**面向对象的特性**

**封装**：可以隐藏实现细节，用类进行封装，实现高内聚、低耦合。内聚指类内部之间的关联程度，耦合指类之间的关联程度。可设置访问级别（private、public、protected）。

**继承**：子类可以继承并拓展基类的功能。C++中有公有继承、保护继承和私有继承，可以修改派生类中基类成员的访问权限。不能被继承的成员：构造函数、析构函数、友元函数、静态成员（数据或函数）。

**多态**：不同对象接到同一消息产生不同的行为。编译时多态：函数重载和模板；运行时多态：继承和虚函数。（多态三个条件：继承、子类重写父类函数、父类引用变量指向子类对象）

**虚函数、纯虚函数、虚继承**

**虚函数**：编译时不确定而运行时再确定的函数，每个类在编译时生成自己的虚函数表，存储虚函数的地址。虚函数表存在.rodata区（只读数据段常量区），虚函数表指针存放在对象内存最前面的位置。构造函数、静态函数不能为虚函数，析构函数最好为虚函数。内联函数可以是虚函数。

（若为多继承，则对象内存先后包括两个父类的两个空间，每个空间最前面为继承该父类的虚指针，在两个空间之后才是自己的成员变量（对于深度继续扩展时也是如此））

**纯虚函数**：前面virtual，后面加= 0。带有纯虚函数的类为抽象类，要求每个派生类定义实现方法，抽象类不能定义对象。

**虚继承**：虚基类在继承体系中只会存在一次，在菱形继承中（B和C均虚继承A，D继承B和C），解决内存重复的问题，避免访问冲突。

（虚继承时会分为两部分，先是自己的虚函数表指针，再是自己的虚基类表指针、自己的成员变量，最后是基类的虚函数表指针、基类成员变量。菱形继承：B的虚函数表指针、B的虚基类表指针、B的成员变量、C的虚函数表指针、C的虚基类表指针、C的成员变量、D的成员变量、A的虚函数表指针、A的成员变量）

<https://www.cnblogs.com/kekec/archive/2013/01/27/2822872.html>

final关键字：在函数末尾用于避免子类重写，

override关键字：在函数末尾用于保证本函数是重写的。

delete this：delete栈对象是未定义的行为。必须保证其在堆上、确保调用的其他成员函数不会读取this对象。

**重写、重载与隐藏**

**重载**：同一作用域下声明几个同名函数，函数的形参列表必须不同，返回值可以不同。（静态函数可以和普通成员函数构成重载）

**重写**：子类重写父类虚函数，参数名称相同，返回值相同或协变（即返回值为指针或引用时可以为子类型）。

**隐藏**：子类有和父类同名函数，对父类函数进行隐藏。（父类指针或引用指向子类对象调用的是父类函数）

**重载运算符**

// 二目运算符左侧

A operator+(const B& other);

// 二目运算符右侧

friend A operator+(const B& left, const A& right);

// 或者再类外定义为全局函数

A operator+(const B& left, const A& right);

// 左自增运算符

A& operator++() {

// 自增实现

return \*this;

}

// 右自增运算符

A operator++(int) {

A res(\*this);

++(\*this);

return res;

}

不能重载的运算符：

成员访问运算符(.)、成员指针访问运算符(->)、域运算符(::)、三目条件运算符(?:)、长度运算符(sizeof)。

运算符优先级：单目>双目（算术>移位>比较>位运算）>三目>赋值

**内联函数、友元函数、静态变量**

**内联函数**：inline修饰，编译时会在调用内联函数的地方展开从而提升效率（可以是虚函数，当使用非多态调用时编译器选择内联）。与宏定义的区别：宏定义是单纯文本替换，inline函数编译器会进行类型检查和语法判断，控制参数和返回值类型。inline函数是在编译阶段处理的而define是在预处理阶段完成的。

