一、C++基础

```
cout<<"hello,wrold"<<endl;
operator<<(cout,"hello,world").operator<<(endl)
```

```
#ifndef __MyTest_h__
#define __MyTest_h__
...
#endif
```

1. 命名空间

（1）三种使用形式

1. using编译指令
2. 作用域限定符
3. using声明机制，一次只引出一个实体。

（2）带命名空间的函数声明

1. 两个命名空间相互调用
2. 自定义命名空间可以扩展（std也可以，但不推荐），结构体不可以扩展

（3）匿名命名空间

（4）嵌套命名空间

2. const

（1）const修饰变量

1. 形式：const int number
2. 必须初始化（与赋值区别）
3. 不能修改，保护数据

（2）与宏定义的不同

1. 宏定义在预处理阶段被使用；const发生的时机在编译阶段被使用。
2. 类型和安全检查不同。宏定义没有类型，不用做安全检查（如果有bug，直到运行时才会发现）；const常量有具体的类型，在编译期间会执行类型检查。

（3）const修饰指针

1. 常量指针(const int * 或 int const *)：不能修改指针所指变量的值，可以修改指针的指向。
2. 指针常量(int * const)：可以修改指针所指变量的值，不可以修改指针的指向。
3. const指针常量(const int * const)：不可以修改指针所指变量的值，不可以修改指针的指向。
4. 函数指针int (*pf)()，指针函数int * func()
5. 数组指针int (*pArray)[]，指针数组int * pArray[]

（4）const修饰对象与成员函数

3. new/delete

（1）malloc与new执行步骤比较

（a）malloc：

1. 申请内存
2. 初始化
3. 赋值
4. 释放空间

（b）new，申请空间、初始化、赋值可以一步完成。

（2）malloc/free与new/delete的异同

1. 不同点

   (1) malloc/free是C的库函数，new/delete的C++的表达式

   (2) malloc申请的是未初始化的原始空间，new申请的空间进行了初始化

   (3) new可以自动分配空间大小，malloc需要传入参数

   (4) new/delete能对对象进行构造和析构函数的调用，进而对内存进行更加详细的工作，而malloc/free不能

2. 相同点：

   都是用来申请堆空间

   都是成对出现，否则就会操作内存泄漏

（3）内存泄漏，内存溢出，踩内存，野指针

- 内存泄漏指使用内存结束后并没有释放使用空间，后续程序再也不能使用这块内存，但内存空间是有限的，久而久之就会造成系统崩溃。
- 内存溢出就是内存越界，指存储的数据超出了制定空间的大小（如在栈空间内分配了超过数组长度的数据，导致多出来的数据覆盖了栈空间其他位置的数据）。
- 内存踩踏，访问了不该访问的内存地址（如访问越界数组，访问已经free掉的内存，栈内存访问越界）。
- 野指针，指访问一个已销毁或者访问受限的内存区域的指针

（4）malloc/free的底层实现

- malloc函数的实质是它有一个将可用的内存块连接为一个长长的空闲链表。调用malloc函数时会沿着空闲链表找到一个大到足以满足用户请求所需要的的内存块。然后将该内存块一分为二。接下来将分配给用户的那块内存存储区域传给用户，并将剩下的那块返回到空闲链表上。
- 调用free函数时，它将用户释放的内存块连接到空闲链表上。到最后空闲链表会被切成很多的小内存片段，如果这是用户申请一个大的内存片段，那么空闲链表上可能没有足以满足用户要求的片段，于是malloc函数请求延时，并开始在空闲链表上检查各内存片段，对它们进行内存整理，将相邻的小空闲块合并成较大的内存块。

(5) 为什么new出来的不能用free

new在最前面躲开了4个字节空间，存放对象数量，free并不会释放最前面的4个字节空间

4. reference

（1）引用的特征

1. 引用的出现就是为了减少指针的使用
2. 与取地址区分就是前面有没有类型
3. 变量的别名，必须初始化
4. 对引用的修改就是对变量本身的修改
5. 底层实现是指针常量（被限制的指针），一旦绑定就不能改变指向

（2）与指针的区别

指针通过某个指针变量指向一个变量后，对它所指向的变量间接操作；引用的操作就是对目标变量的操作

（3）作为函数参数

当调用函数时不需要为引用变量在内存中分配空间

（4）作为函数返回值

返回值是引用的前提：实体的生命周期必须要大于函数的生命周期

（5）什么时候需要常引用

既要利用引用提高程序的效率，又要保护函数的数据不再函数中被改变时需要使用“常引用”

5. 强制类型转换

（1）static_cast

（2）const_cast

去掉常量属性

（3）dynamci_cast

（4）reinterpret_cast

6. 函数重载

（1）C vs C++

C不支持函数重载（不能进行名字改编），C++支持函数重载（名字改编，nm命令可进入.o文件）

（2）extern "C"

