C++语法

参考

- 程序喵大人

基础

- Conception：
  a. C和C++区别、和Java区别、和python区别
  b. 预处理、编译、汇编、链接
  c. 动态链接、静态链接
  d. 动态编译、静态编译
  e. 动态联编、静态联编
  f. 动态绑定、静态绑定
  g. 动态内存分配、静态内存分配
  h. 函数调用过程
- Question：
  a. 预处理、编译、汇编、链接?
    - 预处理：展开头文件、宏替换、去掉注释
    - 编译：生成汇编
    - 汇编：汇编指令转换为机器指令
    - 链接：链接到一起生成可执行程序
  b. C和C++有什么区别?
    - C++是面向对象的语言，C是面向过程的语言
    - C++引入new/delete的概念，取代了C中malloc/free的概念
    - C++引入引用，C没有
    - C++引入类，C没有
    - C++引入函数重载，C没有
  c. C++和Python的区别?
    - C++是编译语言，python是脚本语言
    - C++用花括号，python用缩进
    - C++需要事先定义变量，python不需要定义
    - C++库函数少，python库函数多
  d. C++和Java的区别?
    - Java移植性强
    - Java没有指针的概念
    - Java用接口，C++用抽象类
    - Java自动分配和回收内存，C++需要自己注意

- Java适合Web应用，C++适合桌面程序

e. 从源文件到可执行文件？

- 预处理、编译、汇编、链接

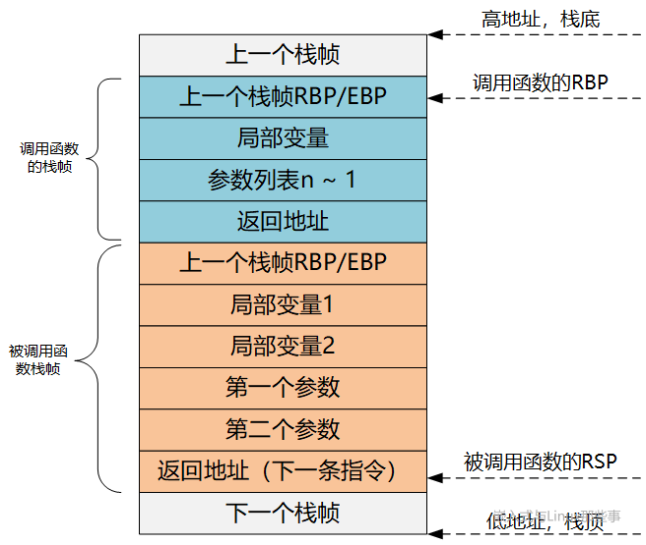
f. 动态链接、静态链接？

- 动态链接：通过记录一系列符号和参数，程序运行时将信息传递给操作系统，操作系统将需要的动态库加载到内存中，运行时链接
  - 优点：共享一个动态库，节省资源
  - 缺点：运行时加载，影响性能
- 静态链接：编译链接时直接将执行的代码拷贝到调用处
  - 优点：不用依赖库，可以独立执行
  - 缺点：体积大；静态库更新需要重新链接

g. **函数调用过程**

```c
#include <stdio.h>

int sum (int a,int b)
{
    return a + b;
}

int mul(int a,int b){
    return a * b;
}
int main()
{
    int x = 5,y = 10,z = 0;
    z = sum(x, y);
    z = mul(x, y);
    return 0;
}
```

- A调用B，B的返回值存在eax寄存器中
- A向上增长，遇到sum()进去执行sum()，然后回退到当前位置；然后再执行后面的函数



C++基础语法

- Conception：

a. 定义相关
b. 声明、定义
c. 全局变量、局部变量
d. 值传递、引用传递、指针传递
e. 指针相关
f. 指针、引用
g. 指针、句柄
h. 函数传递参数（形参、实参 值传递、指针传递、引用传递）
i. 悬挂指针、野指针
j. 指针常量、常量指针
k. 指针数组、数组指针
l. 函数指针
m. 关键字相关
n. static
o. #define
p. typedef
q. const 指针常量、常量指针 顶层const、底层const
r. volatile、mutable、explicit
s. inline
t. #ifdef、#else、#endif、#ifndef
u. #program once
v. extern"C"
w. ++i、i++
x. 函数相关
y. sizeof、strlen
z. strcpy、sprintf、memcpy
aa. RAII、RTTI
ab. 其他
ac. 异常处理（try catch throw exception类）

- Question：
  a. 声明、定义？
     - 声明：不分配地址；可以多个地方声明
     - 定义：分配地址和存储空间；只能定义一次
     - extern int a：修饰的是变量的声明，此变量将在文件外/后面进行定义
  b. **全局变量放在头文件还是cpp文件**
     - .cpp文件
     - C++不允许多次定义变量，声明可以无数次，如果放在头文件，被其他文件包含多次，就定义了多次
  c. 全局变量和局部变量有什么区别？操作系统和编译器怎么知道？

- 全局变量：整个程序都可以访问；程序运行结束释放；存放在全局数据段
- 局部变量：所在模块可以访问；函数调用完毕局部变量消失，内存释放；分配在堆栈
- 操作系统和编译器根据内存分配位置判断

d. 参数传递的方式?
- 值传递
- 引用传递
- 指针传递

e. 值传递、引用传递、指针传递?
- 值传递：
  - 函数内部修改不会影响参数真实的值
- 指针传递：
  - 传入指向实参地址的指针
- 引用传递：
  - 实参的别名，间接寻址找到变量

f. 指针和引用?

	指针	**引用**
	拥有自己的空间	只是一个别名
sizeof大小	4	被引用对象的大小
参数传递	必须解引用才能操作对象	直接可以修改
const	**有const指针**	**没有const引用**
	可以指向别的对象	不能被改变
	多级指针	只有一级
返回动态内容分配的对象或者内存	必须用指针	引用可能引起内存泄露

g. **为什么要有引用?**
- 因为引用和值有一样的语义，而指针不是
- 不存在空引用，必须初始化；保证值不变，保证编译器更加安全
- 加减号、赋值操作符，作用在引用上会触发对象的操作符重载，但作用在指针上不会

h. 指针和句柄?
- 指针：指向系统中物理内存的地址
- 句柄：一种指向指针的指针，Windows用句柄标记系统资源，隐藏系统的信息

i. 数组和指针?
- 相同：作为形参的时候，传入的都是数组的首地址
- 地址是否可变：数组体现出指针常量的特性，不能对首地址进行修改；指针指向的地址可以变化
- 大小：sizeof数组是整个数组大小；sizeof指针是指针大小

- 指针比数组更灵活

j. 数组中a和&a有什么区别?

```
int a[10];
int (*p)[10] = &a;

a是数组名，也是数组元素首地址;
a+1表示地址+1，表示a[1]的地址;
*(a + 1) = a[1];

&a是数组的指针，类型为int (*)[10]（数组指针）
&a + 1表示数组首地址加上整个数组的偏移，表示a尾元素再下一个元素的地址
```

k. 指针相关（常量指针/指针常量、悬挂指针/野指针、指针数组/数组指针、函数指针）

- 5-1常量指针和指针常量?
  - 常量指针：
    - 常量是指针？×
    - 所以不能用这个指针修改变量值，原来的声明可以修改值
    - const * 常量 指针
  - const int *p;
    int const *p;
  - 指针常量：
    - 指针是常量？√
    - 所以指针指向的东西不能改变：指向哪个变量就是哪个，不能修改
    - * const 指针 常量
  - int *const p;
- 5-2悬挂指针和野指针?
  - 悬挂指针：指针所指对象已经被释放
  - 野指针：未初始化的指针
  - 如何避免野指针？①初始化时指向NULL；②释放之后指向NULL；
- 5-3数组指针、指针数组?
  - http://c.biancheng.net/view/335.html
- 5-4函数指针?

```
int(*)(): 函数指针
int(): 函数原型

//函数指针
int (*fp)(int , int) = f;
(*fp)(a, b);


//返回值为指针
int * fp(int , int)
```

- 函数指针的好处：
  - 涉及另一个函数，但是函数名未定，要当成函数指针
- 函数指针的使用场景：
  - 函数回调
  - 虚函数指针
- ((void()())0)()什么含义？
  - void (*)()：函数指针；无参数返回值为空*
  - (void (*)()) 0：把0强制转换成函数指针类型，一个函数存在首地址为0的一块区域内
  - (void ()()) 0：取0地址开始的一段内存中的内容，内容也就是函数的内容
  - (void()()) 0)()：函数调用

l. static关键字？

- 修饰局部变量：静态存储区（全局数据区）分配内存；首次函数调用中初始化，之后的函数调用不再初始化；局部作用域内可见
- 修饰全局变量：静态存储区（全局数据区）分配内存；整个文件内可见，文件外不可见
- 修饰函数：整个文件可以，文件外不可见；避免函数同名冲突（static修饰函数的弊端：**仅在本文件内可见**）
- 修饰成员变量：所有对象共享；类外初始化；不需要对象实例化就可以访问
- 修饰成员函数：所不能访问非静态成员（变量，不能调用非静态成员函数）原因是不能接受this指针，与任何对象无关；只能访问静态成员；不需要对象实例化就可以访问
- C和C++中的区别：C只能修饰局部变量和全局变量、函数，C++还能修饰成员变量和成员函数

m. inline

- inline必须和定义放在一起，和声明放在一起不起作用

```
1  //如下风格的函数Foo 不能成为内联函数:
2  inline void Foo(int x, int y); // inline 仅与函数声明放在一起
3  void Foo(int x, int y)
4  {
5  }
6  //而如下风格的函数Foo 则成为内联函数:
7  void Foo(int x, int y);
8  inline void Foo(int x, int y) // inline 与函数定义体放在一起
9  {
10 }
```

- 定义在类声明之中的成员函数将自动地成为内联函数
- 内联声明只是建议，是否内联由编译器决定，实际不可控
- 递归函数(自己调用自己的函数)是不能被用来做内联函数的
- 优点：
  - 调用地方展开，省去调用时间，提高效率
  - 相比#define宏函数，代码展开时会进行语法安全检查或数据类型转换，更加安全
- 缺点：
  - 代码膨胀，开销大

- 如果内联函数本身执行时间长，效率提升小
- 修改内联函数，所有调用内联函数的文件必须重新编译

n. **virtual虚函数可以是inline内联函数吗**

- 虚函数可以是内联函数，但是当虚函数表现多态时候不能内联；内联是在编译期间进行内联，多态是在运行期间，编译器无法知道调用哪个代码

o. 内联函数和宏的区别?