**友元函数**：类的友元函数可以定义在类外部，但有权访问类的所有私有成员和保护成员。

**静态变量**：static修饰，分为全局变量、类的静态变量和函数的局部静态变量。全局变量和类的静态变量在main执行之前分配内存并初始化，局部静态变量在第一次使用时分配内存并初始化。

**const关键字，constexpr、const数组、指针常量、常量指针**

const修饰普通变量即定义了一个常量，存储在静态存储区或栈区（根据是全局变量还是局部变量而定）。

constexpr修饰变量时，在编译期计算。

const数组：表示数组的每个元素都是常量（即常量指针常量）。

**const默认作用于其左边的东西，否则作用于其右边的东西。**

const int \*p / int const \*p为**常量指针**，指针指向的位置是一个常量，指针可以指向别的位置，但不能修改指向的内容。

int\* const p / const\* int p为**指针常量**，指针本身是一个常量，指针不能指向别的位置，但可以修改指向的内容。

const修饰引用形参：传递外部实参本身无需拷贝，同时参数是const无法修改，保证了外部数据的安全性。（非const引用/指针不能接收const引用/指针，但const引用/指针可以接收非const引用/指针）。

const修饰普通形参/返回值（如int）是没有意义的，同理指针常量（\*const）也是没有意义的，因为他们都是局部变量，不会影响原/返回数据。

const修饰成员函数（放在成员函数括号后面），表示不能对类的任何成员进行修改（this指针指向空间的内容不可被修改，即将this指针类型由指针常量变为常量指针常量）。

volatile关键字：表示易变的，每次访问该变量均需要从内存读取，避免编译器的优化。

**内存中存储区的分布**

**栈**：存放函数的参数、局部变量、返回数据等值，由编译器自动释放。栈的生长方向向下，内存地址由高到低。底层硬件提供支持，存取速度较快。（VS默认1M）

**堆**：由开发人员进行申请和释放（在C#中为引用类型的对象，由CLR释放），堆的生长方向向上，内存地址由低到高。

**静态存储区**：存储程序中明确被初始化的全局变量、静态变量（全局静态变量和局部静态变量）。

**常量区**：存放程序运行期间不能被改变的常量。

**代码区**：存放程序的代码。

**malloc和new的区别？**

malloc是库函数，需要指定分配的字节，而new是运算符，会自行计算。

malloc不会调用构造函数，而new会调用构造函数和析构函数。

malloc返回的是void\*，需要自行转换为具体类型的指针。

malloc需要free来释放，而new需要delete来释放。

delete和delete[]的区别：delete[]会调用数组所有对象的析构函数，而delete只会调用第一个对象的析构函数，对于普通类型无区别，对于自定义析构函数的复杂对象有区别。

new(nothrow)：在分配失败时不再抛出异常，而是返回空指针。

placement new：new (address) ClassConstruct(...);

在用户指定的已经预先分配好的位置上分配内存来构建对象。其中address为指定的内存地址，ClassConstruct为对象的构造函数。

**四种强制类型转换符**

**static\_cast**：适用于低风险转换，如整型和浮点型/字符型之间的转换

**reinterpret\_cast**：进行各种不同类型的指针之间、不同类型的引用之间以及指针和能容纳指针的整数类型之间的转换。转换时，执行的是逐个比特复制的操作。

**const\_cast**：进行去除const属性的转换。

**dynamic\_cast**：将多态基类（基类需要实现虚函数）指针或引用转换为派生类指针或引用，能够检查转换的安全性。（转换失败时对指针返回空指针，对引用抛出异常）

**智能指针**

（auto\_ptr：较早版本的智能指针，在进行拷贝和赋值时，新指针直接接管旧指针资源并将旧指针指向空，在访问旧指针会出现问题。）

**unique\_ptr**：是auto\_ptr的改良版，不能赋值也不能拷贝，保证一个对象同一时间只有一个智能指针。

**shared\_ptr**：一个对象可以有多个智能指针，每多一个引用计数加一，当引用计数为0时自动释放对象。（使用make\_shared可以将两次内存分配减为一次，使指针与引用计数绑定，效率更高。但构造函数是保护或私有时无法使用，以及对象内存可能会无法及时回收，当强弱引用均减为0时才回收）

