C++基础重点复习

一、 基本语法

1、命名空间

定义和使用

- **定义**：C++提供的一种封装机制，用于将全局作用域划分为不同的独立区域，防止冲突
- **作用**
  1. 避免命名冲突
  2. 组织代码，将相关功能分组，如 std 包含标准库
  3. 控制访问范围，通过 namespace 限制变量/函数作用域，减少全局污染
- **使用方式**
  1. using 编译指令

```
#include <iostream>
using namespace std; //using 编译指令

int main(int argc, char * argv[]){
 cout << "hello,world" << endl;
 return 0;
}
```

  2. 作用域限定符

```
namespace wd
{
int number = 10;
}
void test0()
{
      std::cout << "wd::number = " << wd::number << endl;
}
```

3. using 声明机制

```
#include <iostream>
using std::cout;
using std::endl;
```

跨模块调用

- **可以跨模块调用**
  1. 全局变量/函数
  2. 命名空间中的实体，命名空间中的实体跨模块调用时，要在新的源文件中再次定义同名的命名空间（即同一命名空间在跨模块调用时可多次定义，编译时多次定义会被认为是同一个命名空间），在其中通过 extern 引入实体

     (1) externA.cc 中定义

```cpp
// externA.cc
#include <iostream>
using std::cout;
using std::endl;

namespace wd {
    int val = 300;                          // 定义变量 val
    void display() {                        // 定义函数 display
        cout << "wd::display()" << endl;
    }
} // end of namespace wd
```

(2) externB.cc 中调用

```cpp
// externB.cc
#include <iostream>
using std::cout;
using std::endl;

// 重新打开同名命名空间 wd
namespace wd {
    // 通过 extern 声明变量（定义在 externA.cc）
    extern int val;
    // 通过 extern 声明函数（定义在 externA.cc）
    extern void display();
}

void test0() {
    cout << wd::val << endl;   // 访问 wd::val
    wd::display();             // 调用 wd::display
}
```

注意：跨模块调用命名空间实体时避免与全局实体重名

```
// moduleA.cc
int count = 100;   // 全局变量

namespace MyLib {
    int count = 200;   // 命名空间变量
}
```

```
// moduleB.cc
#include <iostream>

extern int count;                 // 声明全局变量（来自 moduleA.cc）
extern int MyLib::count;          // 声明命名空间变量（来自 moduleA.cc）

void test() {
    std::cout << count << std::endl;          // 输出全局的 100
    std::cout << MyLib::count << std::endl; // 输出命名空间的 200
}
```

- **不能跨文件调用**
  1. 静态变量/函数
  2. 匿名空间中的实体

2、const 关键字

- **作用：**被 const 修饰的变量成为 const 常量（本质还是变量），被赋予只读属性，不能修改其值，const 常量在定义时必须初始化
- **const 常量与宏定义常量的区别**
  1. **发生时机不同：**C 语言的宏定义发生时机在预处理时，做字符串的替换；const 常量发生时机在编译时
  2. **类型和安全检查不同：**宏定义没有任何类型，不做任何类型检查；const 常量有具体的类型，在编译期会执行类型检查。在使用中应尽量用 const 替换宏定义常量，减少犯错误的概率

2.1 const 修饰指针

- **指向常量的指针（point to const）**:const int *p/int const *p，const 在*左边，不能通过指针修改其指向的值，但是可以改变这个指针的指向

```
int number1 = 10;
int number2 = 20;

const int * p1 = &number1;//指向常量的指针
*p1 = 100;//error 通过 p1 指针无法修改其所指内容的值
p1 = &numbers;//ok 可以改变 p1 指针的指向
number = 100;//ok 只是不能通过 p1 修改 number 的值，但是 number 本身可以修改

//例子中 p1 称为指向常量的指针（pointer to const），尽管 number1 本身并不是一个 int 常量，但定义指针 p1 的方式决定了无法通过 p1 修改其指向的值。但值得注意的是，修改 p1 的指向是允许的。
```

补充：如果有一个 const 常量，那么普通的指针也无法指向这个常量，只有指向常量的指针才可以，即 const 常量只有指向常量的指针才可以指向

```
const int x = 20;
int * p = &x; //error
const int * cp = &x; //ok
```

- **常量指针（const pointer）：**指针本身是个常量，其保存的地址不能修改，即不能修改指针指向，但是可以修改指针所指向内容的值（const 在*右边）

```
int * const p3 = &number1;//常量指针
*p3 = 100;//ok 通过 p3 指针可以修改其所指内容的值
p3 = &number2;//error 不可以改变 p1 指针的指向
```

- **双重 const 限定指针：**既不能修改指向，又不能修改指向内容的数据

```
const int * const p4 = &number1;//指向和指向的值皆不能进行修改
```

2.2 数组指针和指针数组

- **数组指针**：指向数组的指针，本质是指针，指向数组首地址的指针

```cpp
void test1()
{
    int arr[5] = {1, 2, 3, 4, 5};
    // arr 在涉及计算时会退化为指针
    cout << arr << endl;
    cout << arr + 1 << endl;

    cout << &arr << endl;
    // 这里输出的是整个 arr 数组的下一个地址
    cout << &arr + 1 << endl;
    // 输出整个数组字节数
    cout << sizeof(arr) << endl;

    int (*p)[5] = &arr;
    for (int i = 0; i < 5; i++)
    {
        cout << (*p)[i] << " ";
    }
    cout << endl;
}

// 输出结果如下
0x7ffea3b481b0
0x7ffea3b481b4
0x7ffea3b481b0
0x7ffea3b481c4
20
1 2 3 4 5
```

- **指针数组**：元素都是指针类型的数组，本质是数组，元素是指针

```
void test2()
{
	int num1 = 1, num2 = 2, num3 = 3;
	int *p1 = &num1;
	int *p2 = &num2;
	int *p3 = &num3;

	int *arr[3] = {p1, p2, p3};
	for (int i = 0; i < 3; ++i)
	{
		cout << *arr[i] << " ";
	}
	cout << endl;
}
```

2.3 函数指针和指针函数

- **函数指针**：指向函数的指针，本质为指针，可以通过函数指针调用函数，定义方式：return)type(*pointer_name)(parameter_list)

```c
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int main() {
    // 定义函数指针，指向一个返回 int、接受两个 int 参数的函数
    int (*funcPtr)(int, int);

    funcPtr = add; // 指向 add 函数
    printf("5 + 3 = %d\n", funcPtr(5, 3)); // 输出：8

    funcPtr = sub; // 指向 sub 函数
    printf("5 - 3 = %d\n", funcPtr(5, 3)); // 输出：2

    return 0;
}
```

- **指针函数**：返回值为指针类型的函数，本质为函数，定义方式：return_type* function_name(parameter_list){}

```c
// 指针函数：返回一个 int 指针
int* createArray(int size) {
    int* arr = (int*)malloc(size * sizeof(int));
    return arr; // 返回动态分配的数组指针
}

int main() {
    int* myArray = createArray(10); // 调用指针函数
    free(myArray); // 释放内存
    return 0;
}
```

3、new/delete

- **作用：**动态管理内存，即管理堆上的内存
- **与 malloc/free 对比**
  1. malloc/free 是库函数；new/delete 是运算符，后两者使用时不是函数的写法；
  2. new/delete 是类型安全的
     a. new 表达式的返回值是相应类型的指针，malloc 返回值是 void*；
     b. new 会调用构造函数，delete 会调用析构函数，而 malloc/free 不会，不适合用于 C++对象；
     c. new 失败会抛出 bad_alloc 异常，而 malloc 成功与否需要手动检查
  3. malloc 申请的空间不会进行初始化，获取到的空间是有脏数据的，但 new 表达式申请空间时可以直接初始化；
  4. malloc 的参数是字节数，new 表达式不需要传递字节数，会根据相应类型自动获取空间大小。
- **工作步骤**

  new 表达式工作步骤

  1. 调用 operator new 标准库函数申请未类型化的空间
  2. 在该空间上调用该类型的构造函数初始化对象
  3. 返回指向该对象的相应类型的指针

  delete 表达式工作步骤

  1. 调用析构函数,回收数据成员申请的资源(堆空间)
  2. 调用 operator delete 库函数回收本对象所在的空间

4、引用（超重点）

- **作用与原理：**是被限制的指针，底层是 const pointer，主要是减少对指针的使用，引用一经绑定无法修改

```
void test(){
        int num = 100;
        int num2 = 200;
        int *p = &num;
        cout << &p << endl;
        cout << p << endl;
        cout << &num;

        int & ref = num;// 定义引用 ref 绑定 num 变量
        // 引用的底层也是指针实现的(常量指针 const pointer) 但是无法访问创建的指针变量
        ref = num2;// error，引用一经绑定无法修改
        // 对引用取地址获取到的其实是其绑定的变量的地址
        cout << &ref << endl;
        cout << ref << endl;
        ref = 200;
        cout<<ref<<endl;// 输出 200
        cout<<num<<endl;// 输出 200
}
```

4.1 引用与指针的联系和区别

联系：

1. 引用和指针都有地址的概念，都是用来间接访问变量；
2. 引用的底层还是指针来完成，可以把引用视为一个受限制的指针。（const pointer）

区别：

1. 引用必须初始化，指针可以不初始化；
2. 引用不能修改绑定，但是指针可以修改指向；
3. 在代码层面对引用本身取址取到的是变量本体的地址，但是对指针取址取到的是指针变量的地址

4.2 应用场景

4.2.1 作为函数参数：

a) 可将参数变为传入传出参数，改变传入参数的值；
b) 若参数是较大的对象或数据 ,那么使用引用作为函数参数可以避免复制实参 ,减少开销
c) 若不希望函数体中通过引用改变传入的变量，可以使用常引用作为函数参数

```cpp
// 常引用基本特点
void test1(){
	int num = 10;
	// 定义引用绑定 num  并使用 const 修饰
	const int & ref = 10;
	// 既不能修改指向,也不能通过这个引用修改变量的值
	// ref = 100; // error read only
	num = 100;
	cout << "num = " << num << endl;
	cout << "ref = " << ref << endl;
	// 不能通过引用常引用修改  但是可以通过变量自身修改
}

// 函数不希望通过引用改变变量的值的时候可以使用常引用
// 形参为常引用
void func(const int & x){
	cout << x << endl;
   //   x = 100; //error read only   无法通过常引用修改
}

void test(){
	int num = 1;
	func(num);
	cout << num << endl;
}
```

4.2.2 作为函数返回值

- **返回的变量其声明周期一定要大于函数的声明周期（避免返回值的复制，减少开销）**

```
// 全局变量
int a    = 100;
int func(){
      // func 函数返回的是 a 的一个副本,一个临时变量
      return a;
}

// 全局变量
int b = 200;
// 函数返回值为引用
int & func2(){
      // return 时不会发生复制
      return b; // 返回的实际是一个绑定到 b 的引用
      // 要注意返回的引用所绑定的变量的生命周期要比函数更长
}

void test(){
      cout << func() << endl;
      cout << &a << endl;
      // cout << &func() << endl;// error
      // func()返回的是一个临时变量值,不允许对一个临时变量取地址,一个临时值没有地址返回给调用者

      cout << func2() << endl;
      cout << &func2() << endl; // OK func2 返回的是引用不是值.
}
```

4.3 注意事项

1. **不要返回局部变量的引用。**因为局部变量会在函数返回后被销毁，被返回的引用就成为了"无所指"的引用，程序会进入未知状态

   可以作为函数返回值的引用

1. 静态局部变量的引用
2. 全局变量的引用
3. 通过参数传入的引用（从函数外部传入，即可保证生命周期比函数长）
4. 类成员变量的引用（保证对象的生命周期）

2. **不要轻易返回一个堆空间变量的引用，非常容易造成内存泄漏**

```
int & func3(){
	int *p = new int{10};
	return *p;
}

void test(){
	// func3 调用 1 次就会 new 一次，如果不释放就会内存泄漏
	//cout << func3() << endl;
	//delete &func3();
	// 调用 2 次 func3,释放一次,仍然泄露

	// 完善写法,使用引用接收之后再处理
	int &ref = func3();
	cout << ref << endl;
	// delete
	delete &ref;
}
```