- 内联函数：
  - 编译阶段；
  - 本身是函数，会进行类型检查，更安全
  - 内联函数有作用域和访问权限（private等）
- 宏：
  - 预处理阶段；
  - 本身不是函数，不仅限类型检查，简单的替换
  - 宏没有限定的作用域和访问权限
  - assert是宏不是函数

p. const

- 8-1常量指针、指针常量
  - 常量指针：
    - 常量是指针？×
    - 所以不能用这个指针修改变量值，原来的声明可以修改值
    - const * 常量 指针
  - const int *p;
    int const *p;
  - 指针常量：
    - 指针是常量？√
    - 所以指针指向的东西不能改变：指向哪个变量就是哪个，不能修改
    - * const 指针 常量
  - int *const p;
- 8-2顶层const、底层const
  - 顶层：修饰指针（指针常量），指向不可变，指向的变量内容可变
  - 底层：修饰变量（常量指针）
  - 从右往左读变量声明，先修饰左边，左边没有修饰右边
  - const在*左边，修饰变量
  - const在*右边，修饰指针
- const函数
  - 变量：const函数承诺内部不会修改成员变量（multable）
  - 函数：const函数只能调用const函数，非const函数可以调用const函数
  - 类体外定义的声明和定义处都需要写const

- 如果 const 构成函数重载，const 对象只能调用 const 函数，非 const 对象优先调用非 const 函数。

q. #define作用
   - 定义标识、定义常数、定义函数
   - 提高速度，易读性增强

r. const和#define区别
   - 编译器处理方式不同：#define是预处理阶段展开；const是编译阶段使用
   - 类型和安全检查不同：#define没有类型，不做任何检查；const编译阶段会执行检查
   - 存储方式不同：#define代码展开，多个地方进行替换，不会分配内存；const分配内存，只维持一份拷贝
   - 定义域不同：#define不受定义域限制；const只在定义域内有效

s. typedef和define区别

	typedef	**define**
用法	定义数据类型的别名	定义常量以及宏
执行时间	编译时处理，有类型检查功能	预编译时期，不检查，只是替换，编译时候展开运行发生错误才报错
作用域	有作用域限定	不受作用域约束，define声明后的引用都正确
指针	typedef int*	

t. #ifdef、#else、#endif、#ifndef
   - 条件编译：对一部分内容指定编译条件

```
函数内部编译某些程序段
#ifdef 条件
        程序段
#else
        程序段
#endif
```

   - 用if语句也能达到要求，但是编译了所有语句；条件编译可以减少被编译的语句

```
避免头文件重定义
#ifndef CLASSA_H
#defind CLASSA_H
...
#endif
```

u. 防止C++头文件被重复引用?
   - #ifndef
   - #program once
   - #ifndef兼容性更好，老的编译器不支持#program once

v. extern

- 告诉编译器存在一个变量/函数，在文件的后面/其他的文件中定义

w. extern"C"

- 意义：指示编译器这部分代码按C语言（而不是C++）的方式进行编译，正确实现C++代码调用其他C语言代码
- C不支持函数重载，编译函数时不会带上函数的参数类型，一般只包括函数名
- C++支持函数重载，编译函数时会带上函数的参数类型
- 使用场景：C++代码调用C语言代码；多个人协同开发

```
extern "C"
{
        extern int i;
        extern void func();
}
```

x. **头文件中声明变量有什么问题?**

- 多个cpp文件引用，会报错重复定义
- 解决
  - 定义为const：const**常量**链接性为内部，相当于引用的每一个cpp文件定义一个独立的int a;
  - 声明为extern int a：需要在一个cpp文件中定义：`int a;`

y. volatile

- 表示变量随时可能被改变，编译后程序读取时候直接从地址读入，避免编译器优化从寄存器中读
- 建立内存屏障，多线程环境中
- 使用场景：
  - 多线程中被任务共享的变量
  - 中断服务程序中修改的供其他程序检测的变量
- 一个参数可以既是const又是volatile吗?
  - 可以，表示程序内部只读不能改变（const），程序外部条件变化下改变且编译器不会优化这个变量（vilatile）
  - 原因：const不允许代码改变变量，但没有禁止某段内存的读写特性

z. mutable

- 被修饰对象在任何情况下都可以被改变，突破const的限制
- 在const函数或者const对象中可以进行改变（const函数不能修改**类的成员变量**）

aa. ++i、i++

- 前置++不会产生i你是对象，后置必须产生临时对象

```
int operator++(){
    *this += 1;
    return *this;
}

int operator++(int){
```

```
    int temp = *this;
    ++*this;
    return temp;
}
```

ab. sizeof、strlen的区别?

	sizeof	**strlen**
类型	操作符	库函数
参数	可以是数据类型，也可以是变量	字符串
计算时机	**编译期间算出结果**	**运行时才算出结果**
数组为参	做sizeof参数不退化	传递给strlen就退化为指针了

ac. strcpy、sprintf、memcpy的区别?

	strcpy	**sprintf**	**memcpy**
操作对象	只能为字符串	源为多种数据类型，目的为字符串	复制任意内容
执行效率	中等	低	最高
功能	**字符串直接拷贝**	其他数据类型格式到字符串的转换	**内存块的拷贝**
复制长度	不需要指定长度		void *memcpy(void *destin, void *source, unsigned n) 根据第三个参数决定

ad. RAII、RTTI

- RAII：资源获取即初始化
- RTTI：运行时类型识别

C++面向对象

- Conception：

a. struct和class

b. 访问权限public private protected

c. 三大特征：封装、继承、多态

d. 静态多态/静态绑定、动态多态/动态绑定

e.

f. 初始化、赋值

g. 直接初始化、拷贝初始化

h. 构造函数（默认构造函数、重载构造函数、拷贝构造函数、移动构造函数）

i. 拷贝构造函数、赋值运算符

j. this指针

k. 成员初始化列表

l.

m. 析构函数

n.

o. 虚函数、纯虚函数、虚函数表（virtual table）、虚函数指针

p.

q. 继承 组合

r.

s. 重载、隐藏、重写/覆盖

t.

u. 四个强制类型转换符（static_cast、dynamic_cast、const_cast、reinterpret_const）

v. RTTI

w.

x. 类的大小、空类大小

y. 空类相关

z.

aa. 模板函数、模板类

ab. 模板函数、模板类的特例化

ac.

ad. 深拷贝、浅拷贝

ae. 友元函数、友元类

af. final、override

ag.

ah. friend

- Question：

a. 结构体？

   - 声明时候可以直接初始化
   - 结构体可以直接赋值吗？：同一结构体的不同对象之间可以直接赋值，含有指针时候要小心（多个指针指向同一段内存，可能发生未知的内存释放）

b. C和C++的struct有什么不同？

   - C不能有成员函数；C++可以有成员函数

- C中没有private、public、protected访问限定符；C++中有
- C中没有继承关系；C++中有继承关系

c. C++中struct和class的区别？

- 访问权限：struct默认public，class则默认private
- 继承：struct默认public继承，class则默认private继承
- 抽象层面：class像对象的实现体，struct是数据结构的实现体
- struct不用于定义模板参数，class还能用于定义模板参数
- template<class T>
  template <typename T>
- C++的struct定义必须保证C中struct的向下兼容性，所以类通常定义为class

d. 面向对象三大特征？

- 封装：将客观事务封装成抽象的类；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
- 继承：使用现有类的所有功能，无需重新编写即可实现功能的拓展
- 多态：对不同对象发送同一个消息，不同对象会做出不同的响应
  - 函数重载、函数模板时候体现出来的多态
  - 调用成员函数时候，根据指针指向对象的类型执行不同的函数（基类指针/引用指向派生类对象）

e. 多态的实现方式？

- 静态多态/静态绑定：编译期间确定；函数重载、函数模板
- 动态多态/动态绑定：运行期间确定；虚函数+继承实现，基类指针/引用运行期间再决定使用哪个函数

f. 动态多态/动态绑定作用？必要条件？如何实现？

- 作用：提高代码复用性；使基类指针/引用可以使用派生类功能，接口重用，向后兼容，提高可扩充性和可维护性
- 必要条件：继承、虚函数（virtual）、基类指针/引用
- 如何实现：
  - 编译器发现含有虚函数的类，会创建一个虚函数表，对象中会有一个vptr指向虚函数表；
  - 基类指针指向派生类对象时候，vptr替换，就指向了派生类的虚函数表

g. 访问权限？

- public、private、protected
- 类内部均可相互访问
- 类外部，只能访问public

h. **protected做了什么事情？**

- private成员只能被本类成员（类内）和友元访问，**不能被派生类访问**；
- **protected成员可以被派生类访问。**

- protected继承之后会改变继承来的成员属性，从而继续导致成员变量的访问权限

i. 初始化、赋值？

- int a = 100;
  a = 100

j. 直接初始化、拷贝/复制初始化?

- 直接初始化()：直接选择调用类的各种构造函数
- 拷贝初始化=：先创建一个临时对象（允许跳过），再调用各种拷贝构造函数

```
//直接初始化
string a; //调用默认构造函数
string a("hello"); //调用参数为const char *类型的构造函数
string b(a); //调用拷贝构造函数

//拷贝初始化
string a="hello"; //先调用const char *类型的构造函数创建临时对象，然后调用拷贝构
string b=a; //a已经存在，直接调用拷贝构造函数
```

k. 为什么基类的构造函数不能为定义为虚函数?

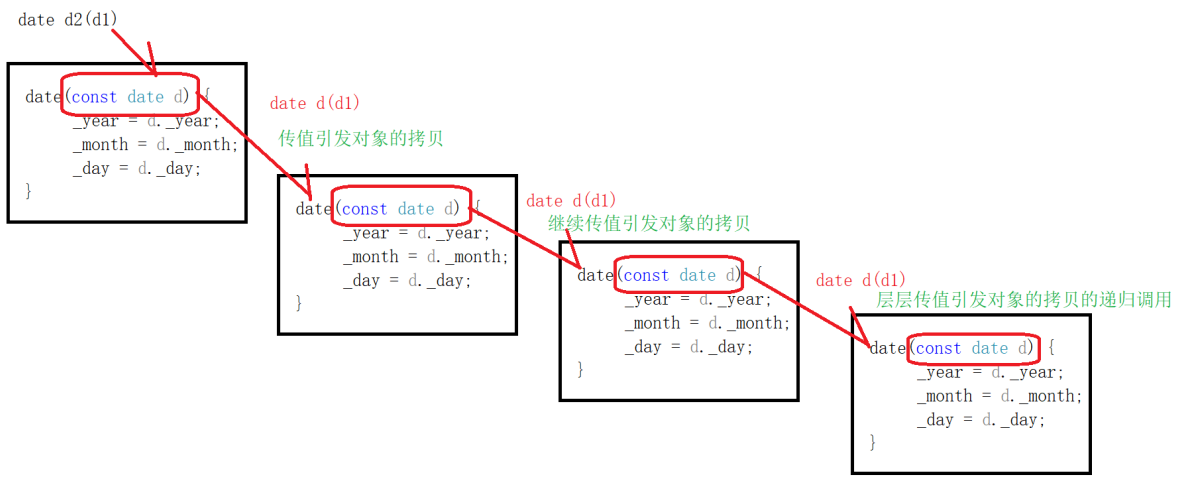
- 类对象的前4个字节是虚函数表指针，构造之后才会生成，只有通过虚函数表才能调用虚函数，而构造函数是虚函数
- 虚函数的意义在于通过父类指针调用子类成员函数，构造函数创建对象时候自动调用，不可能通过父类指针调用

l. 构造函数能抛出异常吗?

- 可以抛出，但尽量不要，会造成内存泄露

m. **拷贝构造函数**

- 拷贝构造函数的参数**必须是引用**，值传递的问题：
  - 无线递归构造，编译时候就会报错



- **显式**调用拷贝构造函数：Date d2(d1);
- **隐式**调用拷贝构造函数：某个函数参数为**值传递**时候，会调用拷贝构造函数构造形参
- 上述讲解

- 浅拷贝：拷贝构造函数**默认**使用**浅拷贝**
- **正常变量深浅拷贝没区别，关键在于指针的深浅拷贝，深拷贝指每一级都是自己的东西**

- 深拷贝：对象中有**动态成员（指针）**时候，析构时候会销毁两次，需要使用深拷贝，其他变量默认浅拷贝也是自己的一份东西

n. 拷贝构造函数、赋值运算符区别？

- 拷贝构造函数：构造新的对象

```
Student S;
Student s1 = s;
Student s2(s);
```

- 赋值运算符：将源对象的内容拷贝到目标对象

```
Student s;
Student s1;
s1 = s;
```

o. delete this指针？

- delete this指针只是做一个标记，而不是立刻释放内存
- delete this之后不能再访问成员变量和虚函数（类对象中只有数据成员和vptr，没有代码内容，类成员函数单独放在代码段中）

p. 虚函数

- 实际的虚函数放在代码段（.text）
- 类的最开始部分有一个虚函数表的指针vptr，虚函数表中放每个虚函数的地址
- 讲解

q. 纯虚函数？

- 作用：实现一个接口，起到规范的作用，想要继承这个类必须覆盖该函数
- virtual void f() = 0

r. 虚函数表

- 虚函数表中存的是虚函数指针，子类覆盖时虚函数地址会被替换
- 虚函数表实际上就是一个**函数指针数组**，有的编译器用的是链表。虚函数表数组中的每一个元素对应一个函数指针指向该类的一个虚函数，同时该类的**每一个对象**都会包含**一个虚函数表指针**，**虚函数表指针指向该虚函数表的地址**。所以当一个类有虚函数的，是占用内存的，占用一个指针大小的内存
- 该类的所有对象共享类的虚函数表（虚函数表伴随类而不是对象），每个对象都必须保存一个指向虚函数表的vptr，vptr地址不同，但指向同一虚函数表

s. 如何让一个类不能被实例化？

- 定义为抽象类（存在纯虚函数）
- 构造函数声明为private

t. **为什么基类的虚构函数需要定义为虚函数？**

- 基类指针指向子类对象，对象销毁时，不定义为虚函数，只能销毁派生类中的部分数据；使得销毁时调用派生类的析构函数

u. 析构函数定义为虚函数有什么缺点？

- 对象将增加虚指针，对象的大小会增加（50%-100%）

v. 析构函数能抛出异常吗?