**weak\_ptr**：避免shared\_ptr的循环引用，可以指向shared\_ptr，并且不会改变shared\_ptr的引用计数。

（unique\_ptr实现原理：禁止拷贝构造函数和拷贝赋值运算符，使用 = delete）

（shared\_ptr实现原理：维护一个引用计数[int\*]，构造函数中指针引用计数设置为1；拷贝构造函数中指针引用计数加1；拷贝赋值运算符中对左操作数指针引用计数减1，对右操作数引用计数加1；析构函数中对引用计数减1；如果引用计数减至0则delete该对象）

**野指针**

野指针产生的原因：指针没有初始化、指针指向的内存被释放但是没有置空、指针超出了数组的边界。

解决野指针的方法：初始化指针、指针指向内存被释放后置空/使用智能指针、避免指针超出数组边界。

**字节对齐**

计算机内存空间按照字节划分，各类型数据按照一定的规则在空间上排列，而不是顺序地一个接一个地排放，这就是对齐。

为什么要字节对齐：优化结构，提高内存访问速度，硬件平台限制。

char(1)、short(2)、int(4)、long(4/8[Linux 64位])、long long(8)、float(4)、double(8)、指针(4/8)。

空类占据1字节的大小。

类的内存对齐：按照最大字节的数据类型进行对齐。

**C++和python的区别？编译型语言和解释性语言的区别？**

编译性语言在程序执行之前进行编译，而解释性语言在执行程序时编译。Python开发效率高，但执行效率低，容易跨平台。

**C++编译的过程？**

预处理：处理源程序中伪指令（以#开头的指令）。（.i）

编译：对预处理后的文件进行词法分析、语法分析和语义分析，生成相应的汇编代码文件。（.s）

汇编：将汇编代码转变成机器可执行的二进制代码（机器码）并生成目标文件。（.o）

链接：解决多个文件之间符号引用的问题。

模板实现为什么放在.h中：编译器为模板生成实例的必要条件是知道模板的实现、知道模板参数的具体类型，在编译时.cpp互相独立，链接器无法链接。

**STL**

包括**容器**、**算法**、**迭代器**（、仿函数、适配器、分配器）。

迭代器：输入迭代器、输出迭代器、前向迭代器、双向迭代器。随机访问迭代器。

算法：sort、reverse、lower\_bound、upper\_bound。

sort：当n小于40时使用插入排序；大于40时，使用快排；当递归深度大于1.5logn时使用堆排。快排时对基准值的优化：均匀取9个点，每三个一组取中间，以最终中间值作为基准。

vector在存储空间满时会申请一块更大的内存，让原内存数据拷贝到新内存里再释放原内存，新的容量为原容量的1.5倍或2倍（GCC）。

vector迭代器失效情况：erase/insert时指向删除节点及以后的节点的迭代器都失效；push\_back时end迭代器失效，若扩容则begin也失效。

vector占用内存：32个字节，即三个指针的大小（\_First, \_Last, \_End）加上一个分配器的大小。

**模板**

泛化、局部特化、完全特化。

template<class T> // 完全泛化

void fun(T a)

{

}

template<class T> // 部分特化

void fun(T\* a)

{

}

template<>

void fun<char\*>(char\*) // 完全特化

{

}

**右值引用、std::move、std::forward**

右值：通常只能放在等号右边的变量，临时的，表达式结束后就不再存在。

右值引用：&&

常量左值引用：const &，既可以访问左值也可以访问右值。

引用折叠规则：右值引用折叠到右值引用上仍然是右值引用。其余只要折叠中有一个左值引用就是左值引用。

std::move：可以将左值变为右值，将对象的状态或者所有权转移，避免拷贝构造、没有内存的搬迁或者内存拷贝。（对应移动构造函数、移动赋值运算符函数，与拷贝构造函数和拷贝赋值运算符函数相对立）

std::forward：完美转发，输入参数原封不动传递到下一个函数中，即输入是左值传递的也是左值，输入是右值传递的也是右值。

**explicit关键字、extern关键字**

explicit：用于类的单参数构造函数之前，避免隐式类型转换。（如Fool(int){}可以通过Fool a = 1实现构造或通过a==1判断）

extern：应用于变量或函数，说明其具有外部链接属性，即这个符号在别处定义，在此处声明，声明可以多次（正常而言声明就是定义）。

extern "C"：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。由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