5、强制转换

- **static_cast**：用于正常状况下的类型转换

(1) 基本数据类型之间的转换

```
int iNumber = 100 ;
float fNumber = 0 ;
fNumber = static_cast<float>(iNumber);
```

(2) 把 void 指针转换成目标类型的指针，但不安全（可能因为指向的实际数据类型和期望类型不一致导致未定义行为，编译器不会检查）

```
void * pVoid = malloc(sizeof(int));
// void * ---> int *
int * pInt = static_cast<int*>(pVoid);
*pInt = 1;
```

(3) 用于类层次结构中基类和子类之间指针或引用的转换（向上转型）

```
class Base {};
class Derived : public Base {};

Derived d;
Base* b = static_cast<Base*>(&d);  // 派生类指针转基类指针
```

- **const_cast（了解）**：用于修改指针/引用的 const 属性
- **dynamic_cast**：用于处理多态类型的向下转换（基类指针/引用转派生类）

```
class Base { virtual void foo() {} };
class Derived : public Base {};

Base* b = new Derived;
Derived* d = dynamic_cast<Derived*>(b);  // 成功

Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // 返回 nullptr
```

- **reinterpret_cast**：万能转换，几乎不进行任何安全检查，一般不用

6、函数重载

- **定义**：在同一作用域内，一组具有相同函数名，不同参数列表的函数
- **条件**：函数名相同，参数列表不同（只有返回类型不同，不能构成重载）

  1. 函数参数的数量不同

2. 数量相同，类型不同
3. 数量，类型都相同，参数的顺序不同

- **实现原理：**名字改编，当函数名称相同时，会根据参数的类型、顺序、个数进行改编
- **赋默认值时要注意冲突**

```
void print(int x);
void print(int x, int y = 0);

print(10);  // 错误：两个版本都匹配
```

7、inline 函数

- **作用与原理：**在普通函数定义之前加上 inline 关键字
  1. inline 是一个建议，并不是强制性的，可能会失效
  2. inline 的建议如果有效，就会在编译时展开，可以理解为是一种更高级的代码替换机制（类似于宏——预处理）
  3. 函数体内容如果太长或者有循环之类的结构，不建议 inline，以免造成代码膨胀；比较短小并且比较常用的代码适合用 inline。
- **与宏函数对比**

类型安全	**提供类型安全，编译器进行类型检查**	**没有类型检查，可能产生不匹配的错误**
编译期替换	编译器决定是否内联（有优化机制）	预处理器简单文本替换
特性	内联函数 (inline)	宏 (#define)
代码可读性和调试性	支持断点调试，可读性和普通函数相似	调试困难，无法跟踪宏的展开过程

副作用	参数只求值一次，不会有多次求值副作用	参数会多次求值，可能导致副作用
代码膨胀	函数被多次内联可能导致代码膨胀	频繁替换也会导致代码膨胀
灵活性	适用于明确类型的函数	可以处理不同类型的参数
性能	小型函数可以避免函数调用开销	无函数调用开销

- **使用场景**

  1、 **内联函数**适用于需要提高性能的小型、频繁调用的函数，特别是需要进行类型检查和避免副作用的场景。对于需要安全性和封装性的代码段，应优先使用内联函数

  2、 **宏**适用于简单的文本替换、条件编译、或者需要通用计算而不考虑类型的情况下。然而，应该尽量避免使用宏函数来实现复杂的逻辑

- **注意事项**

  1、 如果要把 inline 函数声明在头文件中，则必须把函数定义也写在头文件中

  2、 若 Inline 函数定义在源文件中就只能本文件使用

8、内存布局（重点）

- **栈区**：操作系统控制，由高地址向低地址生长
- **堆区**：程序员分配，由高地址向低地址生长
- **全局/静态区**：读写段或数据段（这里说明全局/静态区是可读写的），存放全局变量、静态变量
- **文字常量区**：只读段，存放程序中直接使用的常量，如 const 修饰的变量，字符串常

量等

- **程序代码区**：只读段，存放函数体的二进制代码

9、C 风格字符串

- **定义**：以 '\0' 结尾的字符串
- **形式**

  1、 字符数组形式

```
// 字符数组形式
char str1[] = "hello";
// 等价于下面 最后一位'\0'
char str2[] = {'h','e','l','l','o','\0'};
```

  2、 字符指针形式

```
// 字符指针形式
//c++标准 要使用 const char *
// 即指向常量的指针,不能通过指针修改
const char *str1 = "hello";
const char *str2 = "world";
cout << "str1 = " << str1 << endl;
cout << "str2 = " << str2 << endl;
/* str1[0] = "H"; // error read only*/
str1 = "wd";
cout << "str1 = " << str1 << endl;
```

二、 类与对象

1. 基础

1、 类与对象的概念性知识（重点）

- **面向对象与面向过程的区别**

  过程论：数据和逻辑是分离的，程序世界本质是过程，而数据作为过程处理对象。逻

辑作为过程的形式定义，世界是过程的总体

对象论：数据和逻辑不是分离的，而是相互依存的。相关的数据和逻辑形成个体，即是对象，世界由一个个对象组成

- **面向对象三大基本特征：**封装、继承、多态（若问四种则加上抽象）

2、类的定义

- **访问修饰符（类的默认访问权限为 private）**

  public:公有访问权限，在类外可以通过对象直接访问公有成员

  protected:保护访问权限，在本类和派生类中可以访问，在类外不能通过对象直接访问

  private:私有访问权限，在本类之外不能访问

- **struct 与 class 对比**
  1. C 中的 struct 只能是一些变量的集合体，只能封装数据而不能隐藏数据，而且成员不能是函数
  2. C++中的 struct 对 C 中的 struct 做了扩展，可以定义函数，也可以被继承，基本等同于 class
  3. struct 的默认访问和继承权限都是 public，而 class 的默认访问和继承权限都是 private
- **数据成员：**类中定义的变量，用于存储对象的状态信息，对象的大小只与数据成员有关，与成员函数无关（若有虚函数则需要包括虚函数指针）
- **成员函数定义形式**
  1. 成员函数定义在类内部，默认情况会被认为是内联函数
  2. 成员函数在类内部只进行声明，在类外部完成定义，默认为非内联函数，除非使用 inline 来显式声明
  3. 成员函数在类中声明并使用头文件，成员函数的定义使用实现文件
- **成员类型（内部类和外部类的访问权限问题）**

  1. 外部类访问内部类成员

  a. 外部类总是能访问自己的内部类（无论内部类是 public/protected/private）。

  b. 但访问内部类的具体成员时，仍需遵守该成员的访问权限（如内部类的 private 成员不可直接访问）。

  2. 内部类访问外部类成员

  - 内部类可以访问外部类的 所有成员（包括 private），但必须通过：

i. 外部类的对象（如 outerObj.secret）。

ii. 外部类的指针/引用（如 outerPtr->secret）。

iii. 外部类的 static 成员（如 Outer::static_secret）。

3. **类外代码访问内部类成员**

a. 内部类本身是 public（如 Outer::PublicInner）

b. 内部类的具体成员是 public

3、 特殊数据成员

- **常量数据成员**：当数据成员用 const 修饰后，成为常量数据成员，必须在初始化列表中进行初始化，初始化之后不能再修改其值

```
class Point {
public:
    Point(int ix, int iy)
    : m_ix(ix)
    , m_iy(iy)
    {}
private:
    // C++11 后允许在声明时进行初始化
    //  在这里初始化的值理解为默认值
    const int m_ix = 1;
    const int m_iy = 1;
};

void test(){
    Point p1(10,20);
    Point p2 = p1;

    // p2 = p1; // error  不能进行赋值操作
}
```

- **引用数据成员**：**必须在初始化列表中进行初始化**，C++11 后允许在声明时初始化（绑定），**引用成员需要绑定一个已经存在的、且在这个引用成员的声明周期内始终有效**

的变量

```
class Point
{
public:
    Point(int x,int y,int z)
    : m_ix(x)
    , m_iy(y)
    , m_iz(z) //这样绑定是不行的，
    {}              //z 为值传递，在构造函数结束后就被销毁了


private:
    int m_ix;
    int m_iy;
    int & m_iz;
    // int & m_iz = m_ix; C++11 之后允许在声明时初始化（绑定)
};
```

- **对象成员**：一个类对象作为另一个类对象的数据成员，**对象成员要在初始化列表中进行初始化**（初始化列表中需要写的是对象成员的名称，而不是对象成员的类名）

```
class B
{
public:
	B(int num)
	: m_numB(num)
	{
		cout << "B constructor" << endl;
	}
	~B()
	{
		cout << "B destructor" << endl;
	}
private:
	int m_numB;
};

class C
{
public:
	C(int num)
	: m_numC(num)
	{
		cout << "C constructor" << endl;
	}
	~C()
	{
		cout << "C destructor" << endl;
	}
private:
	int m_numC;


};
```

构造与析构顺序（重点）

1. 创建 A 对象会马上调用 A 的构造函数
2. 在 A 的构造函数执行过程中调用 B 的构造函数和 C 的构造函数（先构造 B
3. A 对象要销毁，就会马上调用 A 的析构函数
4. A 析构函数执行完之后，再根据对象成员声明的反序
5. 通过成员子对象调用 B 和 C 的析构函数
6. m_c 调用析构函数，执行完后，m_b 再调用析构函数

```
C constructor
B constructor
A constructor
A destructor
B destructor
C destructor
```

- **静态数据成员**：使用 static 修饰的数据成员，在编译时就会被创建并初始化
  1. 静态数据成员和静态变量一样，当程序执行时，该成员已经存在，一直到程序结束，任何该类对象都可对其进行访问
  2. 静态数据成员存储在全局/静态区，并不占据对象的存储空间
  3. 静态数据成员被整个类的所有对象共享

静态成员规则

1. private 的静态数据成员无法在类之外直接访问（显然）
2. 对于静态数据成员的初始化，必须放在类外（一般紧接着类的定义，这是规则 1 的特殊情况）
3. 静态数据成员初始化时不能在数据类型前面加 static，在数据成员名前面要加上类名+作用域限定符（int Student::ms_classID = 2;）
4. 如果有多条静态数据成员，那么它们的初始化顺序需要与声明顺序一致（规范）
5. 静态成员在访问时可以通过对象访问，**也可以直接通过类名::成员名的形式**（更常用）

- **指针数据成员：在初始化列表中申请空间，在函数体中复制内容**

4、特殊成员函数

- **静态成员函数**

1. **静态成员函数不依赖于某一个对象**
2. 静态成员函数可以通过对象调用，但更常见的方式是通过类名加上作用域限定符调用
3. 静态成员函数没有 this 指针
4. 静态成员函数中无法直接访问非静态的成员，只能访问静态数据成员或调用静态成员函数（因为没有 this 指针），但是非静态成员函数可以访问静态成员
5. 构造函数、拷贝构造、赋值运算符函数、析构函数比较特殊，可以在静态成员函数中调用
6. 静态成员函数不能是构造函数/析构函数/赋值运算符函数/拷贝构造（因为这四个函数都会访问所有的数据成员，而 static 成员函数没有 this 指针）

- **const 成员函数：**void func() const{}
  1. const 成员函数中，不能直接修改对象的非静态的普通数据成员；
  2. 当编译器发现该函数是 const 成员函数时，会自动将 this 指针设置为双重 const 限定的指针 const Type* const pointer，所以可以进行函数重载，同名的非 const 版本的成员函数和 const 版本的成员函数
- **this 指针：**每个非静态成员函数在编译时，编译器会自动添加一个隐藏的 this 指针加入到参数列表的第一位，指向调用该函数的对象实例

```
class MyClass {
public:
    void print(int x) {
        std::cout << x << ", " << this->data;
    }
    int data = 42;
};
```

编译器处理后的等价形式

```
// 非静态成员函数被改写为普通函数，增加 this 参数
void print(MyClass* const this, int x) {
    std::cout << x << ", " << this->data;
}
```

