` 只是分配指定大小的内存块，不了解所分配内存块的具体用途。

释放内存的方式：

- `delete` 会调用对象的析构函数，然后释放内存。
- `free` 只是简单地释放内存块，不会调用对象的析构函数。

delete 和 free 有什么区别?

类型安全性：

- `delete` 会调用对象的析构函数，确保资源被正确释放。
- `free` 不了解对象的构造和析构，只是简单地释放内存块。

内存块释放后的行为：

- `delete` 释放的内存块的指针值会被设置为 `nullptr`，以避免野指针。
- `free` 不会修改指针的值，可能导致野指针问题。

数组的释放：

- `delete` 可以正确释放通过 `new[]` 分配的数组。
- `free` 不了解数组的大小，不适用于释放通过 `malloc` 分配的数组。

什么是野指针，怎么产生的，如何避免

野指针是指指向已被释放的或无效的内存地址的指针。使用野指针可能导致程序崩溃、数据损坏或其他不可预测的行为。通常由以下几种情况产生

1. 释放后没有置空指针

```
int* ptr = new int;
delete ptr;
// 此时 ptr 成为野指针，因为它仍然指向已经被释放的内存
ptr = nullptr; // 避免野指针，应该将指针置为 nullptr 或赋予新的有效地址
```

2. 返回局部变量的指针

```
int* createInt() {
    int x = 10;
    return &x; // x 是局部变量，函数结束后 x 被销毁，返回的指针成为野指针
}
// 在使用返回值时可能引发未定义行为
```

3. 释放内存后没有调整指针

```
int* ptr = new int;
// 使用 ptr 操作内存
delete ptr;
// 此时 ptr 没有被置为 nullptr 或新的有效地址，成为野指针
// 避免：delete ptr; ptr = nullptr;
```

4. 函数参数指针被释放

```
void foo(int* ptr) {
    // 操作 ptr
    delete ptr;
}
int main() {
    int* ptr = new int;
    foo(ptr);
    // 在 foo 函数中 ptr 被释放，但在 main 函数中仍然可用，成为野指针
    // 避免：在 foo 函数中不要释放调用方传递的指针
}
```

如何避免野指针

- 在释放内存后将指针置为 nullptr

```
int* ptr = new int;
// 使用 ptr 操作内存
delete ptr;
ptr = nullptr;  // 避免成为野指针
```

- 避免返回局部变量的指针

```
int* createInt() {
    int* x = new int;
    *x = 10;
    return x;
}
```

- **使用智能指针（如 `std::unique_ptr` 和 `std::shared_ptr`）：**

```
#include <memory>
std::unique_ptr<int> ptr = std::make_unique<int>(42);
// 使用 std::unique_ptr，避免显式 delete，指针会在超出作用域时自动释放
```

- **注意函数参数的生命周期**， 避免在函数内释放调用方传递的指针，或者通过引用传递指针。

```
void foo(int*& ptr) {
    // 操作 ptr
    delete ptr; // 这里可能造成调用方的指针成为野指针
    ptr = nullptr;
}
int main() {
    int* ptr = new int;
    foo(ptr);
}
```

野指针和悬浮指针的区别

野指针是指向已经被释放或者无效的内存地址的指针。通常由于指针指向的内存被释放，但指针本身没有被置为 `nullptr` 或者重新分配有效的内存，导致指针仍然包含之前的内存地址。使用野指针进行访问会导致未定义行为，可能引发程序崩溃、数据损坏等问题。

悬浮指针是指向已经被销毁的对象的引用。当函数返回一个局部变量的引用，而调用者使用该引用时，就可能产生悬浮引用。访问悬浮引用会导致未定义行为，因为引用指向的对象已经被销毁，数据不再有效。

区别

关联对象类型：

- 野指针涉及指针类型。
- 悬浮指针涉及引用类型。

问题表现：

- 野指针可能导致访问已释放或无效内存，引发崩溃或数据损坏。
- 悬浮指针可能导致访问已销毁的对象，引发未定义行为。

产生原因：

- 野指针通常由于不正确管理指针生命周期引起。
- 悬浮指针通常由于在函数中返回局部变量的引用引起。

如何避免悬浮指针

- 避免在函数中返回局部变量的引用。
- 使用返回指针或智能指针而不是引用，如果需要在函数之外使用函数内部创建的对象。

内存对齐是什么？为什么需要考虑内存对齐?

1. 什么是内存对齐

内存对齐是指数据在内存中的存储起始地址是某个值的倍数。

在C语言中，结构体是一种复合数据类型，其构成元素既可以是基本数据类型（如int、long、float等）的变量，也可以是一些复合数据类型（如数组、结构体、联合体等）的数据单元。在结构体中，**编译器为结构体的每个成员按其自然边界（alignment）分配空间。**各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构体的地址相同。

为了使CPU能够对变量进行快速的访问，变量的起始地址应该具有某些特性，**即所谓的"对齐"，比如4字节的int型，其起始地址应该位于4字节的边界上，即起始地址能够被4整除**，也即"对齐"跟数据在内存中的位置有关。如果一个变量的内存地址正好位于它长度的整数倍，他就被称做自然对齐。

比如在32位cpu下，假设一个整型变量的地址为0x00000004(为4的倍数)，那它就是自然对齐的，而如果其地址为0x00000002（非4的倍数）则是非对齐的。现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

2. 为什么需要考虑内存对齐

需要字节对齐的根本原因在于**CPU访问数据的效率问题**。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。

而如果变量在自然对齐位置上，则只要一次就可以取出数据。一些系统对对齐要求非常严格，比如sparc系统，如果取未对齐的数据会发生错误，而在x86上就不会出现错误，只是效率下降。

各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数据。显然在读取效率上下降很多。

- 大多数计算机硬件要求基本数据类型的变量在内存中的地址是它们大小的倍数。例如，一个 32 位整数通常需要在内存中对齐到 4 字节边界。
- 内存对齐可以提高访问内存的速度。当数据按照硬件要求的对齐方式存储时，CPU可以更高效地访问内存，减少因为不对齐而引起的性能损失。
- 许多计算机体系结构使用缓存行（cache line）来从内存中加载数据到缓存中。如果数据是对齐的，那么一个

缓存行可以装载更多的数据，提高缓存的命中率。

- 有些计算机架构要求原子性操作（比如原子性读写）必须在特定的内存地址上执行。如果数据不对齐，可能导致无法执行原子性操作，进而引发竞态条件。

测试题目

1、以下为WindowsNT 32位C++程序，请计算下面sizeof的值

```
char str[] = "hello";
char* p = str;
int n = 10;
// 请计算
sizeof(str) = ?
sizeof(p) = ?
sizeof(n) = ?

void Func(char str[100])
{
   // 请计算
   sizeof(str) = ?
}
void* p = malloc(100);
// 请计算
sizeof(p) = ?
```

参考答案：

```
sizeof(str) = 6;
```

sizeof()计算的是数组的所占内存的大小包括末尾的 '\0'

```
sizeof(p) = 4;
```

p为指针变量，32位系统下大小为 4 bytes

```
sizeof(n) = 4;
```

n 是整型变量，占用内存空间4个字节

```
void Func(char str[100])
{
 sizeof(str) = 4;
}
```

函数的参数为字符数组名，即数组首元素的地址，大小为指针的大小

```
void* p = malloc(100);
sizeof(p) = 4;
```

p指向malloc分配的大小为100 byte的内存的起始地址，sizeof(p)为指针的大小，而不是它指向内存的大小

2、分析运行下面的Test函数会有什么样的结果

```
void GetMemory1(char* p)
{
    p = (char*)malloc(100);
}

void Test1(void)
{
    char* str = NULL;
    GetMemory1(str);
    strcpy(str, "hello world");
    printf(str);
}

char *GetMemory2(void)
{
    char p[] = "hello world";
    return p;
}

void Test2(void)
{
    char *str = NULL;
    str = GetMemory2();
    printf(str);
}

void GetMemory3(char** p, int num)
{
    *p = (char*)malloc(num);
}

void Test3(void)
{
    char* str = NULL;
    GetMemory3(&str, 100);
    strcpy(str, "hello");
    printf(str);
}

void Test4(void)
{
    char *str = (char*)malloc(100);
```

```
    strcpy(str, "hello");
    free(str);
    if(str != NULL) {
        strcpy(str, "world");
        cout << str << endl;
    }
}
```

参考答案：

Test1(void)：

程序崩溃。 因为GetMemory1并不能传递动态内存，Test1函数中的 str一直都是NULL。strcpy(str, "hello world") 将使程序奔溃

Test2(void)：

可能是乱码。 因为GetMemory2返回的是指向“栈内存”的指针，该指针的地址不是NULL，使其原现的内容已经被清除，新内容不可知。

Test3(void)：

能够输出hello, **内存泄露**。GetMemory3申请的内存没有释放

Test4(void)：

篡改动态内存区的内容，后果难以预料。非常危险。因为 free(str)；之后，str成为野指针，if（str != NULL)语句不起作用。

3、实现内存拷贝函数

```
char* strcpy(char* strDest, const char* strSrc);
```

参考答案：(函数实现)

```
char* strcpy(char *dst,const char *src) {// [1]
    assert(dst != NULL && src != NULL);  // [2]
    char *ret = dst;                     // [3]
    while ((*dst++=*src++)!='\0');       // [4]
    return ret;
}
```

[1] const修饰：

（1）源字符串参数用const修饰，防止修改源字符串。

[2] 空指针检查：

（1）不检查指针的有效性，说明答题者不注重代码的健壮性。

（2）检查指针的有效性时使用 `assert(!dst && !src);`

char *转换为 bool 即是类型隐式转换，这种功能虽然灵活，但更多的是导致出错概率增大和维护成本升高。

（3）检查指针的有效性时使用 `assert(dst != 0 && src != 0);`

直接使用常量（如本例中的0）会减少程序的可维护性。而使用NULL代替0，如果出现拼写错误，编译器就会检查出来。

[3] 返回目标地址：

（1）忘记保存原始的strdst值。

[4] '\0'：

（1）循环写成 `while (*dst++=*src++);` 明显是错误的。

（2）循环写成 `while (*src!='\0') *dst++ = *src++;`

循环体结束后，dst字符串的末尾没有正确地加上'\0'。

（3）为什么要返回char *?

返回dst的原始值使函数能够支持链式表达式

链式表达式的形式如：

`int l=strlen(strcpy(strA,strB));`

又如：

`char * strA=strcpy(new char[10],strB);`

返回strSrc的原始值是错误的。

理由：

1. 源字符串肯定是已知的，返回它没有意义
2. 不能支持形如第二例的表达式
3. 把`const char *作为char *`返回，类型不符，编译报错

4、假如考虑dst和src内存重叠的情况，strcpy该怎么实现

```
char s[10]="hello";

strcpy(s, s+1);
// 应返回 ello

strcpy(s+1, s);
// 应返回 hhello 但实际会报错
// 因为dst与src重叠了，把'\0'覆盖了
```

所谓重叠，就是src未处理的部分已经被dst给覆盖了，只有一种情况：`src<=dst<=src+strlen(src)`

C函数 memcpy 自带内存重叠检测功能，下面给出 memcpy 的实现my_memcpy

```
char * strcpy(char *dst,const char *src)
{
    assert(dst != NULL && src != NULL);
```

```
    char *ret = dst;
    my_memcpy(dst, src, strlen(src)+1);
    return ret;
}

/* my_memcpy的实现如下 */
char *my_memcpy(char *dst, const char* src, int cnt)
{
    assert(dst != NULL && src != NULL);
    char *ret = dst;
    /*内存重叠，从高地址开始复制*/
    if (dst >= src && dst <= src+cnt-1)
    {
        dst = dst+cnt-1;
        src = src+cnt-1;
        while (cnt--)
        {
            *dst-- = *src--;
        }
    }
    else  //正常情况，从低地址开始复制
    {
        while (cnt--)
        {
            *dst++ = *src++;
        }
    }
    return ret;
}
```

5、按照下面描述的要求写程序

已知String的原型为：

```
class String
{
public:
    String(const char *str = NULL);
    String(const String &other);
    ~ String(void);
    String & operate =(const String &other);
private:
  char *m_data;
};
```

请编写上述四个函数

参考答案：

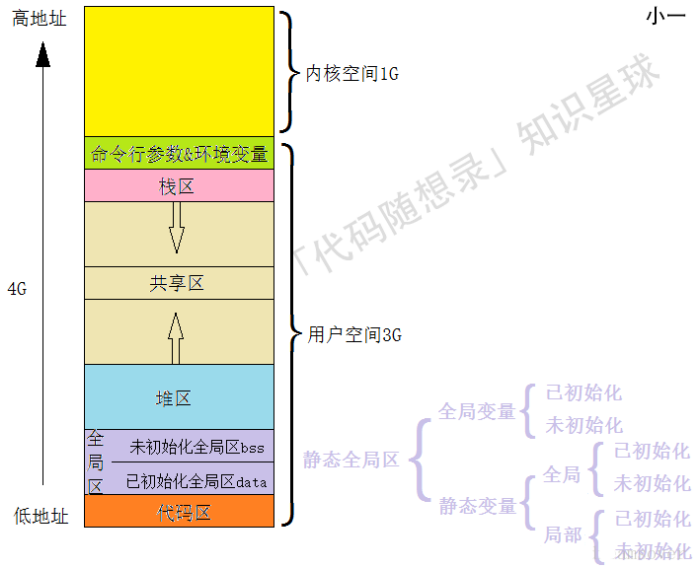
此题考察对构造函数赋值运算符实现的理解。实际考察类内含有指针的构造函数赋值运算符函数写法。

```
// 构造函数
String::String(const char *str)
{
    if(str==NULL)
    {
        m_data = new char[1];  //对空字符串自动申请存放结束标志'\0'
        *m_data = '\0';
    }
    else
    {
        int length = strlen(str);
        m_data = new char[length + 1];
        strcpy(m_data, str);
    }
}

// 析构函数
String::~String(void)
{
    delete [] m_data; // 或delete m_data;
}

//拷贝构造函数
String::String(const String &other)
{
    int length = strlen(other.m_data);
    m_data = new char[length + 1];
    strcpy(m_data, other.m_data);
}

//赋值函数
String &String::operate =(const String &other)
{
    if(this == &other)
    {
        return *this; // 检查自赋值
    }
    delete []m_data; // 释放原有的内存资源
    int length = strlen(other.m_data);
    m_data = new char[length + 1]; //对m_data加NULL判断
    strcpy(m_data, other.m_data);
    return *this; //返回本对象的引用
}
```

6、说一说进程的地址空间分布

参考答案：

对于一个进程，其空间分布如下图所示：



如上图，从高地址到低地址，一个程序由命令行参数和环境变量、栈、文件映射区、堆、BSS段、数据段、代码段组成。

（1）命令行参数和环境变量

命令行参数是指从命令行执行程序的时候，给程序的参数。

（2）栈区

存储局部变量、函数参数值。栈从高地址向低地址增长。是一块连续的空间。

（3）文件映射区

位于堆和栈之间。

（4）堆区

动态申请内存用。堆从低地址向高地址增长。

（5）BSS 段

存放程序中未初始化的 全局变量和静态变量 的一块内存区域。

（6）数据段

存放程序中已初始化的 全局变量和静态变量 的一块内存区域。

（7）代码段

存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。

7、说一说C与C++的内存分配方式

（1）从静态存储区域分配

内存在**程序编译的时候就已经分配好**，这块内存在程序的整个运行期间都存在，如全局变量，static变量。

（2）在栈上创建

在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元**自动被释放**。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从堆上分配(动态内存分配)

程序在运行的时候用malloc或new申请任意多少的内存，程序员负责在何时用free或delete释放内存。动态内存的**生存期自己决定**，使用非常灵活。

8、new、delete、malloc、free关系

参考答案：

如果是带有自定义析构函数的类类型，用 new [] 来创建类对象数组，而用 delete来释放会发生什么？用上面的例子来说明：

```
class A {};
A* pAa = new A[3];
delete pAa;
```

那么 delete pAa; 做了两件事：

1. 调用一次 pAa 指向的对象的析构函数
2. 调用 operator delete(pAa);释放内存

显然，这里只对数组的第一个类对象调用了析构函数，后面的两个对象均没调用析构函数，如果类对象中申请了大量的内存需要在析构函数中释放，而你却在销毁数组对象时少调用了析构函数，这会造成内存泄漏。

上面的问题你如果说没关系的话，那么第二点就是致命的了！直接释放pAa指向的内存空间，这个总是会造成严重的 段错误，程序必然会奔溃！因为分配的空间的起始地址是 pAa 指向的地方减去 4 个字节的地方。你应该传入参数设为那个地址！

计算机中的乱序执行

1、一定会按正常顺序执行的情况

1. 对同一块内存进行访问，此时访问的顺序不会被编译器修改
2. 新定义的变量的值依赖于之前定义的变量，此时两个变量定义的顺序不会被编译器修改

2、其他情况计算机会进行乱序执行

单线程的情况下允许，但是多线程情况下就会产生问题

3、C++中的库中提供了六种内存模型

用于在多线程的情况下防止编译器的乱序执行

（1）memory_order_relaxed

最放松的

（2）memory_order_consume

当客户使用，搭配release使用，被release进行赋值的变量y，获取的时候如果写成consume，那么所有与y有关的变量的赋值一定会被按顺序进行

（3）memory_order_acquire

用于获取资源

（4）memory_order_release

一般用于生产者，当给一个变量y进行赋值的时候，只有自己将这个变量释放了，别人才可以去读，读的时候如果使用acquire来读，编译器会保证在y之前被赋值的变量的赋值都在y之前被执行，相当于设置了内存屏障

