C++ (/tags/#C++) C++基础再探 (/tags/#C++%E5%9F%BA%E7%A1%80%E5%86%8D%E6%8E%A2)

面经汇总 (/tags/#%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB)

面经汇总 C++基础再探 (/tags/#%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB%20C++%E5%9F%BA%E7%A1%80%E5%86%8D%E6%8E%A2)

面经汇总 C++基础再探

面经汇总 C++基础再探

Posted by zhaostu4 on November 28, 2019

> *T:2019/11/28 W:四 17:0:11*

C++ 基础

1、引用和指针的区别?

- 初始化:
  - 引用在定义的时候必须进行初始化，并且不能够改变

    - 指针在定义的时候不一定要初始化，并且指向的空间可变
- 访问逻辑不同:
    - 通过指针访问对象, 用户需要使用间接访问
    - 通过引用访问对象, 用户只需使用直接访问, 编译器负责将其处理为间接访问
- 运算结果不同:
    - 自增运算结果不同
    - `sizeof` 运算的结果不同
    - 下标运算:
        - 指针通过下标运算结果是指针所指值为基地址加上偏移, 且基地址可变.
        - 引用通过下标运算结果是引用的是数组才能有这个操作.
    - 函数参数:
        - 传指针的实质是传值，传递的值是指针内储存的变量地址；
        - 传引用的实质是传地址，传递的是变量的地址。
    - 多级: 有多级指针，但是没有多级引用，只能有一级引用。

- 参考: C++ 引用占用内存？ - toyijiu的专栏 - CSDN博客 (https://blog.csdn.net/toyijiu/article/details/99729949)

2、从汇编层去解释一下引用

- 参考两个语句

```
int a=1;
int &b=a;
//
mov ptr [ebp-4], 1
lea eax, [ebp-4]
mov dword ptr [ebp-8], eax
```

- a 的地址为 ebp-4， b 的地址为 ebp-8，栈地址由高到底分配.
- 可以发现这个和指针的复制几乎一样,所以引用其实是通过指针来实现的

3、C++中的指针参数传递和引用参数传递

- **指针参数传递的本质是值传递, 传递的值是对象的地址**, 在调用时形参会在函数栈中开辟空间用于存放传递过来的对象的地址,此时形参相当于是实参的副本, 对形参的任何操作都不会反映到实参上, 但是通过形参间接访问对象的修改是会反应到函数之外的.
- **引用参数传递的本质是传地址, 传递的是实参变量的地址**, 首先形参会在函数栈中开辟空间用来存放实参变量的地址, 然后对该形参的任何操作都会被处理未间接寻址,即通过形参中的地址访问主调函数中的实参变量, 因为通过形参的任何操作都将被应用于主调函数中.
- 从逻辑上引用相当于对变量起了一个别名, 通过该别名可以对变量进行直接访问, 由编译器负责将直接访问转换为间接访问; 而指针访问变量都是间接访问.

4、形参与实参的区别?

- 形参属于函数内部的局部变量, 在调用函数时才会分配内存, 在函数调用之后会被释放掉, 因此在函数内部才有效
- 实参可以使常量, 表达式, 函数等, 无论是何种类型,在函数调用时都必须有一个确定的值,以便把函数的值传递给形参
- 实参和形参的个数一定要严格匹配(当然可以忽略有默认值形参), 通常情况下函数类型也是应该严格匹配的, 但是允许隐式类型变换,如果类中定义了零参数构造函数,甚至可以使用空初始化列表 {} 的方式调用零参数构造函数
- 实参到形参的传递是单向的
- 形参类型为非指针非引用, 则传递方式为值传递则, 形参为实参的副本, 对形参的任何修改都不会反应在主调函数中

4-2 三种传递方式

- 值传递是通过拷贝构造函数实现的
- 指针传递是属于值传递,实参指针向形参传递的是对象的地址
- 引用传是属于传地址, 相当于对变量起了一个别名, 本质上和指针传递类似传递的都是对象的地址,区别在于对该引用形参的任何操作都会被处理为间接云芝, 也就是会反应到调用函数中

5、 static 的用法

- 主要可以分为五个类型: 全局静态变量, 局部静态变量, 静态函数, 静态成员变量, 静态成员函数
    1. 全局静态变量
    - 在全局变量前加上关键字 `static` ， 全局变量就定义成一个全局静态变量.
    - 内存中的位置：静态存储区，在整个程序运行期间一直存在。
    - 初始化：未经初始化的全局静态变量会被自动初始化为 `0` (对于自动对象,如果没有显示初始化,会调用零参数构造函数,如不存在则编译失败);
    - 作用域：**全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾。**
        1. 局部静态变量
    - 在局部变量之前加上关键字 `static` ， 局部变量就成为一个局部静态变量。
    - 内存中的位置：静态存储区
    - 初始化：未经初始化的全局静态变量会被自动初始化为 `0` (对于自动对象,如果没有显示初始化,会调用零参数构造函数,如不存在则编译失败);
    - 作用域：作用域仍为局部作用域，
        - 当定义它的函数或者语句块结束的时候，作用域结束。
        - 但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变；

1. 静态函数
    - 在函数返回类型前加 `static` ，函数就定义为静态函数。**函数的定义和声明在默认情况下都是 `extern` 的**，但**静态函数只是在声明他的文件当中可见，不能被其他文件所用。**
    - 函数的实现使用 `static` 修饰，**那么这个函数只可在本 `cpp` 内使用，不会同其他 `cpp` 中的同名函数引起冲突；**
    - `warning`： 不要再头文件中声明 `static` 的全局函数，不要在 `cpp` 内声明非 `static` 的全局函数，如果你要在多个 `cpp` 中复用该函数，就把它的声明提到头文件里去，否则 `cpp` 内部声明需加上 `static` 修饰；
2. 类的静态成员
    - 在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。
    - 因此，**静态成员是类的所有对象中共享的成员，而不是某个对象的成员。**对多个对象来说，静态数据成员只存储一处，供所有对象共用
3. 类的静态函数
    - **静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。**因此，对静态成员的引用不需要用对象名。
    - **在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员(这点非常重要)。***如果静态成员函数中要引用非静态成员时，可通过对象来引用。从中可看出，调用静态成员函数使用如下格式：::();*参数表>静态成员函数名>类名>
    - 不能被 `virtual` 修饰,静态成员函数没有 `this` 指针，虚函数的实现是为每一个对象分配一个 `vptr` 指针，而 `vptr` 是通过 `this` 指针调用的，所以不能为 `virtual`； 虚函数的调用关系， `this` -> `vptr` -> `ctable` -> `virtual function`

6、静态变量什么时候初始化

- 静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配好了内存，
- 但在 `C` 和 `C++` 中静态局部变量的初始化节点又有点不太一样。
    - 在 `C` 中，**初始化发生在代码执行之前，编译阶段分配好内存之后，就会进行初始化**，所以我们看到==在 `C` 语言中无法使用变量对静态局部变量进行初始化==，在程序运行结束，变量所处的全局内存会被全部回收。

- 而在 C++ 中，**初始化时在执行相关代码时才会进行初始化**，主要是由于 C++ 引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。所以 **C++ 标准规定为全局或静态对象是有首次用到时才会进行构造**，并通过 `atexit()` 来管理。在程序结束，按照构造顺序反方向进行逐个析构。所以在 **C++ 中是可以使用变量对静态局部变量进行初始化的。**

7、const?

- 一般可以分为如下六种类型
  - `const` 变量(`const int *p1`): 表明标了为 `const` 类型, 通常需要被初始化否则后面将不能被修改, 对该变量的修改操作都会被编译器阻止.(起始就是 `top-level const`)
  - `const` 指针对象(`int * const p2`): 标明该指针为普通的左值类型可以进行修改, 但是不能通过该变量修改做指向的对象, 则通过该指针只能访问 `const` 类型的成员函数.(`bottom-level const`)
  - `const` 引用: 它所绑定的对象不能被修改
  - `const` 形参: 和普通的实参分类一样分为const 变量, const指针对象, const 引用, 作用也类似,表示不能修改该变量.
  - `const` 返回值: 通常是为了表明返回值是一个**const类型防止返回值被修改**, 或则**被当做左值放在赋值运算的左边**
  - `const` 成员函数: 是指成员函数不会修改类对象的任何成员变量, 如果返回值为对象成员的引用则必须返回 `const` 引用, 同时 `const` 成员函数不能调用非 `const` 函数, 其主要是因为 `const` 成员函数所持有的 `this` 指针是一个 `const` 类型的指针, 因为不能调用非 `const` 类型的成员函数.

- 参考: c++函数返回类型什么情况带const - A_zhu - 博客园 (https://www.cnblogs.com/Azhu/p/4352613.html)

8、 const 成员函数的理解和应用?

- ① const Stock & Stock::topval (②const Stock & s) ③const

- ① 处 const： 确保返回的 Stock 对象在以后的使用中不能被修改
- ② 处 const： 确保此方法不修改传递的参数 s
- ③ 处 const： 保证此方法不修改调用它的对象， const 对象只能调用 const 成员函数,不能调用非 const 函数

9、指针和 const 的用法

- 当 const 修饰指针时，由于 const 的位置不同，它的修饰对象会有所不同。
- **(常指针对象)** int *const p2 中 const 修饰 p2 的值,所以理解为 p2 的值不可以改变，即 p2 只能指向固定的一个变量地址，但可以通过 *p2 读写这个变量的值。顶层指针表示指针本身是一个常量
- **(常指针)** int const *p1 或者 const int *p1 两种情况中 const 修饰 *p1 ，所以理解为 *p1 的值不可以改变，即不可以给 *p1 赋值改变 p1 指向变量的值，但可以通过给 p 赋值不同的地址改变这个指针指向。底层指针表示指针所指向的变量是一个常量。

10、 mutable

- 如果需要在 const 成员方法中修改一个成员变量的值，那么需要将这个成员变量修饰为 mutable 。**即用 mutable 修饰的成员变量不受 const 成员方法的限制;**
- 可以认为 mutable 的变量是类的辅助状态，但是只是起到类的一些方面表述的功能，修改他的内容我们可以认为对象的状态本身并没有改变的。实际上由于 const_cast 的存在，这个概念很多时候用处不是很到了。

- 通常情况下
  - const 成员函数时不能被类对象的成员变量的, 但是可以修改被 mutable 修饰的成员变量
  - 通常我们任务 mutable 位类的辅助状态, 只是类的一些表诉功能, **修改它不会改变对象的状态**
  - 通常我们可以是用 const_cast 在 const 成员函数中修改所有的成员变量

11、 extern 用法?

- extern 修饰变量的声明
  - 如果文件 a.c 需要引用 b.c 中变量 int v，就可以在 a.c 中声明 extern int v，然后就可以引用变量 v。
- extern 修饰函数的声明
  - 如果文件 a.c 需要引用 b.c 中的函数，比如在 b.c 中原型是 int fun(int mu)，那么就可以在 a.c 中声明 extern int fun(int mu)，然后就能使用 fun 来做任何事情。
  - 就像变量的声明一样， extern int fun(int mu) 可以放在 a.c 中任何地方，而不一定非要放在 a.c 的文件作用域的范围中。
  - 默认情况情况下函数都是 extern 的, 除非使用 static 对函数进行了隐匿
- extern 修饰符可用于指示 C 或者 C++ 函数的调用规范。
  - 比如在 C++ 中调用 C 库函数，就需要在 C++ 程序中用 extern “C” 声明要引用的函数。这是给链接器用的，告诉链接器在链接的时候用 C 函数规范来链接。主要原因是 C++ 和 C 程序编译完成后在目标代码中命名规则不同。

12、 int 转字符串, 字符串转 int

- C++11 标准增加了全局函数 std::to_string
- 可以使用 std::stoi / std::stol / std::stoll 等等函数

12.1 strcat,strcpy,strncpy,memset,memcpy 的内部实现?

- strcat： char *strcat(char *dst, char const *src);
  - 头文件: #include <string.h>
  - 作用: 将 dst 和 src 字符串拼接起来保存在 dst 上
  - 注意事项:
    - dst 必须有足够的空间保存整个字符串

        - dst 和 src 都必须是一个由 <!JEKYLL@2780@153> 结尾的字符串(空字符串也行)
        - dst 和 src 内存不能发生重叠
    - 函数实现:
        - 首先找到 dst 的 end
        - 以 src 的 <!JEKYLL@2780@159> 作为结束标志, 将 src 添加到 dst 的 end 上
    - Code

```
char *strcat (char * dst, const char * src){
assert(NULL != dst && NULL != src);   // 源码里没有断言检测
char * cp = dst;
while(*cp )
     cp++;                      /* find end of dst */
while(*cp++ = *src++) ;         /* Copy src to end of dst */
return( dst );                  /* return dst */
}
```

- strcpy：char *strcpy(char *dst, const char *src);
    - 头文件: #include <string.h>
    - 作用: 将 src 的字符串复制到 dst 字符串内
    - 注意事项:
        - src 必须有结束符(<!JEKYLL@2780@171>), 结束符也会被复制
        - src 和 dst 不能有内存重叠
        - dst 必须有足够的内存
    - 函数实现:

```
char *strcpy(char *dst, const char *src){   // 实现src到dst的复制
if(dst == src) return dst;              //源码中没有此项
    assert((dst != NULL) && (src != NULL)); //源码没有此项检查，判断参数src和dst的有效性
    char *cp = dst;                         //保存目标字符串的首地址
    while (*cp++ = *src++);                 //把src字符串的内容复制到dst下
    return dst;
}
```

- strncpy：char *strncpy(char *dst, char const *src, size_t len);

- 头文件: `#include <string.h>`
- 作用: 从 `src` 中复制 `len` 个字符到 `dst` 中, 如果不足 `len` 则用 `NULL` 填充, 如果 `src` 超过 `len` , 则 `dst` 将不会以 `NULL` 结尾
- 注意事项:
  - `strncpy` 把源字符串的字符复制到目标数组，它总是正好向 `dst` 写入 `len` 个字符。
  - 如果 `strlen(src)` 的值小于 `len`， `dst` 数组就用额外的 `NULL` 字节填充到 `len` 长度。
  - 如果 `strlen(src)` 的值大于或等于 `len`，那么只有 `len` 个字符被复制到 `dst` 中。这里需要注意它的结果将不会以 `NULL` 字节结尾。
- 函数实现:

```
char *strncpy(char *dst, const char *src, size_t len)
{
assert(dst != NULL && src != NULL);     //源码没有此项
char *cp = dst;
while (len-- > 0 && *src != '<!JEKYLL@2780@201>')
    *cp++ = *src++;
*cp = '<!JEKYLL@2780@201>';                              //源码没有此项
return dst;
}
```

- `memset`： `void *memset(void *a, int ch, size_t length);`
  - 头文件: `#include <string.h>`
  - 作用:
    - 将参数 `a` 所指的内存区域前 `length` 个字节以参数 `ch` 填入，然后返回指向 `a` 的指针。
    - 在编写程序的时候，若需要将某一数组作初始化， `memset()` 会很方便。
    - 一定要保证 `a` 有这么多字节
  - 函数实现:

```
void *memset(void *a, int ch, size_t length){
assert(a != NULL);
void *s = a;
while (length--)
{
    *(char *)s = (char) ch;
    s = (char *)s + 1;
}
return a;
}
```

- memcpy
  - 头文件: `#include <string.h>`
  - 作用:
    - 从 `src` 所指的内存地址的起始位置开始，拷贝 `n` 个字节的数据到 `dest` 所指的内存地址的起始位置。
    - 可以用这种方法复制任何类型的值，
    - **如果 `src` 和 `dst` 以任何形式出现了重叠，它的结果将是未定义的。**
  - 函数实现:

```
void *memcpy(void *dst, const void *src, size_t length)
{
assert((dst != NULL) && (src != NULL));
    char *tempSrc= (char *)src;            //保存src首地址
    char *tempDst = (char *)dst;           //保存dst首地址
    while(length-- > 0)                    //循环length次，复制src的值到dst中
       *tempDst++ = *tempSrc++ ;
    return dst;
}
```

- `strcpy` 和 `memcpy` 的主要区别:
  - 复制的内容不同: `strcpy` 只能复制字符串，而 `memcpy` 可以复制任意内容，例如字符数组、整型、结构体、类等。

- 复制的方法不同： `strcpy` 不需要指定长度，它遇到被复制字符的串结束符 <!JEKYLL@2780@225> 才结束，所以容易溢出。 `memcpy` 则是根据其第 3 个参数决定复制的长度，遇到 <!JEKYLL@2780@228> 并不结束。
- 用途不同: 通常在复制字符串时用 `strcpy`，而需要复制其他类型数据时则一般用 `memcpy`

- 参考: 各种C语言处理函数 strcat，strcpy，strncpy，memset，memcpy 总结 - New World - CSDN博客 (https://blog.csdn.net/nyist_zxp/article/details/80982472)

13、深拷贝与浅拷贝?

- 浅复制:
  - 只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“(浅复制)浅拷贝”，
  - 换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。
- 深复制: 在计算机中开辟了一块新的内存地址用于存放复制的对象。

- 浅复制的问题:
  - 在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如A=B。
  - 这时，如果B 中有一个成员变量指针已经申请了内存，那A 中的那个成员变量也指向同一块内存。
  - 这就出现了问题：当B把内存释放了(如：析构)，这时A 内的指针就是野指针了，出现运行错误。

14、 C++ 模板是什么，底层怎么实现的?

- **编译器并不是把函数模板处理成能够处理任意类的函数**；编译器从函数模板通过具体类型产生不同的函数；

- 编译器会对函数模板进行两次编译：
  - 在声明的地方对模板代码本身进行编译，
  - 在调用的地方对参数替换后的代码进行编译。
- 这是因为**函数模板要被实例化后才能成为真正的函数**，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。
- 模板可以重载返回值, 函数重载不行

- 如果我们试图通过在头文件中定义函数模板, 在 cpp 文件中实现函数模板, 那么我们必须在在实现的那个 cpp 文件中**手动实例化**, 也就是使用你需要使用的参数替换模板, 从而使得编译器为你编译生成相应参数的模板函数.