以C的方式进行编译

（3）混合编程

```
#ifndef __cplusplus
...
#endif
```

使C的源码既能够在C中以C的形式进行调用也能在C++中以C的形式进行调用

7. 默认参数

（1）顺序规定

从右向左

（2）默认参数与重载的函数可能产生二义性

8. inline

（1）与带参数宏定义的区别

使用带参数的宏定义容易出错，宏定义只是简单的将代码进行替换，替换后可能会出现意想不到的由于运算优先级造成的错误；宏定义也无法进行调试，但内联函数是可以进行调试的，在程序的调试版本中函数根本没有真正内联，编译器像普通函数那样为它生成含有调试信息的可执行代码，在程序的发行版本里，编译器才会实施真正的内联。宏定义无法操作类的私有数据成员。函数被内联后，编译器可以通过上下文相关的优化技术对结果代码执行更深入的优化。

（2）放在函数声明前无效

（3）一些短小的函数，不能有for/while等复杂语句

9. throw

```
try{
  //语句块
    throw 表达式;
}catch(异常类型){
  //异常处理
}catch(异常类型){
  //异常处理
}
```

10. 字符串 C vs C++

C风格字符串以'\0'结尾，string字符串实际上是类对象

11. 程序内存分配

（1）3G-4G 内核空间

进程管理、设备管理、虚拟文件管理

（2）0-3G 用户空间

栈区（向下）：由编译器进行分配释放（函数参数，局部变量）

内存映射段：静态库与动态库，文件映射，匿名映射

堆区（向上）：程序员自己控制

读写段：

全局静态区：

全局/静态变量（变量/指针默认初始化为零/空）

只读段：

文字常量区：字符串常量

程序代码区：二进制代码

（3）栈与堆的区别

1. 管理方式：栈由编译器自动管理；堆的申请和释放都由程序员控制。
2. 空间大小：VS下，栈空间默认是1M
3. 分配方式：堆都是动态分配的；静态分配是编译器完成的，栈动态分配和堆不同，由编译器释放，无序手动。
4. 生长方向：堆（向上，内存地址增加）；栈（向下，内存地址减小）。
5. 碎片问题：堆空间会造成大量碎片；栈是先进后出的，在当前变量弹出，后进的栈内容已经被弹出。

12. 内存对齐

1. 原因

（1）性能：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。

（2）平台

2. 规则

（1）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(偏移)为0的地方，以后每个数据成员的对齐按照#pragma pack（8） 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

（2）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。

（3）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。

二、类和对象

抽象、封装、继承、多态

1. 定义

(1) 权限

- public：修饰的成员表示的是该类可以提供的接口、功能、或者服务
- protected：访问权限开放给子类
- private：只能类内访问（封装性的具体体现）
- 类内定义的成员函数都是inline函数

(2) class VS struct

- class默认访问权限是private，struct默认访问权限是public
- C++里面对struct的功能做了一个提升，既可以定义数据也可以定义函数
- 除了默认访问权限外两者一样

(3) 空类

占一个字节，区分出不同的对象。

2. 构造函数

(1) 作用

初始化数据

(2) 调用时机

对象创建时

(3) 默认构造函数

如果程序员没有显式定义它，系统会提供一个默认构造函数。

(4) 可重载

(5) 初始化列表

每个成员在初始化列表之中只能出现一次，初始化的顺序不是由成员变量在初始化列表中的顺序决定的，而是由成员变量在类中被声明时的顺序决定的 。

3. 析构函数

(1) 作用

完成对象的销毁，执行清理任务。

(2) 不可重载

没有返回类型，也不能指定参数，所以只能有一个，不能被重载。

(3) 调用时机

- 对于全局定义的对象，每当程序开始运行，在主函数 main 接受程序控制权之前，就调用构造函数创建全局对象，整个程序结束时，自动调用全局对象的析构函数。
- 对于局部定义的对象，每当程序流程到达该对象的定义处就调用构造函数，在程序离开局部对象的作用域时调用对象的析构函数。
- 对于关键字 static 定义的静态局部变量，当程序流程第一次到达该对象定义处调用构造函数，在整个程序结束时调用析构函数。
- 对于用 new 运算符创建的对象，每当创建该对象时调用构造函数，当用 delete 删除该对象时，调用析构函数 。
- 可以显式调用（但是不建议）

4. 拷贝构造函数

(1) 浅拷贝 VS 深拷贝

- 浅拷贝中拷贝对象与被拷贝对象都指向同一个堆空间的字符串，只拷贝指针地址，当其中一个对象被销毁时，另一个对象就获取不到相应的值，且当一个对象修改时，另一个对象也会随之修改，相当于对同一片地址空间进行操作。
- 深拷贝是指拷贝指针所指空间内容，这样两个对象都拥有各自的独立堆空间字符串，一个对象销毁或修改时就不会影响到另一个对象。

(2) 调用时机

- 当用一个已经存在的对象初始化另一个新对象时，会调用拷贝构造函数。
- 当实参和形参都是对象，进行实参与形参的结合时，会调用拷贝构造函数。
- 当函数的返回值是对象，函数调用完成返回时，会调用拷贝构造函数（要把优化开关打开才可以看到，-fno-elide-constructors）