- 不可以，异常点之后的释放操作没有被执行，造成了内存泄露

w. 组合?

- 人拥有眼睛、鼻子、嘴巴（has a的关系）
- 继承是is a 的关系
- 组合的好处：被包含对象的内部细节对外不可见，封装性好，相互依赖性好
- 缺点：导致系统中的对象过多

x. 多继承存在的问题？如何消除多继承中的二义性?

- 增加程序复杂度，难维护，易出错
- 继承中消除二义性方法：使用`::`；覆盖基类中的同名成员；
- 多继承（从多个基类派生）中消除二义性方法：使用`::`；覆盖基类中的同名成员；使用虚继承，不同继承路径的同名成员只有一份拷贝

y. 重载、隐藏、重写/覆盖

重载	**隐藏**	**重写/覆盖**
一个类中函数的重载	两个类中，可以是也可以不是虚函数，函数名相同，参数返回值相同/不相同	两个类中，必须是虚函数，函数名相同、参数返回值相同

- 形成覆盖的两个函数一定形成隐藏，覆盖/重写是隐藏的特例

	A	B	C	D	E	F
1	**三者**	**作用域**	**有无virtual**	**函数名**	**形参列表**	**返回值类型**
2	重载	相同	可有可无	相同	不同	可同可不同
3	隐藏	不同	可有可无	相同	可同可不同	可同可不同
4	重写	不同	有	相同	相同	相同（协变）

z. 强制类型转换符？（竖着看）

static_cast	**dynamic_cast**	**const_cast:**
通常用于数值类型、向上转换	通常用于多态类型转换；借助RTTI进行类型转换，基类中包含虚函数（RTTI中存在一个继承链，查找能否找到要转换的类型）	删除const、volatile属性（只能改变底层const）
向上转换安全（子类->父类），向下转换不安全（父类->子类）	向上向下转换均可，向上转型始终安全（因为基类有虚函数），向下时有类型检查功能	
	指向派生类的基类指针转换为派生类的指针或引用； Child *pchild = dynamic_cast<Child *>(pfather);	删除的不是变量的特性，删除的是指针/引用的常量性
转换失败：编译期间转换，转换失败抛出编译错误	转换失败：指针转换失败返回NULL；引用转换失败抛出bad_cast异常	

a. RTTI?
   - Runtime Type Information：运行时类型识别
   - 存在继承链
   - 功能由两个运算符实现：
     - typeid：返回对象的类型；允许在运行时确定对象的类型
     - dynamic_cast：运行时检测，实现基类指针/引用向派生类指针/引用的安全转换

b. 类的大小?
   - **成员函数**和**静态成员**无关
   - 类为**数据成员**时候+类的大小（空类的话为1，但是字节对齐之后是4/8）
   - 64位下指针是8字节
   - 单一继承：AB
   - 多继承：A B C D
   - 菱形继承：AB AC D
   - 虚菱形继承：A BC D

c. sizeof一个空类的值为多少/空类大小?
   - 为1
   - 编译器需要区分这个空类的不同实例，分配一个字节，使得空类的不同实例拥有独一无二的地址

d. 空类有哪些成员函数?
   - 构造函数
   - 拷贝构造函数
   - 赋值运算符
   - 析构函数
   - 取址运算符 `class1 * operator & (){}`
   - 取址运算符 const `class1 * operator & () const {}`

e. **friend**
   - 友元函数：在友元函数内部就可以访问该类对象的私有成员了
   - 友元类：类 B 的所有成员函数就都可以访问类 A 对象的私有成员

f. 模板函数、模板类的特例化
   - 模板函数特例化：必须为所有模板参数提供实参

```
template<typename T> //模板函数
int compare(const T &v1,const T &v2)
{
    if(v1 > v2) return -1;
    if(v2 > v1) return 1;
    return 0;
}
//模板特例化,满足针对字符串特定的比较，要提供所有实参，这里只有一个T
template<>
int compare(const char* const &v1,const char* const &v2)
{
    return strcmp(p1,p2);
```

```
13 }
```

- 类模板特例化：不必为所有模板参数提供实参

```
template<typename T>
class Foo
{
    void Bar();
    void Barst(T a)();
};

template<>
class Foo
{
    void Bar();
    void Barst(int a)();
};
```

- 特例化类中的部分成员

```
template<typename T>
class Foo
{
    void Bar();
    void Barst(T a)();
};

template<>
void Foo<int>::Bar()
{
    //进行int类型的特例化处理
    cout << "我是int型特例化" << endl;
}
```

- 总结：
  - 先声明模板，后面放特例化版本
  - 特例化不影响参数匹配，都是最佳原则匹配

g. 内存模型

C++内存管理

- Conception：

a. 内存分区：堆栈全局常量代码区

b. 堆和栈

c. new/delete和malloc/free

d. malloc、calloc、realloc

e. 内存对齐

f. 内存泄露

g. 空类大小、类大小、对象大小

h. 对象复用、零拷贝

i. 内存模型（无多态、有多态单一继承、多重继承、虚拟继承）

- Question：

a. C内存模型

   - 堆、栈、静态全局变量区、常量区
   - 文章

b. C++的内存分区？

   - 栈：编译器分配、清除；存放函数的参数值、局部变量的值
   - 堆：程序员分配，new分配的内存块，程序员不释放的话，程序结束可能由os释放；分配方式类似于链表
   - *自由存储区
   - 全局数据区/静态存储区：全局变量、静态变量
   - 常量存储区：常量字符串
   - 代码区：存放函数体的二进制代码

c. 一个对象在栈上，什么时候被析构？

   - 栈上的对象生命周期结束后，会自动的调用对象的析构函数，释放内存
   - 有成员对象的话，继续调用成员对象的析构函数

d. new/delete和malloc/free？

	new/delete	**malloc/free**
分配内存的位置	自由存储区（C++动态分配和释放的概念，自由存储区可以是堆、全局/静态，具体位置要看具体实现实现，默认是new）	堆（C和操作系统的概念）
	operator new函数分配一块足够大的内存；调用构造函数传入初值；对象构造完成后，返回一个指向该对象的指针	分配内存
	调用析构函数；调用operator delete函数释放内存空间	释放内存
是否调用构造函数与析构函数	调用	不调用
分配成功的返回值	完整类型指针	void*，需要强制转换
分配失败的返回值	bad_alloc异常	返回NULL
分配内存的大小	编译器计算	指定大小
函数重载	允许（全局new有6种重载形式）	不允许（C中的库函数）

是否可以相互调用	operator new的实现可以基于malloc，delete实现可以用free	malloc的实现不可以调用new
已分配内存的扩张	不支持	使用realloc函数重新分配
分配内存时内存不足	可以指定处理函数或重新制定分配器（new_handler、set_new_handler）	返回NULL，无法通过用户代码处理

- 参考1、参考2

e. delete和delete[]有什么区别?
   - 先调用析构函数再释放内存
   - delete对单个对象调用析构函数，delete[]对每个对象逐个调用析构函数
   - 简单类型没有析构函数，delete/delete[]一样；类对象用delete，类对象数组用delete[]（否则第一个之后的对象都不调用析构函数）
   - new/delete、new[]/delete[] 要配套使用

f. malloc申请的内存是否可以使用delete，new申请的内存是否可以使用free?
   - new/delete调用构造/析构函数，malloc/free不调用
   - 不提倡混搭
   -
   - 简单类型可以搭配
   - malloc申请的本身没有构造函数，可以调用delete
   - new申请的类对象中会free会导致没有调用虚构函数，内存泄露

g. **delete如何知道删除多大内存?**
   - malloc本身很复杂，分配时候会用一些内存记录大小，实际分配比真实申请的字节数要大
   - delete p 去获取**指针值减4或减8**处的内容

h. 内存块太小导致new、malloc分配失败怎么办?
   - malloc分配失败返回NULL，分配完需要判断是否是NULL，及时调用return
   -
   - new默认抛出bad_alloc，可以用try...catch...代码块捕获异常
   - 还可以自定义函数处理分配不足问题（**set_new_handler**、new_handler）

```
//set_new_handler、new_handler
namespace std{
    typedef void (*new_handler)(); //函数指针
    std::new_handler set_new_handler( std::new_handler new_p ); //设置自定义
}

//eg:
void out_of_memory() {
    std::cout<<"out of memory!"<<std::endl;
    std::set_new_handler(NULL);
}

int main(){
    set_new_handler(out_of_memory);
```

```
15      int *ptr = new int[100000000]
16  }
```

- new_handler如何设计?
  - 删除其他无用内存
  - 当前new_handler不能解决，设置另一个new_handler函数解决
  - 卸载new_handler（std::set_new_handler(NULL);）
  - 抛出异常
  - abort()、exit()直接退出程序
- 参考

i. 调用new []之后，释放内存使用delete[]，没有指定需要析构的对象的个数，自己设计编译器的话怎么实现operator delete[](void*)?
   - 申请空间首地址用4个字节记录申请了多少空间
   - 操作系统中记录了数组分配的大小，C++记录了析构函数

j. 内存泄露?
   - 场景:
     - 使用未初始化的内存
     - 内存覆盖（越界访问）
     - 没有调用free/delete释放内存
     - 类中存在指针成员，拷贝构造函数/赋值运算符没有合理处理，两个对象一个释放之后另一个再访问出错
     - 基类的析构函数没有设置成虚函数，指向派生类的基类指针析构时候没有释放派生类的成员
   - 定位排查:
     - Linux下可用valgrind、mtrace
   - 如何避免:
     - 结合malloc和memset，避免使用未初始化的内存
     - 使用指针避免越界，避免空指针
     - 利用RAII（资源获取即初始化），避免在堆上分配内存，尽量在栈上分配内存
     - 基类的析构函数设置为虚函数

k. 内存的分配方式?
   - 栈：局部变量、函数参数等；动态扩张收缩
   - 堆：new/delete操作的内存块，动态内存分配，程序员自行申请；动态扩张收缩
   - 自由存储区：malloc/free分配的内存块，和堆很类似
   - 全局/静态存储区：整个程序运行区间都存在的全局变量、局部变量
   - 常量存储区：存放常量，不可修改

l. 堆栈区别?

	A	B	C
1		**栈**	**堆**
2	分配管理方式	编译器管理（动态、静态）	程序员操作分配和释放（动态）
3	产生碎片不同	不存在碎片	频繁分配和释放会产生大量碎片
4	生长方向	高地址向低地址，向下	低地址向高地址，向上
5	申请大小限制	大小固定，超过会栈溢出	不连续的内存区域，大小不定
6	效率	高	机制复杂，低

m. 静态内存分配、动态内存分配?

	A	B	C
1		**静态内存分配**	**动态内存分配**
2	时期	编译期间，按计划分配	运行期间，按需分配
3	位置	栈	堆
4	管理员	编译器管理	程序员控制
5	效率/问题	效率更高	会产生内存泄露

n. 如何构造一个类，使其只能在堆/栈上分配内存?

```
//在栈上分配内存
A a
//使用new在堆上分配内存，两步：operator new()分配内存+构造函数初始化内存
A ptr = new A
```

- 只在栈上：new（分配内存、调用构造函数）会使其在堆上分配，重载operator new()为private
- 只在堆上：编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性，只要非静态函数都会进行检查，析构函数声明为private

o. 浅拷贝、深拷贝?

- 浅：多一个指针指向同一块内存
- 深：创建一个相同的对象

p. 字节对齐、结构体对齐?

