C++

const与static关键字

- static作用
  - 在多个文件间隐藏全局变量与全局函数对于其他文件的可见性；
  - static变量储存于静态区域，具有全局生存期，可以保持变量内容的持久；
  - 由于其存储于静态区，具有默认初始化为0的作用；
  - 类成员声明为static，静态类成员属于整个类，他没有this指针，因此其只能访问静态数据与静态函数，不能将静态成员定义为虚函数，静态数据成员是静态存储的，所以必须对它进行初始化，初始化方式：**<数据类型><类名>::<静态数据成员名>=<值>**
- const作用
  - 修饰变量时，禁止变量值的改变（对于类中的const成员变量必须通过初始化列表进行初始化）
  - 非const变量默认为extern。要使const变量能够在其他文件中访问，必须在文件中显式地指定它为extern
  - const引用是指向const对象的引用，注意常量指针与指针常量
  - const修饰返回值与类成员函数

define与Const

- 编译器处理方式不同：define宏是在预处理阶段展开，生命周期止于编译期。const常量是编译运行阶段使用，const常量存在于程序的数据段.
- 类型与安全检查不同：define宏没有类型，不做任何类型检查，仅仅是展开。const常量有具体的类型，在编译阶段会执行类型检查。
- 存储方式不同：define宏仅仅是展开，有多少地方使用，就展开多少次，不会分配内存。const常量会在内存中分配(可以是堆中也可以是栈中)
- 虚函数机制
- 常见空类，虚函数指针大小

引用与指针

- 联系：
  - 指针与引用本质都是地址的概念，引用其内部是用const指针实现的。
  - 函数传递参数时，指针与引用可以达到相同效果
- 区别：
  - 指针是一个实体，需要分配内存空间。引用只是变量的别名，不需要分配内存空间（类中必须使用初始化列表）；
  - 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。
  - 有多级指针，但是没有多级引用，只能有一级引用。
  - 指针和引用的自增运算结果不一样。sizeof 引用得到的是所指向的变量（对象）的大小，而sizeof 指针得到的是指针本身的大小。
  - 引用访问一个变量是直接访问，而指针访问一个变量是间接访问。使用指针前最好做类型检查，防止野指针的出现；
  - 引用底层是通过指针实现的；

面向对象特征

- 继承：将客观事物封装成抽象的类,并且设计者可以对类的成员进行访问控制权限控制. 这样一方面可以做到数据的隐藏,保护数据安全;另一方面,封装可以修改类的内部实现而不用修改调用了该类的用户的代码.同时封装还有利于代码的 方便复用;
- 封装：继承具有这样一种功能,它可以使用现有类的所有功能；并且可以在不重新编写原有类的情况下对类的功能进行扩展.继承的过程是一般到特殊的过程,即是它们是is-a的关系;基类或父类是一般,而子类或派生类是基类的特殊表现;要实现继承可以通过继承和组合来实现;
- 多态：多态的实现分成两种,一种是编译时的多态,主要是通过函数重载和运算符重

  另外一种是运行时多态,主要是通过函数覆盖来实现的,它需要满足3个条件:基类函数必须是虚函数,并且基类的指针或引用指向子类的时候,当子类中对原有的虚函数进行重新定义之后形成一个更加严格的重载版本的时候,就会形成多态;它是通过动态联编实现的;

valitale关键字

- valitale关键字主要用于告诉编译器不要对访问变量的代码进行优化，从而提供对特殊地址的稳定访问，要求声明的变量系统总是重新从它所在的内存读取数据。
- 使用场景：中断服务程序中修改的供其它程序检测的变量需要加volatile；多任务环境下各任务间共享的标志应该加volatile； 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；
- 非valitale int变量可赋值给valitale int 变量，可用于基本数据类型和用户类型。

C++进程内存空间分配

- 代码段：二进制程序
- 常量区：具有常属性并且初始化的全局和静态变量放在这个区
- 数据区：赋过初值的且不具有常属性的静态和全局变量在数据区，它和BSS段统称为静态区
- BSS段：没有初始化的静态和全局变量;进程一旦被加载这个区所有的数据都被清0;
- 堆区：动态分配的内存;由程序员分配和释放,程序结束的时候如果没有释放,则由OS回收;
- 栈区：由编译器自动分配和释放,不使用的时候会自动的释放.主要用来存放非静态的局部变量,函数的参数和返回值, 临时变量等.
- 命令行参数和环境变量