15、C 语言 struct 和 C++ struct 区别

- struct 在 C语言 中:
  - 是**用户自定义数据类型（UDT）**；
  - 只能是一些**变量的集合体**, 成员不能为函数
  - 没有权限设置
  - 一个**结构标记**声明后，在 C 中必须在**结构标记**前加上 struct ， 才能做**结构类型名**;

- struct 在 C++ 中:
  - 是**抽象数据类型（ADT）**，支持成员函数的定义，(能继承，能实现多态)。
  - 增加了访问权限, 默认访问限定符为 public (为了与 C 兼容)， class 中的默认访问限定符为 private
  - 定义完成之后, 可以直接使用**结构体名字**作为**结构类型名**

- 可以使用模板

16、虚函数可以声明为 `inline` 吗?

- **虚函数要求在运行时进行类型确定，而内敛函数要求在编译期完成相关的函数替换, 所以不能**

- 虚函数用于实现运行时的多态，或者称为晚绑定或动态绑定。
- 内联函数用于提高效率, 对于程序中需要频繁使用和调用的小函数非常有用。它是在**编译期间**，对调用内联函数的地方的代码**替换**成函数代码。

17、类成员初始化方式？构造函数的执行顺序？为什么用成员初始化列表会快一些?

- 概念
  - 赋值初始化，通过在函数体内进行赋值初始化；
  - 列表初始化，在冒号后使用初始化列表进行初始化。
- 这两种方式的主要区别在于：
  - 对于在**函数体中初始化**,是在所有的成员函数分配空间后才进行的。**对于类对象类型成员变量, 则是先调用零参数构造函数, 如果零参数构造函数不存在编译器将会报错.**
  - 列表初始化是给数据成员分配内存空间时就进行初始化,就是说分配一个数据成员只要冒号后有此数据成员的赋值表达式(此表达式必须是括号赋值表达式)。
- 快的原因: 所以对于列表初始化: 只进行了一次初始化操作, 而赋值初始化则先进性了一次初始化,然后调用了一次复制构造函数.

- 一个派生类构造函数的执行顺序如下：
  - **虚基类**的构造函数(多个虚拟基类则按照继承的顺序执行构造函数)。
  - **基类**的构造函数(多个普通基类也按照继承的顺序执行构造函数)。
  - **类类型的成员对象**的构造函数(按照初始化顺序)
  - 派生类**自己的构造函数**。

18、成员列表初始化?

- **必须**使用成员初始化的四种情况
  - 当初始化一个**引用**成员时；
  - 当初始化一个**常量成员**时；
  - **基类**， **无零参数构造函数**时
  - **成员类**， **无零参数构造函数**时
- 成员初始化列表做了什么
  - 编译器在调用用户代码之前, 会按照类成员声明顺序一一初始化成员变量, 如果成员初始化类别中有初值,则使用初值构造成员函数.
  - 初始化顺序由类中的成员声明顺序决定的，不是由初始化列表的顺序决定的；

19、构造函数为什么不能为虚函数？析构函数为什么要虚函数?

构造函数为什么不能为虚函数?

- 首先是没必要使用虚函数:
  - 由于使用间接调用(通过引用或则指针)导致类**类型不可信**, 而使用虚函数机制完成正确的函数调用.
  - 但是构造函数本身是为了初始化对象实例, 创建对象必须制定它的类型, 其类类型是明确的, 因此在编译期间即可确定调用函数入口地址

    - 因而没必要使用虚函数, 其调用在编译时由编译器已经确定.
- 其次不能使用虚函数:
    - 虚函数的调用依赖于虚函数表, 虚函数表储存于静态储存区, 在存在虚函数的对象中都将插入一个指向虚函数表的指针,
    - 在对象中插入一个指向虚函数表的指针是由构造函数完成的, 也就是说在调用构造函数时并没有指向虚函数表的指针, 也就不能完成虚函数的调用.

析构函数为什么要虚函数?

- C++ 中基类采用 virtual 虚析构函数是为了防止内存泄漏。
    - 如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。
    - 假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。
    - 所以，为了防止这种情况的发生， C++ 中基类的析构函数应采用 virtual 虚析构函数。

20、析构函数的作用，如何起作用?

- 析构函数名与类名相同,只是在函数名前增加了取反符号 ~ 以区别于构造函数,其不带任何参数, 也**没有返回值**. 也**不允许重载**.
- 析构函数与构造函数的作用相反, 当对象生命周期结束的时候,如对象所在函数被调用完毕时,析构函数负责结束对象的生命周期. **注意如果类对象中分配了堆内存一定要在析构函数中进行释放.**
- 和拷贝构造函数类似,如果用户未定义析构函数, 编译器**并不是一定**会自动合成析构函数, 只有在**成员变量或则基类**拥有析构函数的情况下它才会自动合成析构函数.
- 如果**成员变量或则基类**拥有析构函数, 则编译器一定会合成析构函数, 负责调用成员变量或则基类的析构函数, 此时如果用户提供了析构函数,则编译器会在用户析构函数之后添加上述代码.
- 类析构的顺序为: 派生类析构函数, 对象成员析构函数, 基类析构函数.

21、构造函数和析构函数可以调用虚函数吗，为什么

- 在C++中，提倡不在构造函数和析构函数中调用虚函数；
- 在构造函数和析构函数调用的所有函数(包括虚函数)都是编译时确定的, 虚函数将运行该类中的版本.
  - 因为**父类对象会在子类之前进行构造**，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而C++不会进行动态联编；
  - 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义。

- 参考: C++构造函数和析构函数的调用顺序 - DoubleLi - 博客园 (https://www.cnblogs.com/lidabo/p/9328323.html)

22、构造函数的执行顺序？析构函数的执行顺序？构造函数内部干了啥？拷贝构造干了啥？

- 构造函数顺序:
  - **基类**构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。
  - **成员类对象**构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。
  - **派生类**构造函数。
- 析构函数顺序:
  - 调用**派生类**的析构函数；
  - 调用**成员类对象**的析构函数；
  - 调用**基类**的析构函数。

23、虚析构函数的作用，父类的析构函数是否要设置为虚函数?

- C++ 中基类采用 virtual 虚析构函数是为了防止内存泄漏。
  - 如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。
  - 假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。
  - 所以，为了防止这种情况的发生， C++ 中基类的析构函数应采用 virtual 虚析构函数。

- **纯虚析构函数一定得有定义**，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层基类的析构函数。**因此缺乏任何一个基类析构函数的定义，就会导致链接失败。**==因此，最好不要把虚析构函数定义为纯虚析构函数。==

24、 构造函数 析构函数 可以调用虚函数吗?

- 在 构造函数 和 析构函数 中最好不要调用虚函数;
- 在 构造函数 和 析构函数 中调用的成员函数都是属于**编译时确定**的,并**不具有虚函数的动态绑定特性**, 有如下原因:
  - **在构造时, 父类对象总是先于子类对象构造**的, 如果父类的析构函数使用虚函数机制调用子类的函数, 结果将是不可预料的
  - **在析构时, 子类的析构函数总是先于父类**执行, 如果父类的析构函数使用虚函数机制调用子类的函数, 结果将是不可预料的

- **参考**: 21、构造函数和析构函数可以调用虚函数吗，为什么

25、 构造函数，析构函数可否抛出异常

- 构造函数异常
  - 后果:
    - **(原因):** C++ 拒绝为**没有完成构造函数的对象**调用**析构函数**，原因是避免开销
    - 构造函数中发生异常，控制权转出构造函数。如果构造函数中申请了堆内存, 则堆内存将无法释放, 从而造成内存泄漏
    - 例如: 在对象 b 的构造函数中发生异常，对象 b 的析构函数不会被调用。**因此会造成内存泄漏。**
  - 解决方案:
    - 使用**智慧指针**来管理堆内存. 其不需要在析构函数中手动释放资源. 在发生异常时, 智慧指针会自动释放资源从而避免了内存泄漏.
    - **一般建议不要在构造函数里做过多的资源分配。**
- 析构函数异常
  - 后果:
    - 在异常传递的**堆栈辗转开解**的过程中, 如果发生析构异常，C++ 会调用 terminate 终止程序
    - 如果析构函数发生发生异常，则异常后面的代码将不执行，无法确保完成我们想做的清理工作。
  - 解决方法：
    - 如果异常不可避免，则应在析构函数内捕获，而不应当抛出。
    - 在析构函数中使用 try-catch 块屏蔽所有异常。
  - 附加说明:
    - **(后果1):** 如果某一个异常发生,某对象的析构函数被调用,而此时析构发生了异常并流出了函数之外,则函数会被立即terminate掉(函数外有catch也不能拯救)

- 参考:
  - More Effective : M9 使用析构函数防止资源泄漏 (https://blog.csdn.net/zzxiaozhao/article/details/102504097#M9__116)
  - More Effective : M10 在构造函数中防止资源泄漏 (https://blog.csdn.net/zzxiaozhao/article/details/102504097#M10__139)

- `More Effective`：M11 禁止异常流出 destructors 之外 (https://blog.csdn.net/zzxiaozhao/article/details/102504097#M11__destructors__142)

26、类如何实现只能静态分配和只能动态分配

- 建立类的对象有两种方式：
  - 静态建立(栈空间)
    - 静态建立一个类对象，就是由编译器为对象在栈空间中分配内存, 然后调用构造函数初始化这片内存空间.
    - 使用这种方法，**直接调用类的构造函数。**
  - 动态建立(堆空间)，`A *p = new A()`；
    - 动态建立类对象, 使用new操作符将在堆空间分配内存, 然后调用构造函数初始化这片内存空间.
    - 这种方法，**间接调用类的构造函数。**
- 只能在堆上建立
  - 分析: 类对象只能建立在堆上，就是不能**静态建立类对象**，即不能**直接调用类的构造函数。**
  - 实现方式: 将**析构函数设为私有或则受保护**
  - 方法分析:
    - 静态建立:
      - 当对象 建立 在栈上面时，是由编译器分配内存空间的，调用 构造函数 来 构造 栈对象。
      - 当对象使用 完 后，编译器会调用 析构函数 来 释放 栈对象所占的空间。
      - 编译器管理了对象的整个生命周期。
    - 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**，
      - 其实不光是析构函数，只要是非静态的函数，编译器都会进行检查。
      - 如果类的析构函数是私有的，则编译器不会在栈空间上为类对象分配内存。
      - 因此，将析构函数设为私有，类对象就无法建立在栈上了。
    - 由此引发的问题:

    - 因为析构函数设置为了私有
    - 需要设置一个 public函数 来调用析构函数
  - 代码如下：

```
class  A
{
protected :
A(){}
~A(){}
public :
static  A* create()
{
    return   new  A();
}
void  destory()
{
    delete   this ;
}
};
```

- 只能在栈上建立
  - 只有使用new运算符，对象才会建立在堆上，因此，只要禁用new运算符就可以实现类对象只能建立在栈上。**将operator new()设为私有即可。**
  - **注意: 重载了 new 就需要重载 delete**
  - 代码如下：

```
class  A
{
private :
    void * operator  new ( size_t  t){}      // 注意函数的第一个参数和返回值都是固定的
    void  operator  delete ( void * ptr){}  // 重载了new就需要重载delete
public :
    A(){}
    ~A(){}
};
```

- 参考:
  - 在堆/栈上建立对象 (https://zhaostu4.github.io/2019/11/28/%E6%80%BB%E7%BB%93%E7%B3%BB%E5%88%97-%E7%BB%93%E8%AE%BA%E9%9B%86%E5%90%88/#%E5%9C%A8%E5%A0%86%E6%A0%88%E4%B8%8A%E5%BB
  - 如何定义一个只能在堆上（栈上）生成对象的类? (https://www.nowcoder.com/questionTerminal/0a584aa13f804f3ea72b442a065a7618)

27、如果想将某个类用作基类，为什么该类必须定义而非声明?

- 因为在继承体系下, 子类会继承父类的成员, 并且编译器会在子类的构造函数和析构函数中插入父类的构造和析构部分, 因而父类必须有定义.

28、什么情况会自动生成默认构造函数?

- 四种情况:
  - 类成员对象带有默认构造函数.
  - 基类带有默认构造函数
  - 类中存在虚函数
  - 继承体系中存在虚继承
- **在合成的默认构造函数中，只有基类子对象和类类型对象会被初始化**，而其他所有的非静态成员(如整数，指针，数组等)，都不会初始化，对他们进行初始化的应该是程序员，而非编译器。
- 注意：值类型的默认值并不是默认构造的初始化。

- 参考: C++关于编译器合成的默认构造函数 - Cheny# - 博客园 (https://www.cnblogs.com/zjc0202/p/4504227.html)

29、什么是类的继承?

- 类与类之间的关系
  - (has-A) 包含关系，即一个类的成员属性是另一个已经定义好的类
  - (use-A) 使用关系, 一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式实现;
  - (is-A) 继承关系, 继承关系，关系具有传递性;
- 继承的相关概念
  - 所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，
  - 被称为子类或者派生类，被继承的类称为父类或者基类;
- 继承的特点
  - 子类拥有父类的所有属性和方法，子类对象可以当做父类对象使用;
  - 子类可以拥有父类没有的属性和方法;
- 继承中的访问控制
  - public 、 protected 、 private
- 继承中的构造和析构函数
  - 子类中构造函数的调用顺序为: 基类构造函数, 成员对象构造函数, 派生类构造函数
  - 子类中析构函数的调用顺序为: 派生类析构函数, 成员对象析构函数, 基类析构函数
- 继承中的兼容性原则
  - 类型兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。
  - 参考: 继承中的类型兼容性原则 - Say舞步 - 博客园 (https://www.cnblogs.com/zhangyaoqi/p/4591571.html)

30、什么是组合?

- 一个类里面的数据成员是另一个类的对象，即内嵌其他类的对象作为自己的成员;
- 如果内嵌类没有零参数构造函数， 则必须使用初始化列表进行初始化

- 构造函数的执行顺序：
  - 按照内嵌对象成员在组合类中的定义顺序调用内嵌对象的构造函数。
  - 然后执行组合类构造函数的函数体，析构函数调用顺序相反。

31、抽象基类为什么不能创建对象?

- 抽象类的定义：**带有纯虚函数的类**为抽象类。
- 抽象类的作用：
  - 抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。
  - 所以抽象类实际上刻画了一组子类的**操作接口**的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。
- 使用抽象类时注意：
  - 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。
  - 如果**派生类中没有给出所有纯虚函数的实现**，而只是继承基类的纯虚函数，则这个**派生类仍然是一个抽象类。**
  - 如果**派生类中给出了所有纯虚函数的实现**，则该**派生类就不再是抽象类**了，它是一个可以建立对象的具体的类。
  - **抽象类是不能定义对象的。**

- 纯虚函数定义: 纯虚函数是一种特殊的虚函数，它的一般格式如下：

```
class <类名>
{
    virtual <类型><函数名>(<参数表>)=0;
    …
};
```

- 纯虚函数引入原因
  - 为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

    - 在很多情况下，基类本身生成对象是不合情理的。
    - 例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。
- 为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数(方法: `virtual ReturnType Function()= 0;`)。
    - 若要使派生类为非抽象类，则编译器要求在派生类中，必须对纯虚函数予以重载以实现多态性。
    - **同时含有纯虚函数的类称为抽象类**，它不能生成对象。

- 相似概念
    - 多态性
        - 指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。
        - C++支持两种多态性：编译时多态性，运行时多态性。
            - 编译时多态性(静态多态)：通过重载函数实现。
            - 运行时多态性(动态多态)：通过虚函数实现。
    - 虚函数
        - 虚函数是在基类中被声明为 `virtual` ，并在派生类中重新定义的成员函数，可实现成员函数的动态重载。
    - 抽象类
        - 包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象。

32、类什么时候会析构?

- 对于静态对象: 当离开作用区域之后, 对象生命周期结束, 编译器会自动调用析构函数
- 对于动态对象: 当对对象指针调用delete时, 会调用析构函数终止对象生命周期并释放内存. 其中对象指针指针可以对象类型的指针, 也可以时基类指针(注意基类析构函数位虚函数)
- 第三种情况: 当对象中存在嵌入对象时, 该对象析构时, 嵌入对象也会被析构

33、为什么友元函数必须在类内部声明?

- 因为编译器必须能够读取这个结构的声明以理解这个数据类型的大、行为等方面的所有规则。
- 有一条规则在任何关系中都很重要，那就是谁可以访问我的私有部分。

- 编译器通过读取类的声明从而进行类的访问权限控制, 而友元函数有权访问本类的所有成员, 因而它必须在类内部进行声明, 使得编译器可以正确处理他的权限.

34、介绍一下C++里面的多态?