- 结构体对齐原则：按结构体中size最大的成员对齐
- 为什么需要进行内存对齐：操作系统不是一字节一字节读取，而是一段一段进行读取（如果int是1-4字节存储，按字节读取的化需要分开读取再组合，效率不高），提高CPU访问速度

- #pragma pack(n)：以n字节方式进行对齐

```cpp
struct alignas(4) Info2 {
  uint8_t a;
  uint16_t b;
  uint8_t c;
};

std::cout << sizeof(Info2) << std::endl;   // 8  4 + 4
std::cout << alignof(Info2) << std::endl;  // 4


//若alignas小于自然对齐的最小单位，则被忽略
//想使用单字节对齐的方式，使用alignas是无效的。应该使用#pragma pack(push,1)或者
```

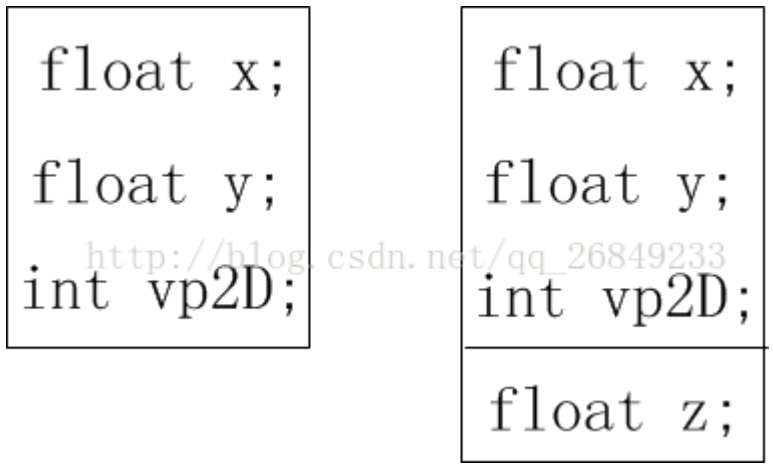
- C++11：alignof可以计算出类型的对齐方式，alignas可以指定结构体的对齐方式
- 64位机器：int4字节，指针*8字节

q. 多重继承、菱形继承

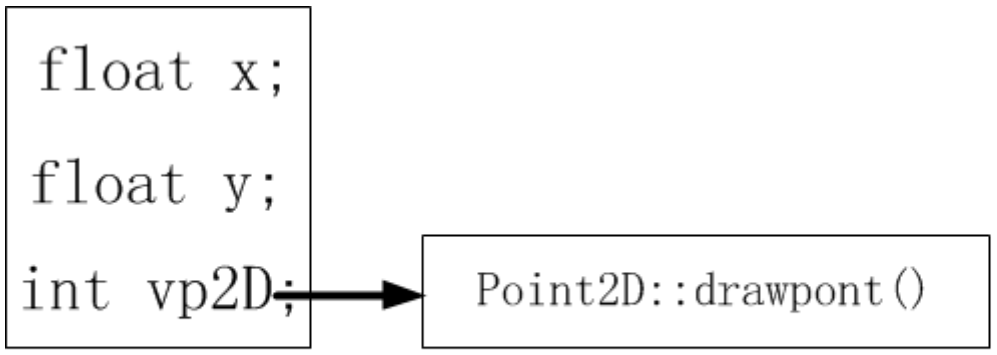
- 多重继承种存在问题，需要作用域限定才能访问祖父基类的内容
- 虚继承解决了问题：父类种不再保存祖父类的内容，而是保存了一份偏移地址，孙子类可以直接调用祖父类的内容

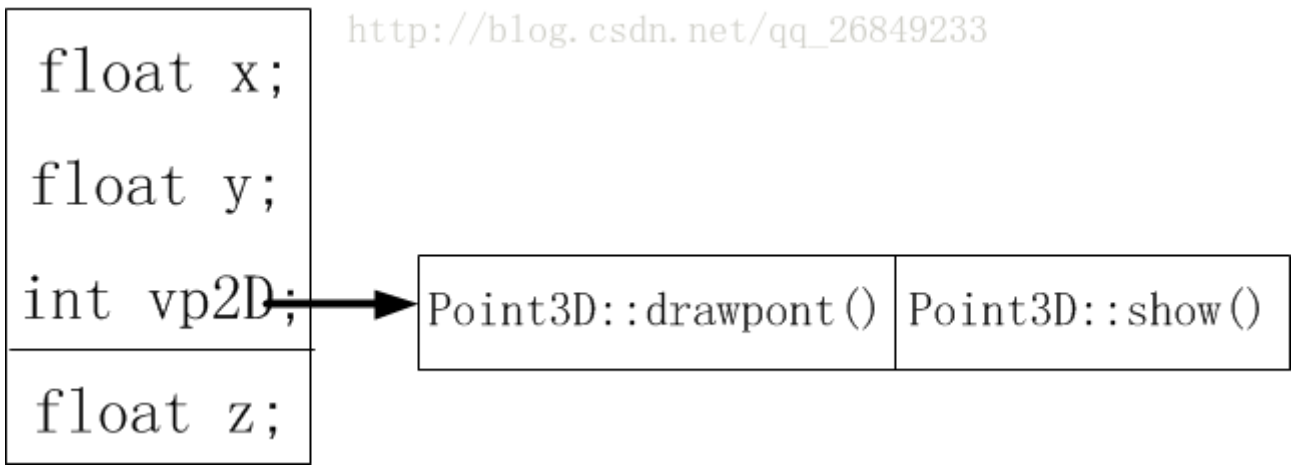
r. 内存模型（无多态、有多态 单一继承、多重继承、虚拟继承）？

- 与成员函数和静态成员无关
- 类为数据成员时候+类的大小（空类的话为1，但是字节对齐之后是4/8）
- 有虚函数指针的大小
- 单一继承无多态

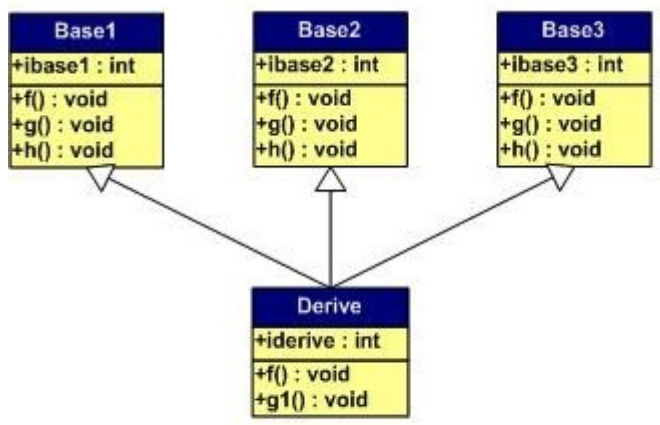


- 单一继承有多态

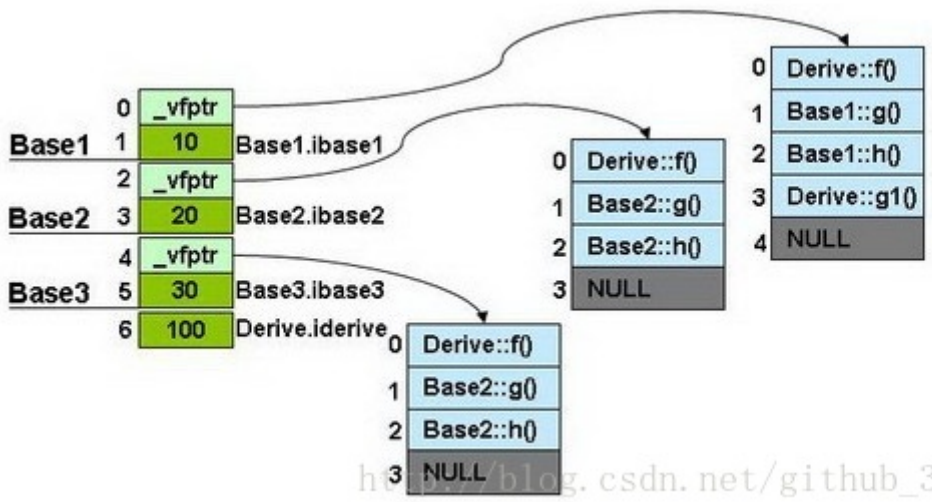




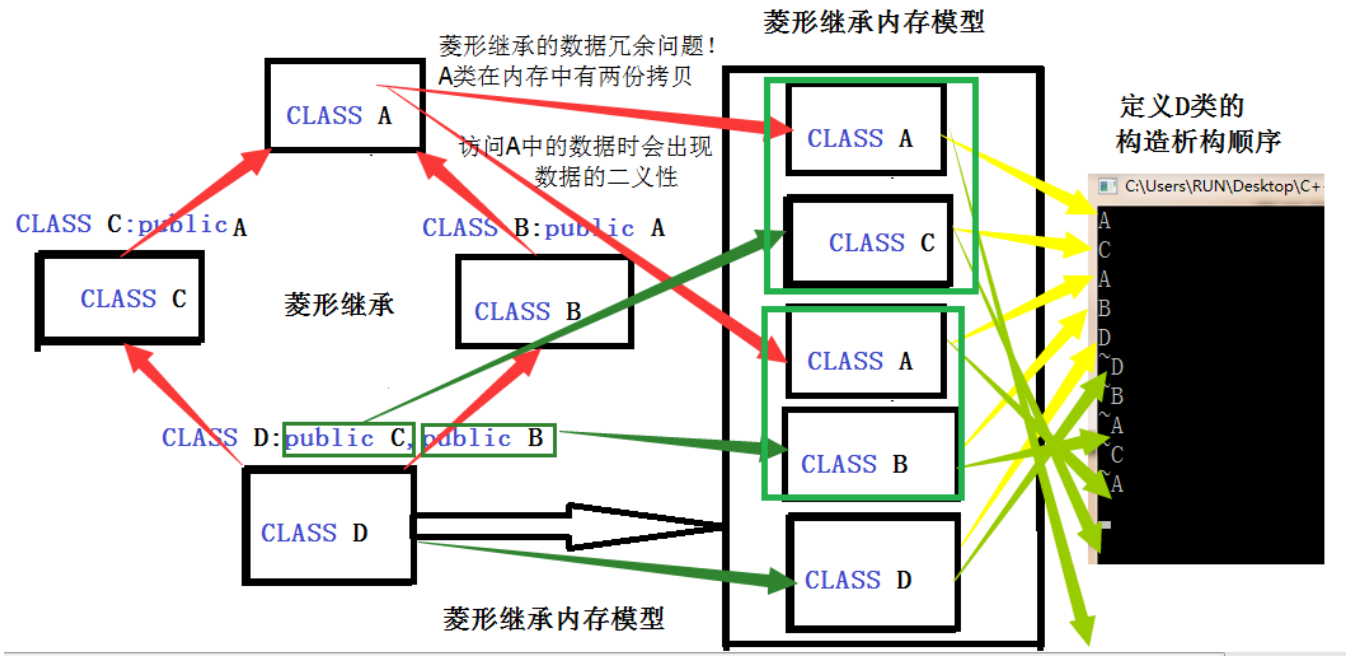
- 多重继承：每个虚函数表中的f都被覆盖



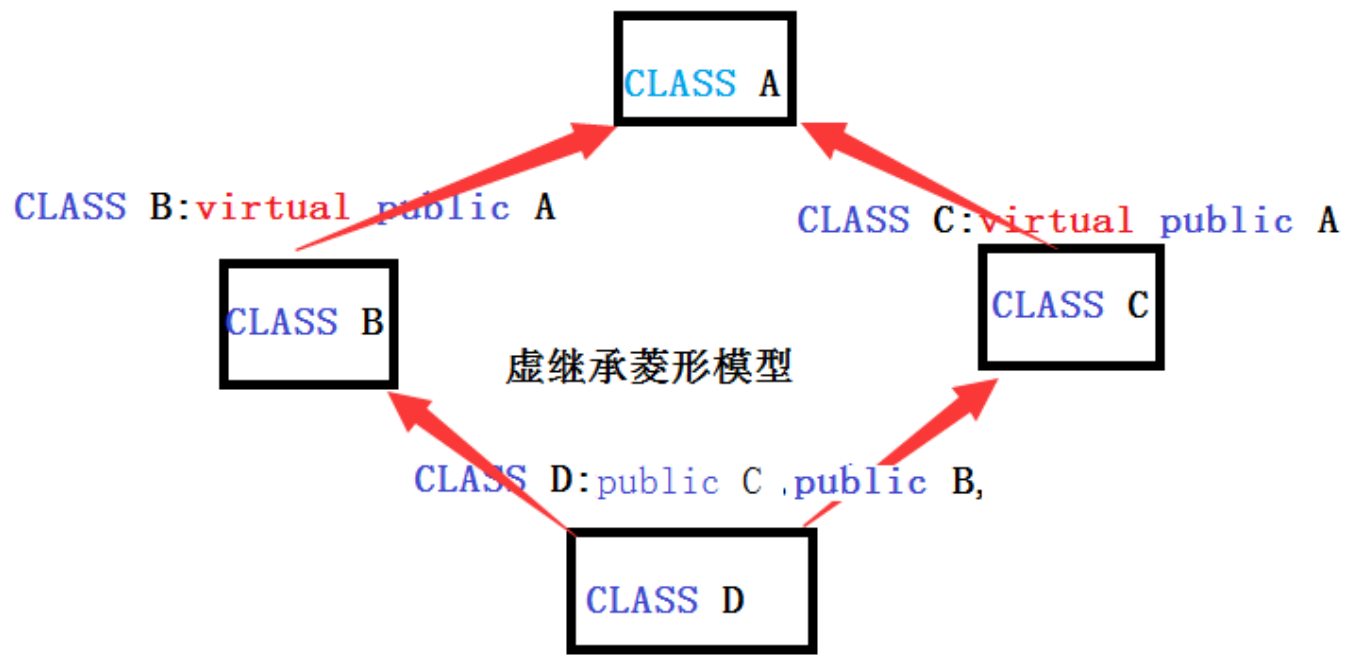
多重继承



- 菱形继承



- A C A B D
- 虚 菱形继承



- A C B D

a.

