C++ 内存管理方式

堆、栈、自由存储区、全局/静态存储区、常量存储区 自由存储区存储malloc申请的内存
(1)从静态存储区域分配 。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如 全局变量， static 变量 。 (2)在栈上创建 。在执行函数时， 函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集。 (3)从堆上分配 ， 亦称动态内存分配 。程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。动态内存的生存期由程序员决定，使用非常灵活，但问题也最多。

extern "C"和extern的作用

extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言的进行编译，而不是C++的。
extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。

为什么需要内存对齐

c++内存问题

1. 缓冲区溢出（buffer overrun）。
   用 std::vector/std::string 或自己编写 Buffer class 来管理缓冲区，自动记住用缓冲区的长度，并通过成员函数而不是裸指针来修改缓冲区。
2. 空悬指针/野指针。
   用 shared_ptr/weak_ptr
3. 重复释放（double delete）。
4. 内存泄漏（memory leak）。
5. 不配对的 new[]/delete。
   把 new[] 统统替换为 std::vector/scoped_array。
6. 内存碎片（memory fragmentation）。

static关键字

- 函数体内static变量：的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值。
- 模块内的static全局变量：可以被模块内所有函数访问，但不能被模块外其他函数访问。
- 模块内的static函数：只可被这一模块内的其他函数调用，这个函数的使用范围被限制在声明它的模块内。起到了隐藏的作用。
- 在类的 static 成员变量：属于整个类所拥有，对类的所以对象只有一份拷贝
- 在类中的 static 成员函数：属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量
  static和extern：(1) extern 表明该变量在别的地方已经定义过了,在这里要使用那个变量.(2) static 表示静态的变量，分配内存的时候, 存储在静态区,不存储在栈上面. **静态全局变量不要放在头文件里** (https://www.cnblogs.com/yc_sunniwell/archive/2010/07/14/1777431.html) 第6条

C++中的static关键字的总结

几个复制的声明

```
void * ( * (*fp1)(int))[10];  //fp1是一个指针，指向一个函数，函数参数为int，函数返回参数是一个指针，指针指向一个数组，数组中有10个元素，每个元素是一个void* 指针。
float (*(* fp2)(int,int,int))(int);  //fp2是一个指针，指向一个函数，函数参数为3个int，函数的返回值是一个指针，指针指向一个函数，函数的参数是1个int，返回float。
int (* ( * fp3)())[10]();  //fp3是一个指针，指向一个函数，函数没有参数，函数返回值为一个指针，指针指向一个数组，数组中有10个元素，每个元素是一个函数指针，函数没有参数，返回int。
```

strlen()和sizeof()

1. 种类：
   sizeof是运算符，并不是函数，结果在编译时得到,因此sizeof不能用来返回动态分配的内存空间的大小。用sizeof来返回类型以及静态分配的对象、结构或数组所占的空间，返回值跟对象、结构、数组所存储的内容没有关系；strlen是字符处理的库**函数**,当数组名作为参数传入时，实际上数组就退化成指针了。。
2. 输入参数：
   sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化），还可以用函数做参数；strlen的参数只能是字符指针且结尾是'¥0'的字符串。
3. 功能：
   sizeof():获得保证能容纳实现所建立的最大对象的字节大小
   strlen():返回字符串的长度。该字符串可能是自己定义的，也可能是内存中随机的，该函数实际完成的功能是从代表该字符串的第一个地址开始遍历，直到遇到结束符NULL。返回的长度大小不包括NULL。
4. 返回：
   当适用于一个结构类型时或变量， sizeof 返回实际的大小， 当适用于一静态地空间数组， sizeof 归还全部数组的尺寸。 sizeof 操作符不能返回动态地被分派了的数组或外部的数组的尺寸
   https://blog.csdn.net/21aspnet/article/details/1539951

变量声明和定义

- 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间；定义要在定义的地方为其分配存储空间。
- 相同变量可以再多处声明（外部变量extern），但只能在一处定义。