5、 最重要的成员函数（重点）

5.1 构造函数

- **构造函数并不是创建了对象，而是为对象初始化数据成员**
- **构造函数初始化和列表初始化的区别：**构造函数初始化是在构造函数体内对成员变量进行赋值操作，而列表初始化是在构造函数的初始化列表中直接对成员变量进行初始化，列表初始化更加高效，可以避免不必要的临时变量和拷贝操作

5.2 析构函数

1. 对象在销毁时会自动调用析构函数，清理为对象分配的资源
2. 析构函数只有一个且无法重载，若为对象分配了堆空间的内存，那么需要自定义析构函数以释放堆空间
3. 不要手动调用析构函数，容易导致各种问题

- **析构函数调用时机（重点）**

1. 对于全局对象，整个程序结束时，自动调用全局对象的析构函数
2. 对于局部对象，在程序离开局部对象的作用域时调用对象的析构函数
3. 对于静态对象，在整个程序结束时调用析构函数
4. 对于堆对象，在使用 delete 删除该对象时，调用析构函数

5.3 拷贝构造函数

1. **形式：**类名（const 类名&）
2. **定义：**拷贝构造函数也是构造函数，用一个已经存在的同类型对象来初始化新对象

- **拷贝构造函数调用时机（重点）**

1. 当使用一个已经存在的对象初始化一个新的同类型的对象时
2. 当函数参数类型是对象，在实参传入与形参结合时
3. 当函数的返回值是对象时

- **拷贝构造函数的形式问题（重点）**

1. **参数中传入引用**是为了避免无限拷贝的问题，若去掉引用，那么实参传入时是值传递，会触发拷贝构造函数，而在拷贝构造函数调用时又会调用拷贝构造函数，会导致无限递归调用导致栈溢出

2. **参数加上 const** 首先是保证原始对象的数据成员不被改变，其次是为了能够复制临时对象的内容，非 const 引用无法绑定临时变量（右值）

- **左值与右值（重点）**

1. **左值：占据内存并可以取地址的对象，也可以说左值是可以放在 '=' 左边的内容**
2. **右值 :没有明确存储位置的临时对象 ,不可取地址 ,只能放在 '=' 的右边 ,**临时变量、临时对象、匿名变量、匿名对象、字面值常量等都是右值

5.4 赋值运算符函数

1. **形式：**类名& operator=(const 类名 &)
2. **定义：**用一个已经存在的对象给另一个已经存在的对象进行赋值，会调用赋值运算符函数
3. **重载原因：**若对象的指针数据成员申请了堆空间，默认的赋值运算符函数会不够用
4. **重载规范**

```
Computer & operator=(const Computer & rhs){
    // 1.自赋值情况判断，若不进行检查，后面的 delete[]会先释放内存
    // 导致访问已释放的内存（rhs 和 this 是同一个对象）
    if(this != &rhs){
        // 回收当前对象中指针成员原来申请的堆空间，避免内存泄漏
        delete [] m_brand;
        // 深拷贝
        m_brand = new char[strlen(rhs.m_brand)]();
        strcpy(m_brand,rhs.m_brand);
        m_price = rhs.m_price; // 其他数据成员的简单赋值
    }
    // 返回当前对象，以便于支持链式赋值（a=b=c）
    return *this;
}
```

- **赋值运算符函数形式问题**

1. 返回值为引用是为了避免不必要的拷贝，减少开销
2. 返回类型不能为 void 是为了支持连续赋值

3. 参数传入引用也是为了避免不必要的拷贝
4. 参数设置为 const 引用首先是为了避免对传入右操作数进行修改，其次是为了能支持传入右值（非 const 引用无法绑定右值）

- **三合成原则**

  拷贝构造函数、赋值运算符函数、析构函数，如果需要手动定义其中的一个，那么另外两个也需要手动定义。

5.5 移动构造函数（重要）

1. **目的：**在构造函数中传入临时对象时减少拷贝，能够分辨右值引用
2. **形式：**类名（类名 &&other）
3. **重载示例**

```
String(String && rhs)
    : m_pstr(rhs.m_pstr)
    {
        cout << "String(String&&)" << endl;
        rhs.m_pstr = nullptr;
    }
```

4. **特点**
   a. 如果没有显式定义构造函数、拷贝构造、赋值运算符函数、析构函数，编译器会自动生成移动构造，对右值的复制会调用移动构造
   b. 如果显式定义了拷贝构造，而没有显式定义移动构造，那么对右值的复制会调用拷贝构造
   c. 如果显式定义了拷贝构造和移动构造，那么对右值的复制会调用移动构造

总结：移动构造函数优先级高于拷贝构造函数

5.6 移动赋值函数（重要）

1. **目的：**避免在使用临时对象给一个已经存在的对象进行赋值时造成的拷贝开销
2. **形式：**类名& operator=（类名&& rhs）
3. **重载示例**

```
String & operator=(String && rhs){
    if(this != &rhs){
        delete [] m_pstr;
        //浅拷贝
        m_pstr = rhs.m_pstr;
        rhs.m_pstr = nullptr;
        cout << "String& operator=(String&&)" << endl;
    }
    return *this;
}
```

4. **特点**
    a. 如果没有显式定义构造函数、拷贝构造、赋值运算符函数、析构函数，编译器会自动生成移动赋值函数。使用右值的内容进行赋值会调用移动赋值函数。
    b. 如果显式定义了赋值运算符函数，而没有显式定义移动赋值函数，那么使用右值的内容进行赋值会调用赋值运算符函数。
    c. 如果显式定义了移动赋值函数和赋值运算符函数，那么使用右值的内容进行赋值会调用移动赋值函数。

总结：移动赋值函数优先级也是高于赋值运算符函数

6. 单例模式（重点）

6.1 定义

确保每次获取的都是同一个唯一的对象，在系统中只能出现类的一个实例，一个类只有唯一的对象

6.2 单例对象的创建

6.2.1 静态单例对象：C++11 后单例标准

1. 将构造和析构函数私有，拷贝构造和赋值运算符函数删除
2. 通过静态成员函数 getInstance 创建局部静态对象，确保对象的生命周期和唯一性，返回值设置为引用，避免复制
3. 设置一个静态的指针数据成员，保存第一次创建出的单例对象地址，后续的使用

和回收都依赖这个指针

4. 设置共有的静态成员函数来回收单例对象

```
class Singleton
{
public:
    // 提供一个静态方法返回静态对象
    // 返回值设置为引用 避免复制
    static Singleton& getInstance()
    {
        // 声明并初始化局部静态变量对象 静态对象只会被初始化一次
        // 后续调用中每次会返回同一对象
        static Singleton instance;
        return instance;
    }
private:
    // 构造函数私有 确保不能在外部创建对象
    Singleton(){
        cout << "default constructor" << endl;
    }
};
void test1(){
    Singleton & instance = Singleton::getInstance();
    cout << &instance << endl;
    Singleton & instance2 = Singleton::getInstance();
    cout << &instance2 << endl;
}
```

隐患：如果单例对象所占空间较大，可能会对静态区造成内存压力（极少出现此种情况）

6.2.2 堆上单例对象（重点）

1. 构造函数私有
2. 通过静态成员函数 getInstance 创建堆上的对象，返回相应类型的指针
3. 通过静态成员函数完成堆对象的回收

```
class Singleton
{
public:
    // 提供一个静态方法返回静态对象
    static Singleton* getInstance()
    {
        // 在堆上创建对象
        if(ms_instance == nullptr){
            // 为 nullptr 才创建
            ms_instance = new Singleton();
        }
        return ms_instance;
    }
    // 提供一个静态方法来销毁对象 释放空间
    static void destroyInstance()
    {
        if(ms_instance != nullptr){
            delete   ms_instance;
            ms_instance = nullptr;
        }
    }
    void func()
    {
        cout << "func()" << endl;
    }
private:
    // 构造函数私有 确保不能在外部创建对象
    Singleton(){
        cout << "default constructor" << endl;
    }
    // 析构函数私有 避免外部删除对象
    ~Singleton()
    {
```

6.3 单例对象的自动释放（重点）

6.3.1 利用局部对象的声明周期管理堆上的单例对象（缺点较多，一般不用）

```cpp
#include <iostream>

using std::cout;
using std::endl;

class AutoRelease;

class Singleton
{
private:
    Singleton()
    {
        cout << "default constructor" << endl;
    }

    ~Singleton()
    {
        cout << "destrouctor" << endl;
    }
    // 静态指针，指向唯一单例对象
    static Singleton *pInstance;
    Singleton(const Singleton &) = delete;
    Singleton &operator=(const Singleton &) = delete;

public:
    friend class AutoRelease;
    // 静态函数，用于获取唯一实例对象
    static Singleton *getInstance()
    {
        if (pInstance == nullptr)
        {
            pInstance = new Singleton();
        }
```

6.3.2 嵌套类+静态局部对象实现单例对象的自动释放（重点）

```cpp
#include <iostream>

using std::cout;
using std::endl;

class Singleton
{
    // 嵌套类，负责自动释放单例对象
    class AutoRelease
    {
    public:
        AutoRelease()
        {
            cout << "AutoRelease()" << endl;
        }

        ~AutoRelease()
        {
            cout << "~AutoRelease()" << endl;
            if (Singleton::pInstance)
            {
                delete Singleton::pInstance;
                Singleton::pInstance = nullptr;
                cout << "Singleton instance destroyed" << endl;
            }
        }
    };

public:
    static Singleton *getInstance()
    {
        if (pInstance == nullptr)
        {
```

优点（与局部对象对比）

（1）无需手动管理生命周期

- 局部 AutoRelease 对象（第一种方式）必须在某个作用域内创建，例如 main 函数内

```
int main() {
    AutoRelease ar(Singleton::getInstance());  // 必须手动创建
    // ...
    return 0;  // ar 析构，释放单例
}
```

- 如果忘记创建 AutoRelease 对象，或者它在不合适的时机析构（比如单例还被其他代码使用时），可能导致问题

（2）更符合单例的全局性

- 单例通常是全局唯一的，而局部 AutoRelease 对象可能会在多个地方创建，导致管理混乱

（3）避免悬空指针问题

- 如果单例在程序运行期间被手动 delete（比如调用 destroyInstance()），局部 AutoRelease 对象可能仍然持有无效指针，导致二次 delete（UB）。
- 静态 AutoRelease 对象始终在 程序结束时 才释放单例，避免了中间状态的不一致

6.3.3atexit 注册 destroy

- **注册函数的调用顺序**：如果注册了多个函数，先注册的后执行，同一个函数注册几次，在程序结束时就会调用几次

```cpp
#include <iostream>
#include <cstdlib>

using std::cout;
using std::endl;

class Singleton
{
private:
    int m_x;
    int m_y;
    static Singleton *pInstance;

private:
    Singleton()
        : m_x(0), m_y(0)
    {
        cout << "Singleton()" << endl;
    }
    ~Singleton()
    {
        cout << "~Singleton()" << endl;
    }
    // 禁止拷贝和赋值
    Singleton(const Singleton &) = delete;
    Singleton &operator=(const Singleton &) = delete;

public:
    // 获取单例实例
    static Singleton *getInstance()
    {
        if (pInstance == nullptr)
        {
```

问题：多线程不安全，当多线程同时进入，由于静态指针 pInstance 由多线程共享，多线程竞争会造成单例对象被创建出多个 ,但是只有最后一个 Singleton 对象会被 pInstance 管理，之前创建的单例对象都会内存泄漏

6.3.4pthread_once + atexit(destroy)

- pthread_once 函数可以确保初始化代码只会执行一次，无论在多少个线程中调用它
- pthread_once 函数参数形式，第一个参数固定，第二个参数为一个静态函数指针

```cpp
#include <iostream>
#include <pthread.h>

using std::cout;
using std::endl;

class Singleton
{
    class AutoRelease
    {
    public:
        AutoRelease()
        {
            cout << "AutoRelease()" << endl;
        }

        ~AutoRelease()
        {
            Singleton::destroy();
        }
    };

private:
    int _x;
    int _y;
    static Singleton *ms_pInstance;
    static AutoRelease ms_autorelease;
    static pthread_once_t ms_once;

private:
    Singleton() = default;
    ~Singleton() = default;