C++ STL

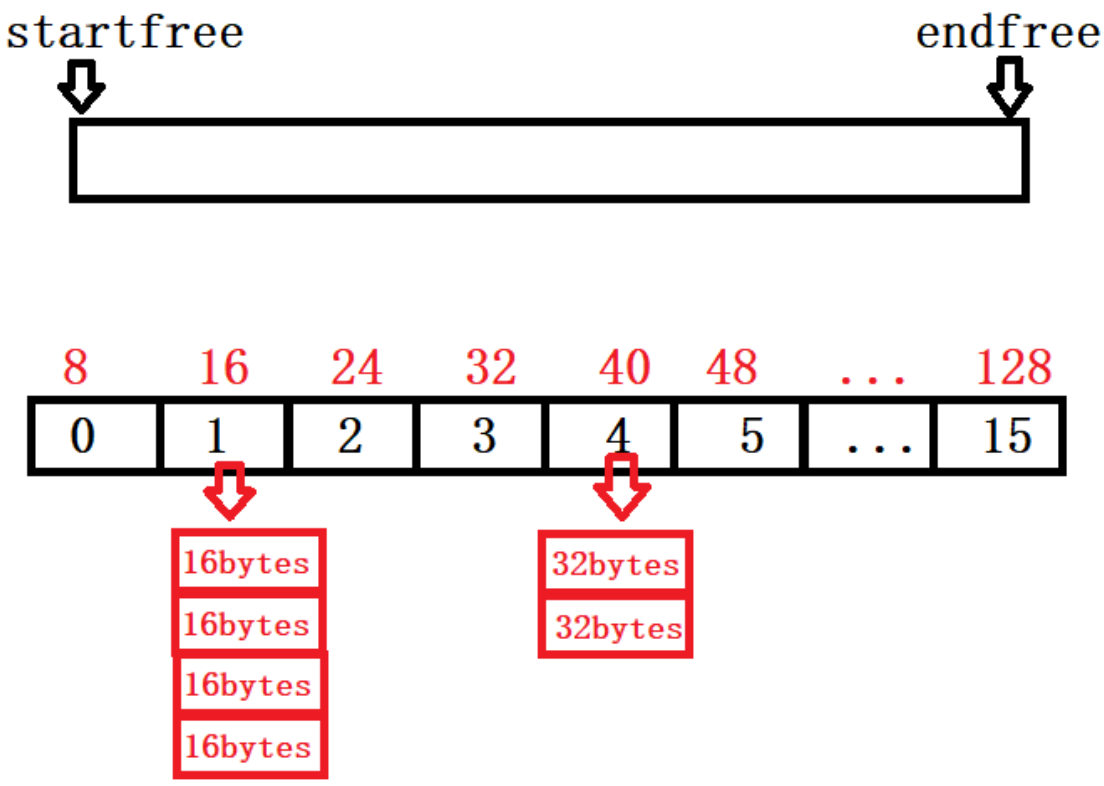
- Conception：
  a. 容器、算法、迭代器、空间配置器、仿函数、适配器
  b. traits机制
  c. vector
  d. list
  e. deque
  f. map、set、multimap、multiset
  g. unordered_map、unordered_set
  h. 迭代器
  i. 算法：lower_bound
- Question：

a. STL是什么?

- 容器、算法、迭代器、空间配置器、仿函数、适配器
- 容器：数据的存放；序列式容器（vector、list、deque）、关联式容器（set、map）
- 算法：排序复制等等
- 迭代器：不暴露容器内部结构的情况下对容器的遍历

b. STL两级空间配置器/内存优化?

- 第一级配置器：大于128byte，使用malloc、free
- 第二级配置器：小于等于128byte



- 维护一个指针数组，管理的内存块都是8的倍数（8-128），每个指针指向一个链表
  - 若链表为空，使用refill函数调用chunk_alloc从内存池中取出一块内存（20*xxbytes），refill函数切分为20个内存块，并组织成链表
- 内存池的分配/chunk_alloc()函数：
  - >= 20：分配20个
  - >=1 && <=20：分配剩余个
  - ==0：
    - 先将剩余的一些内存分配给其他合适的链表；
    - 然后调用malloc分配所需内存的2倍；
      - malloc成功返回给refill；
    - malloc失败先去其他更大的链表搜寻可用内存块，添加到内存池（递归调用chunk_alloc），如果其他链表也被用完，转向第一级空间配置器。

c. **traits萃取机制?**

- 不暴露容器内部结构，迭代器进行访问，如何获取容器类型（模版偏特化）
- **萃取：萃取出不同迭代器的类型调用不同迭代器的xx()函数**

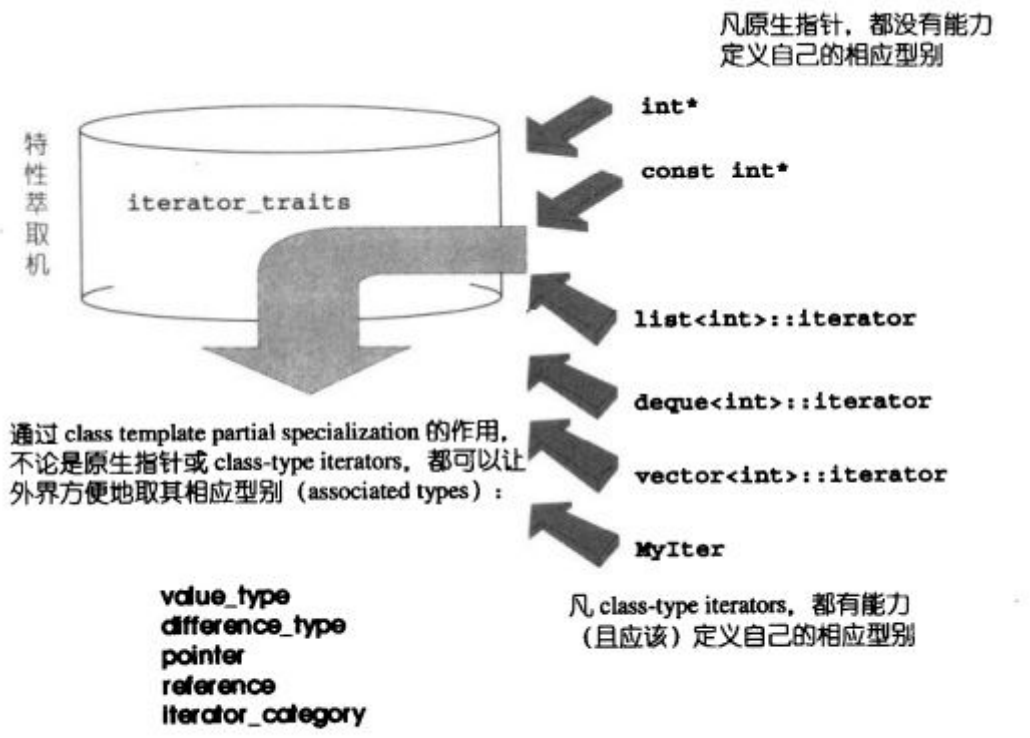


图 3-1 traits 就像一台 “特性萃取机”，榨取各个迭代器的特性（相应型别）

- traits讲解
- iterator和type traits讲解

d. 容器有哪些?

- 序列式容器：
  - array、vector、deque、list、forward_list、string
- 关联式容器：
  - map、set、multiset、multimap、unordered_map、unordered_set、unordered_multiset、unordered_multimap
- 容器适配器：
  - stack、queue、priority_queue

e. vector底层原理?

- 一段连续的线性内存空间，有三个迭代器start、finish、end_of_storage

```
1 //SGI中的三个迭代器
2 _M_start; //起始字节位置
3 _M_finish; //当前最后一个元素的末尾字节
4 _M_end_of_storage; //占用内存空间的末尾字节
```

- 空间不够时，申请1.5/2倍空间，把原来的数据拷贝到新的内存空间，释放原来的内存空间
- 删除时候只删除数据，不释放空间

f. vector中size和capacity区别?

- size：当前有多少元素（finsh - start）
- capacity：当前存储空间能容纳多少元素（end_of_storage - start）

g. vector中reserve和resize区别?
- reserve：仅仅设置capacity这个参数
- resize：容量变大，填充初始值；容量变小，不调整容量，只把前n个元素填充为初始值

h. vector中的元素可以是引用吗?
- 不可以
- 引用不支持一般意义上的赋值操作
- vector中元素的两个要求：元素必须能赋值；元素必须能复制

```
1 int &b = a;
2
3 b = c; //不支持
```

i. vector使迭代器失效的情况?
- erase()：删除位置之后的迭代器、指针、引用失效
- insert()：
  - 插入引起扩容，全部失效
  - 没有引起扩容，插入位置之后的迭代器失效
- 扩容：其他地方开辟新一块内存，原迭代器全部失效
-
- erase()可以返回下一个有效的iterator
- it=q.erase(it);

j. vector1.5/2倍扩容原因?
- VS 下是 1.5倍，在 GCC 下是 2 倍
- 为什么不能太小/为什么不是等长而是倍数增长：
  - 等长时，扩容次数多
  - 倍数时，效率更高，O(n)->O(1)
- 为什么不能太大/为什么不是3、4倍：
  - 理想情况是扩容时候能复用之前释放的空间（1、1.5、2.25、3.375时候能复用前面的空间）
- win下1.5倍，linux下2倍?
  - win下会对释放的内存进行合并，1.5倍为了更好复用
  - linux下伙伴系统（将整个内存区域构建成基本大小basicSize的1倍、2倍、4倍...即要求内存空间分区链均对应2的整次幂倍大小的空间）管理空闲分区，2倍更合适

a. vector如何释放空间?
- vector删除元素内存不释放，析构时候才全部释放，clear()也是清空但不删除
- 空间动态缩小，考虑deque
- 使用swap