(3) const与&可以删除吗

- &如果去掉，在调用拷贝构造函数传参的时候，就会满足拷贝构造函数的调用时机1，此时又会去调用拷贝构造函数，形成递归，但是这个递归是没有出口的，就会导致栈溢出，进而程序崩溃。

```
Point(const Point rhs)
const Point rhs = pt1
```

- const如果去掉，传递是右值的时候，非cosnt左值引用不能绑定到右值，就不能调用拷贝构造函数。（左值：可以取地址的是左值(可以被赋值的变量)；右值：不可以取地址(匿名对象/临时对象/字面值常量)）。

5. 赋值运算符函数

(1) 返回值是否必须为&

必须为引用，否则执行return语句时会符合拷贝构造函数的调用时机3，增大开销，且执行完毕后左操作数是存在的

(2) 返回值类型是否可以为void

不可以，连等pt1 = pt2 = pt3

(3) 深拷贝四部曲

1. 自复制
2. 释放左操作数
3. 深拷贝
4. 返回*this

```
if(this != &rhs){
            delete [] _pstr;
            _pstr = nullptr;
            _pstr = new char[strlen(rhs._pstr)+1]();
            strcpy(_pstr,rhs._pstr);
}
return *this;
```

6. 特殊数据成员的初始化

(1) const数据成员

只能在初始化表达式（初始化列表）中进行，而且不能在构造函数等任何成员函数内部赋值。

(2) &数据成员

只能在初始化表达式中进行初始化，引用成员占据一个指针空间大小（指针大小和64/32位系统有关系）。

(3) 类对象成员

自定义类型创建的对象作为另一个类的对象成员时，必须要在初始化表达式中进行显示初始化，否则会调用子对象对应的默认构造函数。

(4) static成员

- 必须在类外进行初始化，被类的所有对象所共享，如果有头文件与实现文件，需要在实现文件进行定义。
- 不占类的空间。

7. 特殊成员函数

(1) static成员函数

- 静态成员函数没有隐含的this指针
- 只能调用静态数据成员与静态成员函数（因为没有this指针）
- 如果非要在静态成员函数中访问非静态成员，可以传参
- 除了使用对象进行调用，还可以使用类名进行调用静态成员函数

(2) const成员函数

- const成员函数的this指针类型 const 类名 * const this
- 与非const成员函数可以进行重载，且const成员函数不能修改数据成员（因为是双const指针）
- 非const对象可以调用两种，但是默认调用非const成员函数
- const对象只能调用const成员函数

三、new/delete表达式

1. new表达式步骤

1. 调用名为operator new的标准库函数，分配足够大的原始的未类型化的内存，以保存指定类型的一个对象
2. 运行该类型的一个构造函数初始化对象
3. 返回指向新分配并构造的构造函数对象的指针

2. delete表达式步骤

1. 调用析构函数，回收对象中数据成员所申请的资源
2. 调用名为operator delete的标准库函数释放该对象所用的内存

   注意：回收对象中数据成员申请的资源与释放对象所用内存

3. 对象的销毁 == 析构函数?

不等价，对于堆对象而言，析构函数只是对象销毁中的一个步骤

4. 如何只生成栈对象

- 将operator new/delete 设置为私有
- 构造函数和析构函数必须为public（栈对象的销毁编译器自动完成的）

5. 如何只生成堆对象

- 析构函数设置为私有
- 自定义destory函数

```
void destroy(){
        delete this;
}
```

四、C++输入输出流

1. 流的状态

- **badbit** 表示发生系统级的错误，如**不可恢复**的读写错误。通常情况下一旦 badbit 被置位，流就无法再使用了。
- **failbit** 表示发生可恢复的错误，如期望读取一个数值，却读出一个字符等错误。这种问题通常是**可以修改**的，流还可以继续使用。
- 当到达文件的结束位置时， **eofbit** 和 **failbit** 都会被置位。
- **goodbit** 被置位表示流未发生错误。如果 badbit 、 failbit 和 eofbit 任何一个被置位，则检查流状态的条件会失。
- 可通过成员函数查询或者操作流的状态

```
bool bad() const; //若流的badbit置位，则返回true;否则返回false
bool fail() const; //若流的failbit或badbit置位，则返回true;
bool eof() const; //若流的eofbit置位，则返回true;
bool good() const; //若流处于有效状态，则返回true;
iostate rdstate() const; //获取流的状态
void setstate(iostate state); //设置流的状态

//clear的无参版本会复位所有错误标志位*(重置流的状态)
void clear(std::ios_base::iostate state = std::ios_base::goodbit);
```

2. 标准IO

标准输入(cin)、标准输出(cout/cerr/clog)

缓冲区被刷新的几种情况

- 程序正常结束（有一个收尾操作就是清空缓冲区）；
- 缓冲区满（包含正常情况和异常情况）；
- 使用操纵符显式地刷新输出缓冲区，如： endl（还有一个换行） 、 flush 、 ends (没有刷新功能)；
- 使用 unitbuf 操纵符设置流的内部状态；
- 输出流与输入流相关联，此时在读输入流时将刷新其关联的输出流的输出缓冲区 。