结构体和union

1.在存储多个成员信息时，编译器会自动给struct第个成员分配存储空间，struct 可以存储多个成员信息，而Union每个成员会用同一个存储空间，只能存储最后一个成员的信息。
2.都是由多个不同的数据类型成员组成，但在任何同一时刻，Union只存放了一个被先选中的成员，而结构体的所有成员都存在。
3.对于Union的不同成员赋值，将会对其他成员重写，原来成员的值就不存在了，而对于struct 的不同成员赋值 是互不影响的。
未特殊说明时，按结构体中size最大的成员对齐（若有double成员），按8字节对齐。
eg:

```
struct sTest
{
```

```
int a;  //sizeof(int) = 4
char b;  //sizeof(char) = 1
short c;  //sizeof(short) = 2
}x;  #最终实际占用不止4+1+2，因为要考虑内存对齐的问题

union uTest
{
int a;   //sizeof(int) = 4
double b;  //sizeof(double) = 8
char c;  //sizeof(char) = 1
}x;   #分配的内存 size 就是8 byte
```

pragma pack () 取消指定对齐，恢复缺省对齐

malloc/new和free/delete

- malloc和free是标准库函数，支持覆盖；new和delete是运算符，并且支持重载。
- malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险；new和delete除了分配回收功能外，还会调用构造函数和析构函数。
- malloc和free返回的是void类型指针（必须进行类型转换），new和delete返回的是具体类型指针。
- free和delete对应。当delete一个对象数组时，delete只删除了一个，需要使用delete[]

const

宏定义和const函数的区别:一般尽量用const比较好。

- 宏在编译时完成替换，直接进行替换，执行起来更快，但是可能会存在一些风险；函数调用在运行时需要跳转到具体调用函数。如：
- 宏函数属于在结构中插入代码，没有返回值；函数调用具有返回值。
- 宏函数参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
- 一般可以对const常量进行调试，但不能对宏常量进行调试。
- 宏函数不要在最后加分号。

const的分辨(顶层const和底层const)

一般来说，const的分辨可以直接通过看const的最左侧，如果是指针，则指针是const，若为类型，则变量为const。当const在最左侧时，看const右侧。

- 常量指针和指针常量： 常量指针是一个指针，读成常量的指针，指向一个只读变量。如`int const *p`或`const int *p`。
  指针常量是一个常量，指针的值可以改变。如`int *const p`。

指针和引用

区别

1. 指针是具体的变量，需要占存储空间。引用只是别名，不占用具体存储空间。这是最基本的一点，其他的特点也就可想而知了。
2. 指针可以先声明，但是引用声明的时候就必须初始化，不存在空的引用。

3. 指针变量可以改变所指的对象。但是引用一旦声明了就不能再改变引用的对象。

引用相关问题

1. 引用是某个变量的别名，因此定义的时候必须初始化，也不能把该引用再改成其他变量的别名。
2. 声明一个引用并没有定义新变量，引用本身不是一种数据类型。也不占用存储空间。
3. ~~不能建立数组的引用~~。其实这句话的意思是：不能建立引用的数组，例如：int & ref[3] = { 2, 3, 5};
   但是可以建立数组的引用:例如：int arr[3]; int (&tef)[3] = arr;
   原因是：引用时不占空间的，声明引用数组没法分配空间。
4. 将引用作为函数的参数时，可以避免对变量或者对象的复制，因此不会调用对象的拷贝构造函数。当不希望传入的引用参数不被改变时，使用const引用。
5. 函数中不能返回局部变量的引用，不能返回函数内部ne分配的内存的引用。（虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成内存泄露。）。 可以返回类成员的引用，但最好是const。
6. 当类中存在const或者引用时成员变量时，必须使用初始化表。

指针相关问题

指针的相关判断

```
    int *p[10] //表示指针数组，强调数组概念，是一个数组变量，数组大小为10，数组内每个元素都是指向int类型的指针变量。
    int (*p)[10] //表示数组指针，强调是指针，只有一个变量，是指针类型，不过指向的是一个int类型的数组，这个数组大小是10。
    int *p(int)   //函数声明，函数名是p，参数是int类型的，返回值是int *类型的。
    int (*p)(int)  //函数指针，强调是指针，该指针指向的函数具有int类型参数，并且返回值是int类型的。
```

指针与数组名

- 二者均可通过增减偏移量来访问数组中的元素。
- 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
- 当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。

野指针

空悬指针，不是指向null的指针，是指向垃圾内存的指针。
产生原因及解决办法：指针变量未及时初始化 => 定义指针变量及时初始化，要么置空。指针free或delete之后没有及时置空 => 释放操作后立即置空。

指针和数组的区别

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。指针可以随时指向任意类型的内存块。

1. 修改内容上的差别:指针可能指向一块内存，但是指向的常量却无法通过下标计算。

```
char a[] = “hello”;
a[0] = ‘X’;
char *p = “world”; // 注意p 指向常量字符串，指向的是常量区
p[0] = ‘X’; // 编译器不能发现该错误，运行时错误
```