- vector(Vec).swap(Vec); //将Vec的内存清除；

  vector().swap(Vec); //清空Vec的内存；

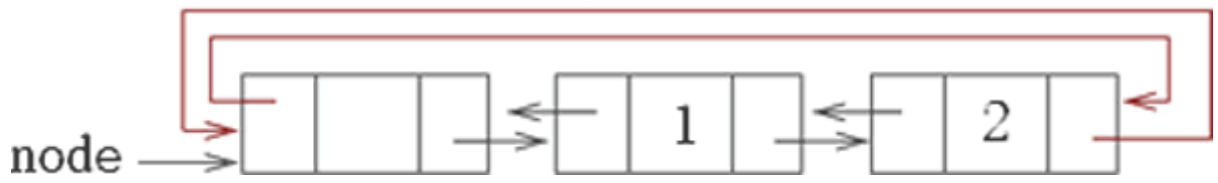
b. 为什么vector的插入操作可能导致迭代器失效?
   - 如果超过容量，2倍扩容是开辟新的内存再拷贝过去，原来的空间释放，迭代器失效

c. 频繁的push_back()对vector性能的影响?
   - 可能影响内存的重新分配，每次都需要开辟新的内存再拷贝过去，释放原来的空间

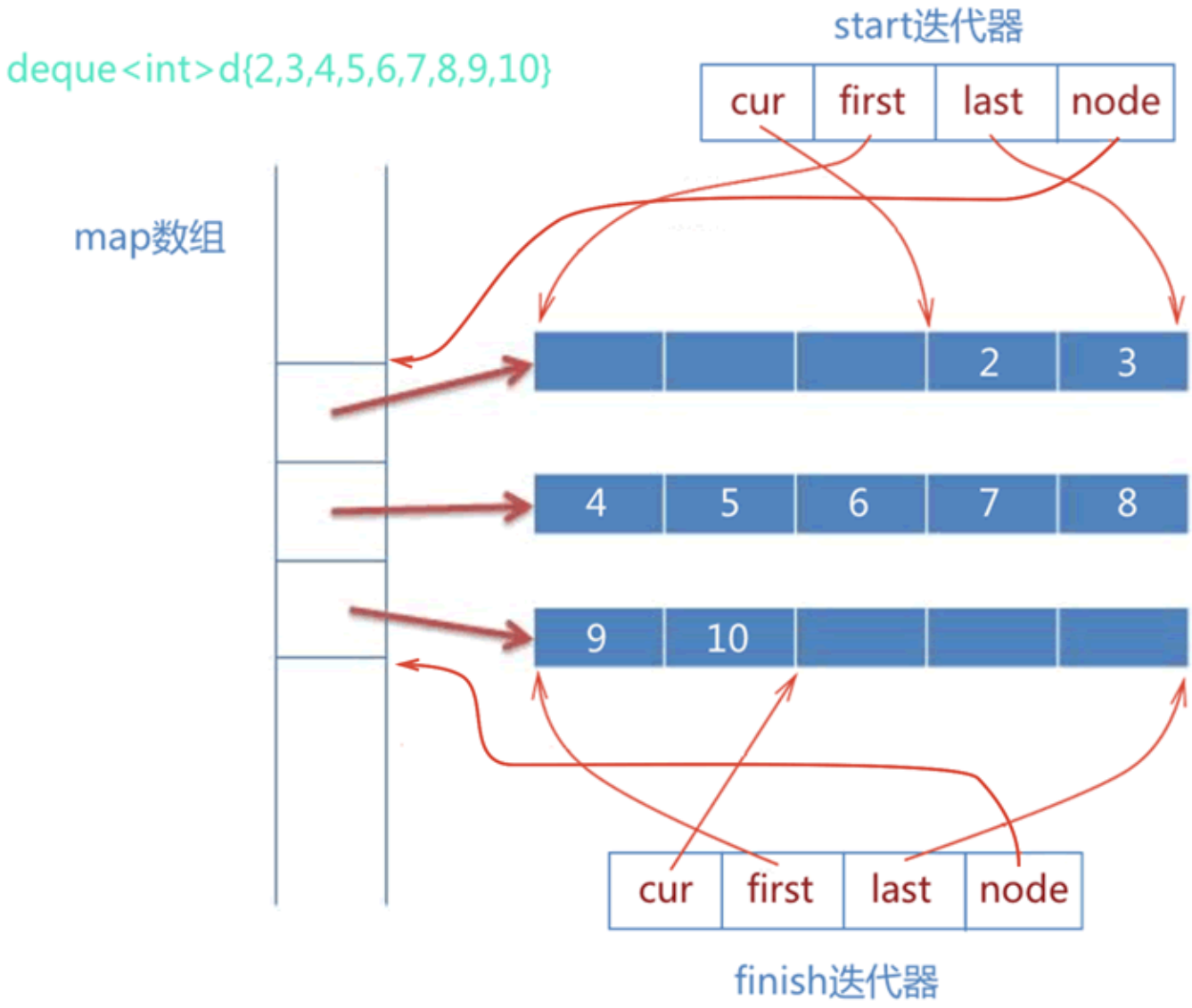
d. vector中push_back的时间复杂度分析?
   - n个元素，倍增因子为m
   - 重新分配的次数大概是logm(n)，第i次分配需要复制m^i个元素
   - n次push_back花费的总时间约为nm/(m - 1)
   - Σ1 ~ logm(n) m^i = nm/(m - 1)
   - m/(m - 1)是一个常数
   - 均摊到每次的时间复杂度为O(1)
   - 参考

e. vector使用注意事项
   - 不要频繁的push_back()，提前看好容量
   - 删除插入要少用，会导致迭代器失效
   - 如果保存的数据是指针类型，需要逐项delete，否则会造成内存泄露

f. list的底层实现原理?
   - 带头节点的双向循环链表



g. deque的底层实现原理?
   - deque由一个map数组组成
   - 每个map指向一段连续的空间
   - 每段连续空间拥有一个迭代器
   - 一个迭代器由4个元素组成，cur、first、last、node
   -
   - ++、--操作需要判断是否到了缓冲区头/尾



- 一些设计特点：
  - deque的start迭代器和finish迭代器并不是一开始就指向map数组的头尾，指向map的中间节点，为了使前后插入留有的空间相同
  - 插入、删除操作为了提高效率，会判断元素偏前还是偏后，偏前移动前面的元素

h. vector、list、deque适用场景？
   - vector：需要随机存取，不关心插入删除
   - list：需要插入删除，不关心随机存取（写多读少的场景）
   - deque：需要随机存取、头尾的随机存取

i. map、set、multiset、multimap底层原理？
   - 红黑树
   - 特点：
     - 节点不是黑就是红
     - 根节点是黑色的
     - 叶子节点都是空节点null、并且是黑色的
     - 红色节点的两个子节点都是黑色节点
     - 从任意节点到叶子节点的路径都包括相同数量的黑色节点（黑子节点的数量称为黑高）
   - 从根节点到叶子节点的最长路径不大于最短路径的 2 倍：最短路径是全黑色节点，最长路径（有红色节点必有黑色节点，红黑色节点数量相同时候，长度=黑色或红色 * 2）
   - 增删改查速度为logn

j. 底层问什么用红黑树不用AVL树？
   - AVL平衡规则太过严格，每次操作几乎都涉及左旋右旋
   - AVL适合**读取查找**型密集任务，红黑树适合**插入**密集型任务

k. 为何map和set的插入删除效率比其他序列容器高，而且每次insert之后，以前保存的iterator不会失效?

- 存储的是节点，不需要内存拷贝和内存移动
- 插入操作只是指针换来换区，节点内存没有改变

l. map的插入方式?

```
//关键字存在插入失败，不会报错，但数据插入不了
m.insert(PII(1, 1));
m.insert(map<int,int>::value_type(1, 1)); //插入value_type数据
m.insert(make_pair(1, 1));

//关键字存在也可以插入，直接覆盖
m[1] = 1;
```

m. map中find和[]的区别?

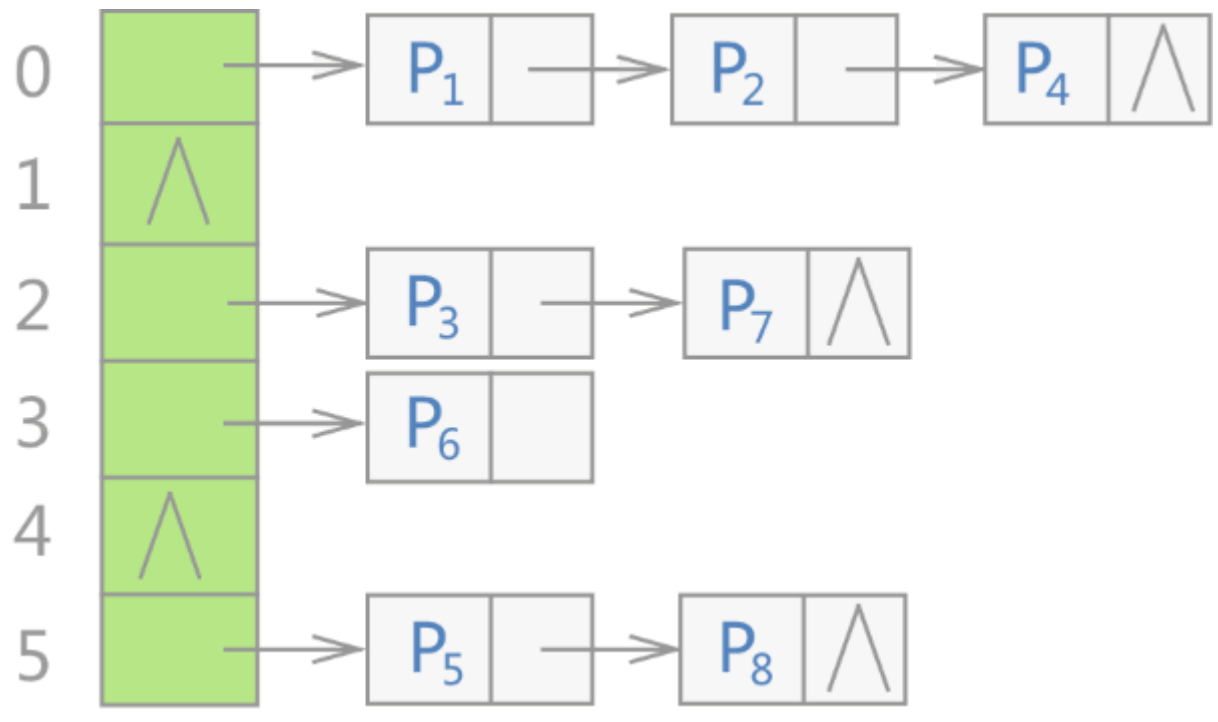
- []：找到key返回这个值；没找到key插入value
- find()：找到返回该位置迭代器；没找到返回尾迭代器

n. map和vector中[]的区别?

- vector中[]会做边界检查
- map中[]找到返回，没找到插入

o. unordered_map、unordered_set、unordered_multimap、unordered_multiset底层原理?

- unordered_和hash_的本质一样，只不过unordered_被纳入标准
- 底层用哈希表，用一个vector数组存储哈希值，并且使用拉链法、链表解决冲突



- vector大小为质数，57、97、193...（28个质数），超过大小新开一个vector，交换两个vector，释放临时vector
- 每个链表称为一个桶
- 总键值对数/桶数=负载因子，负载因子如果超过默认值，自动增加桶数，重新哈希

p. 迭代器底层实现原理?

- 迭代器是连接容器和算法的桥梁，在不了解容器内部原理的情况下遍历容器

- 迭代器相当于一个智能指针
- 迭代器模式
- 链表节点，链表，迭代器操作++、*、->、!=、==

q. 迭代器的型别?

- 迭代器的相应型别：迭代器所指之物的类型（特性）
- value_type：迭代器所指对象类型（T）
- difference_type：两个迭代器之间的距离
- reference_type：迭代器所指对象类型的引用（T&）
- pointer_type：迭代器所指对象类型的指针（*T）
- iterator_categoty：迭代器种类类型

r. 迭代器的种类?