如何定位内存泄漏

linux使用valgrind工具进行检查 添加-g编译参数添加调试信息

运行./valgrind --tool=memcheck --leak-check=yes --show-reachable=yes ./a.out

智能指针

- shared_ptr 代码实现

```
template <typename T>
    class SmartPtr
    {
    public:
        SmartPtr(T* p = NULL) :_ptr(p), _reference_count(new size_t)
        {
            if (p == NULL) (*_reference_count) = 0;
            else (*_reference_count) = 1;
        }

        SmartPtr(const SmartPtr& src)
```

```
        {
            if (src != this)
            {
                _ptr = src._ptr;
                _reference_count = src._reference_count;
                (*_reference_count)++;
            }
        }
        SmartPtr& operator=(const SmartPtr& src)
        {
            if (_ptr == src._ptr)return *this;

            removeCount();
            _ptr = src._ptr;
            _reference_count = src._ptr;
            (*_reference_count)++;
            return *this;
        }
        T& operator*()
        {
            if (_ptr) return *_ptr;
        }
        T* operator->()
        {
            if (_ptr) return _ptr;
        }
        ~SmartPtr()
        {
            if (--(*_reference_count) == 0)
            {
                delete _ptr;
                delete _reference_count;
            }
        }

    private:
        T* _ptr;
        size_t* _reference_count;
        void removeCount()
        {
            if (_ptr)
            {
                --(*_reference_count);
                if ((*_reference_count) == 0)
                {
                    delete _ptr;
                    delete _reference_count;
                }
            }
        }
    };
```

- weak_ptr

弱引用主要用于解决循环引用。lock()，可提升为shared_ptr指针。

- unique_ptr

  auto_ptr在进行赋值时，会移交对象的管理权限，而原指针变为空指针，这样就必须进行非空判断，而unique_ptr则不能直接进行赋值，需要使用移动语义move，否则会出现编译错误。

C++11特性

- auto类型推导，用于从初始化表达式中推断出变量的数据类型，auto实际上实在编译时对变量进行了类型推导，所以不会对程序的运行效率造成不良影响
- nullptr 解决原来C++中NULL的二义性问题而引进的一种新的类型，因为NULL实际上代表的是0。
- decltype编译期类型推导，由于typeid运算符采用的是运行时类型查询，其会为每个类型产生一个type_info类型的数据，typeid则返回其中的数据，RTTI会导致运行时效率降低，且在泛型编程中需要编译期确定类型，则RTTI无法满足要求。
- 序列for循环
- Lambda表达式 [函数对象参数]（操作符重载函数参数）->返回值类型{函数体}

强制类型转换

- static_cast<T*>(content)（编译期处理）静态类型转换，主要用于内置类型数据类型之间的转换，但父类与子类间转换不保证安全性，其中子类指针转换成父类指针是安全的；但父类指针转换成子类指针是不安全的。
- const_cast<T*>(content)（编译期处理）去常转换，去掉/添加类型的const或volatile属性。
- dynamic_cast<T*>(content)(运行时执行)动态类型转换，只用于父类与子类间安全类型转换，转换成功返回类型的指针或引用，失败返回NULL，且父类一定要含有虚函数。
- reinterpret_cast<T*>(content)强制类型转换，转换过程仅仅是bit位的拷贝，不保证安全性。

写程序区分大小端

原理：联合体union的存放顺序是所有成员都从低地址开始存放，而且所有成员共享存储空间

```
void IsBigEndian()
{
    union temp
    {
        short int a;
        char b;
    }temp;
    temp.a = 0x1234;
    if( temp.b == 0x12 )//低字节存的是数据的高字节数据
    {
        cout<<"大端模式"<<endl;
    }
    else
    {
        cout<<"小端模式"<<endl;
    }
}
```