```

2.继承

1、 单继承

- **基本概念**：一个派生类（子类）直接继承自一个基类（父类）
- **访问权限**

继承方式	基类成员访问权限	在派生类中访问权限	在类外派生类对象对基类成员的访问
公有继承 public	public protected private	可以访问（保持public） 可以访问（保持protected） 不可直接访问	可以直接访问 不可直接访问 不可直接访问
保护继承 protected	public protected private	可以访问（protected属性） 可以访问（protected属性） 不可直接访问	均不可直接访问
私有继承 private	public protected private	可以访问(private属性） 可以访问(private属性） 不可直接访问	均不可直接访问

1. **总结**
    a. 不管什么继承方式，派生类内部都不能访问基类的私有成员；
    b. 不管什么继承方式，派生类内部除了基类的私有成员不可以访问，其他的都可以访问；
    c. 不管什么继承方式，派生类对象在类外除了公有继承基类中的公有成员可以访问外，其他的都不能访问。
2. **保护继承和私有继承区别**

a. protected：无限继承

b. private：继承一次就终止

3. **面试常考问题总结**

   a. **派生类在类之外对于基类成员的访问 ，具有什么样的限制？**

      只有公有继承自基类的公有成员，可以通过派生类对象直接访问，其他情况一律都不可以进行访问

   b. **派生类在类内部对于基类成员的访问 ，具有什么样的限制？**

      对于基类的私有成员，不管以哪种方式继承，在派生类内部都不能访问；

      对于基类的非私有成员，不管以哪种方式继承，在派生类内部都可以访问；

   c. **保护继承和私有继承的区别？**

      如果继承层次中都采用的是保护继承，任意层次都可以访问顶层基类的非私有成员；但如果采用私有继承之后，这种特性会被打断。

- **内存结构**

  1. **构造顺序**

     a. 调用派生类的构造函数为派生类对象开辟空间

     b. 在初始化列表时调用基类的构造函数，完成基类数据成员的初始化

     c. 最后执行派生类构造函数的函数体

  2. **析构顺序**

     a. 先调用派生类的析构函数

     b. 再调用派生类中对象成员的析构函数

     c. 最后调用基类的析构函数

- 成员隐藏

  1. **基类数据成员的隐藏**：派生类中声明了和基类的数据成员同名的数据成员，就会对基类的这个数据成员形成隐藏，无法通过派生类对象直接访问基类的这个数据成员，但是基类的这个数据成员依旧存在，只是无法通过派生类对象直接访问了而已

  2. **基类成员函数的隐藏**：当派生类声明了与基类同名的成员函数时，只要名字相同，即使参数列表不同，也只能看到派生类部分的，无法通过派生类对象直接调用基类的同名函数，只需要派生类中定义一个与基类成员函数同名的函数即可，函数返回类型、参数情况都可以不同

2、多继承

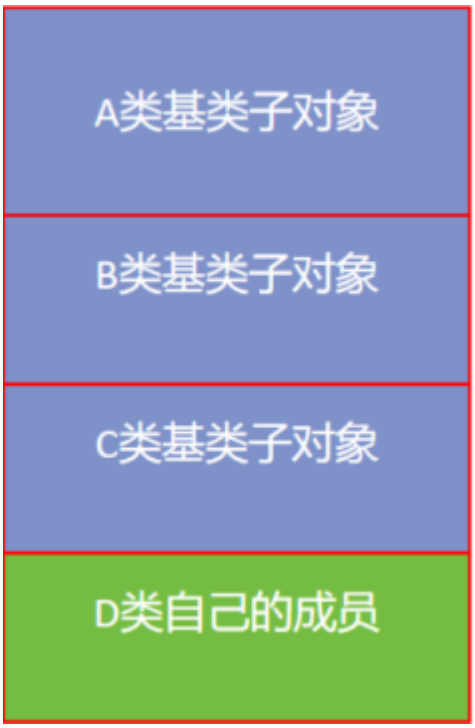
- **构造与析构顺序**

  1. 构造顺序

```
class D
: public A
, public B
, public C
{
public:
    D(){ cout << "D()" << endl; }
    ~D(){ cout << "~D()" << endl; }
};
```

此结构下创建 D 类对象时，先调用 D 类的构造函数，然后根据继承的声明顺序，依次调用 A/B/C 类的构造函数，创建三个类的子对象，即顺序为 A->B->C->D

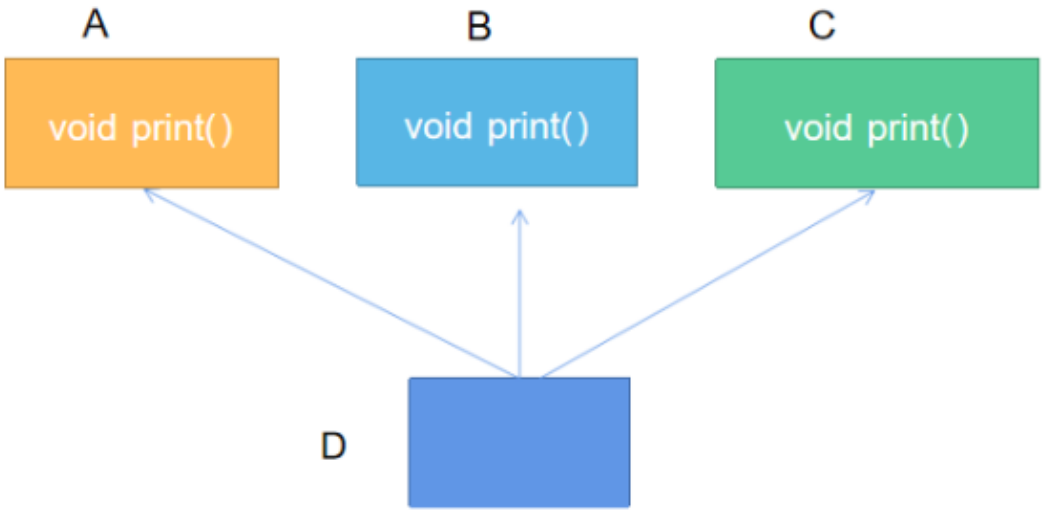
内存布局如下图所示



2. 析构顺序：与构造顺序相反，即 D->C->B->A

- **多继承二义性问题（重点）**

1. **成员名访问冲突二义性**

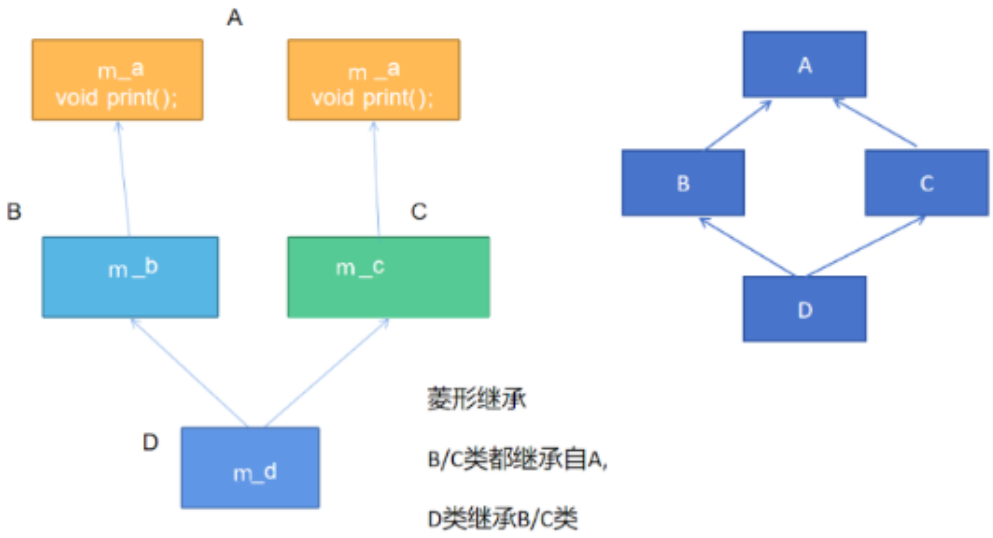



```
//多继承的二义性
void test3()
{
	D d{};
	/* d.print(); //  成员名访问冲突*/
	// member 'print' found in multiple base classes of different types
	//  通过类名::作用域方式解决冲突 ， 指明调用是哪个基类的方法
	d.A::print();
	d.B::print();
	d.C::print();
}
```

如果 D 类中声明了同名的成员，那么会对基类的同名成员造成隐藏效果，此时可以直接通过对象访问

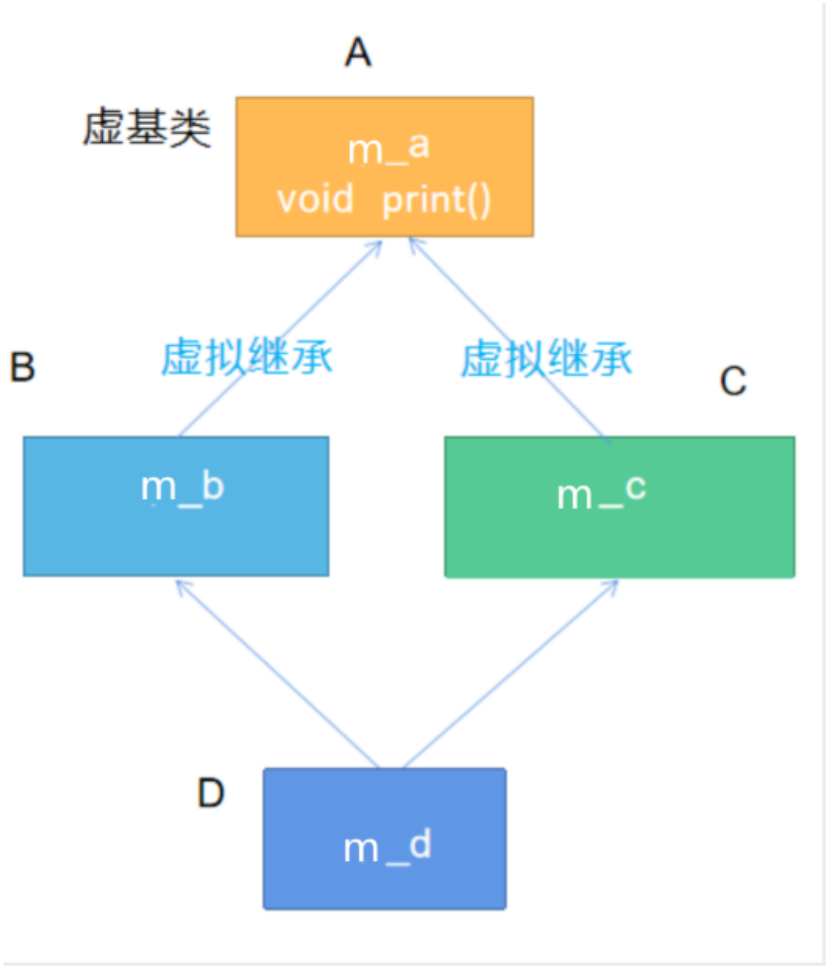
```
D d;
d.A::print();
d.B::print();
d.C::print();
d.print(); //ok
```