- 静态多态(重载, 模板): 是在编译的时候，就确定调用函数的类型。
- 动态多态(覆盖, 虚函数实现): 在运行的时候，才确定调用的是哪个函数，动态绑定。运行基类指针指向派生类的对象，并调用派生类的函数。

- 参考: 理解的虚函数和多态 (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-C++%E5%9F%BA%E7%A1%80/#%E7%90%86%E8%A7%A3%E7%9A%84%E8%99%9A%E5%87%BD%E6%95%B0%E5%92%
- 函数重载:
  - 同一可访问区域内, 存在多个不同参数列表的同名函数, 由编译器根据调用参数决定那个函数应该被调用
  - 函数重载不关心返回值类型, 但是对于函数类型时关心的, 例如类中的两个函数拥有相同参数列表的同名函数, 一个为const类型, 一个为非const类型, 依旧时属于函数重载.
- 函数模板:
  - 模板函数会经历两遍编译:
    - (模板编译)在定义模板函数时对模板本身进行编译

        - (模板实例化)在调用时对参数进行替换, 对替换参数后的代码进行编译
    - 虽然它和函数重载类似都可以根据参数确定将要调用的函数版本, 但是函数模板只会生成将要用到的函数版本, 而函数模板无论是否调用其代码都会生成.
- 覆盖: 是指派生类中重新定义了基类中的 `virtual` 函数
- **隐藏**：是指派生类的函数屏蔽了与其同名的基类函数，只要函数名相同，基类函数都会被隐藏. 不管参数列表是否相同。

35、用C 语言实现C++的继承

- 关键点:
    - 使用函数指针保存函数
    - 将基类放在结构题的头部, 这样强转的就不会出错了 ```cpp #include using namespace std; //C++中的继承与多态 struct A{ virtual void fun() {//C++中的多态:通过虚函数实现 cout<<"A:fun()"<<endl; } int a; }; struct B:public A {//C++中的继承:B 类公有继承A 类 virtual void fun() { //C++中的多态:通过虚函数实现（子类的关键字virtual 可加可不加） cout<<"B:fun()"<<endl; } int b; };

  //C 语言模拟C++的继承与多态 typedef void (*FUN)(); //定义一个函数指针来实现对成员函数的继承 struct _A { //父类 FUN _fun; //由于C 语言中结构体不能包含函数，故只能用函数指针在外面实现 int _a; }; struct _B { //子类 _A _a_; //在子类中定义一个基类的对象即可实现对父类的继承 int _b; }; void _fA() { //父类的同名函数 printf("_A:_fun()\n"); } void _fB() { //子类的同名函数 printf("_B:_fun()\n"); } void Test() { //测试C++中的继承与多态 A a; //定义一个父类对象a B b; //定义一个子类对象b A* p1 = &a; //定义一个父类指针指向父类的对象 p1->fun(); //调用父类的同名函数 p1 = &b; //让父类指针指向子类的对象 p1->fun(); //调用子类的同名函数 //C 语言模拟继承与多态的测试 A _a; //定义一个父类对象_a _B _b; //定义一个子类对象_b _a._fun = _fA; //父类的对象调用父类的同名函数 _b._a._fun = _fB; //子类的对象调用子类的同名函数 _A* p2 = &_a; //定义一个父类指针指向父类的对象 p2->_fun(); //调用父类的同名函数 p2 = (_A*)&_b; //让父类指针指向子类的对象,由于类型不匹配所以要进行强转 p2->_fun(); //调用子类的同名函数 } ```

36、继承机制中对象之间如何转换？指针和引用之间如何转换？

- 派生类的对象可以当做基类对象使用， 例如赋值或则初始化等
- 派生类对象的地址可以赋给指向基类的指针。 在替代之后，派生类对象就可以作为基类的对象使用，但只能使用从基类继承的成员。
- 向上类型转换(派生类转基类, 总是安全的)
  - 将派生类指针或引用转换为基类的指针或引用被称为向上类型转换，**向上类型转换会自动进行**，而且向上类型转换是安全的。
- 向下类型转换(基类转派生类, 不安全)
  - 将基类指针或引用转换为派生类指针或引用被称为向下类型转换，向下类型转换不会自动进行，因为一个基类对应几个派生类，所以向下类型转换时不知道对应哪个派生类，所以在向下类型转换时必须加动态类型识别技术。
  - `RTTI` 技术，用dynamic_cast进行向下类型转换, 只有存在虚函数的类才能使用 `RTTI`

- 参考:
  - 浅谈C++类型转换的安全性 - freshman94的博客 - CSDN博客 (https://blog.csdn.net/qq_22660775/article/details/88715548)
  - 继承的赋值兼容规则 (https://blog.csdn.net/vjhghjghj/article/details/90677092#%C2%A0%E7%BB%A7%E6%89%BF%E7%9A%84%E8%B5%8B%

37、组合与继承优缺点?

- 继承: 继承是Is a 的关系，比如说Student 继承Person,则说明Student is a Person。
- 继承的优点: 是子类可以重写父类的方法来方便地实现对父类的扩展。
- 继承的缺点有以下几点:
  - ①：父类的**内部细节**对子类是**可见**的。(可以自己调用父类的方法)

    - ②：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。
    - ③：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

- 组合(嵌入式对象): 组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量。
- 组合的优点：
    - ①：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的**内部细节**对当前对象时**不可见**的。(必须通过嵌入式对象调用嵌入式对象的方法)
    - ②：当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码。
    - ③：当前对象可以在运行时动态的绑定所包含的对象。可以通过set 方法给所包含对象赋值。
- 组合的缺点：
    - ①：容易产生过多的对象。
    - ②：为了能组合多个对象，必须仔细对接口进行定义。

- 参考: 继承的优点和缺点 (https://blog.csdn.net/u013675978/article/details/82628710)

38、左值右值

- 参考: 什么是右值引用，跟左值又有什么区别？ (https://blog.csdn.net/zzxiaozhao/article/details/102943714#_45)

39、移动构造函数

- 右值的概念: 将亡值, 不具名变量

- 右值引用
  - 概念: 其本身是一个左值, 但是它绑定了一个右值, 此右值的生命周期将和此右值引用一致.
  - 优点:
    - 转移语意
    - 精确语意传递(参数列表分别为**左值引用**和**右值引用**形成参数重载)
- 移动构造函数:
  - 概念: 当我们使用一个即将消亡的对象A初始化对象B时, 使用移动语意可以避免额外的无意义的复制构造操作, 也避免了释放内存, 新分配内存的开销.
  - 实现:
    - 移动构造函数的参数和拷贝构造函数不同，拷贝构造函数的参数是一个左值引用，但是移动构造函数的初值是一个右值引用。
    - 也就是说，只用一个右值，或者将亡值初始化另一个对象的时候，才会调用移动构造函数。
    - **作为参数的右值将不会再调用析构函数。**
    - `move` 语句，就是将一个左值变成一个将亡值。
  - 优点
    - 避免了无畏的对下销毁和构造的开销
    - 当该类对象申请了堆内存, 并在析构函数中进行释放时, 使用拷贝构造函数可能会存产生也野指针, 而使用移动构造可以避免野指针的产生.

40、C 语言的编译链接过程?

- 源代码 - - >预处理 - - >编译 - - >优化 - - >汇编 - - >链接–>可执行文件
- 参考: 源码到可执行文件的过程 (https://blog.csdn.net/zzxiaozhao/article/details/102990773#_1128)

41、vector与list的区别与应用？怎么找某vector或者list的倒数第二个元素

- vector
  - vector 和数组类似，拥有一段连续的内存空间，并且起始地址不变。
  - 因此能高效的进行随机存取，时间复杂度为 o(1)；
  - 连续存储结构：
    - vector 是可以实现动态增长的对象数组，支持对数组高效率的访问和在数组尾端的删除和插入操作，在中间和头部删除和插入相对不易，需要挪动大量的数据。
    - 它与数组最大的区别就是 vector 不需程序员自己去考虑容量问题，库里面本身已经实现了容量的动态增长，而数组需要程序员手动写入扩容函数进形扩容。

```
        - 随机访问
        - 高效的尾部操作(增/删)
        - 不那么高效的非尾部操作(增/删)，后面的迭代器会失效
        - 动态扩容，迁移，迭代器全部失效 - `list`
- `list`是由双向链表实现的，因此内存空间是不连续的。
- 非连续存储结构：
        - `list `是一个双链表结构，支持对链表的双向遍历。
        - 每个节点包括三个信息：元素本身，指向前一个元素的节点`(prev)`和指向下一个元素的节点`(next)`。
        - 因此`list `可以高效率的对数据元素任意位置进行访问和插入删除等操作。由于涉及对额外指针的维护，所以开销比较大。
```

```
        - 高效的插入和删除，后续迭代器不失效
        - 指针维护开销大
        - 不支持随机访问 - 区别：
- `vector `的随机访问效率高，但在插入和删除时(不包括尾部)需要挪动数据，不易操作。
- `list `的访问要遍历整个链表，它的随机访问效率低。
- 但对数据的插入和删除操作等都比较方便，改变指针的指向即可。
- `list `是单向的，`vector `是双向的。`vector `中的迭代器在使用后就失效了，而`list `的迭代器在使用之后还可以继续使用。
```

- `int mySize = vec.size()；vec.at(mySize -2);`
- `list` 不提供随机访问，所以不能用下标直接访问到某个位置的元素，要访问 `list` 里的元素只能遍历，
- 不过你要是只需要访问 `list` 的最后 `N` 个元素的话，可以用**反向迭代器**来遍历：

42、 STL vector 的实现，删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间？

- vector相关函数:
  - `size / capacity`：已用空间 / 总空间
  - `resize / reserve`：改变容器的元素数目 / 概念容器的空间大小
  - `push_back / pop_back`：尾插 / 尾减
  - `insert / erase`：任意位置插入 / 任意位置删除
- 迭代器失效问题:
  - 在 `capacity` 内 `insert` 和 `erase` 都会导致在后续元素发生移动, 进而迭代器失效或则改变
  - 如果 `insert` 或则 `push_back` 导致空间不足, 则会发生整体的移动操作, 所有迭代器都将失效.
- 两倍扩容问题:
  - 为什么呈倍数扩容(时间复杂度更优)
    - 对于 `n` 次插入操作, 采用成倍方式扩容可以保证时间复杂度 `O(n)`，而指定大小扩容的时间复杂度为 `O(n^2)`
  - 为什么是 `1.5` 倍扩容(空间可重用)
    - 当 `k == 2` 时:

        - 第n次扩容的时候需要分配的内存是： `an = a1*q^(n-1) = 4*2^(n-1)`
        - 而前n-1项的内存和为： `Sn = a1*(1-q^(n-1))/(1-q) = 4*(1-2^(n-1)) /(1-2) = 4*2^(n-1)-4`
        - 差值 `= an - Sn = 4 > 0`
        - 所以第 `n` 次扩容需要的空间恰好比前 `n-1` 扩容要求的空间总和要大，那么即使在前 `n-1` 次分配空间都是连续排列的最好情况下，也无法实现之前的内存空间重用
    - 当 `k = 1.5` 时:
        - 第 `n` 次扩容的时候需要分配的内存是： `an = a1*q^(n-1) = 4*1.5^(n-1)`
        - 而前 `n-1` 项的内存和为： `Sn = a1*(1-q^(n-1))/(1-q) = 4*(1-1.5^(n-1)) /(1-1.5) = 8*1.5^(n-1)-8`
        - 差值 `= an - Sn = 8 - 4*1.5^(n-1)`
        - 当 `n` 增长到一定的数值后，差值就会变为小于 `0`，那么如果前 `n-1` 次分配的空间都是连续的情况下, 就可以实现内存空间复用
- 释放空间:
    - 使用 `swap`： `vector<int>().swap(a);`

- 参考:
    - STL中vector 扩容为什么要以1.5倍或者2倍扩容？ - Bryant_xw Is Growing~~ - CSDN博客 (https://blog.csdn.net/bryant_xw/article/details/89524910)
    - C++ STL中vector内存用尽后，为啥每次是两倍的增长，而不是3倍或其他数值？ - 知乎 (https://www.zhihu.com/question/36538542/answer/67929747)

43、容器内部删除一个元素

- 顺序容器

    - erase 迭代器 不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list除外)，所以不能使用 erase(it++) 的方式，但是 erase 的返回值是下一个有效迭代器；
    - it = c.erase(it);
- 关联容器
    - erase 迭代器 只使被删除元素的迭代器失效, 其他迭代器不失效，但是返回值是 void ， 所以要采用 erase(it++) 的方式删除迭代器；
    - c.erase(it++)

44、 STL 迭代器如何实现

- 迭代器 Iterator
    - (总结) Iterator 使用聚合对象, 使得我们在不知道对象内部表示的情况下, 按照一定顺序访问聚合对象的各个元素.
    - Iterator 模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。
    - 由于 Iterator 模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如 STL 的 list 、 vector 、 stack 等容器类及 ostream_iterator 等扩展 iterator 。
- 迭代器的基本思想:
    - 迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符， -> 、 * 、 ++ 、 -- 等。
    - 迭代器封装了指针，是一个“可遍历 STL(Standard Template Library) 容器内全部或部分元素”的对象， 本质是封装了原生指针，是指针概念的一种提升 (lift)，提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的 ++ ， -- 等操作。
    - 迭代器返回的是对象引用而不是对象的值。
- 迭代器产生原因
    - Iterator 类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构就可以实现集合的遍历,是算法和容器之间的桥梁.

- 最常用的迭代器的相应型别有五种： value type 、 difference type 、 pointer 、 reference 、 iterator catagoly ；

45、 set 与 hash_set 的区别

- set 底层是以 RB-Tree 实现， hash_set 底层是以 hash_table 实现的；
- RB-Tree 有自动排序功能，而 hash_table 不具有自动排序功能；
- set 和 hash_set 元素的键值就是实值；
- hash_table 有一些无法处理的型别；(例如字符串无法对 hashtable 的大小进行取模)

46、 hashmap 与 map 的区别

- 底层实现不同；
- map 具有自动排序的功能， hash_map 不具有自动排序的功能；
- hashtable 有一些无法处理的型别；(例如字符串无法对 hashtable 的大小进行取模)

47、 map、 set 是怎么实现的，红黑树是怎么能够同时实现这两种容器？ 为什么使用红黑树?

- map 和 set 都是 STL 中的关联容器，其底层实现都是红黑树(RB-Tree)。由于 map 和 set 所开放的各种操作接口， RB-tree 也都提供了，所以几乎所有的 map 和 set 的操作行为，都只是转调 RB-tree 的操作行为。
- map 中的元素是 key-value(关键字—值)对： 关键字起到索引的作用，值则表示与索引相关联的数据， 红黑树的每个节点包括 key 和 value ；
- set 只是关键字的简单集合，它的每个元素只包含一个 关键字 ， 红黑树每个节点只包括 key 。
- 红黑树的插入删除都可以在 O(logn) 时间内完成, 性能优越

48、如何在共享内存上使用STL标准库?

- 为什么要在共享内存中使用模板容器?
  - 共享内存可以在多进程间共享，到达进程间通信的方式。
  - 共享内存可以在进程的生命周期以外仍然存在。这就可以保证在短暂停止服务(服务进程 `coredump` ， 更新变更)后，服务进程仍然可以继续使用这些共享内存的数据。
  - 如果这些优势在加上 `C++` 容器模板使用方便，开发快速的优势，无疑是双剑合璧，成为服务器开发的利刃。

.....太难了.....

- 参考: C++容器模板在共享内存中的使用 (https://blog.csdn.net/fullsail/article/details/8540078)

49、 `map` 插入方式有几种?

- 下表运算符插入 `[]`
- `insert` 插入 `pair`

50、STL 中 `unordered_map` 和 `map` 的区别， `unordered_map` 如何解决冲突以及扩容

- `unordered_map` 和 `map` 都是键值对不可重复的关联容器,
- 区别:
  - `map` 的底层实现为红黑树, 会根据键值进行排序, 所以键值需要定义小于操作 (`operator<`)

    - `unordered_map` 底层实现为 `hash_table`，不会根据键值进行排序, 但是需要键值提供等于操作 `(operator ==)`，以防止重复键值
- 哈希表解决冲突常见办法:
    - 开放定址法: 线性探测, 二次探测, 二次哈希
    - **(STL使用):** 拉链法: 使用单链表来保存具有相同哈希值得集合
- 哈希表扩容
    - 什么时候扩容: 哈希表键值发生碰撞的概率, 随着负载因子(负载/容量)的增加而增加, 所以当负载因子大于阈值(0.75)的时候就需要扩容了.
    - 怎么扩容 `(resize)`：通过增加桶的数量(两倍扩张)以达到扩容的目的, 然后将原来的所有键值 `rehash` 到新的哈希表中, 增大哈希表并不会影响哈希表的插入删除时间, 那是 `rehash` 需要的时间复杂度为 `n`，所以对实时性非常严格的情况下不要使用

- 参考:
    - 数据结构——哈希表(散列表) (https://blog.csdn.net/chenhanzhun/article/details/38091431)
    - 散列表(哈希表)(散列函数构造、处理冲突、查找) (https://blog.csdn.net/qq_22238021/article/details/78258605)
    - C++ STL hash表用法 (https://www.cnblogs.com/downey-blog/p/10471875.html)
    - hashtable详解 (https://www.cnblogs.com/yyxt/p/4985894.html)
    - 扩容:
        - Hash table详解 (http://zheming.wang/blog/2014/06/17/05E21D24-A791-4D97-993D-98B7E6C88BC2/)
        - HASH TABLE::DYNAMIC RESIZING (Java, C++) (http://www.algolist.net/Data_structures/Hash_table/Dynamic_resizing) - [对 c++ unordered_map 源码的解析 | ZRJ] (https://zrj.me/archives/1248)

51、 vector越界访问下标， map越界访问下标？ vector删除元素时会不会释放空间?

- vector 通过下标访问时不会做边界检查，即便下标越界。
  - 也就是说，下标与 first 迭代器相加的结果超过了 finish 迭代器的位置，程序也不会报错，而是返回这个地址中存储的值。
  - 如果想在访问 vector 中的元素时首先进行边界检查，可以使用 vector 中的 at 函数。
  - 通过使用 at 函数不但可以通过下标访问 vector 中的元素，而且在 at 函数内部会对下标进行边界检查。

  - **我去是真的**: 访问vector元素时的越界问题 (https://blog.csdn.net/zrh_CSDN/article/details/80959258)

- map 通过校表访问会将不存在的 key 插入到 map 中
- map 的下标运算符 [] 的作用是：将 key 作为下标去执行查找，并返回相应的值；如果不存在这个 key ，就将一个具有该 key 和 value 的某人值插入这个 map 。
- erase() 函数，只能删除内容，不能改变容量大小； erase 成员函数，它删除了 itVect 迭代器指向的元素，并且返回要被删除的 itVect 之后的迭代器，迭代器相当于一个智能指针; clear() 函数，只能清空内容，不能改变容量大小;

  - 如果要想在删除内容的同时释放内存，那么你可以选择 deque 容器。(deque 也不总是 erase 之后就会释放内存， 当内存块不在被使用时会释放)
  - 参考： STL容器删除元素时内存释放情况 (https://blog.csdn.net/weixin_30247159/article/details/97269861)

52、 map[]与find的区别?