3. 文件IO

(1) ifstream

文件不存在时，打开文件失败

从文件读数据并输出到屏幕，默认以空格分隔

getline函数

(2) ofstream

文件不存在时创建文件；文件已经存在则清空文件再写

(3) fstream

tellp/tellg

seekp/seekg

(4) 文件的模式

读文件：in，ate（追加，文件指针最初在文件末尾）

写文件：out，app（追加，写入指针始终在文件末尾）

4. 字符串IO

一些实际应用

- 数值转换为字符串并输出
- 将数字转为字符串输出，然后字符串输入到指定格式输出到屏幕
- 读配置文件，结合ifstream进行

五、运算符重载

1. 友元

（1）全局函数

（2）成员函数

（3）友元类

C++ 规则已经极力地将友元的使用限制在了一定范围内，它是单向的、不具备传递性、不能被继承，所以，应尽力合理使用友元。 友元的声明是不受 public/protected/private 关键字限制的。

2. 运算符重载

(1) 实质

函数重载或函数多态

(2) 不可以重载的运算符

. .* ?: :: sizeof （如何证明sizeof不是函数，可以不加括号）

(3) 重载规则

- 为了防止用户对标准类型进行运算符重载， C++ 规定重载的运算符的操作对象必须至少有一个是自定义类型或枚举类型。
- 重载运算符之后，其优先级和结合性还是固定不变的。
- 重载不会改变运算符的用法，原来有几个操作数、操作数在左边还是在右边，这些都不会改变。
- 重载运算符函数不能有默认参数，否则就改变了运算符操作数的个数。
- 重载逻辑运算符（ &&，|| ）后，不再具备短路求值特性。不能臆造一个并不存在的运算符，如@、$等 。

(4) 重载形式

- 采用普通函数的重载形式
- 采用成员函数的重载形式
- 采用友元函数的重载形式

```
返回类型 类名::operator 运算符（参数列表）
{
//...
}
```

(5) 特殊运算符的重载

1. 复合赋值运算符（+=、-=等，推荐以成员函数的形式进行重载，对象本身发生了变化，加&符号）
2. 自增自减运算符

   (a) 区分前置后置形式，后置++参数列表写一个int

   (b) 后置不能加&，前置要加&。前置++之所以比后置++效率高的原因就是因为前置&，后置因为没有&多了两次调用。

   (c) 后置++不能取地址，所以是右值。
3. 赋值运算符
4. 函数调用运算符：对于这种重载了函数调用运算符（）的类创建的对象，我们称为函数对象 （Function Object）。函数也是一种对象。
5. 下表访问运算符（只能以成员函数形式进行，要重载普通版本和const版本）
6. 成员访问运算符（->,*）
7. 输入输出流运算符（由于非静态成员函数的第一个参数是隐含的this指针，代表当前对象本身，这与其要求是冲突的，因此 >> 和 << 不能重载为成员函数，只能是非成员函数，如果涉及到要对类中私有成员进行访问，还得将非成员函数设置为类的友元函数。**因为没有拷贝构造函数，所以一定要加&**）

(6) 运算符重载的建议

- 所有的一元运算符，建议以成员函数重载
- 运算符 = () [] -> ->* ，必须以成员函数重载
- 运算符 += -= /= *= %= ^= &= != >= <= 建议以成员函数形式重载
- 其它二元运算符，建议以非成员函数重载

(7) 类型转换

1. 其他类型向自定义类型转换（隐式转换，int->Point）
2. 自定义类型向其他类型转换

   必须是成员函数
   参数列表中没有参数
   没有返回值，但在函数体内必须以 return 语句返回一个目标类型的变量。

3. string底层实现

(1) 深拷贝

Eager Copy

(2) 写时复制(COW)

1. string设计：count+char
2. **如何区分读写**

   自定义CharProxy类（包含string和size_t两个成员），重载下标访问运算符返回CharProxy类型，对CharProxy类重载 = 运算符、<<运算符，

```
class CharProxy
{
public:
    CharProxy(String &self, size_t idx)
    : _self(self)
    , _idx(idx)
    { }
    char &operator=(const char &ch);//写操作，有隐藏的this指针
    friend std::ostream &operator<<(std::ostream &os, const CharProxy &rhs);//读操作
private:
    String &_self;
    size_t _idx;
};
```

(3) 短字符串优化(SSO)

字符串长度小于15分配到栈

字符串长度大于15分配到堆

(4) 最优策略

- 很短的（0~22）字符串用SSO，23字节表示字符串（包括'\0'）,1字节表示长度
- 中等长度的（23~255）字符串用eager copy，8字节字符串指针，8字节size，8字节capacity.
- 很长的(大于255)字符串用COW, 8字节指针（字符串和引用计数），8字节size，8字节capacity.