i++是否原子操作

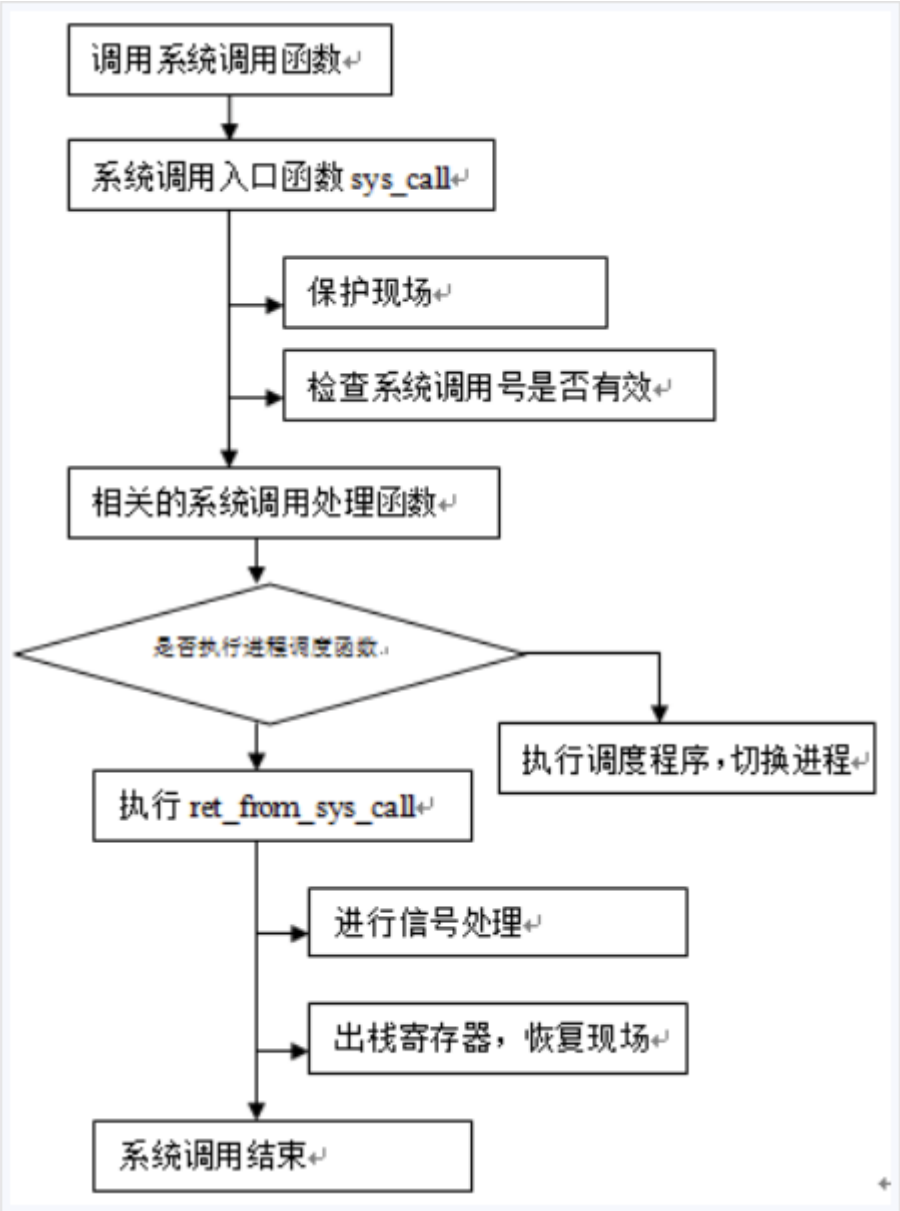
理论上++i更快，实际与编译器优化有关，通常几乎无差别。

```
//i++实现代码为:
int operator++(int)
{
    int temp = *this;
    ++*this;
    return temp;
}//返回一个int型的对象本身

// ++i实现代码为:
int& operator++()
{
    *this += 1;
    return *this;
}//返回一个int型的对象引用
```

系统调用过程

首先指令流执行到系统调用函数时，系统调用函数通过int 0x80指令进入系统调用入口程序，并且把系统调用号放入%eax中，如果需要传递参数，则把参数放入%ebx，%ecx和%edx中。进入系统调用入口程序（System_call）后，它首先把相关的寄存器压入内核堆栈（以备将来恢复），这个过程称为保护现场。保护现场的工作完成后，开始检查系统调用号是不是一个有效值，如果不是则退出。接下来根据系统调用号开始调用系统调用处理程序（这是一个正式执行系统调用功能的函数），从系统调用处理程序返回后，就会去检查当前进程是否处于就绪态、进程时间片是否用完，如果不在就绪态或者时间片已用完，那么就会去调用进程调度程序schedule()，转去执行其他进程。如果不执行进程调度程序，那么接下来就会开始执行ret_from_sys_call，顾名思义，这这个程序主要执行一些系统调用的后处理工作。比如它会去检查当前进程是否有需要处理的信号，如果有则去调用do_signal()，然后进行一些恢复现场的工作，返回到原先的进程指令流中。至此整个系统调用的过程就结束了。