1) map 的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项**插入**这个map。 2) map 的find 函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

53、STL 中list，queue之间的区别

- vector：连续空间存储, 支持随机访问, 高效尾部操作(增/删), 动态空间分配, 迭代器易失效
- list：双向链表, 不支持随机访问(可以反向迭代), 任意位置操作(增/删)高效, 插入时分配空间, 迭代器不易失效
- deque：双向开口的分段连续线性空间，可以在头尾端进行元素的插入和删除, 允许于常数时间内对头端进行插入或删除元素；可以增加一段新的空间, 不过迭代器设置复杂.
- queue：先进先出队列, 默认基于 deque 容器, 可以对两端进行操作，但是只能在队列头部进行移除元素，只能在队列尾部新增元素，可以访问队列尾部和头部的元素，但是不能遍历容器

- deque 和 vector 的差异
  - deque 允许于常数时间内对头端进行插入或删除元素;
  - deque 没有空间包括, 当空间不足时，deque 可以增加一段新的空间, 而不用进行整体迁移
  - vector 的迭代器是对指针的封装，deque 的迭代器相对复杂
- list 和 vector 的差异
  - vector 空间是预先分配的，list 是插入时分配的
  - vector 是连续数组, 增删操作都可能会造成内存迁移, 后续迭代器失效，list 是双向链表, 增删操作都可以在常数时间内完成, 迭代器不会失效

- 参考:
  - STL源码剖析——序列容器之deque (https://blog.csdn.net/chenhanzhun/article/details/39430973)
  - C++ list，STL list(双向链表)详解 (http://c.biancheng.net/view/351.html)
  - STL源码剖析——序列容器之deque (https://blog.csdn.net/chenhanzhun/article/details/39430973)

54、STL 中的allocator,deallocator

- 参考: C++ STL 的内存优化 (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-%E7%B1%BB%E5%92%8C%E6%95%B0%E6%8D%AE%E6%8A%BD%E8%B1%A1/#c-stl-%E7%9A%84%E5%86%85%E5%AD%98%E4%BC%98%E5%8C%96)

55、STL 中hash_map 扩容发生什么?

1) hash table 表格内的元素称为桶 (bucket),而由桶所链接的元素称为节点 (node),其中存入桶元素的容器为 STL 本身很重要的一种序列式容器—— vector 容器。之所以选择 vector 为存放桶元素的基础容器，主要是因为 vector 容器本身具有动态扩容能力， 无需人工干预。

2) 向前操作：首先尝试从目前所指的节点出发，前进一个位置(节点)，由于节点被安置于 list 内，所以利用节点的 next 指针即可轻易完成前进操作，如果目前正巧是 list 的尾端，就跳至下一个 bucket 身上，那正是指向下一个 list 的头部节点。

- hash table 表格内的元素称为桶 (bucket),而由桶所链接的元素称为节点 (node), 由线性表来储存所有的桶, 其底层实现为 vector , 因为它支持随机访问, 和动态扩容
- 哈表表键值发生碰撞的概率和负载因子正相关, 当负载因子过大, 哈希表的性能显著降低, 一般负载因子大于阈值(0.75)则对哈希表进行扩容, 然后通过rehash对所有节点进行重映射, 注意扩容并不会增加哈希表插入删除的复杂度, 但是rehash本身的的时间复杂度

为n, 所以对高时效性的需求下, 要注意.

- 参考:
  - [对 c++ unordered_map 源码的解析 | ZRJ](https://zrj.me/archives/1248) - Hash table详解 (http://zheming.wang/blog/2014/06/17/05E21D24-A791-4D97-993D-98B7E6C88BC2/)
  - HASH TABLE::DYNAMIC RESIZING (Java, C++) (http://www.algolist.net/Data_structures/Hash_table/Dynamic_resizing)
[对 c++ unordered_map 源码的解析	ZRJ](https://zrj.me/archives/1248)

56、map 如何创建?

- `vector` 底层数据结构为数组，支持快速随机访问
- `list` 底层数据结构为双向链表，支持快速增删
- `deque` 底层数据结构为一个中央控制器和多个缓冲区，详细见 `STL` 源码剖析 `P146`，支持首尾(中间不能)快速增删，也支持随机访问，`deque` 是一个双端队列 `(double-ended queue)`，也是在堆中保存内容的.它的保存形式 如下: `[堆1] --> [堆2] -->[堆3] --> ...`, 每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是list 和vector 的结合品.
- `stack` 底层一般用 `list` 或 `deque` 实现，封闭头部即可，不用 `vector` 的原因应该是容量大小有限制，扩容耗时
- `queue` 底层一般用 `list` 或 `deque` 实现，封闭头部即可，不用 `vector` 的原因应该是容量大小有限制，扩容耗时 `(stack` 和 `queue` 其实是适配器,而不叫容器，因为是对容器的再封装)
- `priority_queue` 的底层数据结构一般为 `vector` 为底层容器，堆 `heap` 为处理规则来管理底层容器实现
- `set` 底层数据结构为红黑树，有序，不重复
- `multiset` 底层数据结构为红黑树，有序，可重复
- `map` 底层数据结构为红黑树，有序，不重复
- `multimap` 底层数据结构为红黑树，有序，可重复
- `hash_set` 底层数据结构为 `hash` 表，无序，不重复
- `hash_multiset` 底层数据结构为 `hash` 表，无序，可重复

- `hash_map` 底层数据结构为 `hash` 表，无序，不重复
- `hash_multimap` 底层数据结构为 `hash` 表，无序，可重复

- 红黑树的性质：
  - 每个节点或是红色的，或是黑色的。
  - 根节点是黑色的。
  - 每个叶节点 (`NULL`) 是黑色的。
  - 如果一个节点是红色的，则它的两个孩子节点都是黑色的。
  - 对每个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点。
- 数据结构——红黑树(RB-Tree) (https://blog.csdn.net/chenhanzhun/article/details/38405041)

57、vector 的增加删除都是怎么做的？为什么是1.5 倍?

- 参考: 42、STL vector的实现，删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间? (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-C++%E5%9F%BA%E7%A1%80%E5%86%8D%E6%8E%A2/#42stl-vector%E7%9A%84%E5%AE%9E%E7%8E%B0%E5%88%A0%E9%99%A4%E5%85%B6%E4%B8%AD%E7%9A%84%E5%85%

---

1) 新增元素：vector 通过一个连续的数组存放元素，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； 2) 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； 3) 初始时刻vector 的capacity 为0，塞入第一个元素后capacity 增加为1； 4) 不同的编译器实现的扩容方式不一样，VS2015 中以

1.5 倍扩容，GCC 以2 倍扩容。对比可以发现采用采用成倍方式扩容，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，因此，使用成倍的方式扩容。

```
1) 考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以2 二倍的方式扩容，或者以1.5 倍的方式扩容。
2) 以2 倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间： 3) 向量
```

58、函数指针?

- 什么是函数指针?
  - 函数指针本质是一个指针, 它指向的是函数的入口地址, 它的类型是由函数的参数列表和返回值共同确定.
- 函数指针的声明方法
  - `int (*pf)(const int&, const int&); (1)`
  - `pf` 是一个返回类型为int, 参数为两个const int&的函数。**注意*pf 两边的括号是必须的**
    - 否则上面的定义就变成了:
    - `int *pf(const int&, const int&); // 这声明了一个函数pf, 其返回类型为int *, 带有两个const int&参数.`
- 为什么有函数指针
  - 可以通过函数指针进行函数调用
  - 而且函数指针本质是一个指针, 可以把它指向返回值类型和形参列表相同的不同函数
  - 另外还能将函数指针作为函数参数进行传递.
  - 通过函数指针可以把函数的调用者与被调函数分开。
    - 调用者只需要确定被调函数是一个具有特定参数列表和特定返回值的函数,
    - 而不需要知道具体是哪个函数被调用.
- 两种方法赋值:
  - 指针名 = 函数名
  - 指针名 = &函数名

59、说说你对c 和c++的看法，c 和c++的区别?

- 面向过程 / 面向对象
- C中的函数编译时不会保留形参列表, 也不能重载; 而C++中的函数在编译时会保留形参列表, 有重载
- struct
  - C中: struct是自定义数据类型; 是变量的集合, 不能添加拥有成员函数; 没有访问权限控制的概念; 结构体名称不能作为参数类型使用, 必须在其前加上struct才能作为参数类型
  - C++中: struct是抽象数据类型, 是一个特殊的类, 可以有成员函数, 默认访问权限和继承权限都是public, 结构体名可以作为参数类型使用
- 动态管理内存的方法不一样: `malloc/free` 和 `new/delete`
- C语言没有引用的概念, 更没有左值引用, 右值引用
- C语言不允许只读数据(const修饰)用作下标定义数组, C++允许
- C语言的局部静态变量初始化发生于编译时，所以在函数中不能使用变量对局部静态变量进行初始化， 而C++因为增加了对象的概念，而对象需要调用构造函数进行初始化，所以编译器将局部静态变量的初始化推迟至该变量使用之前，也就是说可以使用变量来初始化局部静态变量。
- C++相比C，增加多许多类型安全的功能，比如强制类型转换
- C++支持范式编程，比如模板类、函数模板等

- `PS`：C/C++的全局变量默认连接属性都是 `extern` 的啊, 参考:C语言：链接属性与存储类型 (https://blog.csdn.net/sinat_27706697/article/details/47679329)
- 参考:
  - C语言：链接属性与存储类型 (https://blog.csdn.net/sinat_27706697/article/details/47679329)
  - 为什么用C语言中const常量定义数组大小会报错？ (https://blog.csdn.net/weixin_43054397/article/details/90417740)
  - 15、C 语言struct 和C++ struct 区别 (https://blog.csdn.net/zzxiaozhao/article/details/103188945#15_C_struct_C_struct__228)

60、c/c++的内存分配，详细说一下栈、堆、静态存储区?

- 栈区（stack）— 由编译器自动分配释放，存放函数的参数值，局部变量的值等其操作方式类似于数据结构中的栈。
- 堆区（heap） — 一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS (操作系统)回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
- 全局区(静态区)（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。
- 文字常量区—常量字符串就是放在这里的。程序结束后由系统释放。
- 程序代码区—存放函数体的二进制代码。

- 参考: C++/C的内存分配 (https://blog.csdn.net/zzxiaozhao/article/details/102943714#CC_116)

61、堆与栈的区别?

- 管理方式: 栈由编译器自动管理，无需我们手工控制；堆需要手动释放不再使用的堆空间 `memory leak` 。
- 空间大小:
  - 32 位系统下, 堆内存可以达到 4G (3G 用户空间，1G 内核空间).
  - 栈空间是受限的, 默认大小为 1M
- 碎片问题:
  - 对于堆来说，频繁的 `new/delete` 势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。
  - 对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，永远都不可能有一个内存块从栈中间弹出
- 生长方向:
  - 对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向;

    - 对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。
- 分配方式：
    - 堆都是动态分配的，没有静态分配的堆。
    - 栈有2种分配方式：静态分配和动态分配。
        - 静态分配是编译器完成的，比如局部变量的分配。
        - 动态分配由alloca ((https://baike.baidu.com/item/alloca/7621487?fr=aladdin))函数进行分配，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放，无需我们手工实现。
- 分配效率：
    - 栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。
    - 堆则是 C/C++ 函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法(具体的算法可以参考数据结构 / 操作系统)在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间(可能是由于内存碎片太多)，就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

- 参考: 动态栈: alloca_百度百科 (https://baike.baidu.com/item/alloca/7621487?fr=aladdin)

62、野指针是什么？如何检测内存泄漏?

- 野指针：指向内存被释放的内存或者没有访问权限的内存的指针。
- “野指针”的成因主要有 3 种：
    - **指针变量没有被初始化。**任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的。
    - **指针被 free 或者 delete 之后，没有置为 NULL ；**
    - **指针操作超越了变量的作用范围。**
- 如何避免野指针：

    - 对指针进行初始化, 或指向有效地址空间
    - 指针用完后释放内存，将指针赋 NULL 。
        - `char * p = NULL;`
        - `char * p = (char * )malloc(sizeof(char));`
        - `char num[ 30] = {0}; char *p = num;`
        - `delete(p); p = NULL;`

- 参考: 野指针和悬空指针 (https://blog.csdn.net/bqxdrs012/article/details/78531357)

63、悬空指针和野指针有什么区别?

- 野指针：野指针指，访问一个已删除或访问受限的内存区域的指针，野指针不能判断是否为 NULL 来避免。指针没有初始化，释放后没有置空，越界
- 悬空指针：一个指针的指向对象已被删除，那么就成了悬空指针。野指针是那些未初始化的指针。

- 参考: 野指针和悬空指针 (https://blog.csdn.net/bqxdrs012/article/details/78531357)

64、内存泄漏

- 内存泄漏
    - 内存泄漏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, 导致此段内存不能被使用；
- 后果
    - 只发生一次小的内存泄漏可能不被注意，但泄漏大量内存的程序将会出现各种证照：**性能下降到内存逐渐用完，导致另一个程序失败；**
- 如何排除
    - 使用工具软件 BoundsChecker ， BoundsChecker 是一个运行时错误检测工具，它主要定位程序运行时期发生的各种错误；调试运行 DEBUG 版程序，运用以下技术： CRT(C run-time libraries) 、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境 OUTPUT 窗口)，综合分析内存泄漏的原因，排除内存泄漏。
- 解决方法
    - 智能指针。
- 检查、定位内存泄漏
    - 检查方法：在 main 函数最后面一行，加上一句 _CrtDumpMemoryLeaks() 。调试程序，自然关闭程序让其退出，查看输出：输出这样的格式 {453} normal block at 0x02432CA8 , 868 bytes long 被 {} 包围的 453 就是我们需要的内存泄漏定位值， 868 bytes long 就是说这个地方有 868 比特内存没有释放。
- 定位代码位置
    - 在 main 函数第一行加上 _CrtSetBreakAlloc(453); 意思就是在申请 453 这块内存的位置中断。然后调试程序，程序中断了，查看调用堆栈。加上头文件 #include <crtdbg.h>

- 参考:
    - BoundsChecker使用 (https://www.cnblogs.com/hrhguanli/p/3890171.html)
    - (转)内存管理：_CrtDumpMemoryLeaks和_CrtSetBreakAlloc (https://www.cnblogs.com/jianqiang2010/archive/2010/12/02/1894327.html)
    - Linux 内存泄露检查工具valgrind简析 (https://blog.csdn.net/fanyun_01/article/details/65938998)

65、 new和malloc的区别?

- 参考: malloc和new的区别 (https://blog.csdn.net/zzxiaozhao/article/details/102604626#mallocnew_430)

66、 delete p;与delete[]p， allocator

- 动态数组管理 new 一个数组时， [] 中必须是一个整数，但是不一定是常量整数，普通数组必须是一个常量整数;
- new 动态数组返回的并不是数组类型，而是一个元素类型的指针;
- delete[] 时，数组中的元素按逆序的顺序进行销毁;
- new 在内存分配上面有一些局限性， new 的机制是将内存分配和对象构造组合在一起，同样的， delete 也是将对象析构和内存释放组合在一起的。
- allocator 将这两部分分开进行， allocator 申请一部分内存，不进行初始化对象，只有当需要的时候才进行初始化操作。
- 参考下一个问题: 67、new和delete的实现原理，delete是如何知道释放内存的大小的额?

67、 new和delete的实现原理， delete是如何知道释放内存的大小的额?

- new
  - 简单类型直接调用 operator new 分配内存;
  - 对于复杂结构，先调用 operator new 分配内存，然后在分配的内存上调用构造函数;
- delete
  - 简单数据类型默认只是调用 free 函数;
  - 复杂数据类型先调用析构函数再调用 operator delete ;

- new[]
  - 对于简单类型， new[] 计算好大小后调用 operator new；
  - 对于复杂数据结构
    - AA* P = new AA[10];
    - new[] 先调用 operator new[] 分配内存, **分配内存时多分配四个字节用于存放元素个数.**, 返回地址为 p
    - p 的最开始的 4 个字节用于存放元素个数 n , 然后从调用 n 次构造函数从 p-4 开始构造对象.
    - 返回地址,也就是 P , 即为 p-4
- delete[]
  - 对于简单类型, 直接调用 free 进行释放(注意简单类型并没有利用 4 个字节保存元素个数, 由编译器自行优化)
  - 对于复制类型,
    - 首先将指针前移 4 个字节获得元素个数 n , 然后执行 n 次析构函数, 最后并释放掉内存.
    - 因为指针指向的是 p-4 并不是内存的起始地址, 所以使用 delete 将无法完成释放, 因为 free 需要通过起始地址进行释放, 而 p-4 不是起始地址

- 参考: **一定要看看:**深入理解C++ new/delete, new/delete 动态内存管理 (https://imgconvert.csdnimg.cn/aHR0cHM6Ly93d3cuY25ibG9ncy5jb20vdHAtMTZiL3AvODY4NDI5OC5odG1s?x-oss-process=image/format,png)

68、 malloc 申请的存储空间能用 delete 释放吗

- 不能
  - malloc /free 主要为了兼容 C， new 和 delete 完全可以取代 malloc /free 的。
  - malloc /free 的操作对象都是必须明确大小的。而且不能用在动态类上。
  - new 和 delete 会自动进行类型检查和大小， malloc/free 不能执行构造函数与析构函数，所以动态对象它是不行的。

- 当然从理论上说使用 `malloc` 申请的内存是可以通过 `delete` 释放的。不过一般不这样写的。而且也不能保证每个 `C++` 的运行时都能正常。

69、malloc 与free 的实现原理?

- 参考: malloc的原理，另外brk系统调用和mmap系统调用的作用分别是什么? (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-%E7%B1%BB%E5%92%8C%E6%95%B0%E6%8D%AE%E6%8A%BD%E8%B1%A1/#malloc%E7%9A%84%E5%8E%9F%E7%9

70、 malloc、 realloc、 calloc、 alloca的区别

- `malloc` 函数: 在堆上申请空间, 随机初始化
  - `void* malloc(unsigned int num_size);`
  - `int *p = malloc(20*sizeof(int)); // 申请20 个int 类型的空间；`
- `calloc` 函数: 省去了人为空间计算； `malloc` 申请的空间的值是随机初始化的， `calloc` 申请的空间的值是初始化为 `0` 的;
  - `void* calloc(size_t n,size_t size);`
  - `int *p = calloc(20, sizeof(int));`
- `realloc` 函数: 给动态分配的空间分配额外的空间，用于扩充容量。(可能会导致内存迁移)
  - `void realloc(void *p, size_t new_size);`
- `alloca` 函数: `_alloca` 是在**栈（stack）**上申请空间,该变量离开其作用域之后被自动释放，无需手动调用释放函数。

71、 __stdcall 和__cdecl 的区别?