- input_iterator：只读，从一个对象中读出元素（==、!=、->、*）
- output_iterator：只写，向一个对象中修改/添加（*i = v，++）
- forward_iterator：读写、单向移动，一次一步（++）
- bidirectional_iterator：读写、双向移动，一次一步（--）
- random access iterator：所有操作，任意读写，还另外支持[i]

<table>
<tr><th colspan="4">category</th><th>characteristic</th><th>valid expressions</th></tr>
<tr><td colspan="4" rowspan="2">all categories</td><td>Can be copied and copy-constructed</td><td><code>X b(a);
b = a;</code></td></tr>
<tr><td>Can be incremented</td><td><code>++a
a++
*a++</code></td></tr>
<tr><td rowspan="6">Random Access</td><td rowspan="5">Bidirectional</td><td rowspan="4">Forward</td><td rowspan="2">Input</td><td>Accepts equality/inequality comparisons</td><td><code>a == b
a != b</code></td></tr>
<tr><td>Can be dereferenced as an rvalue</td><td><code>*a
a->m</code></td></tr>
<tr><td>Output</td><td>Can be dereferenced to be the left side of an assignment operation</td><td><code>*a = t
*a++ = t</code></td></tr>
<tr><td></td><td>Can be default-constructed</td><td><code>X a;
X()</code></td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td><code>--a
a--
*a--</code></td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Supports arithmetic operators + and -</td><td><code>a + n
n + a
a - n
a - b</code></td></tr>
<tr><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td><code>a < b
a > b
a <= b
a >= b</code></td></tr>
<tr><td>Supports compound assignment operations += and -=</td><td><code>a += n
a -= n</code></td></tr>
<tr><td>Supports offset dereference operator ([])</td><td><code>a[n]</code></td></tr>
</table>

Where X is an iterator type, a and b are objects of this iterator type, t is an object of the type pointed by the iterator type, and n is an integer value.

s. 各种容器删除一个元素?

- 顺序容器：迭代器之后的容器失效（list除外），erase**返回值**是下一个**有效迭代器**
- it = c.erase(it);
- 关联容器：被删除元素的迭代器失效，返回值是void
- c.erase(it++);

t. 迭代器失效?

- 插入：vector、deque插入之后的位置失效，list、forward_list、map、set插入操作不失效

- 删除：vector、deque删除之后的位置失效，list、forward_list、map、set仅删除位置失效；递增当前iterator即可获取下一个位置
- 扩容：内存重新分配全部失效
- unordered_迭代器意义不大，stack、queue、priority_queue没有迭代器
- 掘金总结

u. 容器适配器?

v. 为什么要用deque作为stack和queue的底层默认容器?

- stack需要push_back和pop_back()，list、vector、deque均可
- 为什么不用vector：
  - 元素增长时，vector不断2倍扩容效率低，而且很多空间没有用到；deque利用率高，内存不够时候则寻找内存继续放数据
- 为什么不用list：
  - deque插入操作高效，内存使用率高，list会导致更多的内存碎片

w. 如何在共享内存上使用STL?

- 讲解

x. 一个类要封装使用lower_bound，需要注意什么?

- lower_bound传的是begin()、end()、target
- 注意传的迭代器应该是**随机访问迭代器**，因为二分查找需要找中点

y. sort函数源码?

- 数据量很大使用快速排序
- 递归过程中，分段之后数据量很小，使用插入排序（数据大致有序时候为O(n)，快排取元素存在不确定性，快排在数据本身有序的时候是最慢的O(n^2)）
- 递归过程中，递归层次过深，使用堆排序处理（递归层数多浪费时间，堆排序最好最坏都是nlogn，归并也是但需要递归）
- 讲解

C++11

- Conception：
  a. 智能指针
  b. 右值引用、移动构造、完美转发
  c. nullptr
  d. Lambda函数
  e. constexpr
  f. auto、decltype
  g. for()
  h. tuple
  i. bind、functional
  j. noexcept
- Question：
  a. **C++11新特性?**

- 智能指针shared_ptr、weak_ptr
- 右值引用
- 语法：auto、for、nullptr、decltype、constexpr、lambda函数
- STL容器：array、forward_list、unordered_map/set、
- 多线程：thread、atomic、lock_guard、mutex、condition_variable（条件变量）、sem_t（信号量）、future、
- 函数：function、bind

b. **auto、decltype**

c. **using**

```
typedef vector<vector<int>> vvi;
using vvi = vector<vector<int>>;

typedef void (*func)(int, int);
using func = void(*)(int, int);
```

d. **constexpr**

- 对于无 constexpr 关键字的表达式是在运行期执行，对于有 constexpr 关键字的表达式是在编译期执行

e. **bind、function、placeholders、ref、cref**

- void(*)(int ,int)：函数指针
- void(int, int)：函数原型
- placeholders可以指定占位符_1,_2,_3
- ref按引用传递的值，cref按const引用传递的值

```
    function<void(int)> f;
function<void(int ,int)> func1 = bind(func1, 10, 20);
```

```
void f(int n1, int n2, int n3, const int& n4, int n5){

}
using namespce std::palceholders; //针对_1,_2,_3

int n = 7;

auto f1 = std::bind(f, _2, 42, _1, std::cref(n), n);

n = 10

f1(1, 2, 1001);
```

f. **lambda表达式**

```
auto f = [&、=](参数) -> 返回值{内容};
```

g. 线程相关

- thread

```
int main(){
    auto f1 = []{
        for(int i = 0; i < 5; i++){
            cout << i << endl;
        }
    };

    std::thread t(f1);
    //join: 主线程必须等到线程函数执行完之后才会结束
    t.join();
    //detach: 将当前线程对象所代表的执行实例与该线程对象分离，使得线程的执行可以单独进行。一旦线程执行完毕，它所分配的资源将会被释放。但是失去了对于对象的控制权
    t.detach();
    return 0;
}
```

- mutex

```
mutex:
std::mutex mutex_;

lock_guard:
std::uqique_lock<std::mutex> lock(mutex_);
注：离开作用域自动释放锁，不能手动释放

unique_lock:
std::unique_lock<std::mutex> lock(mutex_);
注：相比lock_guard，可以手动释放锁，条件变量必须用unique_lock，因为条件变量在w
```

  - lock_guard和unique_lock都是RAII机制下的锁，即依靠对象的创建和销毁也就是其生命周期来自动实现一些逻辑，而这两个对象就是在创建时自动加锁，在销毁时自动解锁。
  - 所以如果仅仅是依靠对象生命周期实现加解锁的话，两者是相同的，都可以用，因跟生命周期有关，所以有时会用**花括号**指定其生命周期。
  - 但lock_guard的功能仅限于此。unique_lock是对lock_guard的扩展，允许在生命周期内再调用lock和unlock来加解锁以切换锁的状态。
  - **根据linux下条件变量的机制，condition_variable在wait成员函数内部会先调用参数unique_lock的unlock临时解锁，让出锁的拥有权（以让其它线程获得该锁使用权加锁，改变条件，解锁），然后自己等待notify信号，等到之后，再调用参数unique_lock的lock加锁，处理相关逻辑，最后unique_lock对象销毁时自动解锁。**

- atomic

```
std::atomic<int> count;

count--;
count.store(++count);
```

```
5
6 count++;
7 count.store(++count);
8
9 //不用原子变量的话要加锁
```

- call_once：保证某一函数在多线程环境中只调用一次

```
1
```

- condition_variable

```
1 wait()：阻塞等待的同时释放锁
2 notify_all()：唤醒所有阻塞的线程
3 notify_one()：唤醒某一个等待的线程
```

- 异步操作 future、promise、packged_task、async

```
1 future：
2 promise：包装一个值
3 packged_task：包装一个函数
4 async：
5
6 shared_future：可复制而且多个shared_future对象能指代同一共享状态，多个线程访问同一共享状态是安全
7
8 #include <iostream>
9 #include <future>
10 #include <thread>
11
12 int main()
13 {
14     // 来自 packaged_task 的 future
15     std::packaged_task<int()> task([](){ return 7; }); // 包装函数
16     std::future<int> f1 = task.get_future();  // 获取 future
17     std::thread(std::move(task)).detach(); // 在线程上运行
18
19     // 来自 async() 的 future
20     std::future<int> f2 = std::async(std::launch::async, [](){ return 8;
   });
21
22     // 来自 promise 的 future
23     std::promise<int> p;
24     std::future<int> f3 = p.get_future();
25     std::thread( [&p]{ p.set_value_at_thread_exit(9); }).detach();
26
27     std::cout << "Waiting..." << std::flush;
28     f1.wait();
29     f2.wait();
30     f3.wait();
31     std::cout << "Done!\nResults are: "
32               << f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';
33 }
```

```
34
35 输出：
36 Waiting...Done!
37 Results are: 7 8 9
```

h. **智能指针？**

- 智能指针是一个RAII模型（Resource Acquisition is Initialization，资源获取即初始化，构造函数分配内存，析构函数释放内存），用于动态分配内存；将基本类型指针封装为类对象指针，在离开作用域的时候调用析构函数，删除指针所指向的内存空间
- 作用：出路内存泄露和空悬指针问题
- auto_ptr（C++98）：独占式拥有，同一时间只有一个智能指针可以指向该对象；C++11中抛弃；问题在于函数传参时，对象所有权不会返回，存在内存崩溃问题；不能指向数据，不能作为STL容器的成员

```
1 auto_ptr< string> p1 (new string ("I reigned lonely as a cloud."));
2 auto_ptr<string> p2;
3 p2 = p1; //不会报错.
4
5 //p2剥夺了p1所有权，再访问p1会报错
```

- unique_ptr：独占式拥有，同一时间只有一个智能指针可以指向该对象；**不能进行拷贝构造和拷贝赋值，可以进行移动构造和移动赋值，原来的被置为nullptr**

```
1 auto_ptr< string> p1 (new string ("I reigned lonely as a cloud."));
2 auto_ptr<string> p2;
3 p2 = p1; //会报错.
4 p2 = unique_ptr<string>(new string ("You"));   // 临时对象不会报错
5
6 unique_ptr<string> ps1, ps2;
7 ps1 = demo("hello");
8 ps2 = move(ps1); //实现了所有权的转移，ps1为nullptr
9 ps1 = demo("alexia");
10 cout << *ps2 << *ps1 << endl;
```

- shared_ptr：共享式拥有，多个智能指针指向相同的对象；引用计数为0时释放对象；
  - 实现：
    - 构造函数初始为1
    - 析构函数--，为0释放资源
    - 拷贝构造函数++
    - 赋值运算符用move
    - 移动构造（&&）函数++
    - 移动赋值运算符（&&）用move
  - **引用计数：**
    - 如果用一个static变量，指向不同对象的所有指针都用的是同一个变量

```
1 private:
```

```
2     size_t *cnt;
```

- **shared_ptr是线程安全的吗?**
  - shared_ptr不是线程安全的，读安全，写不安全
  - 最简单的方法就是给shared_ptr用锁保护，因为如果想要修改shared_ptr内部的实现来支持多线程，写操作时会涉及到多个地址的更改，用简单的单地址的CAS也是做不到的。所以，不想使用锁的话，最好对shared_ptr只读不写
  - **计数器安全，原子操作，对指向对象的并发读写不安全**
- 以下算是避免**内存泄漏**的一些问题
  - 注意1：不要使用裸指针初始化多个shared_ptr
  - 注意2：用shared_from_this()返回this，因为this本身也是裸指针，存在double free问题
  - 注意3：避免循环引用，导致计数为2，离开作用域后计数为-1，永远不会析构
  - 注意4：**用make_shared不要用new**
    - **shared_ptr 对象在内部指向两个内存位置：**
      - 1、指向对象的指针
      - 2、用于控制引用计数数据的指针。
    - make_shared一次性为int对象和用于引用计数的数据都分配了内存
    - new操作符只是为int分配了内存。
- weak_ptr：对对象的弱引用，不会增加对象的引用计数，搭配shared_ptr使用；只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造；
  - 作用1：shared_from_this()返回的就是weak_ptr
  - 作用2：解决shared_ptr循环引用问题（两个shared_ptr相互引用，两个指针的引用计数永远不可能下降为0，资源永远不会释放）

i. **右值引用、移动构造、完美转发**

```
右值引用（实现移动语义和完美转发，消除不必要的拷贝）：

左值：等号左边，可以取地址
右值：等号右边，不可以取地址
左值持久，右值短暂，右值只能绑定到临时对象，右值引用可以接管引用对象的内容

左值引用&
右值引用&&
左值引用只能绑定左值，右值引用只能绑定右值

//将左值转化为右值，给需要右值的地方把左值转化为右值传递右值
std::move()


引用折叠规则
1.所有右值引用折叠到右值引用上仍然是一个右值引用。（A&& && 变成 A&&）
2.所有的其他引用类型之间的折叠都将变成左值引用。 （A& & 变成 A&; A& && 变成 A&; A
```

```
移动语义（传递的是临时对象的话，原来的东西拷贝完就没用了，新的还得再申请释放）：
```

```
2
3  //拷贝构造函数
4  A(A& a)
5  //移动构造函数
6  A(A&& a)
7  //拷贝赋值函数
8  A& operator=(A& a)
9  //移动赋值函数
10 A& operator=(A&& a)
```