析构虚函数与构造非虚函数

虚析构函数：保证子类能够完整析构，避免内存泄漏，若不是虚函数，则调用的函数依赖于指向的静态类型，即Base。

纯虚析构函数：由于析构函数、构造函数和其他函数不一样，在调用时，编译器需要产生一个调用链，需要实现纯虚析构函数的函数体。纯虚函数语义表明类变成了抽象类，不能产生对象，但并不意味着不能指定函数体。

- 虚函数是动态绑定的，也就是说，使用虚函数的指针和引用能够正确找到实际类的对应函数，而不是执行定义类的函数。
- 构造函数不能是虚函数。而且，在构造函数中调用虚函数，实际执行的是父类的对应函数，因为自己还没有构造好, 多态是被disable的。
- 析构函数可以是虚函数
- 将一个函数定义为纯虚函数，实际上是将这个类定义为抽象类，不能实例化对象。纯虚函数通常没有定义体，但也完全可以拥有。析构函数可以是纯虚的，但纯虚析构函数必须有定义体，因为析构函数的调用是在子类中隐含的。非纯的虚函数必须有定义体，不然是一个错误。
- 派生类的override虚函数定义必须和父类完全一致。除了一个特例，如果父类中返回值是一个指针或引用，子类override时可以返回这个指针（或引用）的派生。例如，在上面的例子中，在Base中定义了 virtual Base*

clone(); 在Derived中可以定义为 virtual Derived* clone()。

extern C作用

- 被 extern 限定的函数或变量是 extern 类型的
- 被 `extern "C"` 修饰的变量和函数是按照 C 语言方式编译和链接的，`extern "C"` 的作用是让 C++ 编译器将 `extern "C"` 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和C语言库中的符号进行链接的问题。

只能在堆上与只能在栈上

静态建立一个类对象，是由编译器为对象在栈空间中分配内存，是通过直接移动栈顶指针，挪出适当的空间，然后在这片内存空间上调用构造函数形成一个栈对象。使用这种方法，直接调用类的构造函数

动态建立类对象，是使用new运算符将对象建立在堆空间中。这个过程分为两步，第一步是执行operator new()函数，在堆空间中搜索合适的内存并进行分配；第二步是调用构造函数构造对象，初始化这片内存空间。这种方法，间接调用类的构造函数。

- 堆：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。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。所以，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。

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

- 栈：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。

```
class A {
private:
    void* operator new(size_t t) {}
    void operator delete(void* ptr) {}
public:
    A() {}
    ~A() {}
};
```