（5）memory_order_acq_rel（acquire/release）

（6）memory_order_seq_cst（squentially consistent）

好处：不需要编译器设置内存屏障，morden c++开始就会有底层汇编的能力

副作用

1、无副作用编程

存在一个函数，传一个参数x进去，里面进行一系列的运算，返回一个y。中间的所有过程都是在栈中进行修改

2、有副作用编程

比如在一个函数运行的过程中对全局变量进行了修改或在屏幕上输出了一些东西。此函数还有可能是类的成员方法，在此方法中如果对成员变量进行了修改，类的状态就会发生改变

3、在多线程情况下的有副作用编程

在线程1运行的时候对成员变量进行了修改，此时如果再继续运行线程2，此时线程2拥有的就不是这个类的初始状态，运行出来的结果会收到线程1的影响

解决办法：将成员方法设为const，此时就可以放心进行调用

信号量

1、binary_semaphore

定义：

可以当事件来用，只有有信号和无信号两种状态，一次只能被一个线程所持有。

使用步骤：

（1）初始创建信号量，并且一开始将其置位成无信号状态

std::binary_semaphore sem(0)

（2）线程使用acquire()方法等待被唤醒

（3）主线程中使用release()方法，将信号量变成有信号状态

2、counting_semaphore

定义：

一次可以被很多线程所持有，线程的数量由自己指定

使用步骤：

（1）创建信号量

指定一次可以进入的线程的最大数量，并在最开始将其置位成无信号状态：std::biinary_semaphore<8> sem(0);

（2）主线程中创建10个线程

并且这些线程全部调用acquire()方法等待被唤醒。但是主线程使用release(6)方法就只能随机启用6个线程。

future库

用于任务链（即任务A的执行必须依赖于任务B的返回值）

1、例子：生产者消费者问题

（1）子线程作为消费者

参数是一个future，用这个future等待一个int型的产品：std::future& fut

（2）子线程中使用get()方法等待一个未来的future，返回一个result

（3）主线程作为生产者,做出一个承诺：std::promise prom

（4）用此承诺中的get_future()方法获取一个future

（5）主线程中将子线程创建出来,并将刚刚获取到的future作为参数传入

（6）主线程做一些列的生产工作,最后生产完后使用承诺中的set_value()方法，参数为刚刚生产出的产品

（7）此时产品就会被传到子线程中,子线程就可以使用此产品做一系列动作

（8）最后使用join()方法等待子线程停止,但是join只适用于等待没有返回值的线程的情况

2、如果线程有返回值

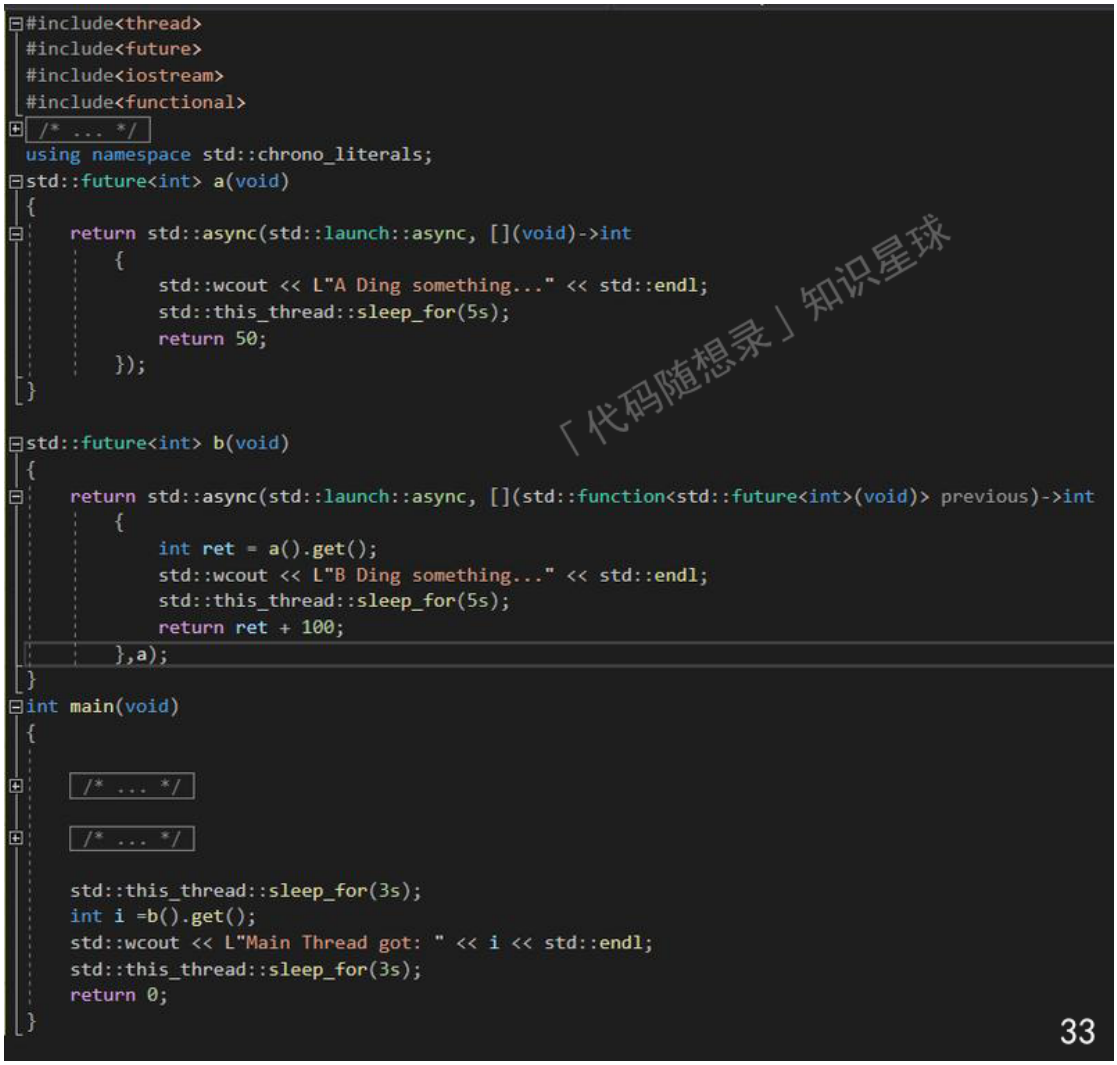
（1）使用async方法可以进行异步执行

参数一： 可以选择是马上执行还是等一会执行（即当消费者线程调用get()方法时才开始执行）

参数二： 执行的内容（可以放一个函数对象或lambda表达式）

（2）生产者使用async方法做生产工作并返回一个future

（3）消费者使用future中的get（）方法可以获取产品



字符串操作函数

常见的字符串函数实现

1、strcpy()

把从strsrc地址开始且含有'\0'结束符的字符串复制到以strdest开始的地址空间，返回值的类型为char*

```
1  char *strcpy( char *strDest, const char *strSrc )
2  {
3    assert( (strDest != NULL) && (strSrc != NULL) );
4    char *address = strDest;
5    while( (*strDest++ = *strSrc++) != '\0');
6    return address;
```

2、strlen()

计算给定字符串的长度。

```
int strlen( const char *str )
{
    assert( strt != NULL ); //断言字符串地址非0
    int len;
    while( (*str++) != '\0' )
    {
        len++;
    }
    return len;
}
```


3、strcat()

作用是把src所指字符串添加到dest结尾处。

```
// strcat的实现
char * strcat(char * dest, const char * src)
{
    assert(dest && src);
    char * ret = dest;
    // 找到dest的'\0'结尾符
    while(*dest)
    {
        dest++;
    }
    // 拷贝(while循环退出时，将结尾符'\0'也做了拷贝)
    while(*dest++ = *src++){}
    return ret;
}

```


4、strcmp()

比较两个字符串设这两个字符串为str1，str2，

若str1 == str2，则返回零
若str1 < str2，则返回负数
若str1 > str2，则返回正数

```cpp
// strcmp的实现
int strcmp(const char * str1, const char * str2)
{
    assert(str1 && str2);
    // 找到首个不相等的字符
    while(*str1 && *str2 && (*str1==*str2))
    {
        str1++;
        str2++;
    }
    return *str1 - *str2;
}

```


C++ 面向对象

面向对象的三大特性

访问权限

C++通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。

在类的内部（定义类的代码内部），无论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。

在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问 public 属性的成员，不能访问 private、protected 属性的成员。

无论共有继承、私有和保护继承，私有成员不能被“派生类”访问，基类中的共有和保护成员能被“派生类”访问。

对于共有继承，只有基类中的共有成员能被“派生类对象”访问，保护和私有成员不能被“派生类对象”访问。对于私有和保护继承，基类中的所有成员不能被“派生类对象”访问。

1. 继承

定义：

让某种类型对象获得另一个类型对象的属性和方法

功能：

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展

常见的继承有三种方式：

1、实现继承：指使用基类的属性和方法而无需额外编码的能力

2、接口继承：指仅使用属性和方法的名称、但是子类必须提供实现的能力

3、可视继承：指子窗体（类）使用基窗体（类）的外观和实现代码的能力

例如：

将人定义为一个抽象类，拥有姓名、性别、年龄等公共属性，吃饭、睡觉等公共方法，在定义一个具体的人时，就可以继承这个抽象类，既保留了公共属性和方法，也可以在此基础上扩展跳舞、唱歌等特有方法。

2. 封装

定义：

数据和代码捆绑在一起，避免外界干扰和不确定性访问;

功能：

把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏，例如：将公共的数据或方法使用public修饰，而不希望被访问的数据或方法采用private修饰。

3. 多态

定义：

同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为（重载实现编译时多态，虚函数实现运行时多态）

功能：

多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作;

简单一句话：允许将子类类型的指针赋值给父类类型的指针。

实现多态有两种方式

1. **覆盖（override）：** 是指子类重新定义父类的虚函数的做法
2. **重载（overload）：** 是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）

例如：

基类是一个抽象对象——人，那学生、运动员也是人，而使用这个抽象对象既可以表示学生、也可以表示运动员。

有哪些访问修饰符

C++提供了三个访问修饰符：`public`、`private` 和 `protected`。这些修饰符决定了类中的成员对外部代码的可见性和访问权限。

`public` 修饰符用于指定类中的成员可以被类的外部代码访问。公有成员可以被类外部的任何代码（包括类的实例）访问。

`private` 修饰符用于指定类中的成员只能被类的内部代码访问。私有成员对外部代码是不可见的，只有类内部的成员函数可以访问私有成员。

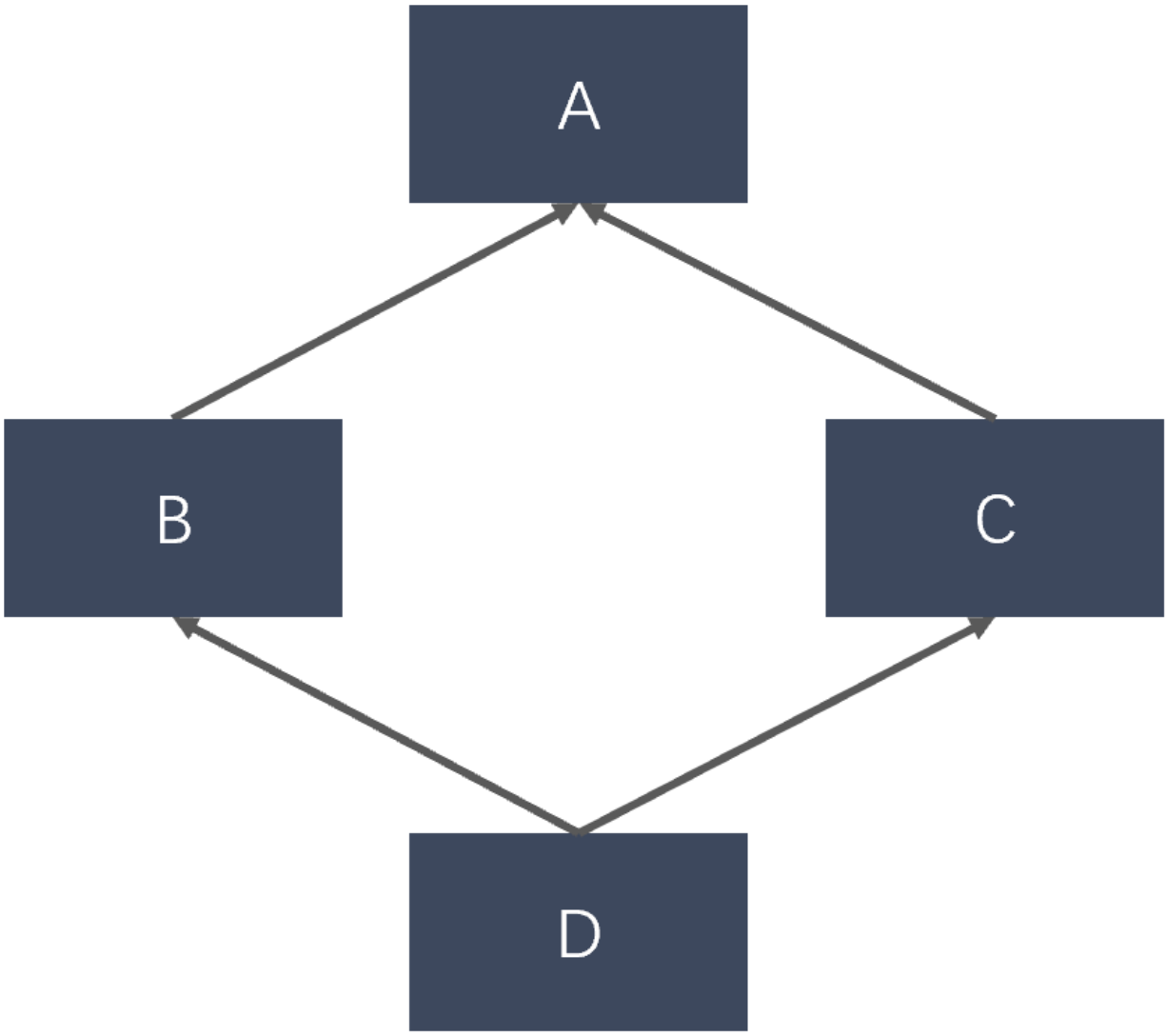
`protected` 修饰符用于指定类中的成员可以被类的派生类访问。受保护成员对外部代码是不可见的，但可以在派生类中被访问。

什么是多重继承?

一个类可以从多个基类（父类）继承属性和行为。在C++等支持多重继承的语言中，一个派生类可以同时拥有多个基类。

多重继承可能引入一些问题，如菱形继承问题, 比如当一个类同时继承了两个拥有相同基类的类，而最终的派生类又同时继承了这两个类时， 可能导致二义性和代码设计上的复杂性。为了解决这些问题，C++ 提供了虚继承, 通过在继承声明中使用 `virtual` 关键字，可以避免在派生类中生成多个基类的实例，从而解决了菱形继承带来的二义性。

```
#include <iostream>

class Animal {
public:
    void eat() {
        std::cout << "Animal is eating." << std::endl;
    }
};

class Mammal : public Animal {
public:
    void breathe() {
        std::cout << "Mammal is breathing." << std::endl;
    }
};

class Bird : public Animal {
public:
    void fly() {
        std::cout << "Bird is flying." << std::endl;
    }
};

// 菱形继承，同时从 Mammal 和 Bird 继承
class Bat : public Mammal, public Bird {
public:
    void navigate() {
        // 这里可能会引起二义性，因为 Bat 继承了两个 Animal
        // navigate 方法中尝试调用 eat 方法，但不明确应该调用 Animal 的哪一个实现
        eat();
    }
};

int main() {
    Bat bat;
    bat.navigate();

    return 0;
}
```




```cpp
#include <iostream>

class Animal {
public:
    void eat() {
        std::cout << "Animal is eating." << std::endl;
    }
};

class Mammal : virtual public Animal {
public:
    void breathe() {
        std::cout << "Mammal is breathing." << std::endl;
    }
};
```

```
class Bird : virtual public Animal {
public:
    void fly() {
        std::cout << "Bird is flying." << std::endl;
    }
};

class Bat : public Mammal, public Bird {
public:
    void navigate() {
        // 不再存在二义性，eat 方法来自于共享的 Animal 基类
        eat();
    }
};

int main() {
    Bat bat;
    bat.navigate();

    return 0;
}
```

简述一下 C++ 的重载和重写，以及它们的区别

1. 重载是指在同一作用域内，使用相同的函数名但具有不同的参数列表或类型，使得同一个函数名可以有多个版本。

```
int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}
```

2. 重写是指派生类（子类）重新实现（覆盖）基类（父类）中的虚函数，以提供特定于派生类的实现。重写是面向对象编程中的多态性的一种体现，主要涉及基类和派生类之间的关系，用于实现运行时多态。

```
class Base {
public:
    virtual void print() {
        cout << "Base class" << endl;
    }
};

class Derived : public Base {
public:
    void print() override {
        cout << "Derived class" << endl;
    }
};
```

1、override是重写（覆盖）了一个方法

以实现不同的功能，一般是用于子类在继承父类时，重写父类方法。

规则：

1. 重写方法的参数列表，返回值，所抛出的异常与被重写方法一致
2. 被重写的方法不能为private
3. 静态方法不能被重写为非静态的方法
4. 重写方法的访问修饰符一定要大于被重写方法的访问修饰符（public>protected>default>private）

2、overload是重载，这些方法的名称相同而参数形式不同

一个方法有不同的版本，存在于一个类中。

规则：

1. 不能通过访问权限、返回类型、抛出的异常进行重载
2. 不同的参数类型可以是不同的参数类型，不同的参数个数，不同的参数顺序（参数类型必须不一样）
3. 方法的异常类型和数目不会对重载造成影响

使用多态是为了避免在父类里大量重载引起代码臃肿且难于维护。

重写与重载的本质区别是,加入了override的修饰符的方法,此方法始终只有一个被你使用的方法。

c++的多态如何实现

C++中的多态性是通过虚函数（virtual function）和虚函数表（vtable）来实现的。多态性允许在基类类型的指针或引用上调用派生类对象的函数，以便在运行时选择正确的函数实现。