- 在进行函数调用的过程中, 参数入栈肯定是调用者干的事, 但是参数出栈, 可以由调用者干, 也可以由被调函数干; 所以就需要对函数调用者和被调函数之间责任进行划分，`stdcall` 和 `cdecl` 正是两种划分方式
- `cdecl`：

    - 是 c语言 的默认定义, 它规定了由调用者负责回复堆栈,
    - 好处: 参数数量可以是任意多个
    - 缺点: 代码存在冗余, 例如100次调用, 就会右100段回复堆栈的代码
- `stdcall`：
    - 一般用于跨语言的协作, 例如系统调用, 都会使用这种方式, 它规定堆栈的恢复由被调函数负责
    - 好处: 不会存在代码冗余, 100次低调用, 只有一段恢复堆栈的代码
    - 缺点: 只能允许规定的参数个数, 无法实现不定参数个数的调用

- `__stdcall`
    - `__stdcall` 是被函数恢复堆栈，只有在函数代码的结尾出现一次恢复堆栈的代码；
    - 在编译时就规定了参数个数，无法实现不定个数的参数调用；
- `__cdecl`
    - `__cdecl` 是调用者恢复堆栈，假设有 `100` 个函数调用函数 `a` ，那么内存中就有 `100` 端恢复堆栈的代码；
    - 可以不定参数个数；
    - 每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用 `__stacall` 函数大。

- 参考: __stdcall与__cdecl的区别 (https://blog.csdn.net/myjisgreat/article/details/46477769)

72、使用智能指针管理内存资源，RAII

- `RAII` 全称是“`Resource Acquisition is Initialization`”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。
- 编译器保证, 栈对象在创建时自动调用构造函数，在超出作用域时自动调用析构函数。

- 所以 RAII 的思想下, 我们使用一个**栈对象**来管理资源, 将资源和对象的生命周期绑定。
- 智能指针 (std::shared_ptr 和 std::unique_ptr) 即 RAII 最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记 delete 造成的内存泄漏。毫不夸张的来讲，有了智能指针，代码中几乎不需要再出现 delete 了。

73、手写实现智能指针类

- 参考: 说一下shared_ptr的实现 (https://blog.csdn.net/zzxiaozhao/article/details/102604626#shared_ptr_257)

- 计数器: 取计数器,
- 指针相关: 取原始指针
- 运算符重载: ++ , -- , -> , + , - , * , =
- 构造函数: 更新计数器
- 复制构造函数: 更新计数器
- 移动构造函数: 计数器不变
- 析构函数: 更新计数器, 按条件释放内存
- 智能指针是一个数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。
- 它会自动记录SmartPointer<T*>对象的引用计数，一旦T 类型对象的引用计数为0，就释放该对象。
- 除了指针对象外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为1，需要一个构造函数。
- 新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存。
- 通过覆写赋值运算符，才能将一个旧的智能指针赋值给另一个指针，同时旧的引用计数减1，新的引用计数加1
- 一个构造函数、拷贝构造函数、复制构造函数、析构函数、移走函数;

74、内存对齐？位域?

- 字节对齐的原因:

    - 更快: 如果数据未对齐自然边界, 则处理器需要两次寻址才能得到完整的数据
    - 通用: 部分硬件平面不支持访问未对齐的数据, 会抛出硬件异常
- 具体操作
    - 自定义对齐系数
        - 可以通过预编译命令 `#pragma pack(n)， n=1，2，4，8，16` 来改变这一系数，其中的 `n` 就是指定的“对齐系数”
    - 数据成员对齐规则:
        - 结构 `(struct)(` 或联合 `(union)`)的数据成员，第一个数据成员放在 `offset` 为 `0` 的地方，以后每个数据成员的对齐按照 `#pragma pack` 指定的数值和这个数据成员自身长度中，比较小的那个进行。
        - 结构体作为成员：
            - 如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储。
    - 结构(或联合)的整体对齐规则：
        - 在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将按照 `#pragma pack` 指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。
- 位域
    - 有些信息在存储时，并不需要占用一个完整的字节， 而只需占几个或一个二进制位。
    - C 语言 又提供了一种数据结构，称为“位域”或“位段”。
    - 所谓“位域”是把一个字节中的二进位划分为几 个不同的区域，并说明每个区域的位数。
    - 位段成员必须声明为 `int`、 `unsigned int` 或 `signed int` 类型 `(short char long)`。 ```c struct 位域结构名{ 位域列表 // 其中位域列表的形式为： 类型说明符 位域名：位域长度 };

struct bs {

int a:8; int b:2; int c:6; };

```

---

- 参考:
```

- 操作系统中的结构体对齐，字节对齐 (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E4%I
- 位域的定义和使用 (https://blog.csdn.net/sty124578/article/details/79456405)

75、结构体变量比较是否相等

- 重载了 == 操作符

```
struct foo {
    int a;
    int b;
    bool operator==(const foo& rhs) { // 操作运算符重载
        return( a == rhs.a) && (b == rhs.b);
    }
};
```

- 元素的话，一个个比；
- 指针直接比较，如果保存的是同一个实例地址，则(p1==p2)为真；

76、位运算

- 若一个数m 满足m = 2^n;那么k%m=k&(m-1)

- 判断奇偶
  - `a&1 == 0; // 偶数`
  - `a&1 == 1; // 奇数`

- int 型变量循环左移 k 次，即 a=a<<k|a>>16-k （设sizeof(int)=16)
- int 型变量 a 循环右移 k 次，即 a=a>>k|a<<16-k （设sizeof(int)=16)
- 整数的平均值
  - 对于两个整数 x , y ， 如果用 (x+y)/2 求平均值，会产生溢出，因为 x+y 可能会大于 INT_MAX ， 但是我们知道它们的平均值是肯定不会溢出的，我们用如下算法：

```
int average(int x, int y) { //返回X,Y 的平均值
     return (x&y)+((x^y)>>1);
}
```

- 判断一个整数是不是 2 的幂,对于一个数 x >= 0 ， 判断他是不是 2 的幂

```
boolean power2(int x){
    return ((x&(x-1))==0)&&(x!=0)；
}
```

- 不用 temp 交换两个整数

```
void swap(int x, int y) {
    x ^= y;
    y ^= x;
    x ^= y;
}
```

- 计算绝对值

```
int abs(int x) {
    int y ;
    y = x >> 31 ;
    return (x^y)-y; //or: (x+y)^y
}
```

- 取模运算转化成位运算 (在不产生溢出的情况下)
  - a % (2^n) 等价于 a & (2^n - 1)
  - a % 2 等价于 a & 1
- 乘法运算转化成位运算 (在不产生溢出的情况下)

    - `a * (2^n)` 等价于 `a<< n`
- 除法运算转化成位运算 (在不产生溢出的情况下)
    - `a / (2^n)` 等价于 `a>> n`
    - 例: `12/8 == 12>>3`
- `if (x == a) x= b; else x= a;` 等价于 `x= a ^ b ^ x;`
- `x` 的相反数表示为 `(~x+1)`

- 参考: 位运算总结 取模 取余 (https://blog.csdn.net/black_ox/article/details/46411997)

77、为什么内存对齐

- 平台原因(移植原因)
    - 不是所有的硬件平台都能访问任意地址上的任意数据的;
    - 某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异
- 性能原因:
    - 数据结构(尤其是栈)应该尽可能地在自然边界上对齐。
    - 原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

78、函数调用过程栈的变化，返回值和参数变量哪个先入栈?

- 调用者函数把被调函数所需要的参数按照与被调函数的形参顺序相反的顺序压入栈中,即:从右向左依次把被调函数所需要的参数压入栈;
- 调用者函数使用 `call` 指令调用被调函数,并把 `call` 指令的下一条指令的地址当成返回地址压入栈中(这个压栈操作隐含在 `call` 指令中);

- 在被调函数中,被调函数会先保存调用者函数的栈底地址（push ebp）,然后再保存调用者函数的栈顶地址,即:当前被调函数的栈底地址（mov ebp,esp）;
- 在被调函数中,从 ebp 的位置处开始存放被调函数中的局部变量和临时变量,并且这些变量的地址按照定义时的顺序依次减小,即:这些变量的地址是按照栈的延伸方向排列的,先定义的变量先入栈,后定义的变量后入栈;
- 关于返回值:
  - 如果 返回值 <= 4字节，则返回值通过寄存器 eax 带回。
  - 如果 4< 返回值 <=8字节，则返回值通过两个寄存器 eax 和 edx 带回。
  - 如果 返回值 > 8字节，则返回值通过产生的临时量带回。

- 参考: C语言是怎么进行函数调用的? (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-C++%E5%9F%BA%E7%A1%80/#c%E8%AF%AD%E8%A8%80%E6%98%AF%E6%80%8E%E4%B9%88%E8%BF%9B%E8%A1
  - 参数逆序入栈
  - 返回地址入栈
  - 调用函数栈顶入栈
  - 被调用函数栈底入栈
  - 局部变量入栈

79、怎样判断两个浮点数是否相等?

- 对两个浮点数判断大小和是否相等不能直接用 == 来判断，会出错!
- 明明相等的两个数比较反而是不相等!
- 对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值!
- 浮点数与 0 的比较也应该注意。与浮点数的表示方式有关。

- `fabs(a-b)<=1.0e-9`

- 参考: 在程序中如何判断两个浮点数相等 (https://www.cnblogs.com/liuyc/p/5933850.html)

80、宏定义一个取两个数中较大值的功能

- `#define MAX(x,y)((x>y?)x:y)`

81、`define`、`const`、`typedef`、`inline`使用方法?

- `const` 与 `#define` 的区别: 作用阶段不同，功能不同，`define`作用丰富，占用的空间不同，作用域
  - 作用阶段不同: `const` 在编译和链接阶段其作用，`define` 在预编译阶段起作用
  - 功能不同:
    - `const` 是定义一个变量, 拥有数据类型, 会进行语义语法检查
    - `define` 是宏定义, 简单的问题替代, 没有类型检查
  - `define` 的作用更丰富: `define` 可以配合条件预编译指令, 完成特殊的逻辑, 例如防止重复引用文件
  - 编译后占用的空间: `const` 定义的是变量, 会储存在数据段空间，`define` 是宏替换, 其值会储存在代码段
  - 作用域不同: `define` 没有作用域限制, 而const定义的变量通常有作用域的限制(全局变量默认为extern)
- `#define` 和别名 `typedef` 的区别 1) 执行时间不同， `typedef` 在编译阶段有效， `typedef` 有类型检查的功能；# `define` 是宏定义，发生在预处理阶段，不进行类型检查； 1) 功能差异， `typedef` 用来定义类型的别名，定义与平台无关的数据类型，与 `struct` 的结合使用等。# `define` 不只是可以为类型取别名，还可以定义常量、变量、编译开关等。 1) 作用域不同， `#define` 没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。而 `typedef` 有自己的作用域。

- `define` 与 `inline` 的区别 1) `#define` 是关键字， `inline` 是函数； 1) 宏定义在预处理阶段进行文本替换， `inline` 函数在编译阶段进行替换； 1) `inline` 函数有类型检查，相比宏定义比较安全；

82、 `printf` 实现原理?

- 函数的调用过程: 参数逆序入栈, 返回地址入栈, 调用函数栈顶入栈, 设置被调函数栈底, 然后是被调函数的局部变量
- 在调用 `printf` 时, 首先获取第一个形参, 也就是字符指针, 然后解析所指向的字符串, 得到后续参数的个数和数据类型,
- 然后计算出偏移量, 并从当前函数栈的栈底往上偏移得到
- `printf("%d,%d",a,b);`

83、 `#include` 的顺序以及尖叫括号和双引号的区别

- 路径不同, 参考: include头文件的顺序以及双引号""和尖括号<>的区别? (https://blog.csdn.net/zzxiaozhao/article/details/102943714#include_77)

84、lambda 函数

- 包括五大部分: 捕获列表, 参数列表, 修饰符, 返回类型, 函数体
- 捕获列表: 对参数的捕获, 捕获方式为 值传递(`[=]`, `[val]`) 和 引用(`[&]`, `[&val]`)
- 参数列表: 参数列表, 和不同函数一样, 如果没有可以省略
- 修饰符: 默认情况下lambda函数总是一个const函数，Mutable可以取消其常量性。在使用该修饰符时，参数列表不可省略。
- 返回类型
- 函数体: 除了可以使用参数外, 还可以使用捕获的参数

- **参考:** C++11中的可变参数模板、右值引用和lambda (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-%E7%B1%BB%E5%92%8C%E6%95%B0%E6%8D%AE%E6%8A%BD%E8%B1%A1/#c11%E4%B8%AD%E7%9A%84%E5%8F

85、hello world 程序开始到打印到屏幕上的全过程?

- 应用程序
- 应用程序载入内存变成进程
- 进程获取系统的标准输出接口
- 系统为进程分配CPU
- 触发缺页中断
- 通过puts系统调用, 往标准输出接口上写字符串
- 操作系统将字符串发送到显示器驱动上
- 驱动判断该操作的合法性, 然后将该操作变成像素, 写入到显示器的储存映射区
- 硬件将该像素值改变转变成控制信号控制显示器显示

- 用户告诉操作系统执行 `HelloWorld` 程序(通过键盘输入等)
- 操作系统：找到 `helloworld` 程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址。
- 操作系统：创建一个新进程，将 `HelloWorld` 可执行文件映射到该进程结构，表示由该进程执行 `helloworld` 程序。
- 操作系统：为 `helloworld` 程序设置 `cpu` 上下文环境，并跳到程序开始处。
- 执行 `helloworld` 程序的第一条指令，发生缺页异常
- 操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行 `helloworld` 程序
- `helloword` 程序执行 `puts` 函数(系统调用)，在显示器上写一字符串

- 操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程
- 操作系统：控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区
- 视频硬件将像素转换成显示器可接收和一组控制数据信号
- 显示器解释信号，激发液晶屏
- OK，我们在屏幕上看到了 HelloWorld

86、模板类和模板函数的区别是什么?

- 函数模板的实例化是由编译程序在处理函数调用时自动完成的
- 类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。在使用时类模板必须加，而函数模板不必

87、为什么模板类一般都是放在一个h 文件中

- 编译器并不是把函数模板处理成能够处理**任意类型**的函数；编译器从**函数模板**通过**具体类型**==产生==不同的**函数**;
- 编译器会对函数模板进行两次编译:
  - 在声明的地方对模板代码本身进行编译,
  - 在调用的地方对参数替换后的代码进行编译。
- 如果模板函数不是定义在 .h 文件中
  - 编译器编译 .cpp 文件时并不知道另一个 .cpp 文件的存在, 也不会去查找(查找通常是链接阶段的事)
    - 在定义模板函数的 .cpp 文件中, 编译器对函数模板进行了第一次编译, 但是它并没有发现任何调用, 故而没有生产任何的函数实例
    - 在调用了模板函数的 .cpp 文件中, 编译器发现调用其他函数, 但是在此 .cpp 文件中并没有定义, 所以将此次调用处理为外部连接符号, 期望链接阶段由连接器给出被调函数的函数地址.
  - 在链接阶段, 连接器找不到被调函数故而报不能识别的外部链接错误.

- 模板定义很特殊。
  - 由 `template<…>` 处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。
  - 在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。
  - 所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。
- 在分离式编译的环境下
  - 编译器编译某一个 `.cpp` 文件时并不知道另一个 `.cpp` 文件的存在，也不会去查找(当遇到未决符号时它会寄希望于连接器)。
  - 这种模式在没有模板的情况下运行良好，但遇到模板时就傻眼了，因为模板仅在需要的时候才会实例化出来，所以，当编译器只看到模板的声明时，它不能实例化该模板，只能创建一个具有外部连接的符号并期待连接器能够将符号的地址决议出来。
  - 然而当实现该模板的 `.cpp` 文件中没有用到模板的实例时，编译器懒得去实例化，所以，整个工程的 `.obj` 中就找不到一行模板实例的二进制代码，于是连接器也黔驴技穷了。

88、C++中类成员的访问权限和继承权限问题。

- 参考: C++中类成员的访问权限 (https://zhaostu4.github.io/2019/11/28/%E9%9D%A2%E7%BB%8F%E6%B1%87%E6%80%BB-%E7%B1%BB%E5%92%8C%E6%95%B0%E6%8D%AE%E6%8A%BD%E8%B1%A1/#c%E4%B8%AD%E7%B1%BB%E6%88%9

89、cout 和printf 有什么区别?

- `cout` 有缓冲区，`printf` 无缓冲区; 有缓冲区意味着
  - 操作系统可以待用户刷新缓冲区时输出, 或则缓冲区存满的时候输出,
  - 如果操作系统空闲的话也会检查缓冲区是否有值, 如果有的话立即输出.
  - `endl` 相当于输出回车后，再强迫缓冲输出。

    - `flush` 立即强迫缓冲输出。
- `cout <<` 是一个函数, 它对常见数据类型进行了重载, 所以能自动识别数据的类型并进行输出.

90、重载运算符?