六、继承

1. 定义

(1) 继承方式

public、private、protected

如果不写默认是私有继承

(2) 派生类的生成步骤

1. 吸收基类的成员
2. 改造基类的成员
3. 添加自己的新成员

(3) 不能从基类继承的

- 构造函数
- 析构函数
- 用户重载的operator new/delete运算符
- 用户重载的operator =运算符
- 友元关系

(4) 派生方式对基类成员的访问权限

通过继承，除了基类私有成员以外的其它所有数据成员和成员函数，派生类中可以直接访问。

private成员是私有成员，只能被本类的成员函数所访问，派生类和类外都不能访问。

public成员是公有成员，在本类、派生类和外部都可访问。

protected成员是保护成员，只能在本类和派生类中访问，是一种区分血缘关系内外有别的成员。

总结：派生类的访问权限规则如下：

- 不管是什么继承方式，派生类中都不能访问基类的私有成员。
- 派生类内部除了基类的私有成员不可以访问外，其他的都可以访问。
- 派生类对象除了公有继承基类中的公有成员可以访问外，其他的一律不能访问 。

2. 派生类对象的构造/销毁

(1) 构造

1.派生类对象构造情况

- 如果派生类有显式定义构造函数，而基类没有显示定义构造函数，则创建派生类的对象时，派生类相应的构造函数会被自动调用，此时都自动调用了基类缺省的无参构造函数。
- 如果派生类没有显式定义构造函数而基类有显示定义构造函数，则基类必须拥有默认（无参）构造函数。
- 如果派生类有构造函数，基类有默认构造函数，则创建派生类的对象时，基类的默认构造函数会自动调用，如果你想调用基类的有参构造函数，必须要在派生类构造函数的初始化列表中显示调用基类的有参构造函数。

- 如果派生类和基类都有构造函数，但基类没有默认的无参构造函数，即基类的构造函数均带有参数，则派生类的每一个构造函数必须在其初始化列表中显示的去调用基类的某个带参的构造函数。如果派生类的初始化列表中没有显示调用则会出错，因为基类中没有默认的构造函数。

2.调用顺序

1. 完成对象所占整块内存的开辟，由系统在调用构造函数时自动完成。
2. 调用基类的构造函数完成基类成员的初始化。
3. 若派生类中含对象成员、 const 成员或引用成员，则必须在初始化表中完成其初始化。
4. 派生类构造函数体执行

3.在创建派生类对象的时候,"先调用基类构造函数，然后调用派生类构造函数",是错误的

在创建派生类对象的时候，会调用派生的构造函数，然后在调用派生类构造函数的过程中，为了完成从基类吸收过程的数据成员的初 始化，所以调用基类的构造函数，完成基类部分的初始化,ok

(2) 销毁顺序

1. 先调用派生类的析构函数
2. 再调用派生类中成员对象的析构函数
3. 最后调用普通基类的析构函数

3. 多基继承

(1) 多基vs单基

多基继承和单基继承的派生类构造函数完成的任务和执行顺序并没有本质不同，唯一一点区别在于：首先要执行所有基类的构造函 数，再执行派生类构造函数中初始化表达式的其他内容和构造函数体。各基类构造函数的执行顺序与其在初始化表中的顺序无关，而 是由定义派生类时继承的基类顺序决定的。

对于每个基类都要写上继承方式，否则就是私有继承。

(2) 成员名冲突二义性

解决该问题的方式比较简单，只需要在调用时，指明要调用的是某个基类的成员函数即可，即使用作用域限定符就可以解决该问题。

(3) 菱形继承存储二义性

中间层虚拟(virtual)继承上层。

4. 基类和派生类的相互转换

- 可以把派生类的对象赋值给基类的对象
- 可以把基类的引用绑定到派生类的对象
- 可以声明基类的指针指向派生类的对象 (向上转型)

5. 派生类对象间的复制控制

- 如果用户定义了基类的拷贝构造函数，而没有定义派生类的拷贝构造函数，那么在用一个派生类对象初始化新的派生类对象时，两对象间的派生类部分执行缺省的行为，而两对象间的基类部分执行用户定义的基类拷贝构造函数。
- 如果用户重载了基类的赋值运算符函数，而没有重载派生类的赋值运算符函数，那么在用一个派生类对象给另一个已经存在的派生类对象赋值时，两对象间的派生类部分执行缺省的赋值行为，而两对象间的基类部分执行用户定义的重载赋值函数。
- 如果用户定义了派生类的拷贝构造函数或者重载了派生类的对象赋值运算符=，则在用已有派生类对象初始化新的派生类对象时，或者在派生类对象间赋值时，将会执行用户定义的派生类的拷贝构造函数或者重载赋值函数，而不会再自动调用基类的拷贝构造函数和基类的重载对象赋值运算符，这时，通常需要用户在派生类的拷贝构造函数或者派生类的赋值函数中显式调用基类的拷贝构造或赋值运算符函数。