1. 基类声明虚函数：在基类中声明虚函数，使用 `virtual` 关键字，以便派生类可以重写（override）这些函数。

```
class Shape {
public:
    virtual void draw() const {
        // 基类的默认实现
    }
};
```

2. 派生类重写虚函数:在派生类中重写基类中声明的虚函数，使用 `override` 关键字

```
class Circle : public Shape {
public:
    void draw() const override {
        // 派生类的实现
    }
};
```

3. 使用基类类型的指针或引用指向派生类对象。

```
Shape* shapePtr = new Circle();
```

4. 调用虚函数：通过基类指针或引用调用虚函数。在运行时，系统会根据对象的实际类型来选择调用正确的函数实现。

```
shapePtr->draw();  // 调用的是 Circle 类的 draw() 函数
```

5. 虚函数表：编译器在对象的内存布局中维护了一个虚函数表，其中存储了指向实际函数的指针。这个表在运行时用于动态查找调用的函数。

成员函数/成员变量/静态成员函数/静态成员变量的区别

1. 成员函数

- 成员函数是属于类的函数，它们可以访问类的成员变量和其他成员函数。
- 成员函数可以分为普通成员函数和静态成员函数。
- 普通成员函数使用对象调用，可以访问对象的成员变量。
- 普通成员函数的声明和定义通常在类的内部，但定义时需要使用类名作为限定符。

2. 成员变量

- 成员变量是属于类的变量，存储在类的每个对象中。
- 每个对象拥有一份成员变量的副本，它们在对象创建时分配，并在对象销毁时释放。
- 成员变量的访问权限可以是 `public`、`private` 或 `protected`。

```
class MyClass {
public:
    int memberVariable;  // 成员变量的声明
    void memberFunction() {
        // 成员函数的实现
    }
};
```

3. 静态成员函数

- 静态成员函数属于类而不是对象，因此可以直接通过类名调用，而不需要创建类的实例。
- 静态成员函数不能直接访问普通成员变量，因为它们没有隐含的 `this` 指针。
- 静态成员函数的声明和定义也通常在类的内部，但在定义时需要使用类名作为限定符。

4. 静态成员变量

- 静态成员变量是属于类而不是对象的变量，它们在所有对象之间共享。
- 静态成员变量通常在类的声明中进行声明，但在类的定义外进行定义和初始化。
- 静态成员变量可以通过类名或对象访问。

```
class MyClass {
public:
    static int staticMemberVariable;  // 静态成员变量的声明
    static void staticMemberFunction() {
        // 静态成员函数的实现
    }
};

int MyClass::staticMemberVariable = 0;  // 静态成员变量的定义和初始化
```

什么是构造函数和析构函数?

1. 构造函数

构造函数是在创建对象时自动调用的特殊成员函数。它的主要目的是初始化对象的成员变量，为对象分配资源，执行必要的初始化操作。构造函数的特点包括：

- **函数名与类名相同：** 构造函数的函数名必须与类名相同，且没有返回类型，包括 void。
- **可以有多个构造函数：** 一个类可以有多个构造函数，它们可以根据参数的类型和数量不同而重载。
- **默认构造函数：** 如果没有为类定义任何构造函数，编译器会自动生成一个默认构造函数。默认构造函数没有参数，也可能执行一些默认的初始化操作。

```
class MyClass {
public:
    // 默认构造函数
    MyClass() {
        // 初始化操作
    }

    // 带参数的构造函数
    MyClass(int value) {
        // 根据参数进行初始化操作
    }
};
```

2. 析构函数

析构函数是在对象生命周期结束时自动调用的特殊成员函数。它的主要目的是释放对象占用的资源、执行必要的清理操作。析构函数的特点包括：

- **函数名与类名相同，前面加上波浪号 `~`：** 析构函数的函数名为 `~ClassName`，其中 `ClassName` 是类名。
- **没有参数：** 析构函数没有参数，不能重载，每个类只能有一个析构函数。
- **默认析构函数：** 如果没有为类定义任何析构函数，编译器会自动生成一个默认析构函数，执行简单的清理操作。

```
class MyClass {
public:
    // 析构函数
    ~MyClass() {
        // 清理操作，释放资源
    }
};
```

C++构造函数有几种，分别什么作用

1. 默认构造函数：没有任何参数的构造函数。如果用户没有为类定义构造函数，编译器会自动生成一个默认构造函数。默认构造函数用于创建对象时的初始化，当用户不提供初始化值时，编译器将调用默认构造函数。

```
class MyClass {
public:
    // 默认构造函数
    MyClass() {
        // 初始化操作
    }
};
```

2. 带参数的构造函数：接受一个或多个参数，用于在创建对象时传递初始化值。可以定义多个带参数的构造函数，以支持不同的初始化方式。

```
class MyClass {
public:
    // 带参数的构造函数
    MyClass(int value) {
        // 根据参数进行初始化操作
    }
};
```

3. 拷贝构造函数：用于通过已存在的对象创建一个新对象，新对象是原对象的副本。参数通常是对同类型对象的引用。

```
class MyClass {
public:
    // 拷贝构造函数
    MyClass(const MyClass &other) {
        // 进行深拷贝或浅拷贝，根据实际情况
    }
};
```

4. 委托构造函数：在一个构造函数中调用同类的另一个构造函数，减少代码重复。通过成员初始化列表或构造函数体内部调用其他构造函数。

```
class MyClass {
public:
    // 委托构造函数
    MyClass() : MyClass(42) {
        // 委托给带参数的构造函数
    }

    MyClass(int value) {
        // 进行初始化操作
    }
};
```

什么是虚函数和虚函数表?

1. 虚函数

C++中的虚函数的作用主要是实现了多态的机制。虚函数允许在派生类中重新定义基类中定义的函数，使得通过基类指针或引用调用的函数在运行时根据实际对象类型来确定。这样的机制被称为动态绑定或运行时多态。

在基类中，通过在函数声明前面加上 `virtual` 关键字，可以将其声明为虚函数。派生类可以重新定义虚函数，如果派生类不重新定义，则会使用基类中的实现。

```
class Base {
public:
    virtual void virtualFunction() {
        // 虚函数的实现
    }
};

class Derived : public Base {
public:
    void virtualFunction() override {
        // 派生类中对虚函数的重新定义
    }
};
```

2. 虚函数表

虚函数的实现通常依赖于一个被称为虚函数表（虚表）的数据结构。每个类（包括抽象类）都有一个虚表，其中包含了该类的虚函数的地址。每个对象都包含一个指向其类的虚表的指针，这个指针被称为虚指针（vptr）。

当调用一个虚函数时，编译器会使用对象的虚指针查找虚表，并通过虚表中的函数地址来执行相应的虚函数。这就是为什么在运行时可以根据实际对象类型来确定调用哪个函数的原因。

虚函数和纯虚函数的区别

1. 虚函数

- **有实现：** 虚函数有函数声明和实现，即在基类中可以提供默认实现。
- **可选实现：** 派生类可以选择是否覆盖虚函数。如果派生类没有提供实现，将使用基类的默认实现。
- **允许实例化：** 虚函数的类可以被实例化。即你可以创建一个虚函数的类的对象。
- **调用靠对象类型决定：** 在运行时，根据对象的实际类型来决定调用哪个版本的虚函数。
- **用 `virtual` 关键字声明：** 虚函数使用 `virtual` 关键字声明，但不包含 `= 0`。

```
class Base {
public:
    // 虚函数有实现
    virtual void virtualFunction() {
        // 具体实现
    }
};
```

2. 纯虚函数

- **没有实现：** 纯虚函数没有函数体，只有函数声明，即没有提供默认的实现。
- **强制覆盖：** 派生类必须提供纯虚函数的具体实现，否则它们也会成为抽象类。
- **禁止实例化：** 包含纯虚函数的类无法被实例化，只能用于派生其他类。
- **用 `= 0` 声明：** 纯虚函数使用 `= 0` 在函数声明末尾进行声明。
- **为接口提供规范：** 通过纯虚函数，抽象类提供一种接口规范，要求派生类提供相关实现。

```
class AbstractBase {
public:
    // 纯虚函数，没有具体实现
    virtual void pureVirtualFunction() = 0;

    // 普通成员函数可以有具体实现
    void commonFunction() {
        // 具体实现
    }
};
```

什么是抽象类和纯虚函数?

抽象类是不能被实例化的类，它存在的主要目的是为了提供一个接口，供派生类继承和实现。抽象类中可以包含普通的成员函数、数据成员和构造函数，但它必须包含至少一个纯虚函数。即在声明中使用 `virtual` 关键字并赋予函数一个 `= 0` 的纯虚函数。

```
class AbstractShape {
public:
    // 纯虚函数，提供接口
    virtual void draw() const = 0;

    // 普通成员函数
    void commonFunction() {
        // 具体实现
    }
};
```

纯虚函数是在抽象类中声明的虚函数，它**没有具体的实现，只有函数的声明**。通过在函数声明的末尾使用 `= 0`，可以将虚函数声明为纯虚函数。**派生类必须实现抽象类中的纯虚函数，否则它们也会成为抽象类。**

```
class AbstractShape {
public:
    // 纯虚函数
    virtual void draw() const = 0;
};
```

简述一下虚析构函数，什么作用

虚析构函数是一个带有 `virtual` 关键字的析构函数。 主要作用是确保在通过基类指针删除派生类对象时，能够正确调用派生类的析构函数，从而释放对象所占用的资源。

通常，如果一个类可能被继承，且在其派生类中有可能使用 `delete` 运算符来删除通过基类指针指向的对象，那么该基类的析构函数应该声明为虚析构函数。

```
class Base {
public:
    // 虚析构函数
    virtual ~Base() {
        // 基类析构函数的实现
    }
};

class Derived : public Base {
public:
    // 派生类析构函数，可以覆盖基类的虚析构函数
    ~Derived() override {
        // 派生类析构函数的实现
    }
};
```

说说为什么要虚析构，为什么不能虚构造

为什么需要虚析构函数?

- 虚析构函数允许在运行时根据对象的实际类型调用正确的析构函数，从而实现多态性。
- 如果基类的析构函数不是虚的，当通过基类指针删除指向派生类对象的对象时，只会调用基类的析构函数，而不会调用派生类的析构函数。这可能导致派生类的资源未被正确释放，造成内存泄漏。

构造函数在对象的创建阶段被调用，对象的类型在构造函数中已经确定。因此，构造函数调用不涉及多态性，也就是说，在对象的构造期间无法实现动态绑定。虚构造函数没有意义，因为对象的类型在构造过程中就已经确定，不需要动态地选择构造函数。

1. 从存储空间角度：虚函数对应一个vtale,这个表的地址是存储在对象的内存空间的。如果将构造函数设置为虚函数，就需要到vtable 中调用，可是对象还没有实例化，没有内存空间分配，如何调用。（悖论）
2. 从使用角度：虚函数主要用于在信息不全的情况下，能使重载的函数得到对应的调用。构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。
3. 从实现上看，vbtl 在构造函数调用后才建立，因而构造函数不可能成为虚函数。从实际含义上看，在调用构造函数时还不能确定对象的真实类型（因为子类会调父类的构造函数）；而且构造函数的作用是提供初始化，在对象生命期只执行一次，不是对象的动态行为，也没有太大的必要成为虚函数。

```
class Base {
public:
    // 错误！不能声明虚构造函数
    virtual Base() {
        // 虚构造函数的实现
    }

    virtual ~Base() {
        // 基类析构函数的实现
    }
};
```

哪些函数不能被声明为虚函数?

常见的不不能声明为虚函数的有：普通函数（非成员函数），静态成员函数，内联成员函数，构造函数，友元函数。

- 构造函数：

构造函数在对象的创建期间调用，对象的类型在构造期间已经确定。因此，构造函数不能是虚函数，因为虚函数的动态绑定是在运行时实现的，而构造函数在对象还未创建完全时就会被调用。

- 普通函数

普通函数（非成员函数）只能被overload，不能被override，声明为虚函数也没有什么意思，因此编译器会在编译时绑定函数。

- 静态成员函数

静态成员函数对于每个类来说只有一份代码，所有的对象都共享这一份代码，他也没有要动态绑定的必要性。

- 友元函数

因为C++不支持友元函数的继承，对于没有继承特性的函数没有虚函数的说法。

- 内联成员函数

内联函数就是为了在代码中直接展开，减少函数调用花费的代价，虚函数是为了在继承后对象能够准确的执行自己的动作，这是不可能统一的。（再说了，*inline函数在编译时被展开，虚函数在运行时才能动态的绑定函数*）

内联函数是在编译时期展开,而虚函数的特性是运行时才动态联编,所以两者矛盾,不能定义内联函数为虚函数

深拷贝和浅拷贝的区别

主要区别在于如何处理对象内部的动态分配的资源。

1. 深拷贝

深拷贝是对对象的完全独立复制，包括对象内部动态分配的资源。在深拷贝中，不仅复制对象的值，还会复制对象所指向的堆上的数据。

主要特点：

- 复制对象及其所有成员变量的值。

- 动态分配的资源也会被复制，新对象拥有自己的一份资源副本。

深拷贝通常涉及到手动分配内存，并在拷贝构造函数或赋值操作符中进行资源的复制。

```
class DeepCopyExample {
public:
    int *data;

    DeepCopyExample(const DeepCopyExample &other) {
        // 手动分配内存并复制数据
        data = new int(*(other.data));
    }

    ~DeepCopyExample() {
        // 释放动态分配的资源
        delete data;
    }

    DeepCopyExample& operator=(const DeepCopyExample &other) {
        // 复制数据
        if (this != &other) {
            delete data;
            data = new int(*(other.data));
        }
        return *this;
    }
};
```

2. 浅拷贝

浅拷贝仅复制对象的值，而不涉及对象内部动态分配的资源。在浅拷贝中，新对象和原对象共享相同的资源，而不是复制一份新的资源。

主要特点：

- 复制对象及其所有成员变量的值。
- 对象内部动态分配的资源不会被复制，新对象和原对象共享同一份资源。

浅拷贝通常使用默认的拷贝构造函数和赋值操作符，因为它们会逐成员地复制原对象的值。

```
class ShallowCopyExample {
public:
    int *data;

    // 使用默认拷贝构造函数和赋值操作符
};
```

运算符重载

重载运算符函数，本质还是函数调用，所以重载后：

（1）可以是和调用运算符的方式调用，data1+data2

（2）也可以是调用函数的方式，operator+(data1, data2)，这就要注意运算符函数的名字是“operator运算符”

在可以重载的运算符里有逗号、取地址、逻辑与、逻辑或

不建议重载：

逗号、取地址，本身就对类类型有特殊定义；逻辑与、逻辑或，有短路求值属性；逗号、逻辑与、或，定义了求值顺序。

运算符重载应该是作为类的成员函数or非成员函数（具体后面各小节会涉及）。有个对应知识点，

注意：

重载运算符，**它本身是几元就有几个参数**，对于二元的，第一个参数对应左侧运算对象，第二个参数对应右侧运算对象。而！类的成员函数的第一个参数隐式绑定了this指针，所以重载运算符如果是类的成员函数，左侧运算对象就相当于固定了是this。

建议非成员：

又因为要访问类的私有成员，多为类的友元。返回值iostream的引用，第一个参数iostream的引用，第二个参数，输出用const、输入非常量。输入的重载里注意判断是否成功，避免输入了不合预期的内容。

一些规则：

（1）算术和关系运算符建议非成员

因为这些运算符是对称性的，形参都是常量引用

（2）赋值运算符必须成员。复合赋值运算符建议成员

（3）下标运算符必须成员

返回访问元素的引用，建议两版本（常量、非常量）

（4）递增递减运算符，建议成员

因其会改变对象状态，后置与前置的区分——接受一个额外的不被使用的int类型形参，前置返回变后的对象引用，后置返回对象的原值（非引用）；解引用（*）建议成员，因其与给定类型关系密切，箭头（->）必须成员。

函数调用运算符：

lambda是函数对象。编译器是将lambda表达式翻译为一个未命名类的未命名对象，‘[’捕获列表‘]’(参数列表){函数体} 对应类中重载调用运算符的参数列表、函数体，捕获列表的内容就对应类中的数据成员。所以捕获列表，值传递时，要拷贝并初始化那些数据成员，引用传递就是直接用。

C++ STL

什么是STL，包含哪些组件

广义上讲，STL分为3类：Algorithm（算法）、Container（容器）和Iterator（迭代器），容器和算法通过迭代器可以进行无缝地连接。

详细的说，STL由6部分组成：容器(Container)、算法（Algorithm）、 迭代器（Iterator）、仿函数（Function object）、适配器（Adaptor）、空间配制器（Allocator）。

STL提供了六大组件，彼此之间可以组合套用，这六大组件分别是:容器、算法、迭代器、仿函数、适配器（配接器）、空间配置器。

STL六大组件的交互关系：

1. 容器通过空间配置器取得数据存储空间
2. 算法通过迭代器存储容器中的内容
3. 仿函数可以协助算法完成不同的策略的变化
4. 适配器可以修饰仿函数。

容器

各种数据结构，如vector、list、deque、set、map等，用来存放数据，从实现角度来看，STL容器是一种class template。

算法

各种常用的算法，如sort、find、copy、for_each。从实现的角度来看，STL算法是一种function tempalte.

