- **C++**
  - 指针
    - 常量指针 底层const
    - 指针常量 顶层const
    - 函数指针
    - 智能指针
    - 野指针和悬浮指针
  - 引用
  - static 关键字
  - inline
  - new delete和malloc free区别
  - 内存管理
    - 内存模型
    - 堆和栈
    - 内存泄漏
    - 内存对齐
  - 运算符重载
    - 前置++ 与后置++
  - 线程安全
    - 信号量
  - 类
    - Struct和类的区别
    - 多重继承
    - 重载和重写（覆盖）的区别
    - 构造函数和析构函数
    - 多态的实现
    - 虚函数和虚函数表
    - 虚函数和纯虚函数区别
    - 虚析构函数
    - 运算符重载
  - 强制类型转换
  - 字符串
  - 面向对象和面向过程
    - 面向对象
      - 三大特性
    - 面向过程语言 C语言
  - STL
    - 组件
    - 容器
    - 迭代器
  - lambda表达式
    - 并发

C++

指针

指针可以为空，不指向有效的地址

常量指针 底层const

指针指向只读对象，不能改变这个对象的值 *靠近变量名

指针常量 顶层const

强调指针只能在定义时初始化，其他地方不能改变，强调的是指针的不可改变性 *在int和const之间

函数指针

指向函数的指针变量，可以存储函数的地址 然后运行时动态选择调用的函数

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

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

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

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

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

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

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

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

    return 0;
}
```

可以将函数指针传给其他函数使用，以便在适当的时候调用
虚函数和函数指针结合使用 可以类似于多态

函数指针和指针函数的区别
指针函数是返回指针类型的函数

智能指针

用于管理动态内存的对象 避免内存泄漏和方便资源管理
auto_ptr自动指针任何时候只能有一个智能指针对象指向那块内存区域，不能有两个对象同时指向那块内存区域
unique_ptr独占智能指针
对动态分配的单一对象的所有权的独占管理　使用 std::unique_ptr，避免显式 delete，指针会在超出作用域时自动释放
shared_ptr(共享智能指针)

允许多个指针共享同一块内存资源，内部使用引用计数来跟踪对象被共享的次数，当计数为零时，资源被释放,会存在循环引用的问题 循环引用指的是不同类实例的共享指针互相指对方
weak_ptr 弱引用智能指针
解决循环引用问题，它的构造和析构不会引起引用记数的增加或减少

野指针和悬浮指针

野指针指向被释放或者无效的内存地址的指针
悬浮指针是指已经被销毁对象的引用 应该避免在函数中返回局部变量的引用。

引用

必须在声明时初始化，而且不能改变后续绑定的对象，没有空引用的概念

常量引用 表示引用的值不能通过引用修改

static 关键字

静态函数 static在类内部使用 不是类的实例 无需创建对象，不能直接访问非静态成员变量和非静态成员函数