2. 用运算符sizeof 可以计算出数组的容量（字节数）。sizeof(p),p 为指针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。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

```
char a[] = "hello world";
char *p = a;
cout<< sizeof(a) << endl; // 12 字节
cout<< sizeof(p) << endl; // 4 字节
//计算数组和指针的内存容量
void Func(char a[100])
{
        cout<< sizeof(a) << endl; // 4 字节而不是100 字节
}
```

volatile

- volatile定义变量的值是易变的，每次用到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。
- 多线程中被几个任务共享的变量需要定义为volatile类型。

堆栈

区别

1. 申请方式不同：栈由系统自动分配，堆由程序员手动分配
2. 申请大小不同：栈顶和栈底都是设定好的，大小固定，可以通过`ulimit -a`查看，通过`ulimit -s`修改。堆向高地址扩展，是不连续的内存区域，大小可以调整。
3. 申请效率不同：栈由系统分配，速度快，没有碎片。堆速度慢，且有碎片。

内存分配

https://blog.csdn.net/nkguohao/article/details/8771867

面向对象

面向对象三大特性

- 封装性：数据和代码捆绑在一起，避免外界干扰和不确定性访问。
- 继承性：让某种类型对象获得另一个类型对象的属性和方法。

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

构造函数和析构函数

1.构造函数、析构函数中都不要调用虚函数

我们知道，构造函数一般不能是虚函数，而析构函数一般必须是虚函数。原理也很清晰，构造函数，由于构造顺序是从基类到派生类，所以调用虚函数，可能派生类还没有构造出来，没有意义。而对于析构函数来说，又必须是虚函数，因为只有先从子类对象进行销毁，才能保证资源不泄露。

在构造函数和析构函数中都不要调用虚函数也是这个道理。

成员变量和成员函数

1.静态成员变量是需要初始化
其实这样说的是有点问题的，应该是静态成员是需要定义的。 因为静态成员属于整个类，而不属于某个对象，如果在类内初始化，会导致每个对象都包含该静态成员，这是矛盾的。
《c++primer》里面说在类外定义和初始化是保证static成员变量只被定义一次的好方法。 但static const int 就可以在类里面初始化

```
class Base{
    public:
        static int class_p;   //只有声明，而没有定义，不能直接调用
};
int Base::class_p=3;   //进行定义
https://blog.csdn.net/qq_16209077/article/details/52602601
```

拷贝构造函数

调用情况：

1. 用一个类的对象去初始化该类的另一个对象时。
2. 函数形参是类的对象时，调用函数将函数的形参和实参结合的时候。
3. 函数返回值是类的对象，函数调用完成返回时。

重写拷贝构造函数

一般会默认生成类的拷贝构造函数，但是当涉及动态分配存储空间时，默认的拷贝构造函数就会有问题，因此需要重写拷贝构造函数，并且采用深拷贝。
浅拷贝和深拷贝：

多态

多态：对于不同对象接收相同消息时产生不同的动作。C++的多态性具体体现在运行和编译两个方面：
编译时多态：函数和运算符的重载。
运行时多态：继承和虚函数。

友元

特性：单向的，传递性，不能继承

标准模板库

编译和调试

编译过程

预处理->编译->汇编->链接

- 预处理：展开宏定义；处理条件编译；处理#include指令；去掉注释；添加行号和文件名标识；保留所有#pragma编译器指令。
- 编译：词法分析；语法分析；语义分析；中间语言生成；目标代码生成与优化。
- 链接：各个源代码模块独立的被编译，然后将他们组装起来成为一个整体，组装的过程就是链接。被链接的各个部分本本身就是二进制文件，所以在被链接时需要将所有目标文件的代码段拼接在一起，然后将所有对符号地址的引用加以修正。

静态库和动态库

二者的不同点在于代码被载入的时刻不同。静态库和动态库的最大区别,静态情况下,把库直接加载到程序中,而动态库链接的时候,它只是保留接口,将动态库与程序代码独立,这样就可以提高代码的可复用度，和降低程序的耦合度。