迭代器

扮演了容器与算法之间的胶合剂，共有五种类型，从实现角度来看，迭代器是一种将operator* , operator-> , operator++,operator–等指针相关操作予以重载的class template.。

所有STL容器都附带有自己专属的迭代器，只有容器的设计者才知道如何遍历自己的元素。

原生指针(native pointer)也是一种迭代器。

仿函数

行为类似函数，可作为算法的某种策略。从实现角度来看，仿函数是一种重载了operator()的class 或者class template

适配器

一种用来修饰容器或者仿函数或迭代器接口的东西。

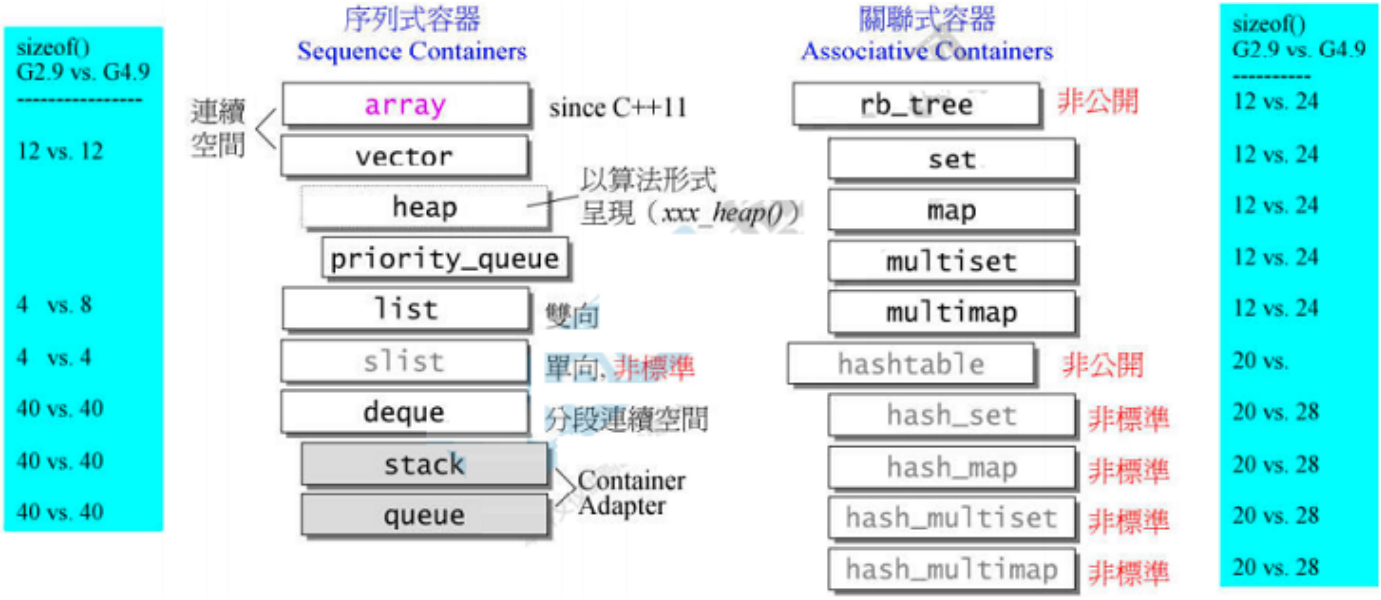
STL提供的queue 和 stack，虽然看似容器，但其实只能算是一种容器配接器，因为它们的底部完全借助deque，所有操作都由底层的deque供应。

空间配置器

负责空间的配置与管理。从实现角度看，配置器是一个实现了动态空间配置、空间管理、空间释放的class tempalte.

一般的分配器的std:alloctor都含有两个函数allocate与deallocte，这两个函数分别调用operator new()与delete()，这两个函数的底层又分别是malloc()and free();但是每次malloc会带来格外开销（因为每次malloc一个元素都要带有附加信息）

容器之间的实现关系以及分类：



STL的优点

STL 具有高可重用性，高性能，高移植性，跨平台的优点。

1、高可重用性：

STL 中几乎所有的代码都采用了模板类和模版函数的方式实现，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。

2、高性能：

如 map 可以高效地从十万条记录里面查找出指定的记录，因为 map 是采用红黑树的变体实现的。

3、高移植性：

如在项目 A 上用 STL 编写的模块，可以直接移植到项目 B 上。

STL 的一个重要特性是将数据和操作分离

数据由容器类别加以管理，操作则由可定制的算法定义。迭代器在两者之间充当“**粘合剂**”,以使算法可以和容器交互运作。

常见的STL容器

1. 序列容器

vector（向量）： `std::vector` 是一个动态数组实现，提供高效的随机访问和在尾部进行插入/删除操作。

list（链表）： `std::list` 是一个双向链表实现，支持在任意位置进行插入/删除操作，但不支持随机访问。

deque（双端队列）： `std::deque` 是一个双端队列实现，允许在两端进行高效插入/删除操作。

array（数组）： `std::array` 是一个固定大小的数组实现，提供对数组元素的高效随机访问。

forward_list（前向链表）： `std::forward_list` 是一个单向链表实现，只能从头到尾进行遍历，不支持双向访问。

2. 关联容器

set（集合）： `std::set` 是一个有序的集合，不允许重复元素，支持快速查找、插入和删除。

multiset（多重集合）： `std::multiset` 是一个有序的多重集合，允许重复元素。

map（映射）： `std::map` 是一个有序的键值对集合，不允许重复的键，支持快速查找、插入和删除。

multimap（多重映射）： `std::multimap` 是一个有序的多重映射，允许重复的键。

unordered_set（无序集合）： `std::unordered_set` 是一个无序的集合，不允许重复元素，支持快速查找、插入和删除。

unordered_multiset（无序多重集合）： `std::unordered_multiset` 是一个无序的多重集合，允许重复元素。

unordered_map（无序映射）： `std::unordered_map` 是一个无序的键值对集合，不允许重复的键，支持快速查找、插入和删除。

unordered_multimap（无序多重映射）： `std::unordered_multimap` 是一个无序的多重映射，允许重复的键。

3. 容器适配器

stack（栈）： `std::stack` 是一个基于底层容器的栈实现，默认使用 `std::deque`。

queue（队列）： `std::queue` 是一个基于底层容器的队列实现，默认使用 `std::deque`。

priority_queue（优先队列）： `std::priority_queue` 是一个基于底层容器的优先队列实现，默认使用 `std::vector`。

pair容器

保存两个数据成员，用来生成特定类型的模板。

使用： `pair<T1, T2>p;`

内部定义

```
namespace std {
    template <typename T1, typename T2>
    struct pair {
        T1 first;
        T2 second;
    };
}
```

pair在底层被定义为一个struct，其所有成员默认是public，两个成员分别是first和second

其中map的元素是pair，pair<const key_type，mapped_type>

可以用来遍历关联容器

```
map<string,int>p;
auto map1 = p.cbegin();
while(map1 != p.cend())
{
  cout<<map1->first<<map1->second<<endl;
   ++map1;
}
```

对map进行插入，元素类型是pair：

```
p.insert({word, 1});
p.insert(pair<string, int>(word, 1));
```

insert对不包含重复关键字的容器，插入成功返回pair<迭代器，bool>迭代器指向给定关键字元素，bool指出插入是否成功。

```
 vector<pair<char, int>>result(val.begin(), val.end());
 sort(result.begin(), result.end(),[](auto &a, auto &b){
     return a.second > b.second;
});
```

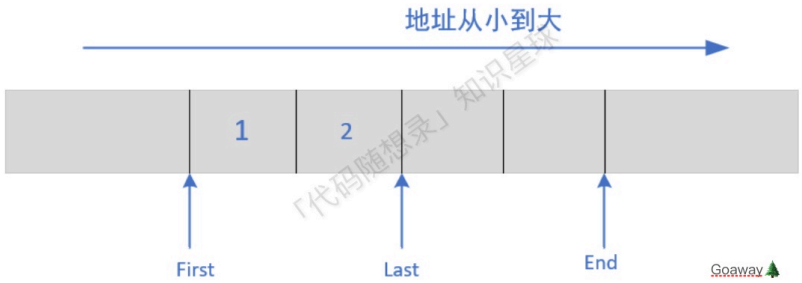
vector容器实现与扩充

1. 底层实现

Vector在堆中分配了一段连续的内存空间来存放元素

1、三个迭代器

（1）first： 指向的是vector中对象的起始字节位置
（2）last ： 指向当前最后一个元素的末尾字节
（3）end ： 指向整个vector容器所占用内存空间的末尾字节



（1）**last - first**： 表示 vector 容器中目前已被使用的内存空间
（2）**end - last** ： 表示 vector 容器目前空闲的内存空间
（3）**end - first**： 表示 vector 容器的容量

2. 扩容过程

如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素

所以对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

size() 和 capacity()

（1）堆中分配内存，元素连续存放，内存空间只会增长不会减少

vector有两个函数，一个是capacity()，在不分配新内存下最多可以保存的元素个数，另一个size()，返回当前已经存储数据的个数

（2）对于vector来说，capacity是永远大于等于size的

capacity和size相等时，vector就会扩容，capacity变大（翻倍）

这里涉及到了vector扩容方式的选择，新增的容量选择多少才适宜呢?

1、固定扩容

机制:

每次扩容的时候在原 capacity 的基础上加上固定的容量，比如初始 capacity 为100，扩容一次为 capacity + 20，再扩容仍然为 capacity + 20;

缺点:

考虑一种极端的情况，vector每次添加的元素数量刚好等于每次扩容固定增加的容量 + 1，就会造成一种情况，每添加一次元素就需要扩容一次，而扩容的时间花费十分高昂。所以固定扩容可能会面临多次扩容的情况，时间复杂度较高;

优点:

固定扩容方式空间利用率比较高。

2、加倍扩容

机制:

每次扩容的时候原 capacity 翻倍，比如初始capcity = 100, 扩容一次变为 200, 再扩容变为 400;

优点:

一次扩容 capacity 翻倍的方式使得正常情况下添加元素需要扩容的次数大大减少（预留空间较多），时间复杂度较低;

缺点：

因为每次扩容空间翻倍，而很多空间没有利用上，空间利用率不如固定扩容。

在实际应用中，一般采用空间换时间的策略。

3、resize()和reserve()

resize()：改变当前容器内含有元素的**数量**(size())，而不是容器的容量

1. 当resize(len)中len>v.capacity()，则数组中的size和capacity均设置为len;
2. 当resize(len)中len<=v.capacity()，则数组中的size设置为len，而capacity不变;

reserve()：改变当前容器的**最大容量**（capacity）

1. 如果reserve(len)的值 > 当前的capacity()，那么会重新分配一块能存len个对象的空间，然后把之前的对象通过copy construtor复制过来，销毁之前的内存;
2. 当reserve(len)中len<=当前的capacity()，则数组中的capacity不变，size不变，即不对容器做任何改变。

3. vector源码

```
templeta<class T,class Alloc=alloc>
class vector {
public:

    typedef T value_type;
    typedef value_type *pointer;
    typedef value_type &reference;
    typedef value_type *iterator;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type
    //嵌套类型定义，也可以是关联类型定义
protected:

    typedef simple_alloc <value_type, Alloc> data_alloctor
    //空间配置器（分配器）
    iterator start;
    iterator finish;
    iterator end_of_storage;
    //这3个就是vector里的数据，所以一个vector就是包含3个指针12byte,下面有图介绍

    void insert_aux(iterator position, const T &x);

    //这个就是vector的自动扩充函数，在下面章节我会拿出来分析
    void deallocate() {
        if (start)
            data_allocator::deallocate(start, end_of_storage);
    }
```

```
    //析构函数的部分实现函数

    void fill_initialize(size_type n, const T &value) {
        start = allocate_and_fill(n, value);
        finish = start + n;
        end_of_storage = finish;
    }

    //构造函数的具体实现

public:

    iterator begin() { return start; };
    iterator end() { return finish; };
    size_type size() const { return size_type(end() - begin()); };
    size_type capacity() const { return size_type(end_of_storage - begin()); }
    bool empty() const { return begin() == end(); }
    reference operator[](size_type n) { return *(begin() + n); };
    //重载[]说明vector支持随机访问

    vector() : start(0), end(0), end_of_storage(0) {};

    vector(size_type n, const T &value)(fill_initialize(n, value););

    vector(long n, const T &value)(fill_initialize(n, value););

    vector(int n, const T &value)(fill_initialize(n, value););

    explicit vector(size_type n) { fill_initialize(n, T()); };

    //重载构造函数
    ~vector() {
        destory(start, finish);//全局函数，析构对象
        deallocate();//成员函数，释放空间
    }
    //接下来就是一些功能函数
    reference front() { return *begin(); };
    reference back() { return *(end() - 1); };
    void push_back(const T &x) {
        if (finsih != end_of_storage) {
            construct(finish, x);
            ++finish;
        } else insert_aux(end(), x);
        //先扩充在添加
    }

    void pop_back() {
        --finish;
        destory(finish);
```

```
    }

    iterator erase(iterator position) {
        if (position + 1 != end())
            copy(position + 1, finish, position);
        --finish;
        destory(finish);
        return position;
    }

    void resize(size_type new_size, const T &x) {
        if (new_size() < size())
            erase(begin() + new_size, end());
        else
            insert(end(), new_size - size(),x);
    }

    void resize()(size_type new_size) { resize(new_size, T()); }
    void clear() { erase(begin(), end()); }

protected:
    //配置空间并填满内容
    iterator allocate_and_fill(size_type n, const T &x) {
        iterator result = data_allocator::allocate(n);
        uninitialized_fill_n(result, n, x);//全局函数
    }
}
```

vector迭代器：由于vector维护的是一个线性区间，所以普通指针具备作为vector迭代器的所有条件，就不需要重载operator+，operator*之类的东西

```
template <class T, class Alloc = alloc>
class vector {
public:
    typedef T value_type;
    typedef value_type* iterator; //vector的迭代器是原生指针
    // ...
};
```

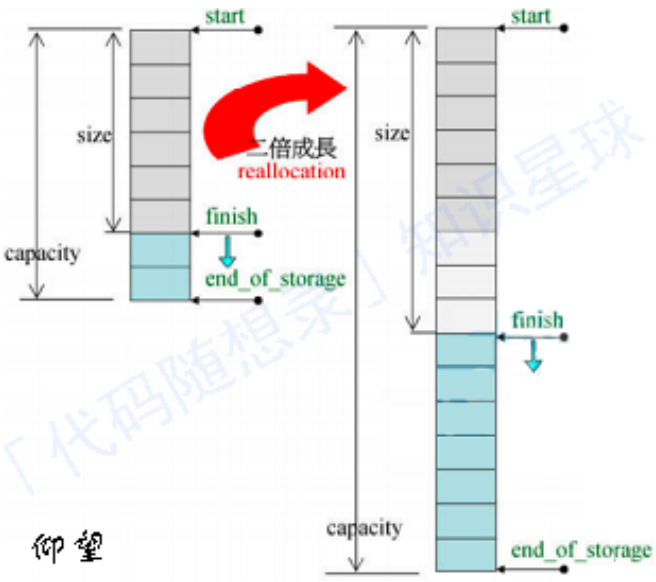
vector的数据结构：线性空间。为了降低配置空间的成本，我们必须让其容量大于其大小。

vector的构造以及内存管理：当我们使用push_back插入元素在尾端的时候，我们首先检查是否还有备用空间也就是说end是否等于end_of_storage，如果有直接插入，如果没有就扩充空间

```
template<class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T &x) {
    if (finish != end_of_storage) {//有备用空间
        consruct(finish, *(finish - 1));//在备用空间处构造一个元素，以vector最后一个元素为其初值
```

```
        ++finish;
        T x_copy = x;
        copy_backward(position, finish - 2, finish - 1);
        *position = x_copy;
    } else {
        const size_type old_size = size();
        const size_type len = old_size != 0 ? 2 * old_size() : 1;
        //vector中如果没有元素就配置一个元素，如果有就配置2倍元素。
        iterator new_start = data_allocator::allocate(len);
        iterator new_finish = new_start;
        try {
            //拷贝插入点之前的元素
            new_finish = uninitialized_copy(start, position, new_start);
            construct(new_finish, x);
            ++new_finish;
            //拷贝插入点之后的元素
            new_finish = uninitialized_copy(position, finish, new_finish);
        }
        catch () {
            destroy(new_start, new_finish);
            data_allocator::deallocate(new_start, len);
            throw;
        }
        //析构并释放原vector
        destory(begin(), end());
        deallocate();
        //调整迭代器指向新的vector
        start = new_start;
        finish = new_finish;
        end_of_storage = new_start + len;
    }
}
```