STL迭代器失效

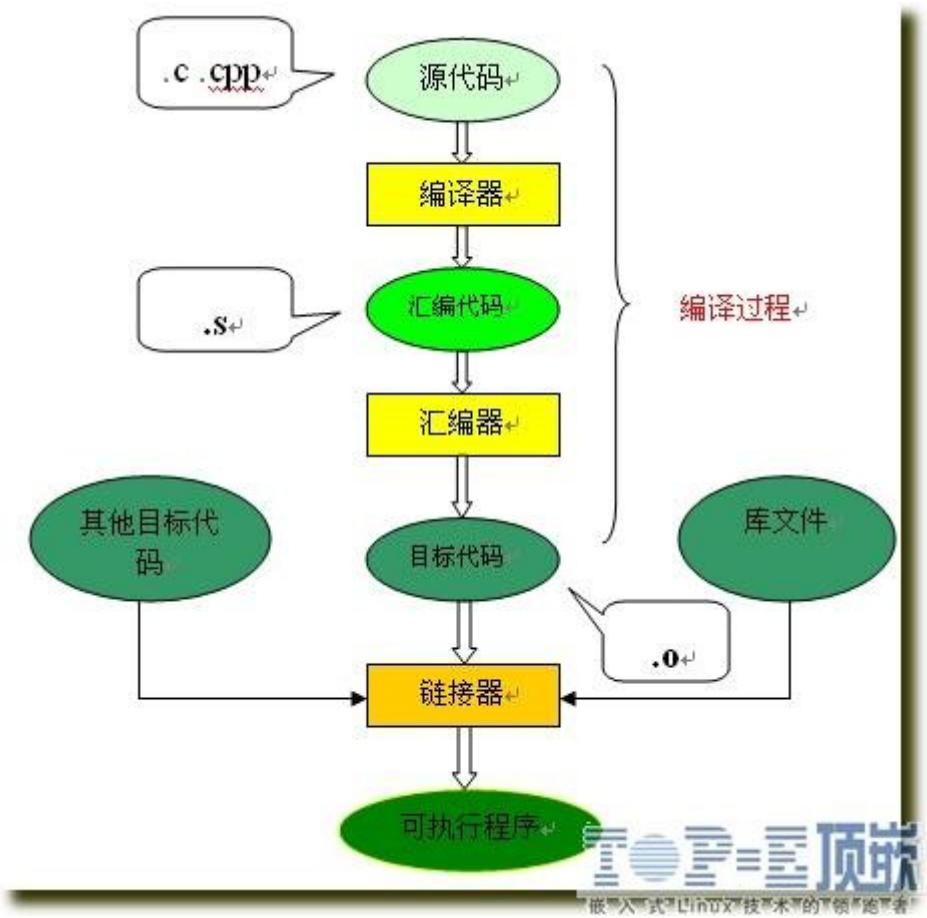
- 对于list链表，删除元素时会导致指向该元素的迭代器失效，而其他元素不受影响；
- 对于vector元素的插入、删除操作会导致指向该元素以及后面的元素都迭代器失效。当插入（push_back）一个元素后，end操作返回的迭代器肯定失效；扩容后导致迭代器失效；当进行删除操作（erase，pop_back）后，指向删除点的迭代器全部失效；指向删除点后面的元素的迭代器也将全部失效。

重载、重写（覆盖）、重定义（隐藏）

- 重载：（同一可访问区内）函数名相同，但是它的参数表列个数或顺序，类型不同。但是不能靠返回类型来判断
- 重写（覆盖）：派生类重新定义基类的虚函数，函数名相同，参数相同，返回值相同/协变，重写函数的访问修饰符可不同。
- 重定义（隐藏）：分别位于派生类与基类，函数名相同，参数不同，此时无论virtual，参数相同需无virtual，返回值可以不同。

C++编译过程

C语言的编译链接过程要把我们编写的一个c程序（源代码）转换成可以在硬件上运行的程序（可执行代码），需要进行编译和链接。编译就是把文本形式源代码翻译为机器语言形式的目标文件的过程。链接是把目标文件、操作系统的启动代码和用到的库文件进行组织，形成最终生成可执行代码的过程。



- 编译预处理，读取c源程序，对其中的伪指令（以# 开头的指令）和特殊符号进行处理。
    - 宏定义指令替换
    - 条件编译指令
    - 头文件包含指令
    - 特殊符号处理，LINE表示当前行号，FILE表示当前被编译的C源程序名称。

- 编译、优化阶段：经过预编译得到的输出文件中，只有常量；如数字、字符串、变量的定义，以及C语言的关键字，如main, if , else , for , while , { , } , + , - , * , \ 等等。编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。
- 汇编：汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个C语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。
- 链接过程：链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。
    - 静态链接：在这种链接方式下，函数的代码将从其所在的静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。
    - 动态链接：在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

  对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小，并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链接要优越。在某些情况下动态链接可能带来一些性能上损害。

C++虚函数机制

- 虚函数的实现，主要依靠虚指针与虚表实现；
- 一个类中，虚函数本身、成员函数（静态与非静态）和静态数据成员都是不占用类对象的存储空间的；
- 每个类都有虚指针和虚表；
- 如果不是虚继承，那么子类将父类的虚指针继承下来，并指向自身的虚表（发生在对象构造时）。有多少个虚函数，虚表里面的项就会有多少。多重继承时，可能存在多个的基类虚表与虚指针；
- 如果是虚继承，那么子类会有两份虚指针，一份指向自己的虚表，另一份指向虚基表，多重继承时虚基表与虚基表指针有且只有一份。