静态成员变量 类所有实例共享一个 然后需要在类外部单独定义

静态局部变量 生命周期在整个程序，但只在声明他的函数可见

inline

将内联函数直接编译插入被调用的地方 减少压栈 跳转和返回 减少函数调用时的开销 不能存在循环，过多的条件判断 不能过大

new delete和malloc free区别

new分配时无需指定内存块大小 malloc需要显式指出所需内存

new/delete 是C++ 运算符，后者是标准库函数

前者是在自由存储区上动态分配对象 malloc是在堆上动态分配

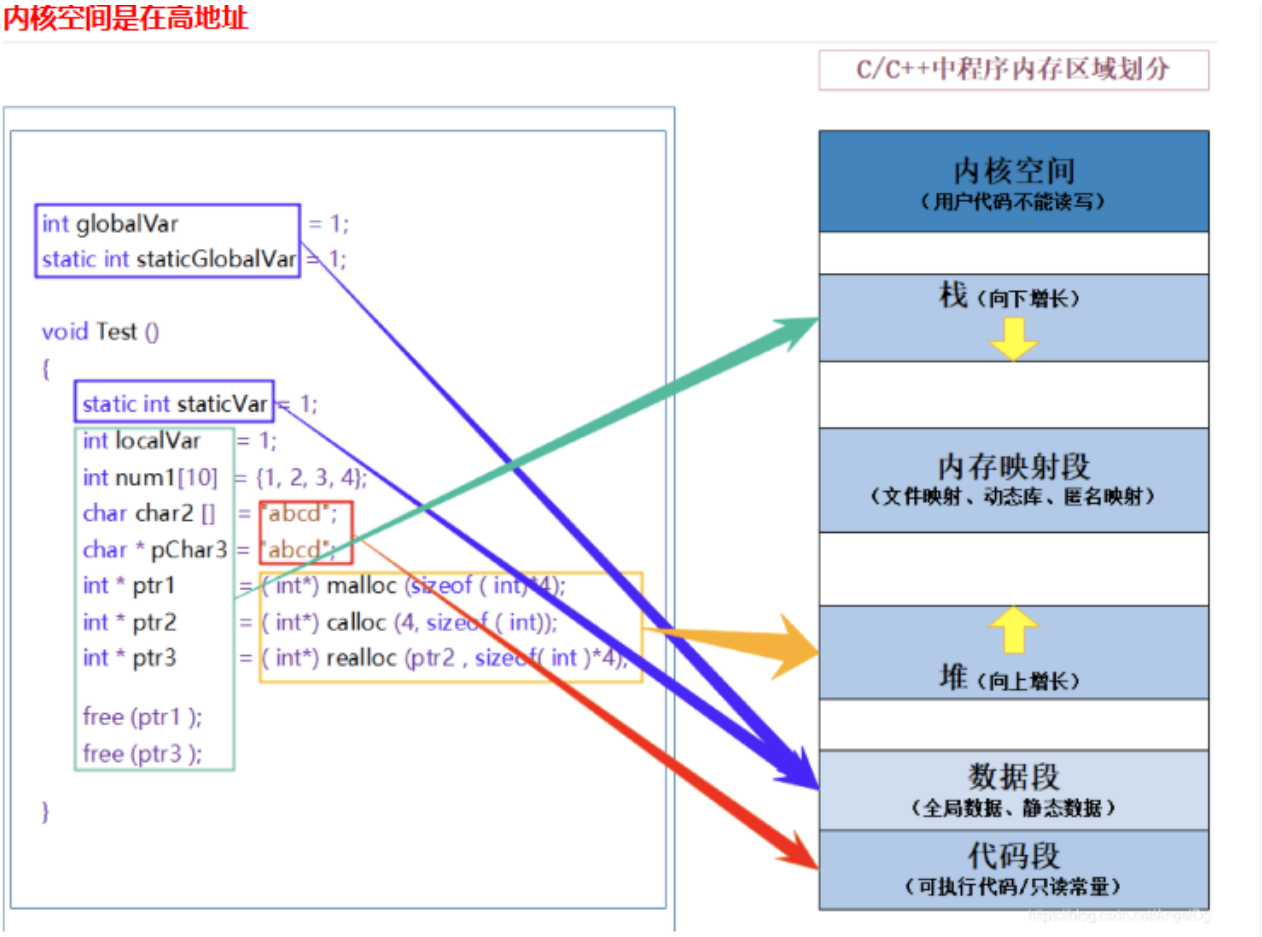
new返回的是具体类型的指针，不需要进行类型转换 malloc返回的是void* 需要进行类型转换 因为malloc不知道分配类型的用途

delete会调用析构函数 free不会 只是简单释放内存块

delete释放内存块后指针会设置成nullptr free不会修改指针的值

内存管理

内存模型



内存映射段是高效的I/O映射方式，用于装载一个共享的 **动态内存库**。用户可使用系统接口创建共享共享内存，做进程间通信

自由存储是 C++ 中通过 new 和 delete 动态分配和释放对象的抽象概念。基本上，所有的 C++ 编译器默认使用堆来实现自由存储。也就是说，默认的全局运算符 new 和 delete 也许会使用 malloc 和free 的方式申请和释放存储空间，这时自由存储区就位于堆上。但程序员也可以通过重载操作符，改用其他内存来实现自由存储，例如全局变量做的对象池，这时自由存储区就不位于堆上了。

堆和栈

栈是有限的内存区域 存储局部变量和函数调用信息什么的 自动释放
堆是动态分配的内存区域 手动分配和释放

内存泄漏

堆内存泄露
手动分配完忘记释放
系统资源泄露
比如Bitmap，Socket等没有使用相应的函数释放掉
没有将基类的析构函数定义为虚函数
基类指针指向子类对象时 如果没有虚函数，那么子类析构不会被调用 释放不正确

使用智能指针解决内存泄漏

内存对齐

比如结构体，每个成员按照他们被声明的顺序存储，4字节对齐
对齐的根本原因在于**CPU访问数据的效率问题**

运算符重载

前置++ 与后置++

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

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

1、为什么后置返回对象，而不是引用

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

2、为什么后置前面也要加const

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

原因：

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

3、处理用户的自定义类型

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

线程安全

a++和int a=b不是线程安全的 他在汇编指令不是原子指令

信号量

binary_semaphore

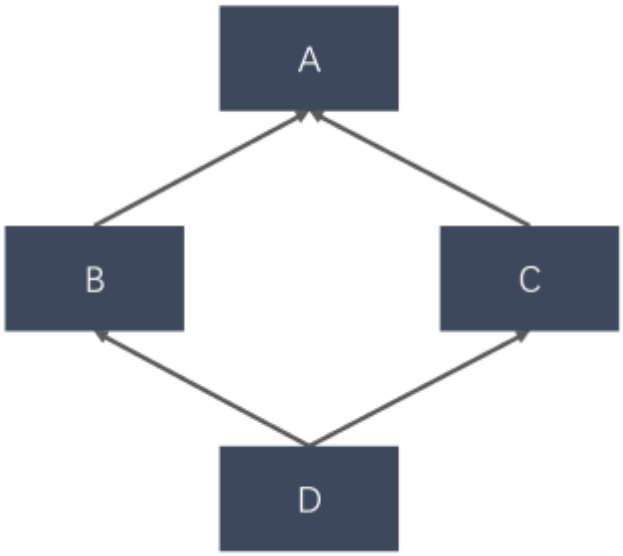
类

Struct和类的区别

前者默认public 后者默认private

多重继承

一个类可以从多个基类（父类）继承属性和行为。在C++等支持多□继承的语言中，一个派生类可以同时拥有多个基类
会引入问题 比如**菱形继承问题** 比如A类的方法在C类调用会有二义性，所以产生虚继承，即B和C虚继承A这样就可以共享A



重载和重写（覆盖）的区别

重写不能为private方法，重写的权限一定大于被重写方法的权限

构造函数和析构函数

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

多态的实现