整个分为3个部分，配置新空间，转移元素，释放原来的元素与空间,因此一旦引起空间配置指向以前vector的所有迭代器都要失效。



vector的部分元素操作：pop_back,erase,clear,insert

```
void pop_back() {
    --finish;
    destory(finish);
}

//erase版本一：清除范围元素

iterator erase(iterator first, iterator last) {
    interator i = copy(last, finish, first);
    destory(i, finish);
    finish = finish - (last - first);
    return first;
}

//版本二：清除某个位置上的元素
iterator erase(iterator position) {
    if (position + 1 != finish) {
        copy(position + 1, finish, position);
        --finish;
        distory(finish);
        return position;
    }
}

void clear() { erase(begin(), end()) };
template<class T, class Alloc>
void vector<T, Alloc>::insert(iterator position, size_type n, const T &x) {
    if (n != 0) {
        if (size_type(end_of_strage - finish) > n) {
            //备用空间大于插入的元素数量
            T x_copy = x;
            //以下计算插入点之后的现有元素个数
```

```
            const size_type elems_after = finish - positon;
            iterator old_finish = finish;
            if (elems_after > n) {
                //插入点之后的元素个数大于要插入的元素个数
                uninitialiazed_copy(finish - n, finish, finish);
                finish += n;//将vector的尾端标记后移
                copy_backward(position, old_finish - n, old_finish);
                fill(position, old_finish, x_copy);//从插入点之后开始插入新值
            } else {
                //插入点之后的元素个数小于要插入的元素个数
                uninitialiazed_fill_n(finish, n - elems_after, finish);
                finish += n - elems_after;
                uninitialiazed_copy(position, old_finish, finish);
                finish += elems_after;
                fill(position, old_finish, x_copy);
            }
            else {
                //备用空间小于要插入元素的个数
                //首先决定新长度，原长度的两倍，或者老长度+新的元素个数
                const size_type old_size = size();
                const size_type len = old_size + max(old_size, n);
                //以下配置新的空间
                iterator new_start = data_allocator::allocate(len);
                iterator new_finish = new_start;
                _STL_TRY {
                    //拷贝插入点之前的元素
                    new_finish = uninitialized_copy(start, position, new_start);
                    //把新增元素（初值皆为n)传入新空间
                    new_finish=uninitialized_fill_n
                    //拷贝插入点之后的元素
                    new_finish=uninitialized_copy(position, finish, new_finish);
                    //这一段有利于理解上面的insert_aux函数
                }

#ifdef _STL_USE_EXCEPTIONS
                catch(){
                    //如果有异常发生
                    destroy(new_start,new_finish);
                    data_allocator::deallocate(new_start,len);
                    throw;
                }
#endif/* _STL_USE_EXCEPTIONS*/
                //析构并释放原vector
                destory(begin(), end());
                deallocate();
                //调整迭代器指向新的vector
                start = new_start;
                finish = new_finish;
                end_of_storage = new_start + len;
```

```
            }
        }
    }
}
```

list（链表）

list设计

每个元素都是放在一块内存中，他的内存空间可以是不连续的，通过指针来进行数据的访问

在哪里添加删除元素性能都很高，不需要移动内存，当然也不需要对每个元素都进行构造与析构了，所以常用来做随机插入和删除操作容器

list属于双向链表，其结点与list本身是分开设计的：

```
template<class T, class Alloc = alloc>
class list {
protected:
    typedef listnode <T> listnode;
public:
    typedef listnode link_type;
    typedef listiterator<T, T &, T> iterator;
protected:
    link_type node;
};
```

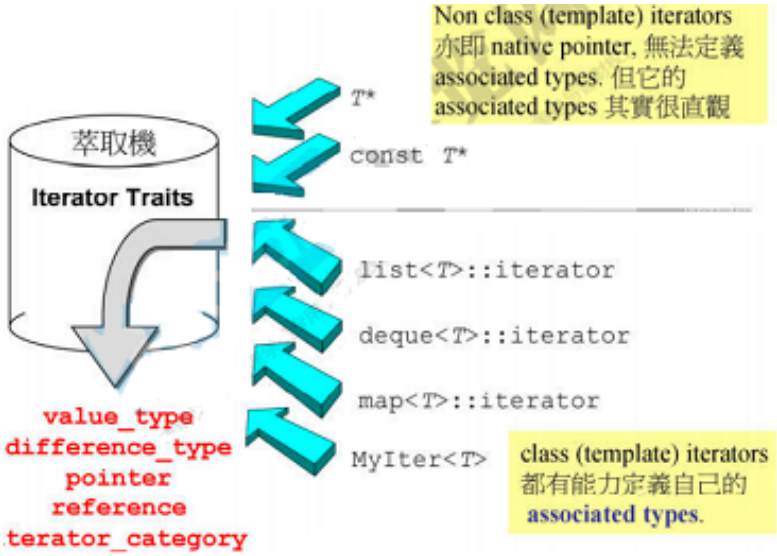
学习到了一个分析方法，拿到这样一个类，先看它的数据比如上面的linktype node，然后我们再看它的前缀，linktype，去上面在linktype,找到typedef listnode linktype;按这个方法继续找到上面的typedef listnode listnode;我们发现list_node是下面的类，我们一层层的寻找，就能看懂这些源码

```
template<class T>
struct _listnode {
    typedef void voidpointer;
    void_pointer prev;
    void_pointer next;
    T data;
};
```

list是一个环状的双向链表，同时它也满足STL对于“前闭后开”的原则，即在链表尾端可以加上空白节点

list的迭代器的设计：

迭代器是泛化的指针所以里面重载了->，--，++，*（），等运算符，同时迭代器是算法与容器之间的桥梁，算法需要了解容器的方方面面，于是就诞生了5种关联类型，(这5种类型是必备的，可能还需要其他类型)我们知道算法传入的是迭代器或者指针，算法根据传入的迭代器或指针推断出算法所想要了解的容器里的5种关联类型的相关信息。由于光传入指针，算法推断不出来其想要的信息，所以我们需要一个中间商（萃取器）也就是我们所说的iterator traits类，对于一般的迭代器，它直接提供迭代器里的关联类型值，而对于指针和常量指针，它采用的类模板偏特化，从而提供其所需要的关联类型的值。




```
// 针对一般的迭代器类型,直接取迭代器内定义的关联类型
template<class I>
struct iterator_traits {
    typedef typename I::iteratorcategory iteratorcategory;
    typedef typename I::valuetype valuetype;
    typedef typename I::differencetype differencetype;
    typedef typename I::pointer pointer;
    typedef typename I::reference reference;
};
•// 针对指针类型进行特化,指定关联类型的值
template<class T>
struct iteratortraits<T> {
    typedef randomaccessiteratortag iteratorcategory;
    typedef T value_type;
    typedef ptrdifft differencetype;
    typedef T *pointer;
    typedef T &reference;
};
// 针对指针常量类型进行特化,指定关联类型的值
template<class T>
struct iteratortraits<const T> {
    typedef randomaccessiteratortag iteratorcategory;
    typedef T valuetype;  // valuetye被用于创建变量,为灵活起见,取 T 而非 const T 作为
value_type
    typedef ptrdifft differencetype;
    typedef const T *pointer;
    typedef const T &reference;
};
```

vector和list的区别

1. vector底层实现是数组；list是双向链表
2. vector是顺序内存,支持随机访问，list不行
3. vector在中间节点进行插入删除会导致内存拷贝，list不会
4. vector一次性分配好内存，不够时才进行翻倍扩容；list每次插入新节点都会进行内存申请
5. vector随机访问性能好，插入删除性能差；list随机访问性能差，插入删除性能好

deque（双端数组）

支持快速随机访问，由于deque需要处理内部跳转，因此速度上没有vector快。

1、deque概述：

deque是一个双端开口的连续线性空间，其内部为分段连续的空间组成，随时可以增加一段新的空间并链接

注意:

由于deque的迭代器比vector要复杂，这影响了各个运算层面，所以除非必要尽量使用vector；为了提高效率，在对deque进行排序操作的时候，我们可以先把deque复制到vector中再进行排序最后在复制回deque

2、deque中控器：

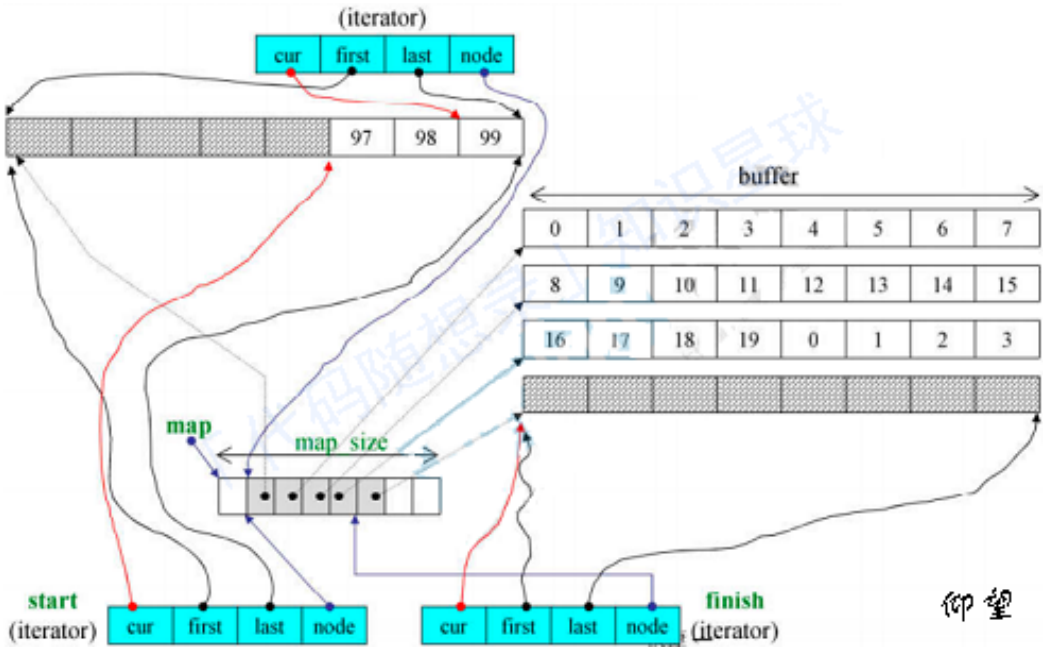
deque是由一段一段的定量连续空间构成。一旦有必要在其头端或者尾端增加新的空间，便配置一段定量连续空间，串接在整个deque的头端或者尾端

好处：

避免“vector的重新配置，复制，释放”的轮回，维护连整体连续的假象，并提供随机访问的接口；

坏处：

其迭代器变得很复杂



deque采用一块map作为主控，其中的每个元素都是指针，指向另一片连续线性空间，称之为缓存区，这个区才是用来储存数据的。

```
template<class T, class Alloc=alloc, size_t Bufsize = 0>
class deque {

public:

    typedef T value_type;
    typedef value_type pointer*;
    typedef size_t size_type;
    // ...

public:

    typedef _deque_iterator<T, T &, T *, BufSiz> iterator;

protected:

    typedef pointer *map_pointer;

protected:

    iterator start;
    iterator finish;
    map_pointer map;//指向map
    size_type map_size;//map内可容纳多少指针
}

//map其实是一个T**
```

deque迭代器:

```
template<class T, class Ref, class Ptr, size_t BufSiz>

struct _deque_iterator {
    typedef _deque_iterator<T, T &, T *, BufSiz> iterator;
    typedef _deque_iterator<T, cosnt T &, const T *, BufSiz> const_iterator;
    static size_t buffer_size() { return _deque_buf_size(BufSiz, sizeof(T)); };
    typedef randem_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T **map_pointer;
    typedef _deque_iterator self;
    T *cur;
    T *first;
    T *last;
    map_pointer node;
```

```
    // ...
    //这是一个决定缓存区大小的函数
    inline size_t _deque_buf_size(size_t n, size_t sz) {
        return n != 0 ? n :
        (sz < 512 ? size_t(512 / sz) : size_t(1));
    }
}
```

deque拥有两个数据成员

start与finish迭代器，分别由deque:begin()与deque:end()传回

```
//迭代器的关键行为，其中要注意的是一旦遇到缓冲区边缘，可能需要跳一个缓存区

void set_node(map_pointer new_node) {
    node = new_node;
    first = *new_node;
    last = first + difference_type(buffer_size());
}

//接下来重载运算子是_deque_iterator<>成功运作的关键
reference operator*() const { return *cur; }

pointer operator->() const { return &(operator*()); }

difference_type operator— (const self &x) const {
    return difference_type (buffer_szie()) * (node-x.node-1)+(cur-first)+(x.last-
x.cur);
}

self &operator++() {
    ++cur;
    if (cur == last) {
        set_node(node + 1);
        cur = first;
    }
    return *this;
}

self operator++(int) {
    self temp = *this;
    ++*this;
    return temp;
}

self &operator--() {
    if (cur == first) {
        set_node(node - 1);
        cur = last;
```

```
    }
    --cur;
    return *this;
}

self operator-(int) {
    self temp = *this;
    --*this;
    return temp;
}

//以下实现随机存取，迭代器可以直接跳跃n个距离
self &operator+=(difference_type n) {
    difference_type offest = n + (cur - first);
    if (offest > 0 && offest < difference_type(buffer_size()))
        cur += n;
    else {
        offest > 0 ? offest / fifference_type(buffer_size()) : -difference_type((-
offest - 1) / buffer_size()) - 1;
        set_node(node + node_offest);
        cur = first + (offest - node_offest * difference_type(buffer_size()));
    }
    return *this;
}

self operator+(differnece_type n) {
    self tmp = *this;
    return tmp += n;
}

self operator-=() { return *this += -n; }

self operator-(difference_type n) {
    self temp = *this;
    return *this -= n;
}

rference operator[](difference_type n) {
    return *(*this + n);
}

bool operator==(const self &x) const { return cur == x.cur; }
bool operator!=(const self &x) const { return !(*this == x); }
bool operatoe<(const self &x) const {
    return (node == x.node) ? (cur < x.cur) : (node - x.node);
}
```

deque数据结构:

deque除了维护一个map指针以外，还维护了start与finish迭代器分别指向第一缓冲区的第一个元素，和最后一个缓冲区的最后一个元素的下一个元素，同时它还必须记住当前map的大小。具体结构和源代码看上面

deque的构造与管理

```
//deque首先自行定义了两个空间配置器
typedef simple_alloc <value_type, Alloc> data_allocator;
typedef simple_alloc <pointer, Alloc> map_allocator;
```

deque中有一个构造函数用于构造deque结构并赋初值

```
deque(int n, const value_type &value) : start(), finish(), map(0), map_size(0) {
    fill_initialize(n, value);//这个函数就是用来构建deque结构，并设立初值
}

template<class T, class Alloc, size_t BufSize)
void deque<T, Alloc, BufSize>::fill_initialize(size_type n, const value_type &value) {
    creat_map_and_node(n);//安排结构
    map_pointer cur;
    _STL_TRY {
        //为每个缓存区赋值
        for (cur=start.node;cur<finish.node;++cur)
        uninitalized_ fill(*cur, *cur+buffer_size(), value);
        //设置最后一个节点有一点不同
        uninitalized_fill(finish.first, finish.cur, value);
    }

    catch() {
        // ...
    }
}

template<class T, class Alloc, size_t Bufsize>
void deque<T, alloc, Bufsize>::creat_map_and_node(size_type num_elements) {
    //需要节点数=元素个数/每个缓存区的可容纳元素个数+1
    size_type num_nodes = num_elements / Buf_size() + 1;
    map_size = max(initial_map_size(), num_nodes + 2);//前后预留2个供扩充

    //创建一个大小为map_size的map
    map = map_allocator::allocate(map_size);

    //创建两个指针指向map所拥有的全部节点的最中间区段
    map_pointer nstart = map + (map_size() - num_nodes) / 2;
    map_poniter nfinish = nstart + num_nodes - 1;
    map_pointer cur;

    _STL_TRY {
        //为每个节点配置缓存区
        for (cur=nstart;cur<nfinish;++cur)
```

```
        +cur=allocate_node();
    }
    catch() {
        // ...
    }

    //最后为deque内的start和finish设定内容
    start.set_node(nstart);
    finish.set_node(nfinish);
    start.cur = start.first;
    finish.cur = finish.first + num_elements % buffer_szie();
}
```

接下来就是插入操作的实现，第一，首先判断是否有扩充map的需求，若有就扩，然后就是在插入函数中，首先判断是否在结尾或者开头从而判断是否跳跃节点。

```
void push_back(const value_type &t) {
    if (finish.cur != finish.last - 1) {
        construct(finish.cur, t);
        ++finish.cur;
    } else
        push_back_aux(t);
}

// 由于尾端只剩一个可用元素空间（finish.cur=finish.last-1），
// 所以我们必须重新配置一个缓存区，在设置新元素的内容，然后更改迭代器的状态

tempalate<class T, class Alloc, size_t BufSize>
void deque<T, alloc, BufSize>::push_back_aux(const value_type &t) {
    value_type t_copy = t;
    reserve_map_at_back();
    *(finish.node + 1) = allocate_node();
    _STL_TRY {
        construct(finish.cur, t_copy);
        finish.set_node(finish.node+1);
        finish.cur=finish.first;
    }
    - STL_UNWIND {
        deallocate_node(*(finish.node + 1));
    }
}
//push_front也是一样的逻辑
```

	deque	vector
组织方式	按页或块来分配存储器的，每页包含固定数目的元素	分配一段连续的内存来存储内容
效率	即使在容器的前端也可以提供常数时间的insert和erase操作，而且在体积增长方面也比vector更具有效率	只是在序列的尾端插入元素时才有效率，但是随机访问速度要比deque快

stack && queue

概述：栈与队列被称之为duque的配接器，其底层是以deque为底部架构。通过deque执行具体操作

源码

```
template<class T, class Sequence=deque <T>>

class stack {//_STL_NULL_TMPL_ARGS展开为<>
    friend bool operator==
    _STL_NULL_TMPL_ARGS(const stack &, const stack &);
    friend bool operator<
    _STL_NULL_TMPL_ARGS(coonst
    stack&,const stack&);

public:

    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_refernece;

protected:

    Sequence c;

public:
```

```
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference top() {
        return c.back();
    }
    const_rference top() const { return c.back(); }
    void push(const value_type &x) { c.push_back(x); }
    void pop_back() { c.pop_back(); }
};

template<class T, class Sequence>
bool operator==(const stack<T, Sequence> &x, const stack<T, Sequence> &y) {
    return x.c == y.c;
}

template<class T, class Sequence>
bool operator<(const stack<T, Sequence> &x, const stack<T, Sequence> &y) {
    return x.c < y.c;
}
```

heap && priority_queue

heap（堆）：

建立在完全二叉树上，分为两种，大根堆，小根堆,其在STL中做priority_queue的助手，即，以任何顺序将元素推入容器中，然后取出时一定是从优先权最高的元素开始取，完全二叉树具有这样的性质，适合做priority_queue的底层

priority_queue:

优先队列，也是配接器。其内的元素不是按照被推入的顺序排列，而是自动取元素的权值排列，确省情况下利用一个max-heap完成，后者是以vector—表现的完全二叉树。

```
template<class T, class Sequence=vector <T>, class Compare=less<typename
Sequence::value_type>>
class priority_queue {
public:

    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_refernece;

protected:

    Sequence c;//底层容器
    Compare comp//容器比较大小标准

public:
```

```
    priority_queue() : c() {}
    explicit priority_queue(const Compare &x) : c(), comp(x) {}
    //以下用到的make_heap(),push_heap(),pop_heap()都是泛型算法

    //任何一个构造函数都可以立即在底层产生一个heap
    template<class InputIterator>
    priority_queue(InputIterator first, InputIterator last const Compare &x)
            :c(first, last), comp(x) { make_heap(c.begin(), c.end(), comp); }

    template<class InputIterator>
    priority_queue(InputIterator first, InputIterator last const Compare &x)
            :c(first, last) { make_heap(c.begin(), c.end(), comp); }

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    const_reference top() const { return c.front(); }
    void push(const value_type &x) {
        _STL_TRy {
            c.push_back(X);
            push_heap(c.begin(), c.end(), comp);
        }
        _STL_UNWIND{c.clear()};
    }

    void pop() {
        _STL_TRY {
            pop_heap(c.begin(), c.end(), comp);
            c.pop_back();
        }
        _STL_UNWEIND{c.clear()};
    }
};

// priority_queue无迭代器
```

map && set的区别和实现原理

map内部实现了一个**红黑树**（红黑树是非严格平衡的二叉搜索树，而AVL是严格平衡二叉搜索树），红黑树有自动排序的功能，因此map内部所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素。因此，对于map进行的查找、删除、添加等一系列的操作都相当于是对红黑树进行的操作。map中的元素是按照二叉树（又名二叉查找树、二叉排序树）存储的，特点就是左子树上所有节点的键值都小于根节点的键值，右子树所有节点的键值都大于根节点的键值。使用中序遍历可将键值按照从小到大遍历出来。

共同点：

都是C++的关联容器,只是通过它提供的接口对里面的元素进行访问，底层都是采用红黑树实现。

不同点：

set：用来判断某一个元素是不是在一个组里面。

map：映射，相当于字典，把一个值映射成另一个值，可以创建字典。

优点：

查找某一个数的时间为O(logn)；遍历时采用iterator，效果不错。

缺点：

每次插入值的时候，都需要调整红黑树，效率有一定影响。

细节

1、为什么要成倍的扩容而不是一次增加一个固定大小的容量呢?