- copy和move的区别

```
1  完美转发（一个函数给另一个函数传参时候，原参数是左值/右值，新函数还能保持左值/右值，
2
3  //不完美的转发
4  void process(int& i){
5      cout << "process(int&):" << i << endl;
6  }
7  void process(int&& i){
8      cout << "process(int&&):" << i << endl;
9  }
10
11 void myforward(int&& i){
12     cout << "myforward(int&&):" << i << endl;
13     process(i);
14 }
15 int main(){
16         myforward(2);  //本来2是右值；到了myforward里有了名字i，再传递process时
17 }
18
19
20 //用来实现完美转发，forward可以实现左右值的相互转换，move只能左值到右值
21 std::forward<T>(u)
22 原则：a.T为左值引用时，u被转换为左值；b.否则u被转换为右值
23
24 std::forward<int>(x) //x转换成右值，b
25 std::forward<int &>(x) //x转换成左值，a
26 std::forward<int &&>(x) //x转换成右值，b
```

- 参考：1知乎、2简书、3CSDN

j. final、override、default、delete、explicit

- final：禁止类进一步派生和虚函数进一步重载

```
1  struct base final{}
```

- **override：表明该函数重写了基类函数；如果基类没有声明这个虚函数，编译报错**
  - 在重写方法时，最好加上这个 override 这个关键字 以 加强代码规范。
- default：声明构造函数为默认构造函数，如果类中有自定义构造函数，编译器就不会隐式生成默认构造函数（不声明时候，不传参数的构造就会报错）
- delete：禁止对象的拷贝和赋值，声明delete

- explicit：**修饰构造函数，只能显式构造，不能被隐式转换**

k. constexpr

- 修饰的是真正的常量，编译期间就会被计算处理
- 作用：修饰函数返回值，尽可能让其被当做一个常量，编译期间没有被计算出来，会被当成一个普通函数处理

```
constexpr int func(int i){
    return i + 1;
}
int main(){
    int i = 2;
    func(i); //普通函数
    func(2); //编译期间就会被计算出来
}
```

l. noexcept

- **指定某个函数不抛出异常**，预先知道函数不会抛出异常有助于某些**优化**操作
- 编译期间不出现问题，如果noexcept的函数执行时出了异常，程序会马上terminate。
- 知乎回答

m. 内存对齐：alignof可以计算出类型的对齐方式，alignas可以指定结构体的对齐方式

n. 新增数据类型、随机数功能、数据结构、算法...

C++14

- Question

a. **函数返回值类型推导**

```
auto func(int i){
    return i;
}
//C++11不支持
```

b. **[[deprecated]]标记**

```
struct [[deprecated]] A {};

//编译时产生警告，将来可能会被丢弃，尽量不要使用
```

c. **std::make_unique**

```
C++11中有std::make_shared，却没有std::make_unique，在C++14已经改善

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

d. **读写锁std::shared_timed_mutex和std::shared_lock**

```
std::shared_lock<std::shared_timed_mutex> lock(mutex_);
std::unique_lock<std::shared_timed_mutex> lock(mutex_);

timed是带超时时间的
```

e. **std::exchange（移动构造，和swap不一样）**

```
template<classT,classU=T>
constexprTexchange(T&obj,U&&new_value){
    T old_value = std::move(obj);
    obj = std::forward<U>(new_value);
    return old_value;
}
```

C++17

- Conception：
- Question：

a. **C++17 之 "constexpr if"**

   - 在编译期间执行，可以将之应用于泛型编程的条件判断

```
template <typename T>
T simpleTypeInfo(T t) {
        if constexpr (std::is_integral_v<T>) {
                std::cout << "foo<integral T> " << t << '\n';
        }
        else {
                std::cout << "not integral \n";
        }
        return t;
}
```

b. **构造函数模板推导**

```
//before C++17
pair<int, double> p(1, 2.2);

//C++17
pair p(1, 2.2);
```

c. **内联变量**

```
之前只有内联函数，C++静态成员变量在头文件中是不能初始化的，有内联变量可以解决

structA{
    inline static const int value=10;
}
```

d. namespace嵌套

```
namespaceA::B::C{
    void func();)
}
```

e. 新增Attribute

```
[[carries_dependency]]：让编译期跳过不必要的内存栅栏指令
[[noreturn]]：函数不会返回
[[deprecated]]：函数将弃用的警告
[[fallthrough]]：switch中可以直接落下去，不需要break，让编译器忽略警告
[[nodiscard]]：修饰的内容不能被忽略，用于修饰函数，返回值一定要被处理，否则有警告
[[maybe_unused]]：编译器修饰的内容可能暂时没有用到，避免产生警告
```

f. std::any：存储任何类型的单个值

```
int main(){//c++17可编译
    std::any a=1;
    cout << a.type().name() << " " << std::any_cast<int>(a) <<endl;
    a = 2.2f;
    cout << a.type().name() << " " << std::any_cast<float>(a) <<endl;
    if(a.has_value()){
        cout << a.type().name();
    }
    a.reset();
    if(a.has_value()){
        cout << a.type().name();
    }
    a = std::string("a");
    cout << a.type().name() << " " << std::any_cast<std::string>(a) <<endl
    return 0 ;
}
```

情景题

- Conception：

  a.

- Question：

  a. 编译器会优化C++程序哪些地方?

     - inline
     - 推荐使用现代的range-for
     - 文章

  b. 设置地址为为0x67a9 的整型变量的值为0xaa66?

  c. int *ptr;
  ptr = (int *)0x67a9;
  *ptr = 0xaa66;

d. 手写实现智能指针

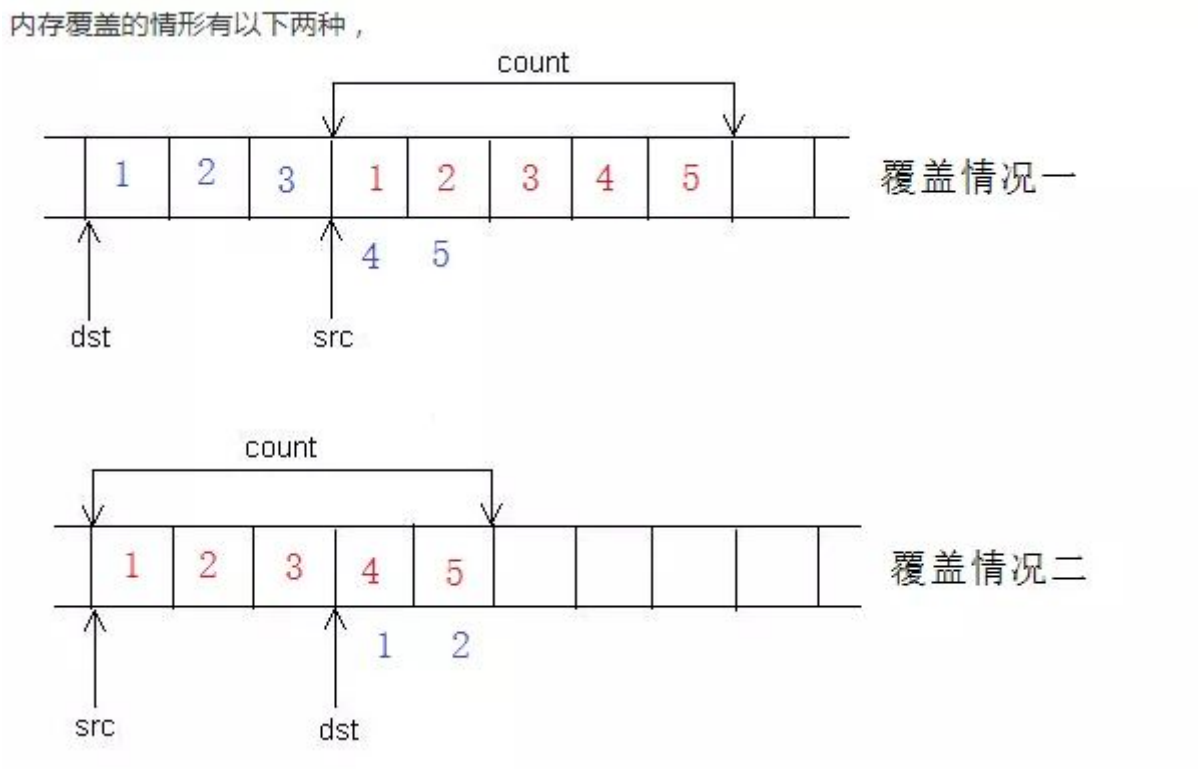
```
//拷贝构造函数中计数初始化为1
//拷贝构造函数计数值+1
//赋值运算符左边-1，右边对象引用计数+1
//析构函数引用计数-1，如果为0:delete释放对象
template<typename T>
class sharedPtr{
  private:
    size_t *cnt;
    T *ptr;
  public:
    sharedPtr()
    ~sharedPtr()
    sharedPtr(const sharedPtr& ptr){

    }
    sharedPtr& operator=(const sharedPtr& ptr){

    }
    sharedPtr(sharedPtr&& ptr){

    }
    void operator=(sharedPtr&& ptr){

    }
}
```

e. strcpy()

```
//src是const
char *strcpy(char *dest, const char *src){
    //ret记录的是开始位置，复制结束后dest已经到了末尾
        char *ret = dest;
    assert( (dest != nullptr) && (src != nullptr));
    while(*src != '\0'){
        *(dest++) = *(src++);
    }
    *dest = '\0';
    return dest;
}
//内存重叠问题
char *strcpy(char *dest, char * src){
    char* ret = dest;
    assert( (dest != nullptr) && (src != nullptr));
    memmove(dest, src, strlen(src) + 1);
    return ret;
}
```

f. memcpy()、memmove()

- 内置版本不考虑内存重叠，考虑内存重叠的版本也就是memmove



- 既然memmove解决了内存重叠问题，为什么还要memcpy呢？memmove有内存拷贝的开销，效率不如memcpy。

```
void *memcpy(void *dest, const void *src, size_t n){
        assert( (dest != nullptr) && (src != nullptr));
    char* s_dest = (char *)dest;
    char* s_src = (char *)src;
    //情况1包括在了不重叠的情况种
    //覆盖情况2
    if(s_src < s_dest && s_dest + size > s_src){
        //指向了字符串末尾
        s_dest = s_dest + n - 1;
        s_src = s_src + n - 1;
        //从后往前
        while(n--) *s_dest-- = *s_src--;
    }else{
        //从前往后
        while(n--) *s_dest++ = *s_src++;
    }
    return dst;
}
```

g. strcat()

- 把src加到dest后面

```
char* strcat(char *dest,const char *src){
    char *ret = dest;
    assert( (dest != nullptr) && (src != nullptr));
    while(*dest!='\0'){
        dest++;
    }
    while(*src!='\0'){
        *(dest++)=*(src++);
    }
    *dest='\0';
    return ret;
}
```

h. strcmp()

```
int strcmp(const char *s1,const char *s2){
    assert( (dest != nullptr) && (src != nullptr));
    while(*s1!='\0' && *s2!='\0'){
        if(*s1>*s2){
            return 1;
        }
        else if(*s1<*s2){
            return -1;
        }
        else{
            s1++,s2++;
        }
    }
    //当有一个字符串已经走到结尾
    if(*s1>*s2){
        return 1;
    }
    else if(*s1<*s2){
        return -1;
    }
    else{
        return 0;
    }
}
```

i. memset()

```
void* memset(void *dst, int val, size_t size) {
    char *s_dst = dst;
    char ch = val;
    if (s_dst == NULL) {
        return NULL;
    }

    while (size--) {
        *s_dst++ = ch;
    }

    return dst;
}
```

j.

遗留问题

- 前置问题：模板完全特例化和非完全（部分）特例化、
- 类型推导，<typename T>推导出来的T是大类型，没法把返回值和参数分开