2. **存储二义性问题（菱形继承问题）：由中间层虚拟继承顶层基类引起**

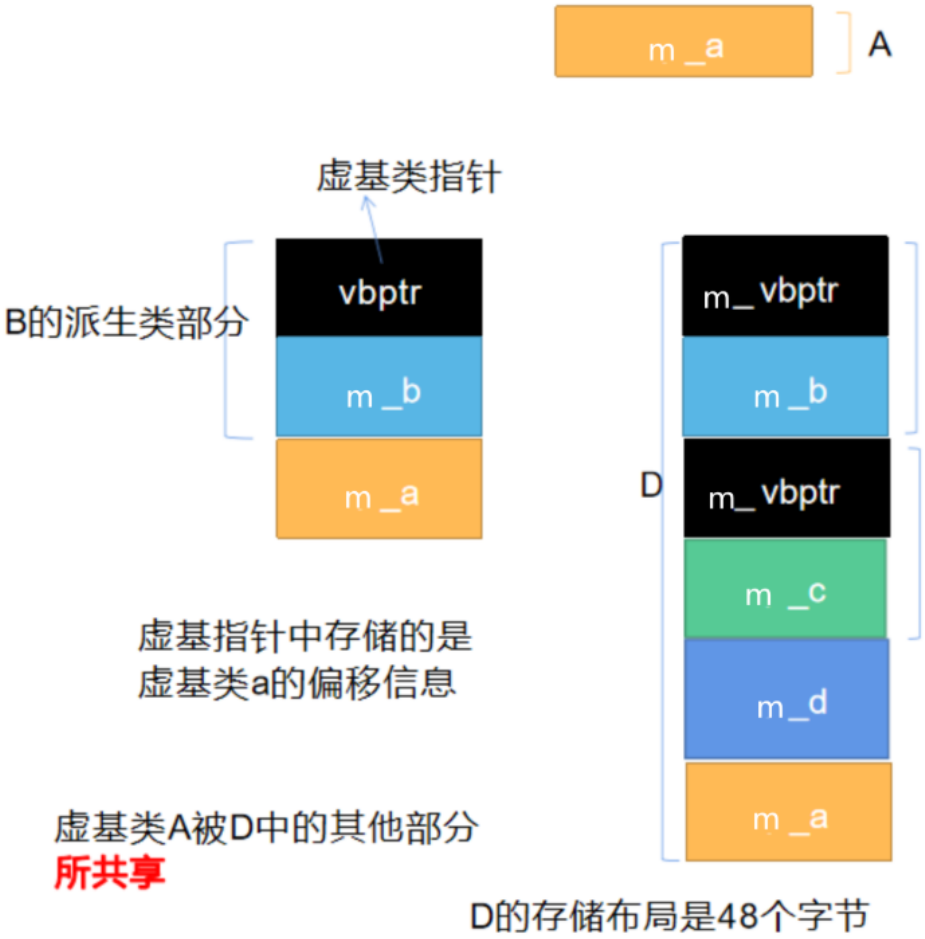


原因：菱形继承情况下，D 类对象的创建会生成一个 B 类子对象，其中包含一个 A 类子对象；还会生成一个 C 类子对象，其中也包含一个 A 类子对象。所以 D 类对象的内存布局中有多个 A 类子对象，访问继承自 A 的成员时会发生二义性**(无论是否涉及 A 类的数据成员，单纯访问 A 类的成员函数也会冲突)**

解决办法：中间层采用虚拟继承顶层基类的方式



内存布局：采用虚拟继承的方式处理菱形继承问题，实际上改变了派生类的内存布局。B 类和 C 类对象的内存布局中多出一个虚基类指针，位于所占内存空间的起始位置，同时继承自 A 类的内容被放在了这片空间的最后位置。D 类对象中只会有一份 A 类的基类子对象



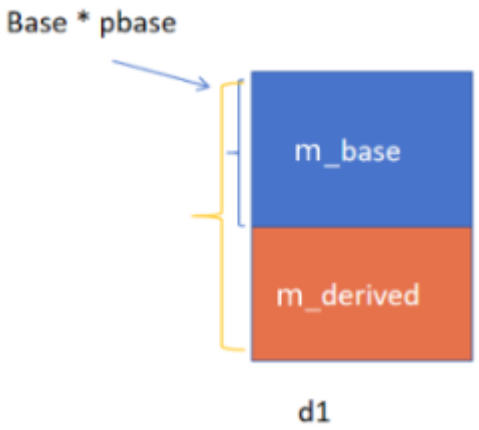
3、 基类与派生类之间的转换

- **向上转型**

  1. **定义：**用派生类型给基类对象/指针/引用赋值

```
Base base;
Derived d1;

// 用基类对象接受派生类对象的赋值
base = d1; //ok
d1 = base; //error

// 用基类引用绑定派生类对象
Base * pbase = &d1; //ok
Derived * pderived = &base //error

// 用基类指针指向派生类对象
Base & rbase = d1; //ok
Derived & rderived = base; //error

```

2. **本质：** Base 类的指针指向 Derived 类的对象，d1 对象中存在一个 Base 类的基类子对象，这个 Base 类指针所能操纵只有继承自 Base 类的部分



- **向下转型**
  1. **定义：**即用基类给派生类对象/指针/引用赋值
  2. **一般不能直接向下转型：**因为派生类指针能操纵的空间一般比基类要大，若向下转型，那么派生类指针将要操作的空间会有一片非法空间
  3. **合理转型（使用 dynamic_cast）**

```
Base base;
Derived d1;
// 这里可以向下转型成功，因为 pbase 本就是一个指向 Derived 对象的指针
Base * pbase = &d1;
Derived * pderived = dynamic_cast<Derived*>;
```

总结：能否向下转型成功要看基类指针原本指向的对象是基类对象还是派生类对象，若原本指向基类对象则无法转型，若指向派生类对象则可以成功转型，可以理解为基类指针指向派生类对象时有一片像派生类指针所能访问的空间大小，只是不属于基类对象的部分它没有权限访问，向下转型之后相当于权限放开了，能访问属于派生类对象大小的空间了

4、派生类对象的复制控制（重点）

1. 原则

a. 当派生类中没有显式定义复制控制函数时，就会自动完成基类部分的复制控制操作；

b. 当派生类中有显式定义复制控制函数时，不会再自动完成基类部分的复制控制操作，需要显式地调用；

2. 显式调用时机

a. 派生类自定义了拷贝构造函数

```
// 这里必须显式调用 Base 类的拷贝构造函数，
// 否则 Base 类的数据成员将保持原有数据不变
Derived(const Derived & rhs)
    : Base(rhs)//调用 Base 的拷贝构造
    , m_derived(rhs.m_derived)
    {
        cout << "Derived(const Derived & rhs)" << endl;
    }

```

b. 派生类自定义了赋值运算符函数

```
    // 这里必须显式调用 Base 类的赋值运算符函数，
   // 否则 Base 类的数据成员将保持原有数据不变
    Derived &operator=(const Derived & rhs){
            //调用 Base 的赋值运算符函数
            Base::operator=(rhs);
            m_derived = rhs.m_derived;
            cout << "Derived& operator=(const Derived &)" << endl;
            return *this;
        }
```

3. 多态

1、 基本概念

- **多态**：一个接口，多种方法，即同一操作作用域不同对象时，产生不同的行为
- **引入多态原因**
  1. 代码复用与简化
  2. 提升代码的可扩展性与灵活性
  3. 支持抽象化与接口统一
  4. 提高代码的可维护性

2、 C++多态类型（重点）

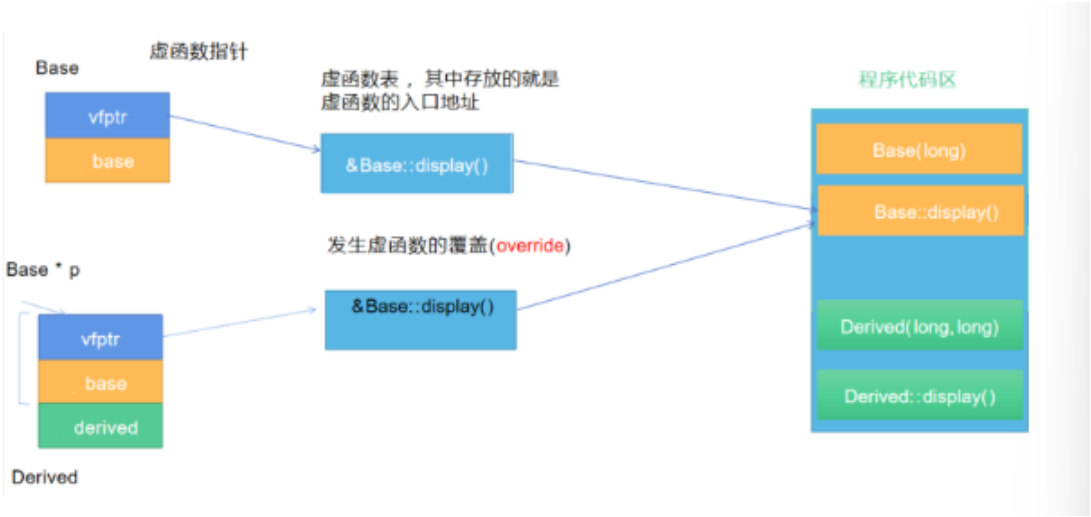
1. **编译时多态（静态多态）：函数重载、运算符重载**就是采用的静态多态，C++编译器根据传递给函数的参数和函数名决定具体要使用哪一个函数，不依赖于对象的实际类型，又称为**静态联编**
2. **运行时多态（动态多态）：在运行阶段，根据对象的实际类型决定调用哪个函数，实现动态绑定，**C++通过虚函数来实现**动态联编**

3、 虚函数机制（重点）

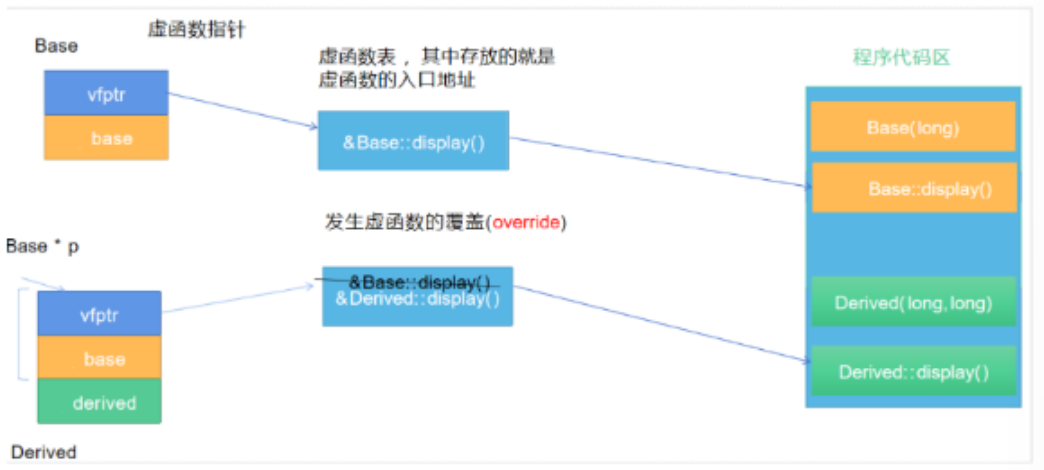
3.1 概念：在成员函数前加上 vitual 关键字，该函数就成为虚函数

- **实现原理**

当成员函数成为虚函数后，该类产生的对象的存储布局就改变了。在存储的开始位置会多加一个虚函数指针，该虚函数指针指向一张虚函数表（简称虚表），其中存放的是虚函数的入口地址