采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到O(n)的时间复杂度。

2、为什么是以两倍的方式扩容而不是三倍四倍，或者其他方式呢

考虑可能产生的堆空间浪费，所以增长倍数不能太大，一般是1.5或2；GCC是2；VS是1.5，k =2 每次扩展的新尺寸必然刚好大于之前分配的总和，之前分配的内存空间不可能被使用，这样对于缓存并不友好，采用1.5倍的增长方式可以更好的实现对内存的重复利用。

C++并没有规定扩容因子K，这是由标准库的实现者决定的。

map && unordered_map的区别

map中元素是一些key-value对，关键字起索引作用，值表示和索引相关的数据。

底层实现：

map底层是基于红黑树实现的，因此map内部元素排列是有序的。

而unordered_map底层则是基于哈希表实现的，因此其元素的排列顺序是杂乱无序的。

map：

优点：

有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作。

map的查找、删除、增加等一系列操作时间复杂度稳定，都为O(logn)。

缺点：

查找、删除、增加等操作平均时间复杂度较慢，与n相关。

unordered_map：

优点：

查找、删除、添加的速度快，时间复杂度为常数级O(1）。

缺点：

因为unordered_map内部基于哈希表，以（key,value）对的形式存储，因此空间占用率高。

unordered_map的查找、删除、添加的时间复杂度不稳定，平均为O(1)，取决于哈希函数。极端情况下可能为O(n)。

问题：

为什么insert之后，以前保存的iterator不会失效?

因为 map 和 set 存储的是结点，不需要内存拷贝和内存移动。但是像 vector 在插入数据时如果内存不够会重新开辟一块内存。map 和 set 的 iterator 指向的是节点的指针，vector 指向的是内存的某个位置

为何map和set的插入删除效率比其他序列容器高?

因为 map 和 set 底部使用红黑树实现，插入和删除的时间复杂度是 O(logn)，而向 vector 这样的序列容器插入和删除的时间复杂度是 O(N)

push_back 和 emplace_back 的区别

1. `push_back` 用于在容器的尾部添加一个元素。

```
container.push_back(value);
```

`container` 是一个支持 `push_back` 操作的容器，例如 `std::vector`、`std::list` 等，而 `value` 是要添加的元素的值。

2. `emplace_back` 用于在容器的尾部直接构造一个元素。

```
container.emplace_back(args);
```

其中 `container` 是一个支持 `emplace_back` 操作的容器，而 `args` 是传递给元素类型的构造函数的参数。与 `push_back` 不同的是，`emplace_back` 不需要创建临时对象，而是直接在容器中构造新的元素。

区别：

- `push_back` 接受一个已存在的对象或一个可转换为容器元素类型的对象，并将其复制或移动到容器中。`emplace_back` 直接在容器中构造元素，不需要创建临时对象。
- `emplace_back` 通常比 `push_back` 更高效，因为它避免了创建和销毁临时对象的开销。
- `emplace_back` 的参数是传递给元素类型的构造函数的参数，而 `push_back` 直接接受一个元素。

```
#include <vector>
#include <string>

int main() {
    std::vector<int> numbers;

    // 使用 push_back
    numbers.push_back(42);

    // 使用 emplace_back
    numbers.emplace_back(3, 4, 5);  // 通过构造函数参数直接构造元素

    return 0;
}
```

vector的实现原理

`std::vector` 是C++标准库中的一个动态数组实现，它的实现原理基于数组数据结构。实现通常包含一个指向数组起始位置的指针、数组的大小和容量等信息。

在尾部进行插入和删除操作时，只需调整尾部指针，不需要移动整个数据块。

当元素数量达到当前内存块的容量时，`std::vector` 会申请一个更大的内存块，将元素从旧的内存块复制到新的内存块，并释放旧的内存块。

由于数组的连续内存结构，通过索引进行访问时可以通过指针运算实现常数时间复杂度的随机访问。

```
template <class T, class Allocator = std::allocator<T>>
class vector {
private:
    T* elements;  // 指向数组起始位置的指针
    size_t size;   // 当前元素数量
    size_t capacity;  // 当前分配的内存块容量
};
```

list的实现原理

`std::list` 是C++标准库中的一个双向链表实现，它的实现原理基于链表数据结构。每个节点包含两个指针，分别指向前一个节点和后一个节点, 以及存储实际数据的部分。

```
template <class T>
struct Node {
    T data;
    Node* prev;
    Node* next;
};
```

`std::list` 维护一个头指针和一个尾指针，它们分别指向链表的第一个和最后一个节点。

在插入和删除操作时，只需调整相邻节点的指针，不需要移动整个数据块。

vector和list的区别

1. 实现上

- `vector`使用动态数组实现。连续的内存块，支持随机访问。在尾部进行插入/删除操作较快，但在中间或头部进行插入/删除可能会涉及大量元素的移动。
- `list`使用双向链表实现。不连续的内存块，不支持随机访问。在任意位置进行插入/删除操作都是常数时间复杂度。

2. 访问上

- vector支持通过索引进行快速随机访问。使用迭代器进行访问时，效率较高。
- List不支持通过索引进行快速随机访问。迭代器在访问时需要遍历链表，效率相对较低。

3. 插入和删除操作

- **vector**：在尾部进行插入/删除操作是常数时间复杂度。在中间或头部进行插入/删除可能涉及大量元素的移动，时间复杂度为线性。
- List: 在任意位置进行插入/删除操作都是常数时间复杂度，因为只需调整相邻节点的指针。

4. 内存管理

- Vector使用动态数组，需要在预估元素数量时分配一块较大的内存空间。
- list由于采用链表结构，动态分配的内存比较灵活。每个元素都有自己的内存块，避免了预分配的问题。

5. 适用场景

- Vector适用于需要频繁随机访问、在尾部进行插入/删除操作的场景。
- list适用于需要频繁在中间或头部进行插入/删除操作、不要求随机访问的场景。

迭代器有什么作用？什么时候迭代器会失效

迭代器为不同类型的容器提供了统一的访问接口， 隐藏了底层容器的具体实现细节， 允许开发者使用一致的语法来操作不同类型的容器。

- 对于序列容器vector，deque来说，使用erase后，后边的每个元素的迭代器都会失效，后边每个元素都往前移动一位，erase返回下一个有效的迭代器。
- 对于关联容器map，set来说，使用了erase后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素，不会影响下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。
- 对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的迭代器，因此上面两种方法都可以使用。

C++ 泛型编程

C++模板全特化和偏特化

模板分为类模板与函数模板，特化分为特例化（全特化）和部分特例化（偏特化）。

对模板特例化是因为对特定类型，可以利用某些特定知识来提高效率，而不是使用通用模板。

对函数模板：

1. 模板和特例化版本应该声明在同一头文件，所有同名模板的声明应放在前面，接着是特例化版本。
2. 一个模板被称为全特化的条件：1.必须有一个主模板类 2.模板类型被全部明确化。

模板函数：

```
template<typename T1, typename T2>
void fun(T1 a, T2 b)
{
 cout<<"模板函数"<<endl;
}

template<>
void fun<int , char >(int a, char b)
{
 cout<<"全特化"<<endl;
}
```

函数模板，只有全特化，偏特化的功能可以通过函数的重载完成。

对类模板：

```
template<typename T1, typename T2>
class Test
{
public:
 Test(T1 i,T2 j):a(i),b(j){cout<<"模板类"<<endl;}
private:
 T1 a;
 T2 b;
};

template<>
class Test<int , char>
{
public:
 Test(int i, char j):a(i),b(j){cout<<"全特化"<<endl;}
private:
 int a;
 char b;
};

template <typename T2>
class Test<char, T2>
{
public:
 Test(char i, T2 j):a(i),b(j){cout<<"偏特化"<<endl;}
private:
 char a;
 T2 b;
}
```

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。

C++ 新特性

C++11的新特性有哪些

1. 语法的改进

   （1）统一的初始化方法

   （2）成员变量默认初始化

   （3）auto关键字： 允许编译器自动推断变量的类型，减少类型声明的冗余。

   （4）decltype 求表达式的类型

   （5）智能指针 std::shared_ptr 和 std::unique_ptr

   （6）**空指针 nullptr: 提供了明确表示空指针的关键字，替代了传统的 `NULL`。**

   （7）**基于范围的for循环: 简化遍历容器元素的语法**

   （8）**右值引用和move语义 引入右值引用和移动构造函数，允许高效地将资源从一个对象移动到另一个对象，提高性能。**

2. 标准库扩充（往STL里新加进一些模板类）

   （9）无序容器（哈希表） 用法和功能同map一模一样，区别在于哈希表的效率更高

   （10）正则表达式 可以认为正则表达式实质上是一个字符串，该字符串描述了一种特定模式的字符串

   （1**1）Lambda表达式： 允许在代码中定义匿名函数**

智能指针

1. shared_ptr

1、**shared_ptr的实现机制是在拷贝构造时使用同一份引用计数**

（1）一个模板指针T* ptr

指向实际的对象

（2）一个引用次数

必须new出来的，不然会多个shared_ptr里面会有不同的引用次数而导致多次delete

（3）重载operator*和operator->

使得能像指针一样使用shared_ptr

（4）重载copy constructor

使其引用次数加一（拷贝构造函数)

（5）重载operator=（赋值运算符）

如果原来的shared_ptr已经有对象，则让其引用次数减一并判断引用是否为零(是否调用delete)，然后将新的对象引用次数加一

（6）重载析构函数

使引用次数减一并判断引用是否为零; (是否调用delete)

2、线程安全问题

（1）同一个shared_ptr被多个线程“读”是安全的;

（2）同一个shared_ptr被多个线程“写”是不安全的;

证明：在多个线程中同时对一个shared_ptr循环执行两遍swap。 shared_ptr的swap函数的作用就是和另外一个shared_ptr交换引用对象和引用计数，是写操作。执行两遍swap之后, shared_ptr引用的对象的值应该不变）

（3）共享引用计数的不同的shared_ptr被多个线程”写“ 是安全的。

2. unique_ptr

1、unique_ptr”唯一”拥有其所指对象

同一时刻只能有一个unique_ptr指向给定对象，离开作用域时，若其指向对象，则将其所指对象销毁（默认delete）。

2、定义unique_ptr时

需要将其绑定到一个new返回的指针上。

3、unique_ptr不支持普通的拷贝和赋值（因为拥有指向的对象）

但是可以拷贝和赋值一个将要被销毁的unique_ptr；可以通过release或者reset将指针所有权从一个（非const）unique_ptr转移到另一个unique。

3. weak_ptr

1、weak_ptr是为了配合shared_ptr而引入的一种智能指针

它的最大作用在于协助shared_ptr工作，像旁观者那样观测资源的使用情况，但weak_ptr没有共享资源，它的构造不会引起指针引用计数的增加。

2、和shared_ptr指向相同内存

shared_ptr析构之后内存释放，在使用之前使用函数lock()检查weak_ptr是否为空指针。

类型推导

1、auto：

auto可以让编译器在编译期就推导出变量的类型

（1）auto的使用必须马上初始化，否则无法推导出类型

（2）auto在一行定义多个变量时，各个变量的推导不能产生二义性，否则编译失败

（3）auto不能用作函数参数

（4）在类中auto不能用作非静态成员变量

（5）auto不能定义数组，可以定义指针

（6）auto无法推导出模板参数

（7）在不声明为引用或指针时，auto会忽略等号右边的引用类型和cv限定

（8）在声明为引用或者指针时，auto会保留等号右边的引用和cv属性

2、decltype：

decltype则用于推导表达式类型，这里只用于编译器分析表达式的类型，表达式实际不会进行运算

decltype不会像auto一样**忽略引用和cv属性**，decltype会保留表达式的引用和cv属性

对于decltype(exp)有：

1. exp是表达式，decltype(exp)和exp类型相同
2. exp是函数调用，decltype(exp)和函数返回值类型相同
3. 其它情况，若exp是左值，decltype(exp)是exp类型的左值引用

auto和decltype的配合使用:

```
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) {
    return t + u;
}
```

右值引用

右值引用、移动语义和完美转发

左值右值：

左值： 可以放在等号左边，可以取地址并有名字
右值： 不可以放在等号左边，不能取地址，没有名字

字符串字面值"abcd"也是左值，不是右值

++i、--i是左值，i++、i--是右值

1、将亡值

将亡值是指C++11新增的和右值引用相关的表达式

将亡值可以理解为**即将要销毁的值**，通过“盗取”其它变量内存空间方式获取的值，在确保其它变量不再被使用或者即将被销毁时，可以避免内存空间的释放和分配，延长变量值的生命周期，常用来完成移动构造或者移动赋值的特殊任务

2、左值引用

左值引用就是对左值进行引用的类型，**是对象的一个别名**

并不拥有所绑定对象的堆存，所以必须立即初始化。 对于左值引用，等号右边的值必须可以取地址，如果不能取地址，则会编译失败，或者可以使用const引用形式

3、右值引用

表达式等号右边的值需要是右值，可以使用std::move函数强制把左值转换为右值。

4、移动语义

可以理解为**转移所有权**，对于移动语义，类似于转让或者资源窃取的意思，对于那块资源，转为自己所拥有，别人不再拥有也不会再使用。

通过移动构造函数使用移动语义，也就是std::move；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

浅拷贝：

> a和b的指针指向了同一块内存，就是浅拷贝，只是数据的简单赋值；

深拷贝：

> 深拷贝就是再拷贝对象时，如果被拷贝对象内部还有指针引用指向其它资源，自己需要重新开辟一块新内存存储资源

5、完美转发

写一个接受任意实参的函数模板，并转发到其它函数，目标函数会收到与转发函数完全相同的实参，通过std::forward()实现

nullptr

nullptr是用来代替NULL，一般C++会把NULL、0视为同一种东西，这取决去编译器如何定义NULL，有的定义为((void*)0)，有的定义为0

C++不允许直接将void* 隐式转换到其他类型，在进行C++重载时会发生混乱

例如：

```
void foo(char *);
void foo(int );
```

如果NULL被定义为 ((void*)0)，那么当编译char *ch = NULL时，NULL被定义为 0

当foo(NULL)时，此时NULL为0，会去调用foo(int)，从而发生混乱

为解决这个问题，从而需要使用NULL时，用nullptr代替：

C++11引入nullptr关键字**来区分空指针和0**。nullptr 的类型为 nullptr_t，能够转换为任何指针或成员指针的类型，也可以进行相等或不等的比较。

范围for循环

基于范围的迭代写法，for（变量：对象）表达式

对string对象的每个字符做一些操作：

```
string str ("some thing");
for (char c : str) cout << c << endl; // 输出字符串str中的每个字符
```

对vector中的元素进行遍历：

```
std::vector<int> arr(5, 100);
for (std::vector<int>::iterator i = arr.begin(); i != arr.end(); i ++) {
    std::cout << *i << std::endl;
}
// 范围for循环
for (auto &i : arr) {
    std::cout << i << std::endl;
}
```

列表初始化

C++定义了几种初始化方式，例如对一个int变量 x初始化为0：

```
int x = 0;      // method1
int x = {0};    // method2
int x{0};       // method3
int x(0);       // method4
```

采用花括号来进行初始化称为列表初始化，无论是初始化对象还是为对象赋新值。

用于对内置类型变量时，如果使用列表初始化，且初始值存在丢失信息风险时，编译器会报错。

```
long double d = 3.1415926536;
int a = {d};    //存在丢失信息风险，转换未执行。
int a = d;      //确实丢失信息，转换执行。
```

lambda表达式

lambda表达式**表示一个可调用的代码单元**，没有命名的内联函数，不需要函数名因为我们直接（一次性的）用它，不需要其他地方调用它。

1、lambda 表达式的语法：

```
[capture list] (parameter list) -> return type {function body }
// [捕获列表] (参数列表) -> 返回类型 {函数体 }
// 只有 [capture list] 捕获列表和 {function body } 函数体是必选的

auto lam =[]() -> int { cout << "Hello, World!"; return 88; };
auto ret = lam();
cout<<ret<<endl;    // 输出88
```