```
class Shape{
    public:
        virtual void draw() const{

        }
}
class Circle:public{
    public:
        void draw() const override{

        }
}
```

虚函数和虚函数表

虚函数的作用主要是实现多态，虚函数允许派生类中重新定义基类定义的函数，然后指针能够根据实际对象动态绑定 在**基类**

虚函数表是一个数据结构 每个类都有一个虚表，包含了该类虚函数的指针，每个对象都包含一个指向其类虚表的指针为虚指针

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

虚函数和纯虚函数区别

虚函数 基类可以有函数声明和实现 派生类可以选择是否重写

纯虚函数 没有函数体 只有函数声明 无实现 派生类必须选择重写，不然也会变为抽象类，同时这个抽象类不能被实例化，只能派生

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

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

虚析构函数

主要是指针删除对象时 能够正确调用析构函数，会在析构函数前加virtual，这样不会导致内存泄漏 实现多态性

构造函数不能虚构造，因为构造函数不涉及多态性。
普通函数也不用虚函数
静态成员函数不用，因为对于每个类来说只有一份，所有对象都共享一份
友元函数不行，因为他不能继承

运算符重载

强制类型转换

static_cast
没有运行类型检查 直接转换 不安全
上行转换 派生类-》基类 安全
下行转换 不安全

dynamic_cast
下行转换 会类型检查（检查虚函数表，如果没有虚函数不能通过）

reinterpret_cast
随便转换 但是开销移植差

const_cast
去掉const属性

字符串

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

strlen(const char *str)

strcat(const char *strsest,const char *strsrc)作用是把src所指字符串添加到dest结尾处。

strcmp(const char *str1,const char *str2)大于返回正，小于返回负

面向对象和面向过程

面向对象

C++是面向对象的语言

访问权限 通过public protected和private来表示权限和继承

三大特性

继承 有实现继承 接口继承 可视继承
封装 将方法公开，不可见的信息隐藏
多态 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（重载载实现 编译时多态，虚函数实现运行时多态）

实现多态有两种方式 一种是**覆盖**（重写）（子类重新定义父类虚函数的做法） 二是**重载** 指允许存在多个同名函数 参数列表不同或者返回类型不同

面向过程语言 C语言

性能比面向对象高，因为类调用时需要实例化，开销比较大

STL

组件

广义 算法 容器 迭代器
详细 容器 算法(比如sort) 迭代器(iterator) 仿函数 适配器 空间配置器
容器去通过空间配置器取得数据存储空间
算法通过迭代器存储容器中的内容
仿函数协助算法完成不同策略的变化
适配器可以修饰仿函数

容器

vector
动态数组实现 线性连续空间 插入和删除代价高

size()和capacity()后者永远大于前者 后者是翻倍的

list双向链表实现 非连续空间

deque双向链表 分段的线性连续空间 有map中控器控制

set红黑树实现

map红黑树变体的平衡二叉树结构 因为红黑树有自动排序功能 所以就是有序的 插入和删除的效率是O(logN)

unorder_map哈希表

pair

栈和队列 底层使用deque

heap大根堆小根堆 和优先队列

迭代器

对于序列容器vector，deque来说，使用erase后，后边的每个元素的迭代器都会失效，后边每个元素都往前 移动一位，erase返回下一个有效的迭代器。

对于关联容器map，set来说，使用了erase后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元 素，不会影响下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。

对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的迭代器，因此上面两种 方法都可以使用。

lambda表达式

```
[capture list] (parameter list) -> return type {function body }
// [捕获列表] (参数列表) -> 返回类型 {函数体 }
// 只有 [capture list] 捕获列表和 {function body } 函数体是必选的
auto lam =[]() -> int { cout << "Hello, World!"; return 88; };
auto ret = lam();
cout<<ret<<endl; // 输出88
```

并发