- **虚函数的覆盖机制（override）**：覆盖的是虚函数表中虚函数的入口地址



（1）覆盖是在虚函数之间的概念，需要派生类中定义的虚函数与基类中定义的虚函数的**形式完全相同**

（2）当基类中定义了虚函数时，派生类去进行覆盖，即使在派生类的同名的成员函数前不加 virtual，依然是虚函数

（3）发生在基类派生类之间，基类与派生类中同时定义形式相同的虚函数。覆盖的是虚函数表中的入口地址，并不是覆盖函数本身

3.2 动态多态（虚函数机制）触发条件（重点）

1. 基类定义虚函数
2. **派生类中要覆盖虚函数**（此时 vfptr 会指向派生类的虚函数表的入口地址）

3. 创建派生类对象

4. 基类的指针指向派生类对象（或基类引用绑定派生类对象）

5. **通过基类指针（引用）调用虚函数**

3.3 虚函数表与虚函数指针

- **面试常考题**

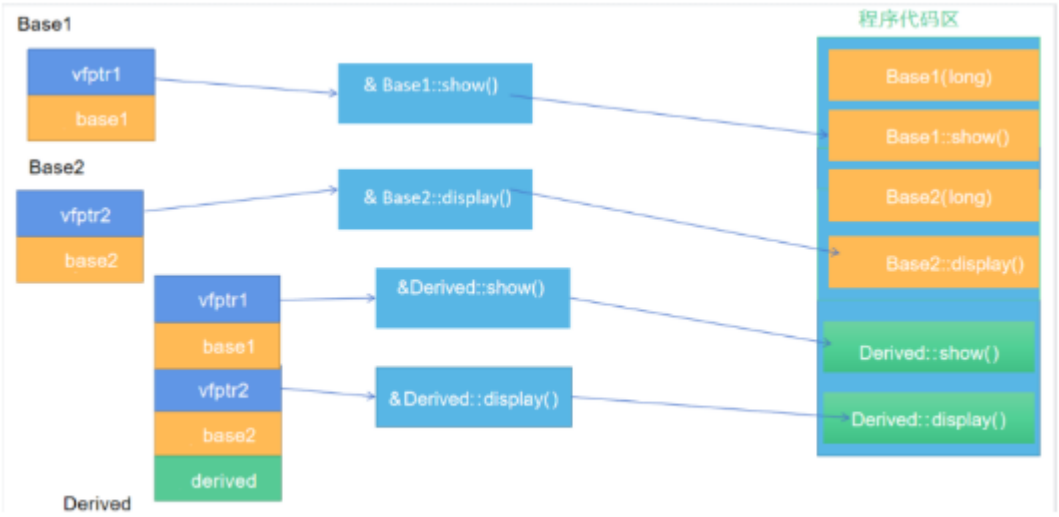
1、 **虚函数表存放在哪里**（一般存放在只读段）

2、 **一个类中虚函数表有几张**

a) 一个类可以没有虚函数表（没有虚函数就没有虚函数表）；

b) 可以有一张虚函数表，只继承单个基类时（**即使这个类有多个虚函数，将这些虚函数的地址都存在虚函数表中**）；

c) 也可以有多张虚函数表（**继承多个有虚函数的基类**）



3、 **虚函数机制的底层实现是怎样的**

虚函数机制的底层是通过虚函数表实现的。当类中定义了虚函数之后，就会在对象的存储开始位置，多一个虚函数指针，该虚函数指针指向一张虚函数表，虚函数表中存储的是虚函数入口地址

4、 **虚函数表的内容是在什么时候写入的**

虚函数表里的内容是在编译阶段确定的。编译器会遍历类的继承层次结构，找出所有的虚函数，并将它们的地址按照一定顺序存放到虚函数表中。每个类对象都包含一个指向其类对应虚函数表的指针，这个指针是在对象构造时由编译器自动设置的

5、 **重载、隐藏、覆盖的区分**

重载(overload)：发生在同一个类的作用域中， 允许多个同名函数 ，函数参数类型、

顺序 、个数任一不同(形参列表不同即可)，返回类型可以相同也可以不同，是编译时多态的体现，编译器根据参数列表来确定调用哪个函数

隐藏(oversee)：发生在基类派生类之间 ，函数名称相同时，就构成隐藏（参数不同也能构成隐藏），即使是基类的虚函数，如果派生类的函数参数列表与基类不同，会发生隐藏而不是覆盖

覆盖(override) :发生在基类派生类之间 ,基类与派生类中同时定义返回类型、参数信息、名字都相同的虚函数，覆盖的是虚函数表中的入口地址，并不是覆盖函数本身，覆盖是运行时多态的体现，当通过基类指针或引用来调用虚函数时，会调用派生类的覆盖版本

- **虚函数的限制**

1. 构造函数不能设为虚函数

构造函数的作用是创建对象，完成数据的初始化，**而虚函数机制被激活的条件之一就是要先创建对象，有了对象才能表现出动态多态。如果将构造函数设为虚函数，那此时构造未执行完，对象还没创建出来，存在矛盾**

2. 静态成员函数不能设为虚函数

虚函数的实际调用：this -> vfptr -> vtable -> virtual function，但是静态成员函数没有 this 指针，所以无法访问到 vfptr

3. Inline 函数不能设为虚函数

因为 inline 函数在编译期间完成替换，而在编译期间无法展现动态多态机制，所以起作用的时机是冲突的。如果同时存在，inline 失效

4. 普通函数不能设为虚函数

虚函数要解决的是对象多态的问题，与普通函数无关

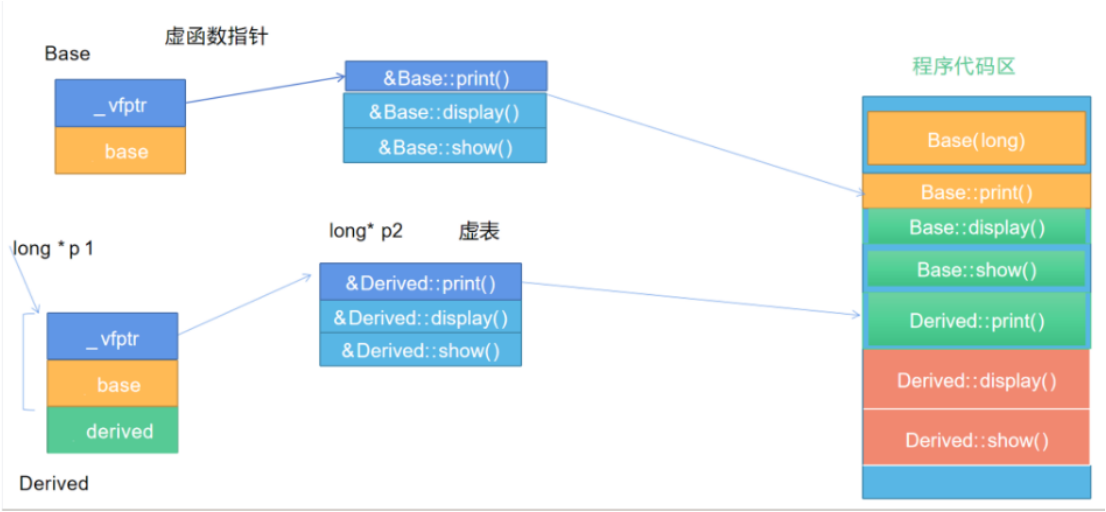
3.4 虚函数调用过程

（1）通过基类指针获取派生类对象的虚函数指针 vfptr

（2）通过 vfptr 找到派生类的虚函数表 vTable

（3）在虚函数表中查找调用的虚函数的条目，得到该虚函数的实际地址

（4）执行找到的函数（派生类的而非基类的）



4、虚函数的各种访问情况

4.1 通过派生类对象直接调用虚函数

并没有满足动态多态触发机制的条件，此时只是 Derived 中定义 display 函数对 Base 中的 display 函数形成了隐藏

4.2 在构造函数和析构函数中可以访问虚函数

a. 在构造函数/析构函数执行期间，虚函数的行为如同普通函数，不会触发动态绑定。
b. 调用的虚函数版本是当前正在构造/析构的类（或其基类）定义的版本，而不是派生类覆盖的版本

```cpp
class Base {
public:
    Base() {
        callVirtual(); // 调用 Base::callVirtual()，而非派生类版本
    }
    virtual void callVirtual() {
        cout << "Base::callVirtual" << endl;
    }
};

class Derived : public Base {
public:
    void callVirtual() override {
        cout << "Derived::callVirtual" << endl;
    }
};

int main() {
    Derived d; // 输出 Base::callVirtual
    return 0;
}
```

4.3 在普通成员函数中调用虚函数

```
class Base{
public:
	Base(long x)
	: m_base(x)
	{}

	virtual void display() const
	{
		cout << "Base::display()" << endl;
	}

	void func1()
	{
		display();
		cout << m_base << endl;
	}

	void func2()
	{
		Base::display();
	}
private:
	long m_base = 10;
};


class Derived
: public Base
{
public:
	Derived(long base,long derived)
	: Base(base)
	  m_derived(derived)
```

4.4 通过指针调用虚函数

此时基本上一定会走虚表

5、抽象类

5.1 定义

抽象类是至少声明了一个纯虚函数的类，未经覆盖直接继承纯虚函数的类也是抽象类；抽象类不能实例化对象，派生类只有实现基类全部的纯虚函数才能创建对象，若只实现了部分纯虚函数，那么该派生类也是抽象类

5.1.1 纯虚函数

一种特殊的虚函数，在基类中不能对虚函数给出实现，它的实现留给该基类的派生类去做

```
class 类名 {
public:
    virtual 返回类型 函数名(参数 ...) = 0;
};
```

5.1.2 类似于抽象类的类：只定义了保护属性的构造函数的类

```
class PseudoAbstract {
protected:
    PseudoAbstract() {}   // 只能被派生类构造
public:
    virtual ~PseudoAbstract() {}
};
```

注意：此时 PseudoAbstract 类不能在外部被实例化，但仍然可以在成员函数或派生类中实例化

（1）由于构造函数为 protected 属性，因此外部不能直接实例化

```
int main() {
    PseudoAbstract obj;  //      编译错误：构造函数不可访问
    return 0;
}
```

（2）可以在成员函数或派生类中构造

```
// 提供 static 方法给外界访问
class PseudoAbstract {
protected:
    PseudoAbstract() {}
public:
    virtual ~PseudoAbstract() {}
    static PseudoAbstract create() { return PseudoAbstract(); }  //      合法
};
```

```
// 派生类中依然可以进行构造
class Derived : public PseudoAbstract {
public:
    Derived() : PseudoAbstract() {}  //      合法
};
```

6、虚析构函数（重点）

1. **定义原因**：确保通过基类指针删除派生类对象时，能够正确调用派生类的析构函数，从而避免资源泄露和不完全销毁的问题
2. **总结**：在实际的使用中，如果有通过基类指针回收派生类对象的需求，都要将基类的析构函数设为虚函数
3. **规范建议**：一个类定义了虚函数，而且需要显示定义析构函数，就将它的析构函数设为虚函数

7、内存布局与虚表布局（重点）

7.1 单继承下虚表布局

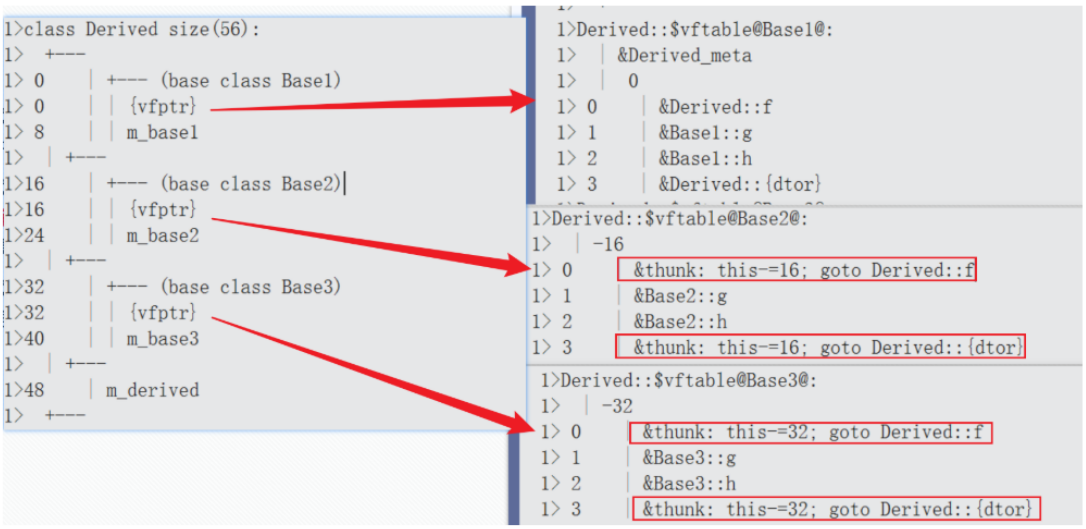
1. 虚函数表中记录的是虚函数的地址
2. 虚函数表中元素的排列顺序遵循基类中声明虚函数的顺序
3. 派生类若定义新的虚函数，新的虚函数地址会在已有的虚函数表的最后位置