-> int： 代表此匿名函数返回int，大多数情况下lambda表达式的返回值可由编译器猜测得出，因此不需要我们指定返回值类型。

2、lambda 表达式的特点：

（1）变量捕获才是成就lambda卓越的秘方

1. [] 不捕获任何变量,这种情况下lambda表达式内部不能访问外部的变量

2. [&] 以引用方式捕获所有变量（保证lambda执行时变量存在）
3. [=] 用值的方式捕获所有变量（创建时拷贝，修改对lambda内对象无影响)
4. [=, &foo] 以引用捕获变量foo, 但其余变量都靠值捕获
5. [&, foo] 以值捕获foo, 但其余变量都靠引用捕获
6. [bar] 以值方式捕获bar; 不捕获其它变量
7. [this] 捕获所在类的this指针

```
int a = 1, b = 2, c = 3;
auto lam2 = [&, a](){        //b,c以引用捕获，  a以值捕获
    b = 5;
    c = 6;                //a = 1，a不能赋值
    cout << a << b << c << endl;  //输出 1 5 6
};
lam2();

void fcn() {              //值捕获
    size_t v1 = 42;
    auto f = [v1] {return v1;};
    v1 = 0;
    auto j = f();          //j = 42，创建时拷贝，修改对lambda内对象无影响
}

void fcn() {              //可变lambda
    size_t v1 = 42;
    auto f = [v1] () mutable {return ++v1;};  //修改值捕获可加mutable
    v1 = 0;
    auto j = f();          //j = 43
}

void fcn() {              //引用捕获
    size_t v1 = 42;          //非const
    auto f = [&v1] () {return ++v1;};
    v1 = 0;
    auto j = f();          //注意此时 j = 1
}
```

（2）lambda最大的一个优势是在使用STL中的算法(algorithms)库

例如：数组排序

```
int arr[] = {6, 4, 3, 2, 1, 5};
bool compare(int& a, int& b) {    //谓词函数
    return a > b;
}
std::sort(arr, arr + 6, compare);

//lambda形式
std::sort(arr, arr + 6, [](const int& a, const int& b){return a > b;});    //降序

std::for_each(begin(arr), end(arr), [](const int& e){cout << "After:" << e << endl;});
//6, 5, 4, 3, 2, 1
```

并发

1. std::thread

default (1)	**thread () noexcept;**
initialization (2)	template <class Fn, class... Args> explicit thread (Fn&& fn, Args&&... args);
copy [deleted] (3)	thread (const thread&) = delete;
move (4)	thread (thread&& x) noexcept;

1、默认构造函数，创建一个空的 thread 执行对象。

2、初始化构造函数，创建一个 thread对象，该 thread对象可被 joinable，新产生的线程会调用 fn 函数，该函数的参数由 args 给出。

3、拷贝构造函数(被禁用)，意味着 thread 不可被拷贝构造。

4、move 构造函数，move 构造函数，调用成功之后 x 不代表任何 thread 执行对象。

注意：

可被 joinable 的 thread 对象必须在他们销毁之前被主线程 join 或者将其设置为 detached.

std::thread在使用上容易出错，即std::thread对象在线程函数运行期间必须是有效的。什么意思呢?

```
#include <iostream>
#include <thread>

void threadproc() {
    while(true) {
        std::cout << "I am New Thread!" << std::endl;
    }
}

void func() {
    std::thread t(threadproc);
}
```

```
int main() {
    func();
    while(true) {} //让主线程不要退出
    return 0;
}
```

以上代码再main函数中调用了func函数，在func函数中创建了一个线程，乍一看好像没有什么问题，但在实际运行是会崩溃。

崩溃的原因：

在func函数调用结束后，func中局部变量t（线程对象）被销毁，而此时线程函数仍在运行。所以在使用std::thread类时，必须保证线程函数运行期间其线程对象有效。

std::thread对象提供了一个detach方法，通过这个方法可以让线程对象与线程函数脱离关系，这样即使线程对象被销毁，也不影响线程函数的运行。

只需要在func函数中调用detach方法即可，代码如下：

```
// 其他代码保持不变
void func() {
    std::thread t(threadproc);
    t.detach();
}
```

2. lock_guard

ock_guard是一个互斥量包装程序，它提供了一种方便的RAII（Resource acquisition is initialization ）风格的机制来在作用域块的持续时间内拥有一个互斥量。

创建lockguard对象时，它将尝试获取提供给它的互斥锁的所有权。当控制流离开lockguard对象的作用域时，lock_guard析构并释放互斥量。

它的特点如下：

1. 创建即加锁，作用域结束自动析构并解锁，无需手工解锁
2. 不能中途解锁，必须等作用域结束才解锁
3. 不能复制

3. unique_lock

unique_lock是一个通用的互斥量锁定包装器，它允许延迟锁定，限时深度锁定，递归锁定，锁定所有权的转移以及与条件变量一起使用。

简单地讲，uniquelock 是 lockguard 的升级加强版，它具有 lock_guard 的所有功能，同时又具有其他很多方法，使用起来更强灵活方便，能够应对更复杂的锁定需要。

特点如下：

1. 创建时可以不锁定（通过指定第二个参数为std::defer_lock），而在需要时再锁定
2. 可以随时加锁解锁
3. 作用域规则同 lock_grard，析构时自动释放锁

4. 不可复制，可移动
5. 条件变量需要该类型的锁作为参数（此时必须使用unique_lock）

《Effective STL》

条款 01

> 慎重选择容器类型

1、需要在容器任意位置插入元素，就选择序列容器（vector string deque list ）

2、不关心容器中的元素是否是排序的，哈希容器可行

3、 你选择的容器是c++标准的一部分，就排除了哈希容器，slist(单链表)和rope（“重型”的string）

4、随机访问迭代器：vector，deque,string .rope, 双向迭代器：避免使用slist与哈希容器的一个实现

5、当发生元素的插入和删除，避免移动原来容器的元素移动很重要，那么避免使用连续内存的容器

6、需要兼容c，就vector

7、元素的查找速度为关键：哈希容器，排序的vector，标准关联容器（按速度顺序）

8、介意引用计数，就要避免string与rope

9、回滚能力（错了能改）：就要基于节点的容器。if 对多个元素的插入操作需要事务语义就list ;使用连续内存的容器也可以获得事务语义

10、基于节点的容器不会使迭代器、指针、引用变为无效

11、在string上使用swap会使迭代器、指针、或引用变为无效

12、 如果序列容器的迭代器是随机访问，只要没有删除操作发生，且插入操作只在末尾，则指向数据的引用和指针不会失效-deque

条款 02

> 不要试图编写独立于容器类型的代码

试图编写对序列容器和关联容器都适用的代码

条款 03

> 确保容器中的对象拷贝正确而高效

1、存入容器的是你的对象的拷贝

2、剥离问题：

向基类对象的容器添加派生类对象会导致，派生类对象所特有的信息被抹去。智能指针是个解决问题的好办法。

条款 04

> 调用empty而不是检查size()是否为0

理由很简单，empty()对所有的标准容器都是常数时间操作，而size()对于list耗费的是线性时间。

条款 05

> 区间成员函数优先于与之对应的单元素成员函数

好处：效率高易于理解，更能表达你的意图

区间创建、删除、赋值（assign）、插入可以用到区间成员函数

条款 06

> 当心c++编译器的烦人的分析机制—尽可能的解释为函数声明

```
/*三种同样形式的声明*/
int f(double d);
int f(double (d));
int f(double);

/*g是一个函数，该函数的参数是一个指向不带任何参数的函数的指针*/
int g(double (* pf) ());
int g(double pf ());
int g(double ());

list <int>data(istreamiterator<int>(datafile),istreamiterator<int>());
// 由上面分析可知，这是一个函数声明，跟我们想要做的事，大相径庭
// 第一个参数的名称是datafile ,其括号可省略，类型是istream_iterator<int>
// 第二个参数是一个函数指针，返回一个istream_iterator<int>
```

```
class Weight{...};
Weight w();
```

我是我们刚开始学习类的时候容易犯下的错误，我们想声明一个Weight函数，向进行默认初始化，编译器却给我什么了一个函数声明。

该函数不带任何参数，并返回一个Weight

解决方法1：

给函数参数加上括号

```
list <int>data( (istreamiterator<int>(datafile)) ,istreamiterator<int>());
```

解决方法2：

避免是使用匿名的istream_iterator迭代器对象，而是给这些迭代器一个名称(更好一点)

```
ifstream datafile("ints.dat");
istream_iterator<int> dateBegin(datafile);
istream_iterator<int> dataEnd;
list<int>data(dataBegin,dateEnd);
```

条款 07

> 容器包含指针

如果在容器中包含了通过new操作创建的对象的指针，切记在容器对象调用析构函数之前将指针delete掉

解决方法:

最简单的方法用智能指针代替指针容器，这里的智能指针通常是被引用计数的指针

```
void dosomething() {
    typedef boost::shardptr <Weight> SPW; //令SPW=shardptr<Weight>
    vector <SPW> vwp;
    for (int i = 0; i < SOMEMAGICNUMBER; ++i) {
        vwp.push_back(SPW(new Weight))//这里不会发生内存泄露，即使前面抛出异常。
        ...
    }
}
```

条款 08

> 切勿创建包含 auto_ptr 的容器

无论是被拷贝还是被复制，源对象都将失去对其资源的所有权。

而STL容器又是需要元素具有拷贝可赋值的属性的。

条款 09

当你复制一个auto_ptr时，它所指对象的所有权被移交到复制的对象，而他自身被置为NULL

条款 10

> 慎重选择删除元素的方法

1、要删除容器中有特定值的所有对象

如果容器时vector、string或duque, 则使用erase—remove

如果是list 则使用list::remove

如果容器是一个关联容器，则使用它的erase成员函数

2、要在循环内部做某些（除了删除对象的操作之外）操作

如果容器是一个**标准序列容器**，则写一个循环来遍历容器中的元素，记住每次调用erase时，要用它的返回值更新迭代器

如果是**关联容器**，写一个循环来遍历容器中的元素，记住当把迭代器传给erase时，要对他进行后缀递增

```
for(specalcontainer<int>::iterator i=c.begin();i!=c.end()) {
    if(badvalue(*i)){
        logFile<<"..."<<*i<<endl;
        i=c.erase(i);//对于vector，string，deque删除的一个元素不仅会导致这个元素的迭代器失效，同时会导致所有 的迭代器失效，所以必须更新迭代器。
    } else {
        ++i;
    }
}
```

要删除容器中满足判别式的所有对象

1. 如果容器时vector、string或duque, 则使用erase—remove_if
2. 如果是list 则使用list::remove_if

如果是关联容器，则使用removecopyif和swap（把我们需要的值复制到新容器，然后交换容器），或者写一个循环来编译容器中的元素，记住当把迭代器传给erase时，要对他进行后缀递增

```
container<int>c;
...
for(container<int>::iterator i=c.begin();i!=c.end();) {
    if(badvalue(*i)) c.erase(i++);//当该元素被删除的时候，该元素所有的迭代器都会失效，所以我们使用i++
    else ++i;
}
```

条款 11

> 了解分配子的约定与概念

分配子最初是作为内存模型的抽象，后来为了有利于开发作为对象形式的内存管理器，STL内存分配子负责分配和释放内存

1、首先分配子能够为它所定义的内存模型中的指针和引用提供类型定义

分别为 `allocator<T>::pointer` 与 `allocator<T>::reference` ,用户定义的分配子也应该提供这些类型定义，创建这种具有引用行为特点的对象是使用代理对象的一个例子，而代理对象会导致很多问题

2、库实现者可以忽略类型定义而直接使用指针和引用

允许每个库实现者假定每个分配子的指针类型等同于`T*`，引用为`T&`

3、STL实现者可以假定所有属于同一类型的分配子都是等价的

4、大多数标准容器从来没有单独使用过对应的分配子

例如list,当我们添加一个节点的时候我们并不是需要T的内存，而是要包含T的listNode的内存

所以说list从未需要allocator做任何内存分配，该list的分配子不能够提供list所需的分配内存的功能。

所以它会利用分配子提供的一个模板，根据list中T来决定listNode的分配子类型为：Allocator::rebind::other

这样就得到listNode 的分配子，就可以为list分配内存

new与allocator在分配内存的时候，**他们的接口不同**

```
void operator new(sizet bytes);
pointer allocator<T>::allocator(sizetype numberjects);
// pointer是个类型定义总是T
// 两者都带参数说名要分配多少内存，但是new 是指明一定的字节
// 而allocator ，它指明的内存中要容纳多少个T对象
// new返回的是一个void，而allocator<T>::allocate返回的是一个T
// 但是返回的指针并未指向T对象
// 因为T为被构造
// STL会期望allocator<T>::allocate的调用者最终在返回的内存中创建一个或者多个T对象
```

条款 12

> 编写自定义分配子需要什么

你的分配子是个模板，T代表为它分配对象的类型

提供模板类型定义，分别为allocator::pointer与allocator::reference

通常，分配子不应该有非静态对象

new返回的是一个void ，而allocator::allocate返回的是一个T ，但是返回的指针并未指向T对象，因为T为被构造，STL会期望allocator::allocate的调用者最终在返回的内存中创建一个或者多个T对象

一定要提供rebind模板

条款 13

> 理解分配子的用法

把STL容器中的内容放在共享内存中
把STL容器中的内容放到不同的堆中

条款 14

> 切勿对STL容器的线程安全性有不切实际的依赖

STL自身对多线程的支持非常有限，在需要修改STL容器或这调用STL算法时需要自己加锁。

为了实现异常安全，最好不要手动加锁解锁，多使用RAII。

条款 15

> 当你在动态分配数组的时候，请使用 vector 和 string

条款 16

> 使用reserve来避免不必要的重新分配

STL容器会自动增长以便容纳下其中的数据，只要没有超出他们的限制

vector与string 的增长实现过程：

分配一块大小为旧内存两倍的新内存，把容器中所有的元素复制到新内存中，析构旧内存的对象，释放旧内存。

reserve函数能够把你重新分配内存的次数减到最小，从而避免重新分配和指针、迭代器、引用失效带来的开销，所以应该尽早的使用reserve，最好是容器在被刚刚构造出来的时候就使用

4个易混函数(只有vector与string提供所有的这4个函数)

1. size()：告诉你容器中有多少个元素
2. capacity()：告诉你该容器利用已分配的内存能够容纳多少个元素，这是容器能够容纳元素的总数
3. resize(Container::size_type n)：强迫容器改变到包含n个元素的状态，如果size返回的数<n，则容器尾部的元素就会被析构，如果>n，则默认构造新的元素添加到容器的末尾，如果n要比当前的容器容量大，那么就会在添加元素之前，重新分配内存
4. reserve(Container::size_type n),强迫容器改变容量变为至少n，前提是不比当前的容量小，这会导致重新分配。

有两种方式避免不必要的内存分配

1. 你提前已经知道要用多少的元素，你此时就可以使用reserve。
2. 先预留足够大的空间，然后在去除多余的容量（如何去除，参照使用swap技巧）

条款 17

> string实现的多样性

1. string 的值可能会被引用计数
2. string对象的大小可能是char*的大小的1~7倍
3. 创建一个新的字符串可能会发生0，1，2次的动态分配内存
4. string也可能共享其容量，大小信息
5. string可能支持针对单个对象的分配子
6. 不同的实现对字符内存的最小分配单位有不同的策略

条款 18

> 了解如何把vector和string数据传给旧的API

```
if(!v.empty())
do something(&v[0],v.size());
or dosomething(v.c_str());
```

如何用来至C API的元素初始化一个vector

```
sizet fillArray(doublepArray,size_t arraySize);
vector< double >vd(maxnumbers);
vd.resize(fillArray(&v[0],vd.size()));
sizet fillString(charpArray,size_t arraySize);
vector< char >vc(maxnumbers);
size_t charWritten=fillString(&v[0],vd.size(0))
string s(vc.begin(),v.end()+charWrittrn)
```

条款 19

> 使用“swap技巧”删去多余的容量

```
vector<C> cs.swap(cs);
string s;
string (s).swap(s);
```

swap还可以删去一个容器

```
vector< C>().swap(cs);
string s;
string().swap(s);
```

条款 20

> swap的时候发生了什么

在swap的时候，不仅两个容器的元素被交换了，他们的迭代器，指针和引用依然有效（string除外），只是他们的元素已经在另一个容器里面。

条款 21

> 避免使用vector< bool >,用deque< bool >和bitset代替它

对于vector来说：

第一，它不是一个STL容器。
第二，它并不容纳bool。除此以外，就没有什么要反对的了。

条款 22

> 理解等价与相等

相等基于operator\==，一旦x==y则返回真，则x与y相等

等价关系是在已经排好序的的区间中对象值的相对顺序，每一个值都不在另一个值的前面。

!=（x < y)&&!=(y < x)。每个标注关联容器的比较函数是用户自定义的判别式，每个标准关联容器都是通过`key_comp`成员函数使排序判别式可被外部使用

条款 23

> 熟悉非标准散列容器

1. hashmap
2. hashset
3. hashmultimap
4. hashmultiset

条款 24

> 为包含指针的关联容器指定比较类型，而不是比较函数，最好是准备一个模板

```
struct Dfl {
    template<typename ptrtype>
    bool operator()(ptrtype pT1, ptrtype pT2) const {
        return pT1 < pT2;
    }
}
```

条款 25

> 切勿直接修改 set 或 multiset 中的键

set/multiset 的值不是const，map/multimap的键是const。

如何修改元素：

1. 找到想要修改的元素
2. 为将要修改的元素做一份拷贝。在map/multimap的情况下，不要把该拷贝的第一部分申明为const
3. 修改拷贝
4. 把该元素重容器中删除，一般用erase
5. 把新的值插入到容器中