深浅拷贝、零拷贝

- 在未显示调用构造函数的情况下，系统会调用默认的拷贝构造—浅拷贝，完成对成员的按位赋值，若类成员中存在指针变量，则有两个指针指向同一个地址，则会析构两次，而导致指针悬挂现象，所以必须采用深拷贝。深拷贝与浅拷贝的区别就在于深拷贝会在堆内存中另外申请空间来储存数据，从而也就解决了指针悬挂的问题。
- 零拷贝：在引入右值引用，转移构造函数，转移复制运算符之前，通常使用push_back()向容器中加入一个右值元素（临时对象）的时候，首先会调用构造函数构造这个临时对象，然后需要调用拷贝构造函数将这个临时对象放入容器中。原来的临时变量释放。这样造成的问题是临时变量申请的资源就浪费。引入了右值引用，转移构造函数后，emplace_back()右值时就会调用构造函数和转移构造函数。

内存对齐及作用

- 从0位置开始
- 数据类型自身的对齐值，变量的起始位是自身的整数倍
- 结构体总体大小是其最大元素的整数倍，不足补齐。
- 结构体中的结构体，从结构体中最大元素的整数倍开始。
- 若加入pragma pack(n) 取n和变量自身大小较小者。

C++ core dump

- 调试：
  - 编译时添加编译参数 -g 添加调试信息
  - ulimit -a 查看 core文件；修改core文件大小为无限大 ulimit -c unlimited
  - 运行程序 ./test
  - 调试core文件 gdb ./coretest core (bt 查看出错的堆栈信息，p 打印变量 n下一步 q退出 b设置断点 l 显示源码，并且可以看到对应的行号)
- 原因
  - 内存访问越界
  - 多线程程序使用了线程不安全的函数。
  - 多线程读写的数据未加锁保护。
  - 非法指针
  - 堆栈溢出

模板编程

- 未实例化的模板其成员变量不可用，即使是静态变量、枚举变量
- 成员函数只有使用才会被实例化（空间和时间效率；如果只"实例化"真正使用的函数，template就能够支持哪些原本可能造成编译期错误的类型）
- 模板在实例化之前只进行有限的类型检验。

new（运算符）和delete是如何实现的，new 与 malloc（库函数）的异同处

- new运算符不可重载，其主要调用operator new分配足够的空间，并调用相关对象的构造函数
- operator new只分配要求的空间，可被重载，返回类型必须被声明为void*，重载时，第一个参数必须为表达分配空间的大小（size_t）,可带其他参数。
- new分配成功返回对象类型的指针，符合类型安全性的操作符，而malloc内存分配成功则返回void*，需要经过强制类型转换成所需的类型。
- new内存分配失败抛出bac_alloc异常，malloc分配内存失败返回NULL。new无需显示指定内存大小，编译器会根据类型信息进行计算，而malloc需要指出内存大小。
- new较malloc多了调用构造函数的过程，new可用于对数组的处理，而malloc则只负责分配一块原始的内存。
- malloc能够直观的重新分配内存，可以使用realloc进行内存重新分配实现内存扩容。realloc先判断当前指针是否含有足够的连续空间，若有，则原地扩容，若没有，则分配到新区域。

shared_from_this

如果使用shared_ptr指针来管理对象，则必须初始化后都使用智能指针，避免出现使用已释放的对象。若需要在类对象内部使用类对象的shared_ptr，有两种方法：

- 类对象的外部shared_ptr作为参数传给类的需要引用类对象自身的函数--显然这种方法很丑，且不是所有的情况都可行。
- 类对象自身存储某种信息，在需要自身shared_ptr时来产生一个临时shared_ptr（使用weak_ptr）。
- 何时初始化weak_ptr？因为类对象生成时还没有生成相应的用来管理该对象的shared_ptr.boost中实现首先生成类 A： 会依次调用 enable_shared_from_this 的构造函数（定义为 protected ），以及类 A 的构造函数。在调用 enable_shared_from_this 的构造函数时，会初始化定义在 enable_shared_from_this 中的 weak_ptr （调用其默认构造函数），这时这个 weak_ptr 是无效的（或者说不指向任何对象）。接着：外部程序会把指向类 A 对象的指针作为初始化参数来初始化一个 shared_ptr 。
- 首先需要注意的是：这个函数仅在shared_ptr的构造函数被调用之后才能使用。原因是enable_shared_from_this::weak_ptr并不在enable_shared_from_this构造函数中设置，而是在shared_ptr的

构造函数中设置。