- 引入运算符重载，是为了实现类的多态性；
- 只能重载已有的运算符；对于一个重载的运算符，其 优先级 和 结合律 与内置类型一致才可以；不能改变运算符操作数个数；
- `.`，`：`，`？`，`sizeof`，`typeid` `**` 不能重载；
- 两种重载方式，成员运算符和非成员运算符，成员运算符比非成员运算符少一个参数；下标运算符、箭头运算符(重载的箭头运算符必须返回类的指针)、解引用运算符必须是成员运算符；
- 当重载的运算符是成员函数时， `this` 绑定到左侧运算符对象。成员运算符函数的参数数量比运算符对象的数量少一个；至少含有一个类类型的参数；
- 下标运算符必须是成员函数，下标运算符通常以所访问元素的引用作为返回值，同时最好定义下标运算符的常量版本和非常量版本；
- 当运算符既是一元运算符又是二元运算符(`+`， `-`， `*`， `&`)，从参数的个数推断到底定义的是哪种运算符；

91、函数重载函数匹配原则

- 首先进行名字查找, 确定候选函数
- 然后按照以下顺序进行匹配:
    - 精确匹配：参数匹配而不做转换，或者只是做微不足道的转换，如数组名到指针、函数名到指向函数的指针、 `T` 到 `const T`；
    - 提升匹配：即整数提升(如 `bool` 到 `int` 、 `char` 到 `int` 、 `short` 到 `int` 、 `float` 到 `double`)，；
    - 使用标准转换匹配：如 `int` 到 `double` 、 `double` 到 `int` 、 `double` 到 `long double` 、 `Derived*` 到 `Base*` 、 `T*` 到 `void*` 、 `int` 到 `unsigned int`；
    - 使用用户自定义匹配；

- 使用省略号匹配：类似于 `printf` 中省略号参数。

- 参考: 重载函数的调用匹配规则 (https://www.cnblogs.com/bonelee/p/5951718.html)

92、定义和声明的区别

- 如果是指变量的声明和定义
  - 从编译原理上来说，
    - 变量声明是仅仅告诉编译器，有个某类型的变量会被使用，但是编译器并不会为它分配任何内存。
    - 变量定义就是分配了内存。
- 如果是指函数的声明和定义
  - 函数声明：一般在头文件里，对编译器说：这里我有一个函数叫 `function()` 让编译器知道这个函数的存在。
  - 函数定义：一般在源文件里，具体就是函数的实现过程写明函数体。

93、 C++ 类型转换有四种

- `const_cast`：
  - 用来移除 `const` 或 `volatile` 属性。但需要特别注意的是 `const_cast` 不是用于去除变量的常量性，而是去除**指向常数对象的指针或引用**的常量性，其去除常量性的对象必须为**指针或引用。**
  - 如果对一个指向常量的指针,通过 `const_cast` 移除 `const` 属性, 然后进行修改, 编译通过,但是运行时会报段错误
- `static_cast`：静态类型转换(不能移除 `const/volatile` 属性)是最常看到的类型转换, 几个功能.
  - **内置类型之间的转换**, 精度耗损需要有程序员把握
  - **继承体系中的上下行转换**(上行:子类转父类,安全转换; 下行:父类转子类, 不安全转换)
  - **指针类型转换**：空指针转换成目标类型的空指针，把任何类型转换成`void` 类型 。

- `dynamic_cast`：主要用在继承体系中的安全向下转型
  - 它能安全地将指向基类的 `指针/引用` 转型为指向子类的 `指针/引用`，转型失败会返回 `null` (转型对象为指针时)或抛出异常 `bad_cast` (转型对象为引用时)。
  - `dynamic_cast` 会利用运行时的信息（`RTTI`）来进行动态类型检查，因此dynamic_cast 存在一定的效率损失。
  - 而且 `dynamic_cast` 进行动态类型检查时, 利用了虚表中的信息, 所以只能用于函数虚函数的类对象中.
- `reinterpret_cast` 强制类型转换,**非常不安全**
  - 它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针(先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值)。

- 参考:
  - C++开发必看 四种强制类型转换的总结 (https://www.cnblogs.com/lidabo/p/3651049.html)
  - C++系列总结——volatile关键字 (https://www.cnblogs.com/yizui/archive/2019/03/30/10628020.html)

94、全局变量和static 变量的区别

- static变量分为两个类型: 全局静态变量(在全局变量的类型前加上static)和局部静态变量(在局部变量的类型前加上static).
- 从储存形式看: 他们没有区别, 都储存于静态数据区
- 从作用域看:
  - 全局变量默认具有extern属性, 它的作用域为整个项目, 可能和其他cpp文件中的全局变量发生命名冲突.
  - 全局静态变量,作用域受限, 它的作用域仅限于定义它的文件内有效, 不会和其他cpp文件中的全局变量发生命名冲突.
  - 局部静态变量, 作用域依旧不管, 当时当离开作用域时不会变量不会被释放, 其值保持不变只是被屏蔽了, 直到再次进入作用域, 其也只会被初始化一次.

`static` 函数与普通函数有什么区别?

- `static` 函数与普通函数有什么区别?
  - `static` 函数与普通的函数作用域不同。
    - 普通函数默认为 `extern` 属性, 作用域为整个项目, 可能会和其他 `cpp` 文件中的函数发生命名冲突.
    - `static` 修饰的函数, 作用域受限仅为定义的文件, 不会和其他 `cpp` 文件中的函数发生命名冲突.

95、静态成员与普通成员的区别

- 储存位置不同: 普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；
- 声明周期不同:
  - 静态成员变量从类被加载开始到类被卸载，一直存在；
  - 普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；
- 初始化位置: 普通成员变量在类中初始化；**静态成员变量在类外初始化**；
- 拥有则不同: 静态成员变量可以理解为是属于类的变量,可以通过类名进行访问, 为本类的所有对象所共享；普通成员变量是每个对象单独享用的, 只能通过对象进行访问；

96、说一下理解 `ifdef endif`

- 从源文件到可执行程序的过程, 通常要经历: `预编译` , `编译` , `汇编` , `链接` 等过程
- `ifdef` , `endif` 为条件预编译指令, 生效于预编译阶段, 根据条件可以完成一些特殊的逻辑, **例如防止文件重复引用**
- `#ifdef` , `#else` , `#endif` 为完整的逻辑, 分别表示, 如果定义了某个标识符, 则编译后续程序段, 否则编译另外一个程序段
- 因为预编译阶段处于编译链的第一阶段, 它可以直接影响应用程序的大小.

97、隐式转换，如何消除隐式转换?

- 隐式转换，是指不需要用户干预，编译器私下进行的类型转换行为。很多时候用户可能都不知道进行了哪些转换,
  - 例如:
    - 类型提升: `(bool, int)`; `(short, int)`; `(float, double)`
    - 类型转换: `(int, float)`; `(int, double)`, `(Derived*, Base*)`
- 基本数据类型的转换, 通常发生于从小到大的变换, 以保证精度不丢失
- 对于用户自定义类型, 如果存在单参数构造函数, 或则除一个参数外其他参数都有默认参数的, 此时编译器可能完成由此参数类型到自定义类型的隐式变换, 消除方式为使用关键字 `explicit` 禁止隐式转换.

98、虚函数的内存结构，那菱形继承的虚函数内存结构呢

- 如果一个类存在虚函数, 则会发生以下几个变化
  - 如果不存在构造函数, 则编译器一定会合成默认构造函数
  - 编译器会为类生成一个虚表(储存在静态区, 不占用对象内存), 并给该类的每个对象插入一个指向虚表的指针(通常此指针位于对象的起始位置), 虚函数表的每一项为函数的入口地址.
- 如果派生类的基类存在虚函数,则
  - 编译器会复制基类的虚表形成一个副本, 然后给该派生类对象插入一个指向该虚表副本的指针
  - 如果该派生类对基类的虚函数进行了重定义, 则会替换虚表副本中的对应函数入口地址
  - 如果该派生类新增了虚函数, 则对该虚表副本增加对应的项
- 如果存在菱形结构的继承关系, 则通常回使用虚继承的方式, 防止同一类中存在基类的多个副本
  - 虚表的继承方式和普通继承一样, 但是在 - 如果不存在构造函数, 则编译器一定会合成默认构造函数
  - 如果类 `B` 虚拟继承自类 `A` , 则类 `B` 中存在一个虚基类表指针,指向一个虚基类表(储存在静态区, 不占用对象内存), 此虚基类表中存储中虚基类相对于当前类对象的偏移量.
  - 不同的编译器对虚基类表指针的处理方式不同, 例如 `VS` 编译器将虚基类表指针插入到对象中(会占用对象内存), 而 `SUN/GCC` 公式的编译器则是插入到虚函数表中(不占用对象内存)

- 参考: 给你一个类，里面有static，virtual，之类的，说一说这个类的内存分布 (https://blog.csdn.net/zzxiaozhao/article/details/102990773#staticvirtual_731)

99、多继承的优缺点，作为一个开发者怎么看待多继承

- C++ 允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。
- 优点: 对象可以调用多个基类中的接口;
- 缺点:
  - 如果基类重存在多个相同的基类或则方法, 则会出现二义性(解决方案是调用时加上全局限定符)
  - 容易存在菱形继承, 从而导致存在多个基类的副本(解决方案是使用虚拟继承)

- 个人觉得挺方便的, 虽然有缺点,但是也都用对应的解决方案

100、迭代器++it,it++哪个好，为什么

- 略

101、C++如何处理多个异常的?

- C++ 中的错误情况:
  - **语法错误(编译错误)**：比如变量未定义、括号不匹配、关键字拼写错误等等编译器在编译时能发现的错误，这类错误可以及时被编译器发现，而且可以及时知道出错的位置及原因，方便改正。

  - **运行时错误**：比如数组下标越界、系统内存不足等等。这类错误不易被程序员发现，它能通过编译且能进入运行，但运行时会出错，导致程序崩溃。为了有效处理程序运行时错误， C++ 中引入异常处理机制来解决此问题。
- C++ 异常处理机制：
  - 异常处理基本思想：执行一个函数的过程中发现异常，可以不用在本函数内立即进行处理， 而是抛出该异常，让函数的调用者直接或间接处理这个问题。
  - C++ 异常处理机制由3 个模块组成： try(检查) 、 throw(抛出) 、 catch(捕获)
  - 首先是: 抛出异常的语句格式为： throw 表达式；
  - 如果 try 块中程序段发现了异常则抛出异常, 则依次尝试通过 catch 进行捕获, 如果捕获成功则调用相应的函数处理段, 如果捕获失败, 则条用terminal终止程序.

```
try{
// 可能抛出异常的语句；(检查)
} catch(类型名[形参名]){ //捕获特定类型的异常
//处理1；
} catch(类型名[形参名]){//捕获特定类型的异常
//处理2；
} catch (…){ //捕获所有类型的异常
}
```

- C++ 标准的异常
  - std::exception：所有标准 C++ 异常的父类。
  - std::logic_error：逻辑错误(无效的参数, 太长的 std::string , 数组越界)
  - std::runtime_error：运行时错误(数据溢出)
- 我们可以通过这些类派生出自己的错误类型,尤其是对 logic_error 进行重载

- 参考: [C++ 异常处理 | 菜鸟教程](https://www.runoob.com/cplusplus/cpp-exceptions-handling.html)

102、模板和实现可不可以不写在一个文件里面？为什么?

- 参考: 87、为什么模板类一般都是放在一个h 文件中

104、智能指针的作用;

- C++11 中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露(忘记释放)，二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。
- 三个指针指针: unique_ptr 、 shared_ptr 、 weak_ptr
- unique_ptr
  - 语意为**唯一**拥有所指向对象
  - 其只支持移动语义, 不允许拷贝语义, 不允许强制剥夺, 有条件支持赋值语义(等号右边为右值的时候).
  - 当unique_ptr指针生命周期结束, 且没有被使用移动语义, 则会将所指向对象释放掉.
- shared_ptr
  - 语义为**共享**的拥有多指向的对象, 其支持拷贝语义, 支持移动语义, 支持赋值语义.
  - shared_ptr 内部存在一个计数器, 为指向该对象的所有 shared_ptr 所共享,
  - 每减少一个 shared_ptr 则计数器减一, 没多一个则计数器加一
  - 当计数器为零时则释放所指向的对象.
- weak_ptr : 解决交叉引用问题, 房子内存泄漏.

105、 auto_ptr 作用

- 已经被 unique_ptr 替代, 其允许强制剥夺所有权, 会存在野指针风险.

1) auto_ptr 的出现，主要是为了解决“有异常抛出时发生内存泄漏”的问题；抛出异常，将导致指针 p 所指向的空间得不到释放而导致内存泄漏； 2) auto_ptr 构造时取得某个对象的控制权，在析构时释放该对象。我们实际上是创建一个 auto_ptr<Type> 类型的局部对象，该局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏； 3) auto_ptr 的构造函数是 explicit ，阻止了一般指针隐式转换为 auto_ptr 的构造，所以不能直接将一般类型的指针赋值给 auto_ptr 类型的对象，必须用 auto_ptr 的构造函数创建对象； 4) 由于 auto_ptr 对象析构时会删除它所拥有的指针，所以使用时避免多个auto_ptr对象管理同一个指针； 5) Auto_ptr 内部实现，析构函数中删除对象用的是 delete 而不是 delete []，所以 auto_ptr 不能管理数组； 6) auto_ptr 支持所拥有的指针类型之间的隐式类型转换。 7) 可以通过 * 和 -> 运算符对 auto_ptr 所有用的指针进行提领操作； 8) T* get() ,获得 auto_ptr 所拥有的指针； T* release() ，释放 auto_ptr 的所有权，并将所有用的指针返回。

106、class、union、struct 的区别

- struct 在 C 和 C++ 中是不同的
  - C 语言中:
    - struct 为自定义数据类型, 结构体名不能单独作为类型使用, 其结构名前必须加 struct 才行
    - struct 为变量的集合, 不能存定义函数(但是可以存在函数指针变量)
    - struct 不存在访问权限控制的概念
  - C++ 中:
    - struct 为抽象数据类型, 只一个特殊的 class , 支持成员函数的定义, 可以继承和实现多态
    - 增加了访问权限控制的概念, 但是默认访问和继承权限为 public
    - 结构体名字可以为直接做为类型使用
- C++ 中 struct 和 class 的区别
  - 默认的访问和继承权限不同
  - 注意 C++ 中 struct 可以使用模板
- union
  - C语言 中:
    - union 是一种数据格式，能够存储不同的数据类型，但只能同时存储其中的一种类型。

        - union 的数据成员是共享内存的, 以成员最大的做为结构体的大小
        - 每个数据成员在内存中的起始地址是相同的。
    - C++ 中:
        - union 结构式一种特殊的类。 默认访问权限是 public 。
        - 能包含访问权限、成员变量、成员函数(可以包含构造函数和析构函数)。
        - 不能包含虚函数和静态数据变量。也不能被用作其他类的基类，它本身也不能从某个基类派生而来。
        - union 成员是共享内存的，以 size 最大的结构作为自己的大小。
        - 每个数据成员在内存中的起始地址是相同的。
    - 无论是 C/C++ ， union 的储存方式都是小端模式储存的

- 参考:
    - 15、C 语言struct 和C++ struct 区别 (https://blog.csdn.net/zzxiaozhao/article/details/103188945#15_C_struct_C_struct__228)
    - C++中struct和class的区别 (https://blog.csdn.net/zzxiaozhao/article/details/102943714#Cstructclass_35)
    - C++的struct可以使用template (https://blog.csdn.net/weixin_30817749/article/details/98037298)

107、 动态联编与静态联编

- 在 C++ 中，**联编是指一个计算机程序的不同部分彼此关联的过程。**按照联编所进行的**阶段**不同，可以分为**静态联编**和**动态联编**;
- **静态联编**
    - 是指联编工作在编译阶段完成的，这种联编过程是在程序运行之前完成的，又称为早期联编。
    - 要实现静态联编，在编译阶段就必须确定程序中的操作调用(如函数调用)与执行该操作代码间的关系，确定这种关系称为束定，在编译时的束定称为静态束定。
    - **静态联编**对**成员函数**的选择是**基于指向对象的指针或者引用的类型。**

    - 其优点是**效率高**，但**灵活性差。**
- **动态联编**
    - 是指联编在程序运行时动态地进行，根据当时的情况来确定调用哪个同名函数，**实际上是在运行时虚函数的实现。**这种联编又称为晚期联编，或动态束定。
    - **动态联编**对**成员函数**的选择是**基于对象的类型**，针对不同的对象类型将做出不同的编译结果。
    - C++ 中一般情况下的联编是静态联编，但是当涉及到**多态性**和**虚函数**时应该使用动态联编。
    - 动态联编的优点是**灵活性强**，但**效率低。**
    - **动态联编规定，只能通过** 指向基类的指针或基类对象的引用 **来调用虚函数**，其格式为：
        - 指向基类的指针变量名 -> 虚函数名(实参表);
        - 或基类对象的引用名 . 虚函数名(实参表)

- 实现动态联编三个条件：
    - 必须把动态联编的行为定义为**类的虚函数**；
    - **类之间应满足子类型关系**，通常表现为一个类从另一个类公有派生而来；
    - 必须先**使用基类指针指向子类型的对象**，然后直接或间接使用**基类指针调用虚函数**；

- 参考: c++动态联编与静态联编 (https://blog.csdn.net/neiloid/article/details/6934129)

108、 动态编译与静态编译

- 静态编译
    - 静态编译，编译器在编译可执行文件时，把需要用到的对应动态链接库中的部分提取出来，连接到可执行文件中去
    - 缺点: **编译慢**, **可执行程序大**

    - 优点: 使可执行文件在运行时**不需要依赖于动态链接库**;
- 动态编译
    - 动态编译的可执行文件需要附带一个动态链接库，在执行时，需要调用其对应动态链接库的命令。
    - 优点:
        - 一方面是**缩小**了执行文件本身的体积，
        - 另一方面是**加快了编译速度**，节省了系统资源。
    - 缺点:
        - 哪怕是很简单的程序，只用到了链接库的一两条命令，也需要附带一个相对庞大的链接库;
        - 二是如果其他计算机上没有安装对应的运行库，则用动态编译的可执行文件就不能运行。

- 参考: 动态编译、静态编译区别(转) (https://blog.csdn.net/weixin_38907560/article/details/81478981)

109、 动态链接和静态链接区别

- **静态链接**： 1) 函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。 1) **空间浪费**：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一个目标文件在多个程序内都存在一个副本； 1) **更新困难**：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。 1) **运行速度快**：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。
- **动态链接**： 1) 动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。 1) **共享库**：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多个副本，而是这多个程序在执行时共享同一份副本； 1) **更新方便**：更新时只

需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。 1) **性能损耗**：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

- 区别
  - 使用静态链接生成的可执行文件可能会存在共享库的多个复本, 而使用动态链接库的可执行文件只有存在一份
  - 使用静态链接库的可执行程序不需要依赖动态链接库, 依赖关系简单; 而使用动态链接库的可执行程序需要引用动态链接库, 故而依赖关系复杂
  - **静态链接生成的静态链接库不能再包含其他的动态链接库或则静态库, 而动态链接库可以包括其他的动态库或则静态库.**

- 参考: 动态编译、静态编译区别(转) (https://blog.csdn.net/weixin_38907560/article/details/81478981)

110、在不使用额外空间的情况下，交换两个数?