条款 26

> 考虑用排序的vector替代关联容器

当程序使用数据结构的方式是：设置阶段、查找阶段、重组阶段,使用排序的vector容器可能比使用关联容器的效率要更好一点(当在使用数据结构的时候，查找操作不与删除添加操作混在一起的时候在考虑vector)

好处：

消耗更少的内存，运行的更快一些

注意：

当你使用vector来模仿map<const k,v>时，存储在vector中的是pair<k,v>，而不是pair<const k,v>；需要自己写3个自定义比较函数（用于排序的比较函数，用于查找的比较函数）

```
typedef pair<string, int> Data;
class Datacompare {
```

```
public:
    bool operator()(const Data &lhs, const Data &rhs) const {
        return keyless(lhs.first, rhs.first)
    }//用于排序的比较函数

    bool operator()(const Data &lhs, const Data::first_type &k) const {
        return keyless(lhs.first, k)
    }//用于查找的比较函数

    bool operator()(const Data::first_type &k, const Data &rhs) const {
        return keyless(k, rhs.first)
    }//用于查找的比较函数

private:
    bool keyless(const Data::firsttype &k1, const Data::firsttype &k2) const {
        return k1 < k2;
    }//为了保证operator () 的一致性
}
```

条款 27

> 更新一个已有的映射表元素

如果要更新一个已有的映射表元素，则应该选择operator[]，如果是添加元素，那么最好还是选择insert。

条款 28

> Iterator

iterator优先于constiterator,reserveiterator,constreserveiterator

条款 29

> 使用distance和advance将容器的const_iterator转换为iterator

```
typedef deque<int> IntDeque;
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;
IntDeque d;
ConstIter ci;
...
Iter i(d.begin());
advance(i, distance<ConstIter>(i, ci));
```

条款 30

> 正确理解由reserve_iterator

正确理解由reserve_iterator的base（）成员函数所产生的iterator的用法

1. 对于插入操作，ri和ri.base()是等价的
2. 对于删除操作，ri和ri.base()不是等价的
3. v.erase((++ri).base());

条款 31

> istreambuf_iterator

对于逐个字符的输入请考虑使用istreambuf_iterator

1. `ifstream inputFile("sdsdsa.txt");`
2. `string filedate((istreambufiterator<char>(inputFile),istreambufiterator<char>());`

条款 32

> 如果所使用的算法需要指定一个目标空间

如果所使用的算法需要指定一个目标空间，确保目标区间足够大或确保它会随着算法的运行而增大。

要在算法执行过程中中增大目标区间，请使用插入型迭代器：backinserter,frontinserter,ostream_iterator【插入器，它接受一个容器，生成一个迭代器，能实现向给定容器添加元素】

条款 33

> 了解各种与排序相关的选择

1. 如果需要对vector string，deque，或者数组中的元素进行一次完全排序，那么可以使用sort和stable_sort
2. 如果有一个vector，string，deque或者数组，并且只需要对等价性最前面的n个元素进行排序，那就是可以使用partial_sort
3. 如果有一个vector，string，deque或者数组，并且只需要找到第n个位置上的元素，或者，并且只需要找到等价性最前面的n个元素，并不需要排序，那么nth_element就行
4. 如果需要将一个标准序列容器中的元素按照是否满足某个特定区间区分开，那么就选择partition、stable_partition
5. 如果你的数据在list中，那么你可以选择内置的sort和stablesort算法。同时如果你需要活得partitionsort或nth_element算法的效果可采用一些间接的方法【effective stl p114】
6. 对于排序算法的选择应该基于功能而选择，而不是基于性能

条款 34

> 如果要删除元素，需要在remove后面使用erase

1. remove并没有删除容器中的元素：因为remove它不是成员函数（list除外），所以它不知道要删除那个容器的元素。它会把不被删除的元素排在前面，要删除的元素排在后面，所有它返回的一个指针指向最后一个不被删除的元素的后面的那个元素
2. 我们就要使用在remove后面使用erase函数。v.erase(remove(v.begin(),v.end(),elemnet),v.end())
3. 还有两类函数也是这种情况：remove_if,unique;

4. 但是list函数把remove与erase结合在一起生成的list::remove比原先的remove-erase的效率高。unique和remove_if同理

条款 35

> 对包含指针的容器使用remove这一类算法要小心

原因：由于remove是将那些要被删除的指针被那些不需要被删除的指针覆盖了，所以没有指针指向那些被删除指针所指向的内存和资源，所有资源就泄露了

做法：使用智能指针或者在使用remove-erase之前手动删除指针并把他们置为空

条款 36

> 了解那些算法要求使用排序的区间作为参数

```
template<typename InputIterator
typename OutputIterator
typename predicate>
OutputIterator copy_if(InputIterator begin, InputIterator end, OutputIterator
destbegin, predicate p) {
    while (begin != end) {
        if (p(*begin))
            destbegin++ = begin;
        ++begin;
    }
    return destbegin;
}
```

条款 37

> 使用accumulate或者for_each进行区间统计

1. accumulate (innerproducet、adjacentdifference、partial_sum)位于< numeric >中
2. for_each(区间，函数对象)
3. accumulate（begin,end,初始值）；accumulate（初始值，统计函数)

条款 38

> 遵循按值传递的原则来设计函数子类

如果做能够允许函数对象可以很大、或者保留多态，又可以与STL所采用的按值传递函数指针的习惯保持一致：将数据和虚函数从函数子类中分离出来，放到一个新的类中；然后在函数子类中包含一个指针，指向这一个心类的对象。

```
template<typename T>
class Bs : public:unary_function<T, void> {
private:
    Weight w;
    int x;
```

```
...
virtual ~Bs();
virtual void operator()(const T &val) const;
friend class B<T>
}

template<typename T>
class B : public:unary_function<T, void> {
private:
    BS <T> *p;
public:
    virtual void operator()(const T &val) const {
        p->
        operator()(val);
    }
}
```

> 这样的设计模式：effective c++ 34有介绍

条款 41

> 确保判别式是“纯函数”

对于用作判别式的函数对象，使用时它会被拷贝存起来，然后再使用这个拷贝。这一特性要求判别式函数必须是纯函数。

条款 42

> 使你的函数子类可配接

为什么：

1.可配接的函数对象能够与其他STL组件默契的协同工作
2.能够让你的函数子类拥有必要的类型定义

为什么：

4个标准的函数配接器（not1,not2,bind1st,bind2nd)要求这些类型定义

为什么not1等需要这些定义：能够辅助他们完成一些功能

如何使函数可配接：让函数子重特定的基类继承：unaryfuntion与binaryfunction

注意：`unaryfuntion<operator所带参数类型，返回类型>` `binaryfunction<operator 1, operator 2,返回类型>`

条款 43

> 理解ptrfun && memfun && memfunref

在函数和函数对象被调用的时候，总是使用非成员函数形式发f(),而当你使用成员函数的形式时x.f(),p->f();

这将通不过编译，所有使用上面的那些东西，就能调整成员函数，使其能够以成员函数的形式调用函数和函数对象。

每次将成员函数传给STL组件的时候，就要使用他们。

条款 44

> 确保less< T >与operator<的语义相同

一般情况下我们使用less< T >都是默认通过operator<来排序。

如果你想要实现不同的比较，最好是重新写一个类，而不是修改特化修改less

条款 45

> 算法的调用优先于手写的循环

1. 效率高
2. 自己手写的循环更容易出错
3. 算法代码比我们自己写的更简单明了，利于维护

条款 46

> 容器的成员函数优先于同名函数

1. 成员函数往往速度快。
2. 成员函数通常与容器结合地更紧密，这是算法所不能比的。

条款 47

> 正确区分以下关键字

count、find、binarysearch、lowerbound,upperbound、equalrange

effective stl p167,表格详细介绍了算法的使用

条款 48

> 使用函数对象作为STL算法的参数

条款 49

> 避免产生“直写行”的代码

不利于阅读和维护

条款 50

> 包含正确的头文件

1. 几乎所有的STL容器都被声明在与之同名的头文件中
2. 除了4个STL算法外，其他所有的算法都被声明在< algorithm >中；accumulate (innerproducet、adjacentdifference、partial_sum)位于< numeric >中

3. 特殊类型的迭代器（istreamiterator,istreambufiterator）被声明在< iterator >
4. 标注的函数子（less< T >）和函数配接器< not1,bind2nd >b被声明在头文件< funtional >中

条款 51

> 学会分析于STL相关的编译器的诊断信息

条款 52

> 熟悉于STL相关的web站点

1. SGI STL
2. STLport
3. Boost

《Effective C++》

条款 05

了解C++默默编写并调用哪些函数

一个空类，编译器会自动声明：

1. 默认构造函数（对于非空类：只在没有自定义任何构造函数的时候，才会由编译器补充）
2. 拷贝构造函数
3. 拷贝赋值运算符
4. 析构函数（非虚函数）

注意：

当一个class内含有reference/const成员时，编译器不提供拷贝赋值运算符的补充，只能由程序员自己编写

这是因为C++不允许改变reference成员的指向，也不允许更改const成员

条款 06

问题：

比如说想要禁止一个类对象的拷贝操作，就要禁止拷贝构造函数和拷贝赋值运算符。但是不声明这两个函数，编译器可能会自动生成；要想避免编译器自动生成，又要自己声明一份；怎样都避免不了产生这两个函数。

解决方案1：

将拷贝构造函数、拷贝赋值运算符声明为private函数，这样在类外部不能以拷贝构造和复制的形式公然调用这两个函数

```
class A {

public:

    A() = default;

private:

    A(const A&);
    A& operator=(const A&);
};
```

解决方案1存在的问题：

以class A为例，A的其他成员函数和友元函数还是能调用class A的private函数。由此引出解决方案2

解决方案2：

定义一个基类专门阻止拷贝动作

```
class unCopyable {
protected:

    unCopyable() {}
    ~unCopyable() {}

private:

    unCopyable(const unCopyable&) {}
    unCopyable& operator=(const unCopyable&) {}
};


class A : private unCopyable{
public:

    A() = default;

};
```

此时，就算是class A的成员函数和友元函数，在尝试拷贝A的对象时，编译器会“试着“生成拷贝构造函数和拷贝赋值运算符，但是这种”试着“会被基类阻止，因为基类的对应函数时private，子类不能调用

条款 07

为多态基类声明virtual析构函数

当一个基类指针指向一个子类对象时，若基类的析构函数不是virtual函数，那么在delete基类指针的时候，只会释放基类对象的资源，不会释放子类对象的资源，从而造成内存泄漏

任何带有virtual函数的class都应该有一个virtual析构函数

虚函数的实现原理：

1. 每个含有virtual函数的class都有一个vtbl（虚函数表），vtbl中存储的是指向各个虚函数的函数指针；
2. 每个对象内存空间内存在一个vptr（虚指针），这个vptr指向类的vtbl
3. 当对象调用某个virtual函数的时候，编译器根据对象的vptr找到类的vtbl，在vtbl中寻找适当的函数指针

注意：

不作为基类使用的class不要声明析构函数为virtual函数，因为虚表和虚表指针会占用额外的内存；同时，std::string和STL容器的析构函数都是non-virtual，不要作为基类使用

条款 08

别让异常逃离析构函数

1. C++并不禁止析构函数抛出异常，但是不建议这样做
2. 有两个异常存在的情况下，程序不是结束执行就是导致不明确的行为

如果无法避免析构函数产生异常的可能性，有两种办法：

1、若析构函数爬出异常，就调用abort结束程序

```
A::~A() {
    try {a.func();}
    catch(...) {
        //记录调用a.func()过程中出现异常
        abort();
    }
}
```

2、吞下异常，当做什么也没发生过

```
A::~A() {
    try {a.func();}
    catch(...) {
        //记录调用a.func()过程中出现异常
    }
}
```

条款 09

绝不在构造和析构过程中调用virtual函数

子类对象调用子类构造函数之前，会先调用父类构造函数。若子类重写了父类中的一个虚函数func()，父类的构造函数中调用了func()函数，那么当子类对象调用父类的构造函数执行到这一句时，会调用父类版本的func()函数。

这是因为：

父类的构造函数的执行早于子类的构造函数，当父类的构造函数执行时，子类的成员变量尚未初始化，如果此时调用的虚函数下降至子类层次，会存在使用子类未初始化成员的风险，因此C++禁止这种危险。

更根本的原因是：

子类对象在调用父类构造函数期间，对象类型是基类而不是子类，不仅虚函数会使用父类版本，此时使用dynamic_cast和typeid，也会把对象视为基类类型

析构函数也是一样的原因

条款 10

令operator=返回一个绑定到*this的引用

形式：

```
class A {
public:
    ...
    A& operator=(const A& other) {
        ...
        return *this;
    }
}
```

这是为了实现连锁赋值，即：

```
int x, y, z;
x = y = z = 100;
//运行原理
x = (y = (z = 100))
```

条款 11

在operator=中处理自我赋值

自我赋值的意思就是：

等号右边的东西和等号左边的东西是同一份（地址相同）

```
a[i] = a[j];// i == j成立
```

一般的做法是在赋值前做一个检查：

```
A& A::operator=(const A& other) {
    if (this == &other) // 比较双方地址
        return *this;
    delete pb;
    pb = new B(*other.pb);
    return *this;
}
```

条款 12

赋值对象时勿忘其每一个成分

复制函数包括：拷贝构造函数、拷贝赋值运算符

当自定义了复制函数后，编译器不会再生成补充版本（即便自定义的有问题）

1. 当为class添加一个成员变量，必须同时修改复制函数
2. 为子类自定义复制函数时，要注意对基类成员的复制。基类成员通常是private，子类无法直接访问，应该让子类的复制函数调用相应的基类复制函数

不能为了简化代码，就在拷贝构造函数中调用拷贝赋值运算符，也不能在拷贝赋值运算符中调用拷贝构造函数。因为构造函数用来初始化新对象，而赋值运算符只能用于已初始化的对象之上。

要想消除复制函数的重复代码，可以建立一个新的成员函数给复制函数调用，这个函数通常是private且命名为init

条款 16

成对使用new和delete

new 创建，delete 删除
new[] 创建，[]delete 删除

条款 30

了解inline的里里外外

inline函数：

对函数的每一个调用都用函数本体替代，调用不承受额外开销，编译器对其执行语境相关最优化。增加目标码大小，额外的换页行为，降低缓存击中率，效率损失。

对虚函数进行inline无意义，虚函数是运行时确定，inline是在编译期替换。

编译器一般不对“通过函数指针进行调用”提供inline，是否inline取决于调用的方式。

条款 34

区分接口继承和实现继承

public继承由函数接口继承+函数实现继承组成。

纯虚函数两个特性：

1. 它们必须被任何继承了它们的具象class重新声明
2. 在抽象class中通常没有定义

声明一个纯虚函数的目的是为了让派生类只继承函数接口。
只提供接口，派生类根据自身去实现。

声明非纯虚函数的目的是让派生类继承函数的接口和缺省实现。
必须支持一个虚函数，如果不想重新写一个（override），可以使用基类提供的缺省版本。

声明非虚函数的目的是令派生类继承函数接口和一份强制性实现
任何派生类都不应该尝试修改次函数，non-virtual函数代表不变性>特异性，不应该在派生类被重新定义。

条款 35

考虑virtual函数以外的其他选择

1. non-virtual interface，以public non-virtual成员函数包裹较低访问性（private/protected）的虚函数。
2. virtual函数替换成“函数指针成员变量”。
3. tr1::function成员变量替换virtual。

条款 36

绝不重新定义继承而来的非虚函数

public继承说明，每个派生类对象都是基类对象，非虚函数(静态绑定)一定会继承基类的接口和实现。

重新定义则设计出现矛盾。派生类重新定义使得出现特化，这样就不一定适用于基类，那么就不应该public。

条款 37

绝不重新定义继承而来的缺省参数值

因为缺省参数值是静态绑定，虚函数是动态绑定。

静态类型是程序中被声明时采用的类型，动态类型是目前所指对象的类型。

条款 39

明智而审慎的使用private继承

如果派生类需要访问基类保护的成员，或需要重新定义继承来的虚函数，采用private继承。

条款 40

明智而审慎的使用多重继承

多重继承：一个类同时继承多个类

多继承中实现派生类中只有一份数据，虚继承。

虚继承会增加大小，速度，初始化等成本。

最好不要使用虚继承或者虚基类中不放置数据。

条款 44

将与参数无关的代码抽离 templaes

Template 生成多个 classes 和多个函数，所以任何 template 代码都不该与某个造成膨胀的 template 参数产生相依关系。

因非类型模板参数（non-type template parameters）而造成的代码膨胀，往往可消除，做法是以函数参数或 class 成员变量替换 template 参数。

因类型参数（type parameters）而造成的代码膨胀，往往可降低，做法是让带有相同二进制表述的具现类型共享实现码。

条款 45

运用成员函数模板接受所有兼容类型

如果你声明 member templates 用于“泛化 copy 构造”或“泛化assignment 操作”，你还是需要声明正常的 copy 构造函数和 copy assignment 操作符。

条款 46

需要类型转换时请为模板定义非成员函数

当我们编写一个class template，而它所提供之“与此template相关的”函数支持“所有参数之隐式类型转换”时，

请将那些函数定义为“class template 内部的 friend 函数”。

条款 47

请使用 traits classes 表现类型信息

Traits classes 使得“类型相关信息”在编译期可用。

它们以 templates 和“templates 特化”完成实现。整合重载技术后，traits classes 有可能在编译期队类型执行 if...else 测试。

条款 48

认识 template 元编程

Template metaprogramming（TMP，模板元编程）可将工作有运行期移往编译期，因而得以实现早起错误侦测和更高的执行效率。

TMP 可被用来生成“基于政策选择组合”（based on combinations of policy choices）的客户定制代码，

也可用来避免生成对某些特殊类型并不合适的代码。