七、多态

1. 多态定义

静态多态：函数重载、运算符重载、模板（发生在编译时）

动态多态：虚函数（发生在运行时）

2. 虚函数定义

格式要求

- 与基类的虚函数有相同的参数个数
- 与基类的虚函数有相同的参数类型
- 与基类的虚函数有相同的返回类型

3. 虚函数的实现机制

当基类定义了虚函数之后，就会在**基类的对象的存储布局之中产生一个虚函数指针**(vfptr，该虚函数指针位于对象的最前面)，虚函数指针会指向基类自己的虚表(虚函数表)，虚表存放的是虚函数的入口地址,当派生类继承基类的时候，就会吸收基类的虚函数，然后在派生类创建的对象自己的存储布局中产生虚函数指针，该虚函数指针指向派生类自己的虚函数表，该虚函数表存放是虚函数的入口地址，如果派生类有 自己重定义(重写)该虚函数，就会在派生类虚表中将从基类继承过来的虚函数的地址进行覆盖。

4. 动态多态的激活条件

1. 基类定义虚函数
2. 派生类重定义（覆盖、重写）虚函数
3. 创建派生类对象
4. 基类的指针指向派生类对象
5. 基类指针（引用）调用虚函数

5. 哪些函数不能被设置为虚函数

- 普通函数（非成员函数）：定义虚函数的主要目的是为了重写达到多态，所以普通函数声明为虚函数没有意义，因此编译器在编译时就绑定了它。
- 静态成员函数：静态成员函数对于每个类都只有一份代码，所有对象都可以共享这份代码，他不归某一个对象所有，所以它也没有动态绑定的必要。(静态函数发生在编译时，虚函数体现多态发生在运行时)
- 内联成员函数：内联函数本就是为了减少函数调用的代价，所以在代码中直接展开。但虚函数一定要创建虚函数表，这两者不可能统一。另外，内联函数在编译时被展开，而虚函数在运行时才动态绑定。
- 构造函数：这个原因很简单，主要从语义上考虑。因为构造函数本来是为了初始化对象成员才产生的，然而虚函数的目的是为了在完全不了解细节的情况下也能正确处理对象，两者根本不能“ 好好相处 ”。因为虚函数要对不同类型的对象产生不同的动作，如果将构造函数定义成虚函数，那么对象都没有产生，怎么完成想要的动作呢
- 友元函数：当我们把一个函数声明为一个类的友元函数时，它只是一个可以访问类内成员的普通函数，并不是这个类的成员函数，自然也不能在自己的类内将它声明为虚函数。**(当友元函数是成员函数的时候是可以设置为虚函数的，比如在自己类里面设置为虚函数，但是作为另一个类的友元)**

6. 虚函数的访问

（1）指针访问

使用指针访问非虚函数时，编译器根据指针本身的类型决定要调用哪个函数，而不是根据指针指向的对象类型；使用指针访问虚函数时，编译器根据指针所指对象的类型决定要调用哪个函数(**动态联编**)，而与指针本身的类型无关。

（2）引用访问

使用引用访问虚函数，与使用指针访问虚函数类似，表现出动态多态特性。不同的是，引用一经声明后，引用变量本身无论如何改变，其调用的函数就不会再改变，始终指向其开始定义时的函数。因此在使用上有一定限制，但这在一定程度上提高了代码的安全性，特别体现在函数参数传递等场合中，可以将引用理解成一种 “受限制的指针” 。

（3）对象访问

和普通函数一样，虚函数一样可以通过对象名来调用，此时编译器采用的是**静态联编**。通过对象名访问虚函数时，调用哪个类的函数取决于定义对象名的类型。对象类型是基类时，就调用基类的函数；对象类型是子类时，就调用子类的函数。

（4）成员函数中访问

在类内的成员函数中访问该类层次中的虚函数，采用**动态联编**，要使用 this 指针。

（5）构造函数和析构函数中访问

构造函数和析构函数是特殊的成员函数，在其中访问虚函数时，C++采用**静态联编**，即在构造函数或析构函数内，即使是使用 “this->虚函数名” 的形式来调用，编译器仍将其解释为静态联编的 “本类名::虚函数名” 。即它们所调用的虚函数是自己类中定义的函数，如果在自己的类中没有实现该函数，则调用的是基类中的虚函数。但绝不会调用任何在派生类中重定义的虚函数，因为派生类在构造函数中还未产生或在析构函数中已经销毁。

7. 纯虚函数及抽象类

（1）纯虚函数

```
virtual void func() = 0//形式
```

（2）抽象类

- 一个类可以包含多个纯虚函数。只要类中含有一个纯虚函数，该类便为抽象类。
- 将构造函数设置为protected，也是抽象类

一个抽象类只能作为基类来派生新类，不能创建抽象类的对象。

8. 虚析构函数

（1）将基类的析构函数设置为虚函数，派生类的析构函数自动成为虚函数，目的：防止内存泄漏（也可以使用dynamic_cast释放内存）

（2）原理

根据虚函数可以被重写这个特性，如果基类的析构函数设置为虚函数后，那么派生类的析构函数就会重写基类的析构函数。但是他们的 函数名不相同，看起来违背了重写的规则，但是实际上编译器对析构函数的名称做了特殊的处理，编译后析构函数的名称统一为~destructor()。之所以可以这样做，是因为在每个类里面，析构函数是独一无二的，不能重载，所以可以这么设计。