7.2 带虚函数的多继承

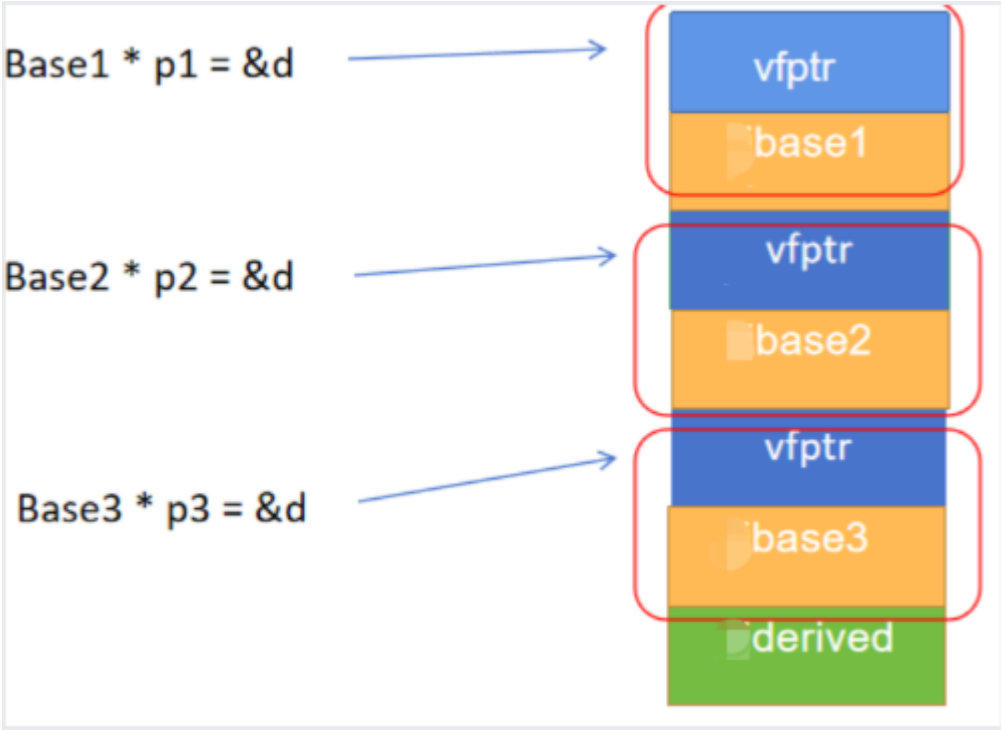
7.2.1 举例

```
class Base1
{
public:
    Base1()
    : m_base1(10)
    { cout << "Base1()" << endl; }

    virtual void f()
    {
        cout << "Base1::f()" << endl;
    }

    virtual void g()
    {
        cout << "Base1::g()" << endl;
    }

    virtual void h()
    {
        cout << "Base1::h()" << endl;
    }

    virtual ~Base1() {}
private:
    double m_base1;
};

class Base2
{
    //...
private:
    double m_base2;
};
```

vs 验证布局和虚函数表存放的内容



注意：三种不同的基类类型指针指向派生类对象时，实际指向的位置是相应类型的基类子对象的位置



7.2.2 布局规则

1. 每个基类都有自己的虚函数表（前提是基类定义了自己的虚函数）
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中

```
1>Derived::$vftable@Base1@:
1>  | &Derived_meta
1>  |  0
1> 0    | &Derived::f
1> 1    | &Base1::g
1> 2    | &Base1::h
1> 3    | &Derived::{dtor}
```

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(**有虚函数的基类会往上放——希望尽快访问到虚函数**)

   如果继承顺序为 Base1/Base2/Base3，在 Derived 对象的内存布局中就会先是 Base1 类的基类子对象，然后是 Base2、Base3 基类子对象

   此时，如果 Base1 中没有定义虚函数，那么内存排布上会将 Base1 基类子对象排在 Base2、Base3 基类子对象之后

4. 菱形继承时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；其它的虚函数表中对应位置存放的并不是真实的对应的虚函数的地址，而是一条跳转指令 —— 指示到哪里去寻找被覆盖的虚函数的地址

```
1>Derived::$vftable@Base2@:
1>  | -16
1> 0    | &thunk: this-=16; goto Derived::f
1> 1    | &Base2::g
1> 2    | &Base2::h
1> 3    | &thunk: this-=16; goto Derived::{dtor}
1>Derived::$vftable@Base3@:
1>  | -32
1> 0    | &thunk: this-=32; goto Derived::f
1> 1    | &Base3::g
1> 2    | &Base3::h
1> 3    | &thunk: this-=32; goto Derived::{dtor}
```

7.3 单继承与多继承在虚函数表上的差异

1. **单继承**

   - 派生类未重写基类虚函数时，派生类对象会共享基类的虚函数表
   - 若派生类重写了基类虚函数或自身新增了虚函数，那么派生类对象会有属于自身

的新的虚函数表

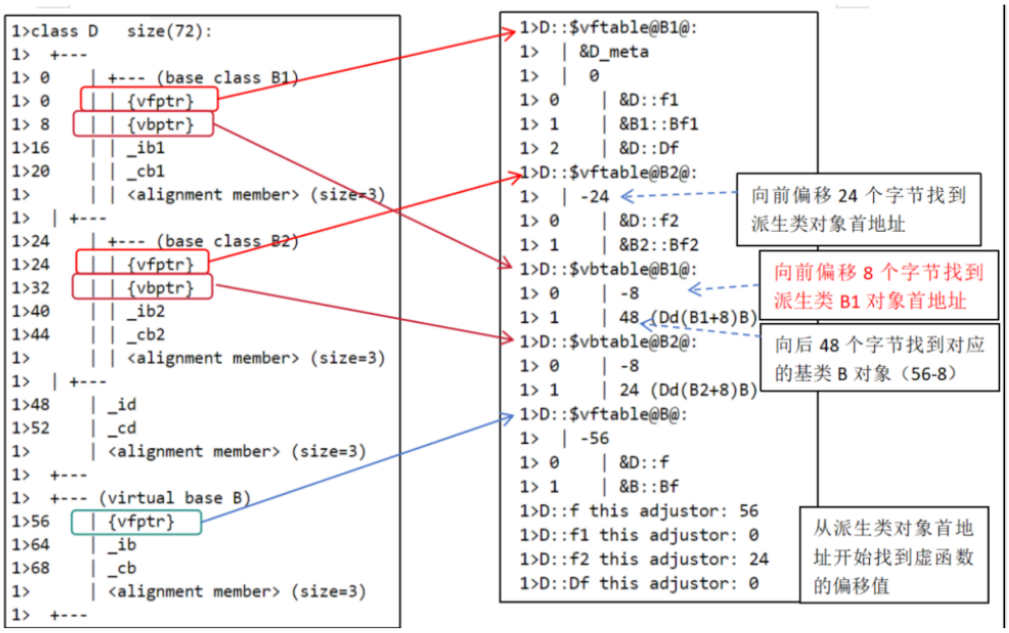
2. **多继承（派生类子对象不会拥有自己的虚表）**
    - 每个包含虚函数的基类子对象都有自己的虚函数指针和虚函数表
    - 派生类的虚函数会被合并到相关基类的虚函数表中

7.4 带虚函数的菱形继承的二义性

7.4.1 带虚函数的虚拟菱形继承

```
class B
{
public:
	virtual void f()
	{
		cout << "B::f()" << endl;
	}

	virtual void Bf()
	{
		cout << "B::Bf()" << endl;
	}
private:
	//double _b = 1;
	int _ib;
	char _cb;
};

class B1 : virtual public B {
public:
	virtual void f() { cout << "B1::f()" << endl; }
	virtual void f1() { cout << "B1::f1()" << endl; }
	virtual void Bf1() { cout << "B1::Bf1()" << endl; }
private:
	//double _b1 = 10;
	int _ib1;
	char _cb1;
};

class B2 : virtual public B {
public:
	virtual void f() { cout << "B2::f()" << endl; }
	virtual void f2() { cout << "B2::f2()" << endl; }
```

7.4.2 内存布局



- 中间基类会生成一个虚基指针指向虚基表，虚基表中记载着从当前子对象到虚基类首地址的偏移量

三、运算符重载

1、友元

友元能打破类的封装性，使得其它的类或函数能在类外访问某个类的私有成员

1.1.1 友元之普通函数形式

在类中将普通函数设置为友元，可通过该普通函数访问类中私有成员

```
class MyClass
{
public:
        MyClass(){}
        MyClass(int data)
        :m_data(data)
        {}
        // 声明友元函数
        friend void friendFunc(const MyClass & cls);
    }

    // 使用了 const
void friendFunc(const MyClass & cls)
{
        // 友元函数访问类中私有成员
        cout << cls.m_data << endl;
        cls.privateFunc();
}
```

1.1.2 友元之成员函数形式

前提：将 B 类前向声明，然后在 A 类中给出成员函数的声明，函数的定义写在类外

```
//前向声明 ：让编译器认识 MyClass 类
class MyClass;

class FriendClass
{
public:
	void display(MyClass & cls);
};

class MyClass
{
public:
	MyClass(){}
	MyClass(int data)
	:m_data(data)
	{}
	// 成员函数作为友元声明
	friend void FriendClass::display(MyClass & cls);

private:
	int m_data;
	void privateFunc(){
		cout << "privateFunc" << endl;
	}
};

void FriendClass::display(MyClass & cls)
{
	cout << cls.m_data << endl;
	cls.privateFunc();
}
```

1.1.3 友元类

```
class B {
    //...
    friend class A;
    //...
};
```

此时 A 类中的所有成员函数都可以访问 B 类的私有成员

2、 运算符重载的规则

2.1 不能重载的运算符

- . 成员访问运算符
- .* 成员指针访问运算符
- ?: 三目运算符
- :: 作用域限定符
- sizeof 长度运算符

2.2 运算符重载的规则和形式（重点）

1. 运算符重载时，**其操作数类型必须要有自定义类类型或枚举类型，不能全都是内置类型**
2. 运算符重载的本质是函数重载，其优先级和结合性还是固定不变的
3. **操作符的操作数个数是保持不变的**
4. **运算符重载时，不能设置默认参数**，如果设置了默认值，其实也就是改变了操作数的个数
5. 逻辑与 && 逻辑或 || 就不再具备短路求值特性，进入函数体之前必须完成所有函数参数的计算，不推荐重载
6. 不能臆造一个并不存在的运算符

3.运算符重载的方式

1. **采用友元函数的重载形式**

```
class Complex{
    //...
    friend Complex add(const Complex & lhs, const Complex & rhs);
    //...
};

Complex add(const Complex & lhs, const Complex & rhs){
    return Complex(lhs.m_real + rhs.m_real,   // 实部相加
                   lhs.m_imag + rhs.m_imag); // 虚部相加
}
```

2. 采用普通函数的重载形式（不推荐使用）

```
class Complex {
public:
    //...
    double getReal() const { return m_real; }
    double getImage() const { return m_image; }
    //...
};

Complex operator+(const Complex & lhs, const Complex & rhs)
{
    return Complex(lhs.getReal() + rhs.getReal(),
                   lhs.getImage() + rhs.getImage());
}
```

3. 采用成员函数的重载形式

```
class Complex{
public:
    //...
    Complex operator+(const Complex & rhs)
    {
        return Complex(m_real + rhs.m_real, m_image + rhs.m_image);
    }
};
```

注意：除非有特别的理由，尽量使重载的运算符与其内置的、广为接受的语义保持一致

4.重载形式的选择

4.1 友元函数方式重载适用场景

- 不会修改操作数的值的运算符
- 具有对称性的运算符可能转换任意一端的运算对象，即运算符两端的操作数可以互换位置，而运算逻辑不变，例如相等性、位运算符等

4.2 成员函数方式重载适用场景

- 会修改操作数的值的运算符
- 赋值=、下标[ ]、调用()、成员访问->、成员指针访问->* 运算符必须是成员函数形式重载
- 与给定类型密切相关的运算符，如递增、递减和解引用运算符