- 静态库的代码在编译过程中已经被载入可执行程序,程序运行时将不再需要该静态库，因此可执行程序体积比较大。在Linux中以`.a`结尾
- 动态库(共享库)的代码在可执行程序运行时才载入内存，在编译过程中仅简单的引用，因此代码体积比较小,在程序运行时还需要动态库存在。不同的应用程序如果调用相同的库,那么在内存中只需要有一份该动态库(共享库)的实例。在Linux中以`.so`结尾
  当静态库和动态库同名时， gcc命令将优先使用动态库.为了确保使用的是静态库，编译时可以加上 -static 选项，因此多第三方程序为了确保在没有相应动态库时运行正常，喜欢在编译最后应用程序时加入-static
  优**缺点**：
  1.动态库运行时会先检查内存中是否已经有该库的拷贝，若有则共享拷贝，否则重新加载动态库（C语言的标准库就是动态库）。静态库则是每次在编译阶段都将静态库文件打包进去，当某个库被多次引用到时，内存中会有多份副本，浪费资源。
  2.动态库更新很容易，当库发生变化时，接口没变只需要用新的动态库替换掉就可以。静态库需要重新编译。
  3.静态库静态库一次性完成了所有内容的绑定，运行时就不必再去考虑链接的问题了，执行效率会高一些。

安全相关

类型安全

类型安全很大程度上可以理解为内存安全。类型安全的代码不会试图去访问自己没有被授权的内存区域。
对于C语言来说，很多操作都不是类型安全的。例如打印的时候:`printf("%f\n",10) //编译通过，没有报错，结果为0.000000`.
对于C++来说，有些操作也不是类型安全的，比如不同类型指针之间可以强制转换(reinterpret cast) 注：C#、Java是类型安全的

C++使用得当，可以远比C更有类型安全性。 （1）操作符new返回的指针类型严格与对象匹配，而不是void*；
（2）C中很多以void*为参数的函数可以改写为C++模板函数，而模板是支持类型检查的；
（3）引入const关键字代替#define constants，它是有类型、有作用域的，而#define constants只是简单的文本替换；
（4）一些#define宏可被改写为inline函数，结合函数的重载，可在类型安全的前提下支持多种类型，当然改写为模板也能保证类型安全；
（5）C++提供了dynamic_cast关键字，使得转换过程更加安全，因为dynamic_cast比static_cast涉及更多具体的类型检查。

线程安全

如果代码在多线程运行和单线程运行具有相同的结果，那就是线程安全的。
线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

异常安全

当异常抛出时，带有异常安全的函数会： （1）不泄露任何资源 一般采用RAII技术，即以对象管(智能指针)理资源来防止资源泄漏。 (2)不允许数据被破坏（例如正常指针变野指针） (3)少些try catch，因为大量的try catch会影响代码逻辑。导致代码丑陋混乱不优雅
解决异常安全的问题： 1.多使用RAII，使用智能指针来管理内存。由于unwind机制的保证，当异常发生时，函数栈内已构造的局部对象的析构函数会被一一调用，在析构函数内释放资源，也就杜绝了内存泄漏的问题。
2.做好程序设计。特别是异常发生时的回滚机制的正确使用，copy-and-swap是有效的方法。
3.注意需要异常保证的函数内部的调用函数，异常安全等级是以有最低等级异常保证的函数确定的。一个系统即使只有一个函数不是异常安全的，那么系统作为一个整体就不是异常安全的。
4.流对象，资源对象，new对象，不应该直接作为参数，一旦抛出异常，就可能会导致严重的问题，函数也许会被错误的执行，资源也许会泄漏。
5.减少全局变量的使用。
6.如果不知道如何处理异常，就不要捕获异常，直接终止比吞掉异常不处理要好。
7.保证构造、析构、swap不会失败

类型安全 与 线程安全 、异常安全、事务安全

其他

为什么说栈比堆要快

1. 分配和释放：堆在分配和释放时都要调用函数（MALLOC,FREE），比如分配时会到堆空间去寻找足够大小的空间（因为多次分配释放后会造成空洞），这些都会花费一定的时间，具体可以看看MALLOC和FREE的源代码，他们做了很多额外的工作，而栈却不需要这些。
2. 访问时间，访问堆的一个具体单元，需要两次访问内存，第一次得取得指针，第二次才是真正得数据，而栈只需访问一次。
3. 堆的内容被操作系统交换到外存的概率比栈大，栈一般是不会被交换出去的。

c++协程的实现

ucontext-人人都可以实现的简单协程库

设计模式

单例模式

工厂方法

观察者模式

怎么判断两个结构体变量是否相等？

1，元素的话，一个个比咯：if(p1->age==p2->age)...有一个元素不等，即是两个实例不相等！没什么效率高的方法吧！

2，指针直接比较，如果保存的是同一个实例地址，则(p1==p2)为真！

3，重载==运算符；

Refeence:

常见C++面试题