9. 重载/覆盖/隐藏

（1）重载

发生在同一个类中，函数名称相同，但参数的类型、个数、顺序不同。

（2）覆盖（重写、重定义）

发生在基类与派生类中，同名虚函数，参数亦完全相同。

（3）隐藏

发生在基类与派生类中，指的是在某些情况下，派生类中的函数屏蔽了基类中的同名函数。(同名数据成员也有隐藏)

10. 虚表存在性测试

创建一个派生类对象，解引用对象地址强转为（long *）可得虚表地址，解引用虚表地址强转为（long *）可得虚函数地址。

单继承时一个类只有一张虚表，被所有的对象共享，虚表存在于只读段

11. 多基派生

1. 每个基类都有自己的虚函数表
2. 派生类如果有自己的虚函数，会被加入到第一个虚函数表之中
3. 内存布局中，其基类的布局按照基类被声明时的顺序进行排列
4. 派生类会覆盖基类的虚函数，只有第一个虚函数表中存放的是真实的被覆盖的函数的地址；其它的虚函数表中存放的并不是真实的对应的虚函数的地址，而只是一条跳转指令

12. 虚拟继承

(1) 虚继承与继承的区别

结论一：单个虚继承，不带虚函数

1. 多了一个虚基指针
2. 虚基类位于派生类存储空间的最末尾

结论二：单个虚继承，带虚函数

1. 如果派生类没有自己的虚函数，此时派生类对象不会产生虚函数指针
2. 如果派生类拥有自己的虚函数，此时派生类对象就会产生自己本身的虚函数指针，并且该虚函数指针位于派生类对象存储空间的开始位置

(2) 虚拟继承时派生类对象的构造和析构

注意在派生类中需要显示调用基类的构造函数。在 C++ 中，如果继承链上存在虚继承的基类，则最底层的子类要负责完成该虚基类部分成员的构造。即我们需要显式调用虚基类的构造函数来完成初始化，如果不显式调用，则编译器会调用虚基类的缺省构造函数，不管初始化列表中次序如何，对虚基类构造函数的调用总是先于普通基类的构造函数。 如果虚基类中没有定义的缺省构造函数，则会编译错误。因为如果不这样做，虚基类部分会在存在的多个继承链上被多次初始化。

13. 菱形继承

(1) 虚基指针所指向的虚基表的内容

虚基指针的第一条内容表示的是该虚基指针距离所在的子对象的首地址的偏移

虚基指针的第二条内容表示的是该虚基指针距离虚基类子对象的首地址的偏移

(2) 对于虚继承的派生类对象的析构，析构函数的调用顺序为

1. 先调用派生类的析构函数
2. 然后调用派生类中成员对象的析构函数
3. 再调用普通基类的析构函数
4. 最后调用虚基类 的析构函数

14. 虚字定义

C++ 中的 virtual 关键字采用第一个定义，即被 virtual 所修饰的事物或现象在本质上是存在的，但是没有直观的形式表现，无法直接描述或定义，需要通过其他的间接方式或手段才能够体现出其实际上的效果。

关键就在于存在、间接和共享这三种特征

(1) 虚函数

虚函数是存在的

虚函数必须要通过一种间接的运行时（而不是编译时）机制才能够激活（调用）的函数

共享性表现在基类会共享被派生类重定义后的虚函数

(2) 虚继承

存在即表示虚继承体系和虚基类确实存在

间接性表现在当访问虚基类的成员时同样也必须通过某种间接机制来完成（通过虚基表来完成）

共享性表现在虚基类会在虚继承体系中被共享，而不会出现多份拷贝 （所以可以用来解决菱形继承）

八、移动语义+资源管理

1. 为什么需要移动语义

在程序运行的过程中，会产生大量的临时对象。临时对象只在某一个表达式执行的过程中创建。表达式执行完毕时，临时对象马上就被 销毁。其作用是用来进行过渡的，它带来了资源拷贝的不必要的浪费。

2. 左值 vs 右值

左值可以取地址，右值不能取地址

3. 右值引用(&&)

右值引用不能绑定到左值

4. std::move()

将左值转为右值，表明不想再继续使用该值了，如果还想继续使用必须重新赋值，仅针对自定义类型移动堆上的资源。

1. C++ 标准库使用比如vector::push_back 等这类函数时,会对参数的对象进行复制,连数据也会复制.这就会造成对象内存的额外创建, 本来原意是想把参数push_back进去就行了,通过std::move，可以避免不必要的拷贝操作。
2. std::move是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝所以可以提高利用效率,改善性能.。