- 算术

```
x = x + y;
y = x - y;
x = x - y;
```

- 异或

```
// 原理 x ^= y ^= x; 能对int,char..
x = x^y;
y = x^y;
x = x^y;
```

111、strcpy 和memcpy 的区别

- 复制的内容不同。 strcpy 只能复制字符串，而 memcpy 可以复制任意内容，例如字符数组、整型、结构体、类等。
- 复制的方法不同。 strcpy 不需要指定长度，它遇到被复制字符的串结束符 <!JEKYLL@2780@1056> 才结束，所以容易溢出。 memcpy 则是根据其第 3 个参数决定复制的长度。

112、执行int main(int argc, char *argv[])时的内存结构

- 参数的含义是程序在命令行下运行的时候，需要输入 argc 个参数，每个参数是以 char 类型输入的，依次存在数组里面，数组是 argv[]，所有的参数在指针 char * 指向的内存中，数组的中元素的个数为 argc 个，第一个参数为程序的名称。

- main 函数是用户代码的入口函数, 其调用过程依旧是函数调用过程, 区别在于main函数的参数有固定的规范
  - main函数参数规范如下:
    - 第一个参数为: int 型, 表示参数的个数
    - 第二个参数为: char* 数组 , 每一个 char* 元素指向一个以字符串形式储存在内存中的参数的首地址, 其中第一个参数为程序的名字
- 函数调用过程如下:
  - 首先将参数以字符串的形式保存在内存中, 然后利用字符串起始字符指针组成char* 数组, 并计算参数的个数.
  - 然后将进行函数调用,
    - 首先, 将参数逆序入栈, 也就是(参数指针数组, 参数个数)
    - 然后返回地址入栈
    - 然后调用则栈顶入栈
    - 将当前栈顶设置为被调函数栈底, 并将栈底入栈
    - 然后被调函数建立形参以及局部变量, 处理相应的逻辑

113、 volatile 关键字的作用?

- volatile 关键字是一种类型修饰符，被它修饰的变量拥有三大特性: 易变性, 不可优化性, 顺序性
  - 易变性: 编译器对 valatile 的访问总是从内存中读取数据, 即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。
  - 不可优化性: volatile 告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。
  - 顺序性: 保证 Volatile 变量间的顺序性，编译器不会进行乱序优化。**但是可能会被CPU优化**
- 声明时语法： int volatile vInt；
- volatile 用在如下的几个地方： 1) 中断服务程序中修改的供其它程序检测的变量需要加 volatile； 2) 多任务环境下各任务间共享的标志应该加 volatile； 3) 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义；

- 参考: C/C++ Volatile关键词深度剖析 (https://www.cnblogs.com/god-of-death/p/7852394.html)

114、讲讲大端小端，如何检测(三种方法)

大端模式：是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址端。 小端模式，是指数据的高字节保存在内存的高地址中，低位字节保存在在内存的低地址端。

- 直接读取存放在内存中的十六进制数值，取低位进行值判断(在GCC中测试,不可行!)

```
int a = 0x12345678;
int *c = &a;
c[0] == 0x12 大端模式
c[0] == 0x78 小段模式
```

- 用union来进行判断(union总是小端储存)

```
union w{
    char ch;
    int i;
};
union w p;
p.i = 1;
bool flag = p.ch==1;
```

115、查看内存的方法

- 首先打开vs 编译器，创建好项目，并且将代码写进去，这里就不贴代码了，你可以随便的写个做个测试;
- 调试的时候做好相应的断点，然后点击开始调试;
- 程序调试之后会在你设置断点的地方暂停，然后选择调试->窗口->内存，就打开了内存数据查看的窗口了。

116、空类会默认添加哪些东西？怎么写?

- 默认构造函数
- 拷贝构造函数
- 析构函数
- 赋值运算符 (operator=)
- 两个取址运算符 (operator&) (const 和 非const)
- **当然所有的这些函数都是需要才生成, 例如你都没使用过复制运算, 肯定不会生成的**

- 参考:编译一个空类会默认生成哪些函数? (https://blog.csdn.net/bug07250432/article/details/10099453)

117、标准库是什么?

1) C++ 标准库可以分为两部分: - **标准函数库**: 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C语言 。 - **面向对象类库**: 这个库是类及其相关函数的集合。

- 标准函数库: 输入 / 输出 I/O 、字符串和字符处理、数学、时间、日期和本地化、动态分配、其他、宽字符函数
- 面向对象类库: 标准的 C++ I/O 类、 String 类、数值类、 STL 容器类、 STL 算法、 STL 函数对象、 STL 迭代器、 STL 分配器、本地化库、异常处理类、杂项支持库

118、 const char* 与 string 之间的关系，传递参数问题?

- string 是 c++ 标准库里面其中一个，封装了对字符串的操作，实际操作过程我们可以用 const char* 给 string 类初始化
- 三者的转化关系如下所示:
  - string 转 const char*

```
string s = “abc”;
const char* c_s = s.c_str();
```

  - const char* 转 string , 直接赋值即可

```
const char* c_s = “abc”;
string s(c_s);
```

  - string 转 char*

```
string s = “abc”;
char* c;
const int len = s.length();
c = new char[len+1];
strcpy(c,s.c_str());
```

  - char* 转 string

```
char* c = “abc”;
string s(c);
```

- const char* 转 char*

```
const char* cpc = “abc”;
char* pc = new char[strlen(cpc)+1];
strcpy(pc,cpc);
```

- char* 转 const char*，直接赋值即可

```
char* pc =
**** “abc”;
const char* cpc = pc;
```

119、new、delete、operator new、operator delete、placement new、placement delete

- new operator
  - new operator 完成了两件事情：用于**申请内存**和**初始化对象**。
  - 例如： string* ps = new string("abc")；
- operator new
  - operator new 类似于 C 语言中的 malloc，只是负责申请内存。
  - 例如：

```
void* buffer = operator new(sizeof(string)); // 注意这里new 前要有个operator。
```

- placement new
  - 用于在给定的内存中初始化对象。
  - 例如：

```
void* buffer = operator new(sizeof(string));
buffer = new(buffer) string("abc");
```

    - 调用了 `placement new`，在 `buffer` 所指向的内存中创建了一个 `string` 类型的对象并且初始值为“ `abc` ”。

- 因此可以看出：
    - `new operator` 可以分解 `operator new` 和 `placement new` 两个动作，是 `operator new` 和 `placement new` 的结合。
- 与 `new` 对应的 `delete` 没有 `placement delete` 语法
    - 它只有两种，分别是 `delete operator` 和 `operator delete`。
    - `delete operator` 和 `new operator` 对应，完成**析构对象**和**释放内存**的操作。
    - 而 `operator delete` 只是用于**内存的释放**，与 `C语言` 中的 `free` 相似。

120、为什么拷贝构造函数必须传引用不能传值？

- 拷贝构造函数的作用就是用来复制对象的，在使用这个对象的实例来初始化这个对象的一个新的实例。
- 两种不同的参数传递方式：
- 值传递:
    - 对于内置数据类型的传递时，直接赋值拷贝给形参(注意形参是函数内局部变量)；对于类类型的传递时，需要首先调用该类的拷贝构造函数来初始化形参(局部对象)；
    - 如 `void foo(class_type obj_local){}`，如果调用 `foo(obj)`；首先 `class_type obj_local(obj)` ,这样就定义了局部变量 `obj_local` 供函数内部使用
- 引用传递:
    - 无论对内置类型还是类类型，传递引用或指针最终都是传递的**地址值**！而地址总是指针类型(属于简单类型), 显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用(对于类类型).

- 拷贝构造函数使用值传递会产生无限递归调用，内存溢出。
- 拷贝构造函数用来初始化一个非引用类类型对象，如果用传值的方式进行传参数，那么构造实参需要调用拷贝构造函数，而拷贝构造函数需要传递实参，所以会一直递归。

121、空类的大小是多少？为什么?

- C++ 空类的大小不为 0，不同编译器设置不一样， vs 设置为 1；
- C++ 标准指出，不允许一个对象(当然包括类对象)的大小为 0，**因为不同的对象不能具有相同的地址**；
- 带有虚函数的 C++ 类大小不为 1，因为每一个对象会有一个 vptr 指向虚函数表，具体大小根据指针大小确定；
- C++ 中要求对于类的每个实例都必须有独一无二的地址,那么**编译器自动为空类分配一个字节大小**，这样便保证了每个实例均有独一无二的内存地址。

122、你什么情况用指针当参数，什么时候用引用，为什么?

- 使用引用参数的主要原因有两个：
  - 程序员能**修改**调用函数中的数据对象
  - 通过传递引用而不是整个数据–对象，可以提高程序的**运行速度**
- 一般的原则：
  - 对于使用数据对象不做修改的函数：
    - 如果数据对象很 小(内置数据类型或者小型结构)，则按**照值传递**；
    - 如果数据对象是 数组，则使用指针 **(唯一的选择)**，并且指针声明为 **const 的指针**；
    - 如果数据对象是 较大的结构，则使用 **const 指针或者引用**，已提高程序的效率。这样可以节省结构所需的时间和空间；
    - 如果数据对象是 类对象，则使用 **const 引用**(传递类对象参数的标准方式是按照引用传递)；
  - 对于修改函数中数据的函数：

- 如果数据是 内置数据类型， 则使用**指针**
- 如果数据对象 是数组， 则只能使用**指针**
- 如果数据对象是 结构， 则使用**引用或者指针**
- 如果数据是 类对象， 则使用**引用**

123、大内存申请时候选用哪种？ C++ 变量存在哪？变量的大小存在哪？符号表存在哪？

- 大内存申请时，采用堆申请空间，用 new 申请， 当大于 128K 的时候会在映射区分配内存.
- 变量存储位置:
  - 全局变量
  - 静态变量
  - 局部变量
  - 堆对象:大, 小
- 符号表只存在于编译阶段, 符号表的每一项分别对应变量名和变量地址, 但是 C++ 对变量名不作存储，在汇编以后不会出现变量名，变量名作用只是用于方便编译成汇编代码，是给编译器看的，是方便人阅读的

124、为什么会有大端小端， htol 这一类函数的作用

- 计算机以字节为基本单位进行管理, 每个地址单元都对应着一个字节，一个字节为 8bit 。但是我们常用到大于一个字节的数据类型, 例如 short， int， float 等, 此时就会存在字节如何放置的问题, 从而出现了大端模式和小端模式.
- 大端: 低字节放于高地址处(网络字节序为大端)
- 小端: 低字节放于低地址处(通常主机字节序为小端)
- 例如(16bit 的 short型 x)
  - 在内存中的地址为 0x0010， x 的值为 0x1122， 那么 0x11 为高字节， 0x22 为低字节。
  - 对于大端模式，就将 0x11 放在低地址中，即 0x0010 中， 0x22 放在高地址中，即 0x0011 中。小端模式，刚好相反。

125、静态函数能定义为虚函数吗？常函数?

- 不能！
  - `static` 成员不属于任何类对象或类实例，没有this指针(静态与非静态成员函数的一个主要区别)。
  - 虚函数调用链为: `vptr -> vtable -> virtual function`
  - 但是访问 `vptr` 需要使用 `this` 指针但是 `static` 成员函数没有 `this` 指针, 从而无法实现虚函数的调用

  - 虚函数依靠 `vptr` 和 `vtable` 来处理。 `vptr` 是一个指针，在类的构造函数中创建生成，并且只能用 `this` 指针来访问它，因为它是类的一个成员，并且 `vptr` 指向保存虚函数地址的 `vtable`. 对于静态成员函数，它没有 `this` 指针，所以无法访问 `vptr`. 这就是为何 `static` 函数不能为 `virtual`. 虚函数的调用关系： `this -> vtable -> virtual function`

126、 this 指针调用成员变量时，堆栈会发生什么变化?

- 当我们在类中定义非静态成员函数时, 编译器会为此成员函数添加一个参数(最后一个形参), 类型为当前类型的指针
- 当我们进行通过对象或则对象指针调用此成员函数时, 编译器会自动将对象的地址传给作为隐含参数传递给函数，这个隐含参数就是 `this` 指针。即使你并没有写 `this` 指针，编译器在链接时也会加上 `this` 的，对各成员的访问都是通过 `this` 的。
- 函数调用时，`this` 指针首先入栈，然后成员函数的参数从右向左进行入栈，最后函数返回地址入栈。

127、静态绑定和动态绑定的介绍

- 对象的静态类型：**对象在声明时采用的类型。**是在编译期确定的。
- 对象的动态类型：**目前所指对象的类型。**是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。
- 静态绑定：绑定的是对象的静态类型，某特性(比如函数)依赖于对象的静态类型，发生在编译期。
- 动态绑定：绑定的是对象的动态类型，某特性(比如函数)依赖于对象的动态类型，发生在运行期。

128、设计一个类计算子类的个数

- 为类设计一个 `static` 静态变量 `count` 作为计数器;
- 类定义结束后初始化 `count` ;
- 在构造函数中对 `count` 进行 `+1` ;
- 设计拷贝构造函数，在进行拷贝构造函数中进行 `count +1` ，操作;
- 设计复制构造函数，在进行复制函数中对 `count+1` 操作;
- 在析构函数中对 `count` 进行 `-1` ;

129、怎么快速定位错误出现的地方

- 如果是简单错误, 通常可以分析编译器辗转解栈过程, 定位到输出位置, 通常都是解栈的靠后位置
- 如果错误较复杂, 就最好使用gdb调试模式, 进行调试, 逐步定位错误位置, 或者添加更多的输出信息.

130、虚函数的代价?

1）带有虚函数的类，每一个类会产生一个虚函数表，用来存储指向虚成员函数的指针，增大类； 1）带有虚函数的类的每一个对象，都会有有一个指向虚表的指针

131、类对象的大小

1) 类的非静态成员变量大小，静态成员不占据类的空间，成员函数也不占据类的空间大小； 2) 内存对齐另外分配的空间大小，类内的数据也是需要进行内存对齐操作的； 3) 当该该类是某类的派生类，那么派生类继承的基类部分的数据成员也会存在在派生类中的空间中，也会对派生类进行扩展。 4) 虚函数的话，会在类对象插入vptr 指针，加上指针大小； 5) 如果是虚拟继承而来的话, 还会存在一个虚基类表指针, 不同的编译器对这个虚基类指针的处理是不一样的, gcc是存放在虚函数表中(意味着虚函数表指针和虚基类表指针只会存在一个), vc是存放在对象中的(意味着可能会虚函数表指针和虚基类表指针共存)

132、移动构造函数

- 移动构造函数是C++11中引入的移动语义的具体实现. 它的主要目的是避免无谓的构造和析构
- 例如: 当我们用右值初始化一个左值时, 通常是使用复制构造函数构造左值,然后对右值调用析构函数, 此时存在大量的浪费. 而且复制构造函数对于指针通常是浅复制, 容易产生野指针.
- 移动构造函数的参数为右值引用, 它的作用就是将此右值的内容**转移**到左值内, 从而避免右值调用构造函数. 也避免了左值分配内存进行构造.

```
Example6 (Example6&& x):ptr(x.ptr){
    x.ptr = nullptr;
}
// move assignment
Example6& operator= (Example6&& x){
    delete ptr;
    ptr = x.ptr;
    x.ptr=nullptr;
    return *this;
}
```

133、何时需要合成构造函数

- 如果一个类没有构造函数，一共四种情况会合成构造函数:
  - **存在虚函数**的情况
  - **存在虚基类**的情况
  - **基类成员**存在构造函数的情况
  - **对象成员对象**存在构造函数的情况

134、何时需要合成复制构造函数

- 有三种情况会以一个对象的内容作为另一个对象的初值： 1) 对一个对象做显示的初始化操作， X xx = x；2) 当对象被当做参数交给某个函数时； 3) 当函数传回一个类对象时；
- 如果一个类没有拷贝构造函数，合成复制构造函数的情况:
  - 成员对象有拷贝构造函数
  - 基类拷贝构造函数
  - 存在虚函数
  - 存在虚基类

135、何时需要成员初始化列表？过程是什么?

- 需要成员初始化列表:
  - 引用类型的成员变量
  - const类型的成员变量
  - 基类不存在零参数构造函数
  - 成员对象不存在零参数构造函数
- 过程:
  - 编译器会根据成员变量定义顺序一一初始化成员变量, 如果相应成员在成员初始化列表中有初始化参数, 则用成员初始化列表中的参数进行构造
  - 发生在用户自定义代码段之前.

136、程序员定义的析构函数被扩展的过程?

- 析构函数的执行顺序(和构造相反):
  - 析构函数函数体被执行

    - 本类的成员对象析构函数被调用, 调用顺序和声明的顺序相反
    - 非虚基类拥有析构函数，会以声明的相反顺序被调用；
    - 虚基类被析构

- 参考: C++虚基类构造函数详解(调用顺序)之一 (https://www.cnblogs.com/haoyuanyuan/archive/2013/04/25/3041250.html)

137、构造函数的执行算法?

- 扩展过程:
    - 虚基类按照定义顺序被构造
    - 基类按照定义顺序被构造
    - 成员变量被构造
    - 执行程序员所提供的代码
- 一个类被构造的执行过程:
    - 虚基类按照定义顺序被构造
    - 基类按照定义顺序被构造
    - 然后是按照定义顺序构造成员变量, 如果某个成员在初始化成员变量列表内存在初始化参数, 则调用初始化成员变量列表内的参数初始化该成员变量
    - 然后是执行构造函数函数体内用户提供的代码.
- 注意事项:
    - 在构造函数函数体内的对虚函数的调用将不具备动态绑定的特性

- 参考:

- C++ 构造函数执行原理 (https://blog.csdn.net/qingdujun/article/details/26626605)
- 构造函数和析构函数中可以调用调用虚函数吗 (https://www.cnblogs.com/sylar5/p/11523992.html)

138、构造函数的扩展过程?

- 虚基类按照定义顺序被构造
- 基类按照定义顺序被构造
- 成员变量被构造
- 执行程序员所提供的代码

- 参考: 构造函数和析构函数中可以调用调用虚函数吗 (https://www.cnblogs.com/sylar5/p/11523992.html)

139、哪些函数不能是虚函数

- 构造函数: 首先是没必要使用虚函数, 其次不能使用虚函数
- 内联函数: 表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型确定，所以内联函数不能是虚函数;
- 静态函数: 静态函数不属于对象属于类，静态成员函数没有this 指针，因此静态函数设置为虚函数没有任何意义。
- 友元函数: 友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说法。
- 普通函数: 普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚函数。

- 参考: 19. 构造函数为什么不能为虚函数？析构函数为什么要虚函数?