**decltype关键字与auto关键字**

decltype关键字用于在编译时期进行自动类型推导，decltype(exp)根据表达式exp来推断类型，若exp为函数则为函数返回值类型，若exp被括号包围或为左值则为exp类型的引用。

auto关键字要求变量初始化，根据等号右边的初始值推导出变量的类型，将变量类型和初始值绑定。

**Lambda表达式和闭包**

[捕获列表](参数列表) 可变规则 异常说明 ->返回类型 {函数体}

捕获列表：[]不捕获任何变量、[var]值传递捕获变量var、[&var]引用传递捕获变量var、[=]值传递捕获父作用域所有变量、[&]引用传递捕获父作用域所有变量。

可变规则：mutable修饰符，可以取消Lambda函数的常量性。

异常说明：noexcept修饰符，可以指示 Lambda 表达式不会引发异常。

返回类型：返回类型可自动推导，因此可以不必指定。

函数体：与普通函数主体相类似。

仿函数：用一个类重载()，可以用类来表示函数，为Lambda表达式底层原理，捕获列表为类的成员。

闭包：带有上下文/状态的函数（能读取外部变量的函数），可通过匿名函数、重载operator()实现。

**enum和enum class**

区别在于是否限定其作用域。enum的成员的可见范围被提升至该枚举类型所在的作用域内，这被认为有可能污染了外部的作用域。枚举类（enum class）用于解决此问题，使用时需要限定枚举类型。

enum可以和int进行隐式类型转换，而enum class不能。

**指针和引用的区别**

指针可以为空，引用不能为空。指针存储了对象的地址，而引用为对象取了别名，必须要初始化。

**异常处理**

throw来抛出异常，使用try来执行一系列代码，catch来捕获异常。

构造函数中抛出异常：对象的析构函数将不会执行，对象父类析构函数正常执行，对象已构造完成的成员对象的析构函数正常执行。如果在构造函数中分配了内存（new），就会造成内存泄漏。

不要在析构函数中抛出异常：由于异常点之后的程序不会执行，可能会造成资源泄露的问题；异常处理会调用已经构造好的对象的析构函数来释放资源，此时若其他析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常，会造成程序崩溃。

noexcept关键字：告诉编译器此函数不会抛出异常。（若抛出异常则程序会终止。）

**静态库.lib/.a和动态库.dll/.so**

静态库：在编译时与程序代码直接链接，生成的可执行文件中包含了库中所有需要的代码。程序加载时间更快，跨平台需要重新编译。多个程序公用同一个静态库时每个程序都会有一份拷贝，浪费空间。

动态库：在编译时不直接链接到程序中而在程序运行时动态链接，生成可执行文件中只包含库的引用。程序需要加载和链接动态库，启动时间较长。多个程序可以共享同一份动态库从而节省空间。

**函数调用入栈基本步骤**

1. 参数入栈：将参数从右向左依次压入系统栈中。

2. 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。

3. 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

4. 栈帧调整：保存当前栈帧状态值，将当前栈帧切换到新栈帧，给新栈帧分配空间。

**C++是类型安全语言吗？**

类型安全：类型安全的代码不会试图访问自己没被授权的内存区域。

C++不是类型安全语言，同一片内存可使用不同方法解释，如int可以强转bool，reinterpret\_cast强制类型转换等。

**缺省成员函数**

1. 缺省构造函数

2. 缺省拷贝构造函数

3. 缺省析构函数

4. 缺省赋值运算符

5. 缺省取址运算符

6. 缺省取址运算符const

**如何定义一个只能在栈上或者只能在堆上的类？**

只能在栈上：关闭new的分配方式。

只能在堆上：析构函数设为私有。

**实现string**

class string {

public:

string(const char\* s = "") : size(strlen(s)) {

data = new char[size + 1];

strcpy(data, s);

}

string(const string& s) : size(s.size) {

data = new char[size + 1];

strcpy(data, s.data);

}

string& operator=(const string& s) {

if (this != &s) {

delete[] data;

data = new char[s.size + 1];

strcpy(data, s.data);

size = s.size;

}

return \*this;

}

~string() {

delete data;

data = nullptr;

size = 0;

}

private:

int size;

char\* data;

};