5. 移动构造和移动赋值

仅仅移动数据成员，不会分配新的内存。

在自定义**移动赋值**运算符时，需要检查是否存在自赋值，也就是说如果要赋值的对象与自己的地址一样，则不需要做任何事情。

6. 资源管理(RAII)

Resource Acquisition Is Initialization

利用对象的生命周期来管理资源，不允许赋值与复制

7. auto_ptr

拷贝（复制）之后资源会进行转移，原指针不能再使用

8. unique_ptr

保证独有权，具有移动语义，不允许赋值或复制，具有移动语义

9. shared_ptr

强引用：只要有一个引用就不能进行资源回收

循环引用要通过weak_ptr解决

10. weak_ptr

弱引用：并不增加对象的引用计数，但知道对象是否存在

通过weak_ptr访问对象的成员的时候，要提升为shared_ptr

11. 智能指针的误用

（1）不要把一个裸指针同时交给不同的unique_ptr

（2）std::enable_shared_from_this（可以修改this指针为shared_ptr）

九、模板

普通函数优于函数模板执行。

函数模板不能分成头文件和实现文件的形式，在头文件中不能看到具体的实现。（C++头文件都是使用模板进行编写的，而模板的的特点是必须知道所有实现才能进行正常编译）

1. 函数模板

2. 可变模板参数

3. 类模板

十一、关键字

1. inline

内联函数

2. explicit

防止隐式转换

3. override

检测虚函数重写是否正确

4. voliate

volatile提醒编译器它后面所定义的变量随时都有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。

十二、设计模式

1. 单例模式

(1) 实现步骤

1. 将构造函数私有化
2. 在类中定义一个静态的指向本类型的指针变量
3. 定义一个返回值为类指针的静态成员函数

(2) 代码实现

```
class Singleton{
public:
    static Singleton *getInstance(){
        if(nullptr == _pInstance){
            _pInstance = new Singleton();
        }
        return _pInstance;
    }
    static void destroy(){
        if(_pInstance){
            delete _pInstance;
```

```
            _pInstance = nullptr;
        }
    }
private:
    Singleton(){
        cout << "Singleton()" << endl;
    }
    ~Singleton(){
        cout << "~Singleton()" << endl;
    }
private:
    static Singleton *_pInstance;
};

Singleton *Singleton::_pInstance = nullptr;

int main()
{
    Singleton *ps1 = Singleton::getInstance();
    Singleton *ps2 = Singleton::getInstance();
    cout << "ps1 = " << ps1 << endl
         << "ps2 = " << ps2 << endl;
    Singleton::destroy();
    return 0;
}
```

(3) 自动释放

1. 友元

```
class AutoRelease
{
    public:
    AutoRelease()
    {
        cout << "AutoRelease()" << endl;
    }
    ~AutoRelease()
    {
        cout << "~AutoRelease()" << endl;
        if (Singleton::_pInstance)
        {
            delete Singleton::_pInstance; //1、调用析构函数 2、operator delete
            Singleton::_pInstance = nullptr;
        }
    }
};内部类+静态成员的方法
```

2. 内部类+静态成员的方法

当设置为私有时，要创建一个全局AutoRelease类对象(如果不是全局对象也属于Singleton类对象的一部分，这时如果想执行析构函数，两个对象的析构函数就会相互等待)

3. atexit函数

   先注册，进程结束时才执行

   线程不安全

(4) 饿汉/饱汉(懒汉)

1. 饿汉

   _pInstance初始化时直接getInstance()

2. 饱汉

   _pInstance初始化时直接nullptr，多线程下不安全

3. pthread_once

   保证多线程下安全

```
class Singleton
{
public:
    static Singleton *getInstance()
  {
    pthread_once(&_once, init);
    return _pInstance;
  }
    static void init()
  {
    _pInstance = new Singleton();
    atexit(destroy);
  }
    static void destroy()
  {
    if(_pInstance)
    {
      delete _pInstance;
      _pInstance = nullptr;
    }
  }
private:
    static Singleton *_pInstance;
  static pthread_once_t _once;
}
Singleton *Singleton::_pInstance = nullptr; //饱汉模式(懒汉模式)
/* Singleton *Singleton::_pInstance = getInstance();//饿汉模式 */
pthread_once_t Singleton::_once = PTHREAD_ONCE_INIT;
```

2. Pimpl模式

通过一个私有的成员的指针，将指针所指向的类的内部实现数据进行隐藏，实现中用到了嵌套类，又称“编译防火墙”。

优点：

- 提高编译速度；
- 实现信息隐藏；
- 减小编译依赖，可以用最小的代价平滑的升级库文件；
- 接口与实现进行解耦；
- 移动语义友好。

3. 代理模式

解决自定义string写时复制区分读写问题

十三、开源库

1. log4cpp

(1) Layout（格式化器）

%d-日期，%c-catergory，

(2) Appender（输出器）

(3) Category（日志记录器）： 要用shutdown回收

(4) Priority（优先级）： emerg、fatal、alert、error、warn、info、notice、debug

RollingFileAppender（回滚文件）

2. xml