140. sizeof 和strlen 的区别

- sizeof 是一个**取字节运算符**，计算变量所占的内存数(字节大小), 可以用于任意类型
- strlen 是个**函数**, 计算字符串的具体长度(只能是字符串)，不包括字符串结束符(<!JEKYLL@2780@1245>)。
- strlen 是个不安全的函数, 如果没有 <!JEKYLL@2780@1247> 将会发生段错误。
- sizeof 和 strlen 对同一个字符串求值, 结果差一.
- 数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针；

141、简述strcpy、 sprintf 与memcpy 的区别

- 复制操作: strcpy , memcpy
  - 复制类容不一样: strcpy 是用于复制字符串的, 不能用去其他类型, 而 memcpy 是用于复制任意类型的数据类型
  - 复制防止不一样: strcpy 是通过检测支付中的 <!JEKYLL@2780@1260> 判断结束的, 存在溢出风险 (strncpy)；而 memcpy 是需要指定复制的字节数的.
- 字符串格式化: sprintf
  - 将格式化的数据写入字符串中
  - 注意 sprintf 对写入字符串没有限制大小, 也就存在溢出风险, 建议采用 snprintf

142、编码实现某一变量某位清0 或置1

```
#define BIT3 (0x1 << 3 ) Satic int a; //设置a 的bit 3:
void set_bit3( void ){
        a |= BIT3; //将a 第3 位置1
}
//清a 的bit 3
void set_bit3( void ){
        a &= ~BIT3; //将a 第3 位清零
}
```

143、将“引用”作为函数参数有哪些特点?

1) 传递引用给函数与传递指针的效果是一样的。 1) 这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象(在主调函数中)的操作。 2) 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作； 1) 而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本； 2) 如果传递的是对象，还将调用拷贝构造函数。 3) 因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。 3) 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用”*指针变量名”的形式进行运算，这很容易产生错误且程序的阅读性较差； 1) 另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

- 引用传递 从逻辑上就好像是对主调函数中的实参取了一个别名, 在被调函数中对该别名的任何操作都会反应在主调函数中, 实际的实现过程中, 传递的其实是对象的地址, 和指针传递相似, 区别在于对该引用的任何操作都会被处理为间接寻址
- 引用传递 并没有对 对象 进行拷贝, 只是对指针进行了拷贝, 避免了对 对象 的复制, 效率更高.
- 引用传递 逻辑上相当于对主调函数中的实参取了一个别名, 阅读性更好.

144、分别写出BOOL,int,float,指针类型的变量a 与“零”的比较语句。

```
BOOL : if ( !a ) or if(a)
int : if ( a == 0)
float : const EXPRESSION EXP = 0.000001 // 1.0e-10 浮点数有精度限制，所以只能通过阈值来判断是否相等
if ( a < EXP && a >-EXP)
pointer : if ( a != NULL) or if(a == NULL)
```

145、局部变量全局变量的问题?

- 局部会屏蔽全局。
  - 要用全局变量，需要使用”::”, 局部变量可以与全局变量同名，在函数内引用这个变量时，会用到同名的局部变量，而不会用到全局变量。
  - 对于有些编译器而言，在同一个函数内可以定义多个同名的局部变量，比如在两个循环体内都定义一个同名的局部变量，而那个局部变量的作用域就在那个循环体内。
- 如何引用一个已经定义过的全局变量，可以用引用头文件的方式，也可以用 extern 关键字，如果用引用头文件方式来引用某个在头文件中声明的全局变理，假定你将那个变写错了，那么在编译期间会报错，如果你用 extern 方式引用时，假定你犯了同样的错误，那么在编译期间不会报错，而在连接期间报错。
- 全局变量可不可以定义在可被多个 .C 文件包含的头文件中，在不同的 C 文件中以 static 形式来声明同名全局变量。可以在不同的 C 文件中声明同名的全局变量，前提是其中只能有一个 C 文件中对此变量赋初值，此时连接不会出错

- 局部屏蔽全局
- 引用另一个文件中的变量, 使用 extern 关键字, 或则引用头文件
- 全局变量冲突

146、数组和指针的区别?

- 对数组使用 sizeof 操作符可以计算出数组的容量(字节数). 对指针使用 sizeof 操作符得到的是一个指针变量的字节数，而不是 p 所指的内存容量。
- 编译器为了简化对数组的支持，实际上是利用指针实现了对数组的支持。具体来说，就是将表达式中的数组元素引用转换为指针加偏移量的引用。
- 在向函数传递参数的时候，如果实参是一个数组，那用于接受的形参为对应的指针。也就是传递过去是数组的首地址而不是整个数组，能够提高效率;
- 在使用下标的时候，两者的用法相同，都是原地址加上下标值，不过数组的原地址就是数组首元素的地址是固定的，指针的原地址就不是固定的。

- 数组作为 sizeof 参数时, 不会退化
- 数组在内存中是连续存放的，开辟一块连续的内存空间;
  - 数组所占存储空间： sizeof(数组名);
  - 数组大小： sizeof(数组名)/sizeof(数组元素数据类型);
- 指针也可以使用下标, 表示指针指向地址 + 偏移
  - 参考: C语言指针变量加下标的作用意思意义 (https://blog.csdn.net/qianxuedegushi/article/details/81699033)

147、C++如何阻止一个类被实例化？一般在什么时候将构造函数声明为 private?

1) 将类定义为 抽象基类 或者将 构造函 数声明为 private； 2) 不允许类外部创建类对象(也就是杜绝了静态构建的可能性)，只能在类内部创建对象(成员函数通过 new 构建)

148、如何禁止自动生成拷贝构造函数?

1) 为了阻止编译器默认生成拷贝构造函数和拷贝赋值函数，我们需要手动去重写这两个函数，某些情况下，为了避免调用拷贝构造函数和拷贝赋值函数，我们需要将他们设置成private，防止被调用。 2) 类的成员函数和friend 函数还是可以调用private 函数，如果这个private 函数只声明不定义，则会产生一个连接错误； 3) 针对上述两种情况，我们可以定一个base 类，在base 类中将拷贝构造函数和拷贝赋值函数设置成private,那么派生类中编译器将不会自动生成这两个函数，且由于base 类中该函数是私有的，因此，派生类将阻止编译器执行相关的操作。

- 参考: 高效C++; 条款6：阻止编译器自动生成拷贝构造函数和赋值函数 (https://blog.csdn.net/qq_29422251/article/details/77850312)

---

1) 拷贝构造函数的定义后面使用 `=delete` 关键字 2) 将 `base` 类的拷贝构造函数和拷贝赋值构造函数设置为 `private` , 这样编译器就不会自动生成这两个函数, 且由于 `base` 类的该函数为 `private` , 所以编译器会阻止相关操作.

149、 assert 与NDEBUG

1) `assert` 宏的原型定义在 `<assert.h>` 中，其作用是如果它的条件返回错误，则终止程序执行，原型定义： `c #include <assert.h> void assert( int expression );` - `assert` 的作用是计算表达式 `expression` ，如果其值为假(即为 `0`)，那么它先向 `stderr` 打印一条出错信息，然后通过调用 `abort` 来终止程序运行。如果表达式为真， `assert` 什么也不做。 2) `NDEBUG` 宏是 `Standard C` 中定义的宏，专门用来控制 `assert()` 的行为。 - 如果定义了这个宏，则 `assert` 不会起作用。 - 定义 `NDEBUG` 能避免检查各种条件所需的运行时开销，当然此时根本就不会执行运行时检查。

150、 Debug 和release 的区别

1) 调试版本，包含调试信息 - 体积 比 Release 大很多，并且 不进行任何优化 (优化会使调试复杂化，因为源代码和生成的指令间关系会更复杂)，便于程序员调试。 - Debug 模式下生成两个文件，除了 .exe 或 .dll 文件外，还有一个 .pdb 文件，该文件记录了代码中断点等调试信息； 3) 发布版本，不对源代码进行调试，编译时对应用程序的速度进行优化，使得程序在代码大小和运行速度上都是最优的。(调试信息可在单独的 PDB 文件中生成)。 Release 模式下生成一个文件 .exe 或 .dll 文件。 4) 实际上， Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本。

151、 main 函数有没有返回值

1) 程序运行过程入口点 main 函数， main() 函数返回值类型必须是 int ， 这样返回值才能传递给程序激活者(如操作系统)表示程序正常退出。 2) main(int args，char**argv) 参数的传递。参数的处理，一般会调用 getopt() 函数处理，但实践中，这仅仅是一部分，不会经常用到的技能点。 3) main 函数事调用用户代码逻辑的接口有着固有的规范(或则逻辑): - 返回值: int 程序退出状态 - 参数: 用于传递到用户代码中 - int args : 参数个数 - char** argv : 参数以字符串的形式储存, 然后将字符串首地址组成指针数组作为参数进行传递.

- 参考: 命令行选项解析函数(C语言)：getopt()和getopt_long() (https://www.cnblogs.com/chenliyang/p/6633739.html)

152、写一个比较大小的模板函数

```
#include<iostream> using namespace std;
template<typename type1,typename type2>//函数模板
type1 Max(type1 a,type2 b){
        return a > b ? a : b;
}
void main(){
        cout<<"Max = "<<Max(5.5,'a')<<endl;
}
```

153、 c++怎么实现一个函数先于main 函数运行

1) 全局对象/全局静态变量的生存期和作用域都高于 mian 函数, 在 main 函数之前初始化

```
        class simpleClass{
        public:
                simpleClass( ){
                        cout << "simpleClass constructor.." << endl;
                }
        };
        simpleClass g_objectSimple; //step1 全局对象
        int _tmain(int argc, _TCHAR* argv[]) { //step3
                return 0;
        }
```

2) GCC编译器可以使用 __attribute((constructor/deconstrucor)) 在 main **之前**和**之后**注册函数 cpp // 在main之前 __attribute((constructor)) void before_main(){ printf("befor\n"); } // 在main之后 __attribute((deconstructor)) void after_main(){ printf("befor\n"); }

- 附加
  - Main 函数执行之前，主要就是初始化系统相关资源；
    - 设置栈指针
    - 初始化 static 静态和 global 全局变量，即 data 段的内容
    - 将未初始化部分的全局变量赋初值(即 .bss 段的内容)：
      - 数值型 short， int， long 等为 0，
      - bool 为 FALSE
      - 指针为 NULL
    - 全局对象初始化，在 main 之前调用构造函数
    - 将 main 函数的参数， argc， argv 等传递给 main 函数，然后才真正运行 main 函数
  - Main 函数执行之后
    - 全局对象的析构函数会在 main 函数之后执行；
    - 可以用 _onexit 注册一个函数，它会在 main 之后执行;
- 参考:
  - 在main()之前执行前运行 (https://blog.csdn.net/zzxiaozhao/article/details/102604626#main_228)
  - C++中_onexit()用法简述 (https://blog.csdn.net/cafuc46wingw/article/details/38587473)

154、虚函数与纯虚函数的区别在于

1) 纯虚函数只有定义没有实现，虚函数既有定义又有实现； 2) 含有纯虚函数的类不能定义对象，含有虚函数的类能定义对象；

155、智能指针怎么用？智能指针出现循环引用怎么解决？

1) unique_ptr：独占式拥有一个对象, 当 unique_ptr 被销毁时，它所指向的对象也被销毁。

2) shared_ptr：- 初始化: - shared_ptr<int> p =make_shared<int>(42); - 通常用 auto 更方便， auto p =make_shared<int>(42); - shared_ptr<int> p2(new int(2)); - 每个 shared_ptr 都有一个关联的计数器，通常称为引用计数，一旦一个 shared_ptr 的计数器变为 0，它就会自动释放自己所管理的对象； - shared_ptr 的析构函数就会递减它所指的对象的引用计数。 - 如果引用计数变为 0， shared_ptr 的析构函数就会销毁对象，并释放它占用的内存。 3) weak_ptr :是一种不控制所指向对象生存期的智能指针，它指向由一个 shared_ptr 管理的对象，将一个 weak_ptr 绑定到一个 shared_ptr 不会改变引用计数，一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放，即使有 weak_ptr 指向对象，对象还是会被释放。

4) 弱指针用于专门解决 shared_ptr 循环引用的问题， weak_ptr 不会修改引用计数，即其存在与否并不影响对象的引用计数器。循环引用就是：两个对象互相使用一个 shared_ptr 成员变量指向对方。弱引用并不对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。

156、 strcpy 函数和 strncpy 函数的区别？哪个函数更安全?

1) 函数原型 c char* strcpy(char* strDest, const char* strSrc) char* strncpy(char* strDest, const char* strSrc, int pos) 2) strcpy 函数: - 如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出 (bufferOverflow) 的错误情况，在编写程序时请特别留意，或者用 strncpy() 来取代。 - strncpy 函数：用来复制源字符串的前 n 个字符， src 和 dest 所指的内存区域不能重叠，且 dest 必须有足够的空间放置 n 个字符。 4) 长度关系: - 如果 目标长 > 指定长 > 源长 ，则将源长全部拷贝到目标长，自动加上 <!JEKYLL@2780@1410> ; - 如果 指定长 < 源长 ，则将源长中按指定长度拷贝到目标字符串，不包括 <!JEKYLL@2780@1413> ; - 如果 指定长 > 目标长 ，运行时错误;

157、为什么要用 static_cast 转换而不用 c 语言中的转换?

1) 更加安全； 2) 更直接明显，能够一眼看出是什么类型转换为什么类型，容易找出程序中的错误； - 可清楚地辨别代码中每个显式的强制转； - 可读性更好，能体现程序员的意图

158、成员函数里memset(this,0,sizeof(*this))会发生什么

1) 如果类中的所有成员都是内置的数据类型的, 则不会存在问题 2) 如果有以下情况之一会出现问题: - 存在对象成员 - 存在虚函数/虚基类 - 如果在构造函数中分配了堆内存, 而此操作可能会产生内存泄漏

159、方法调用的原理(栈，汇编)

- 每一个函数都对应一个栈帧:
  - 帧栈可以认为是程序栈的一段
  - 它有两个端点
    - 一个标识起始地址, 开始地址指针ebp;
    - 一个标识着结束地址，结束地址指针esp;
- 函数调用使用的参数, 返回地址等都是通过栈来传递的.
- 函数调用过程:
  - 参数逆序入栈(主调函数)
  - 返回地址入栈(主调函数)(被调函数栈底往上4个子节为返回地址)

  - 主调函数栈底入栈(被调函数)
  - 栈顶给栈底赋值(被调函数)
  - 被调函数局部变量…
  - 被调函数局部变量析构
  - 恢复主调函数栈帧
  - 获取返回地址, 继续执行主调函数代码
  - 关于返回值:
    - 如果 返回值 <= 4字节 ，则返回值通过寄存器 eax 带回。
    - 如果 4< 返回值 <=8字节 ，则返回值通过两个寄存器 eax 和 edx 带回。

- 如果 返回值 > 8字节 ，则返回值通过产生的临时量带回。

---

4) 过程调用和返回指令 - call 指令 - leave 指令 - ret 指令

160、 MFC 消息处理如何封装的?

161、回调函数的作用

- 回调函数一般可以分为两个类型:
- 中断处理函数
  - 当发生某种事件时，系统或其他函数将会自动调用你定义的一段函数;
  - 此时回调函数就相当于是中断处理函数, 由系统在符合你设定的条件时自动调用。
  - 为此我们需要进行
    - 函数声明
    - 函数定义
    - 设置中断触发, 就是把回调函数名称转化为地址作为一个参数，以便于系统调用;
- 通过函数指针调用的函数
  - 如果函数的指针(地址)作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数;
  - 因为可以把调用者与被调用者分开。调用者只需要确定被调函数是一个具有特定参数列表和特定返回值的函数, 而不需要知道具体是哪个函数被调用.

162、随机数的生成

```
#include<time.h> srand((unsigned)time(NULL));
cout<<(rand()%(b-a))+a;
```

- 由于 `rand()` 的内部实现是用线性同余法做的，所以生成的并不是真正的随机数，而是在一定范围内可看为随机的伪随机数。
- 种子写为 `srand(time(0))` 代表着获取系统时间，电脑右下角的时间，每一秒后系统时间的改变，数字序列的改变得到的数字

164、C++临时对象产生的时机

- 为了使函数调用成功而进行隐式类型转换的时候。
  - 传值方式
  - const 引用传递(!!!!)
- 当函数返回对象的时候。
- 参考: 转：C++中临时对象及返回值优化 (https://www.cnblogs.com/xkfz007/articles/2506022.html)

PREVIOUS
UNIX网络编程 学习笔记 (二)
(/2019/11/26/UNIX_NETWORK_PROGRAMMING_02/)

NEXT
深度探索C++对象模型
(/2019/11/28/CPLUSPLUS_OBJECT_MODEL/)

FEATURED TAGS (/tags/)

C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B) C/C++ (/tags/#C/C++)
后台开发 (/tags/#%E5%90%8E%E5%8F%B0%E5%BC%80%E5%8F%91) C (/tags/#C) 网络编程 (/tags/#%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B)
STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90) Linux (/tags/#Linux)
操作系统 (/tags/#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 程序设计 (/tags/#%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)
优化 (/tags/#%E4%BC%98%E5%8C%96) UML (/tags/#UML) UNIX (/tags/#UNIX) 学习笔记 (/tags/#%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0)
面试 (/tags/#%E9%9D%A2%E8%AF%95) Java (/tags/#Java) 读书笔记 (/tags/#%E8%AF%BB%E4%B9%A6%E7%AC%94%E8%AE%B0) go (/tags/#go)
阅读笔记 (/tags/#%E9%98%85%E8%AF%BB%E7%AC%94%E8%AE%B0)

FRIENDS

WY (http://zhengwuyang.com) 简书·JF (http://www.jianshu.com/u/e71990ada2fd) Apple (https://apple.com)
Apple Developer (https://developer.apple.com/)

● (https://www.facebook.com/wangpengcheng) ● (https://github.com/wangpengcheng)

Theme on GitHub (https://github.com/wangpengcheng/wangpengcheng.github.io.git) | Star 12