5.运算符重载的案例

5.1()运算符的重载

1. **函数对象** :重载了函数调用运算符的类创建的对象称为函数对象 ,使得对象能像函数一样被调用

```
class FunctionObject{
        //...
        // 重载函数调用运算符
        ReturnType operator()(ParameterList)
        {
                // do something....
        }
};

void test0(){
        FunctionObject fo{ };
        fo();   //让对象像一个函数一样被调用
}
```

形式：函数调用运算符必须以成员函数的形式进行重载

```
ReturnType operator()(ParameterList);
```

6、可调用实体

6.1 可调用实体

成员函数、成员函数指针、可调用对象（函数对象、函数指针、普通函数）

6.1.1 函数指针

（1）返回类型（*指针名称)(参数类型列表)

```
void print(int x){
	cout << "print:" << x << endl;
}

void display(int x){
	cout << "display:" << x << endl;
}

int main(void){
	//省略形式
	void (*p)(int) = print;
	p(4);
	p = display;
	p(9);

	//完整形式
	void (*p2)(int) = &print;
	(*p2)(4);
	p2 = &display;
	(*p2)(9);
}
```

（2）typedef void(*Function)(int);

```
Function f;
// 指向 print 函数
f = print;
// 调用 print 函数
f(19);
// 指向 display 函数
f = display;
// 调用 display 函数
f(27);
```

6.1.2 成员函数指针

- **对于具体对象**

```
(对象.*指针名称)(参数);        // 对于具体对象
(对象指针->*指针名称)(参数); // 对于对象指针(指向堆上的对象)
```

```
void (FFF::*p)(int) = &FFF::print;
FFF ff;
(ff.*p)(4);
```

- **对于堆上对象**

```
typedef void (FFF::*MemberFunction)(int); //定义成员函数类型为
                                          //MemberFunction
MemberFunction mf = &FFF::print; //定义成员函数指针
FFF * fp = new FFF{};
(fp->*mf)(65);//通过指针调用成员函数指针
```

注意：成员函数指针指向的成员函数需要是 FFF 类的公有成员函数

7、类型转换函数

7.1 其他类型向自定义类型转换

- 由构造函数实现，只有当类中定义了合适的构造函数时，转换才能通过，又称为隐式转换，若在构造函数前加上 explicit 关键字，则无法进行隐式转换

```
class A
{
public:
    // explicit //禁止隐式转换
    A(int num)
    : m_num(num)
    {
        cout << "A constructor" << endl;
    }
private:
    int m_num;
};

void test(){
    // 隐式转换 使用 int 数据创建出了一个 A 类型对象
    A a = 1;
}
```

7.2 由自定义类型向其他类型转换

7.2.1 类型转换函数形式与特征

- **形式：**operator 目标类型(){ }
- **特征**
  1. 必须是成员函数
  2. 没有返回类型

3. 没有参数
4. 在函数执行体中必须要返回目标类型的变量

7.2.2 自定义类型向内置类型转换

```
class Point{
public:
//...
operator int(){
   cout << "operator int()" << endl;
   return m_x + m_y;
}
      //...
};

int main(){
Point pt(1,2);
int a = 10;
//将 Point 类型对象转换成 int 型数据
a = pt;
cout << a << endl;
}
```

7.2.3 自定义类型向自定义类型转换

（1）使用类型转换函数

```
// Complex 类向 Point 类转换
class Complex
{
//...
operator Point(){
   cout << "operator Complex()" << endl;
   return Point(m_real,m_image);
}
};

pt = cx;
```

（2）使用隐式转换（特定形式的构造函数）

```
// 实现 Complex 类向 Point 类的转换
Class Complex;
Class Point{
      Point(const Complex& rhs)
      :_ix(rhs._real)
      ,_iy(rhs._image)
      {}
};
```

（3）使用赋值运算符函数

```
// 实现 Complex 类向 Point 类的转换
Class Point{
      Point& operator=(const complex& rhs){
            _ix = rhs._real;
            _it = rhs._image;
      }
};
```

优先级：赋值运算符函数>类型转换函数>构造函数隐式转换

8、String 的底层实现（重点）

8.1 深拷贝

- 在拷贝构造和赋值运算符函数中直接使用深拷贝，缺点是要频繁申请空间、赋值内容，效率会很底下

8.2 写时复制（COW）

- **共享数据**：当多个对象引用同一份数据时，先共享底层内存，而非立即复制
- **延迟复制**：只有在某个对象尝试修改数据时，系统才会创建该数据的独立副本，确保修改不影响其它共享对象

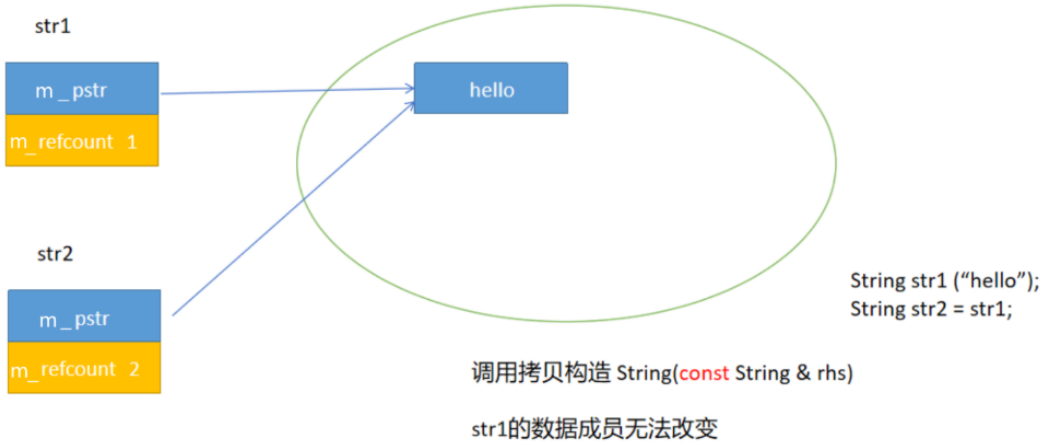
8.2.1 何时回收堆空间的字符串内容

- 使用一个引用计数 refCount，当字符串进行复制操作时，引用计数+1
- 当字符串对象被销毁时，引用计数-1
- 只有当引用计数减为 0 时，才真正会输堆空间字符串
- **单独创建对象需要新开辟空间，只有进行复制时才有优化空间**

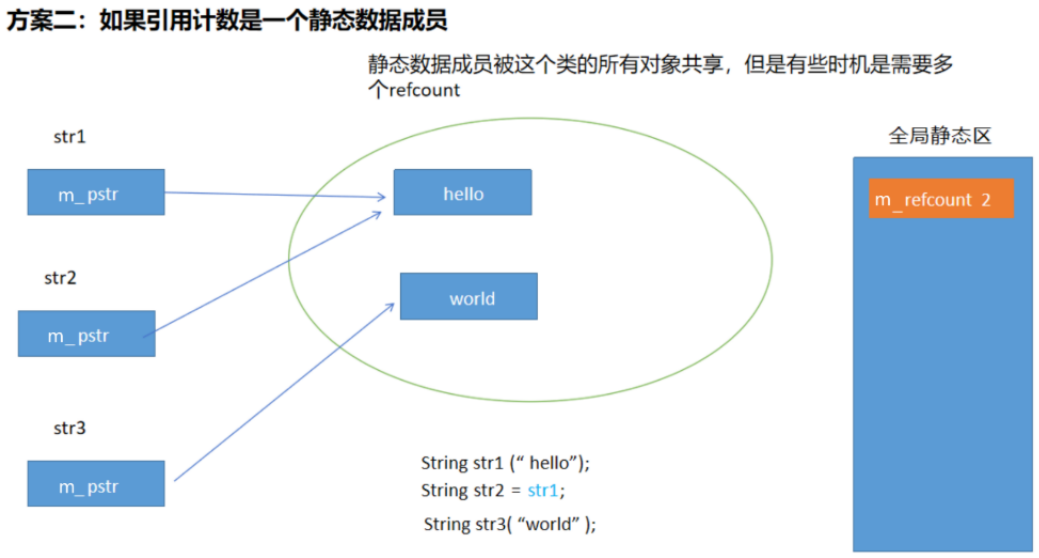
8.2.2 引用计数放置位置应如何选择

- **引用计数为普通数据成员**：当多个对象同时指向一个字符串时，任意复制或销毁对象的操作都需要改变其他所有对象的引用计数，显然不现实

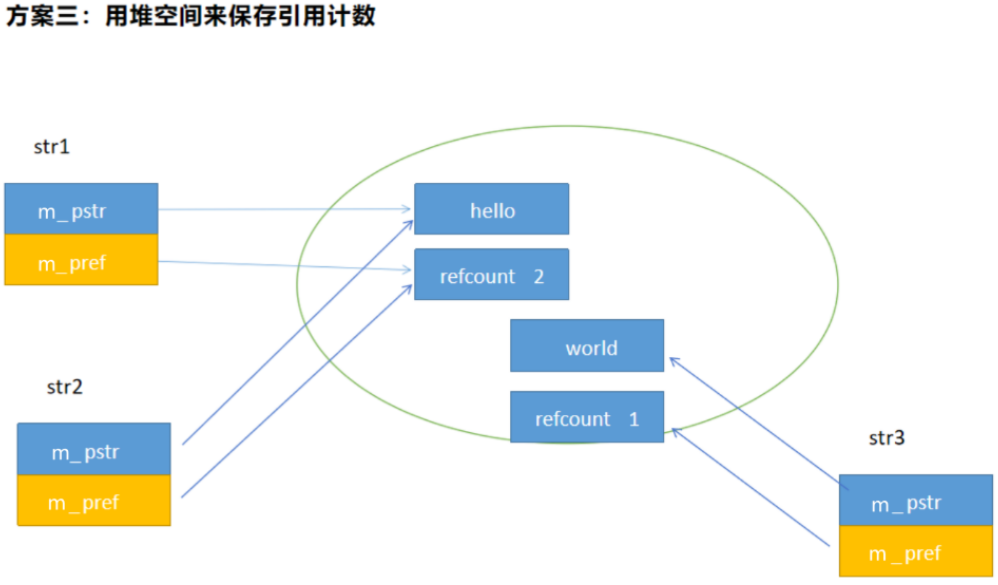
方案一：如果引用计数是一个普通的数据成员



- **引用计数为静态数据成员**：静态数据成员被该类所有对象共享，此时 str1 和 str2 的引用计数都是 2，但 str3 的引用计数应该是 1，显然无法满足需求

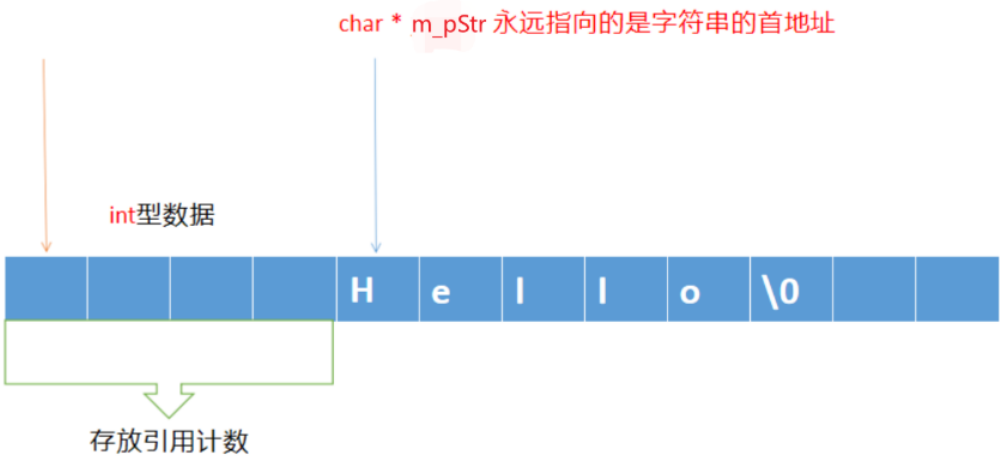


- **引用计数保存在堆空间**：String 类使用两个指针，一个指向字符串，另一个指向引用计数，方案可行但是可以优化



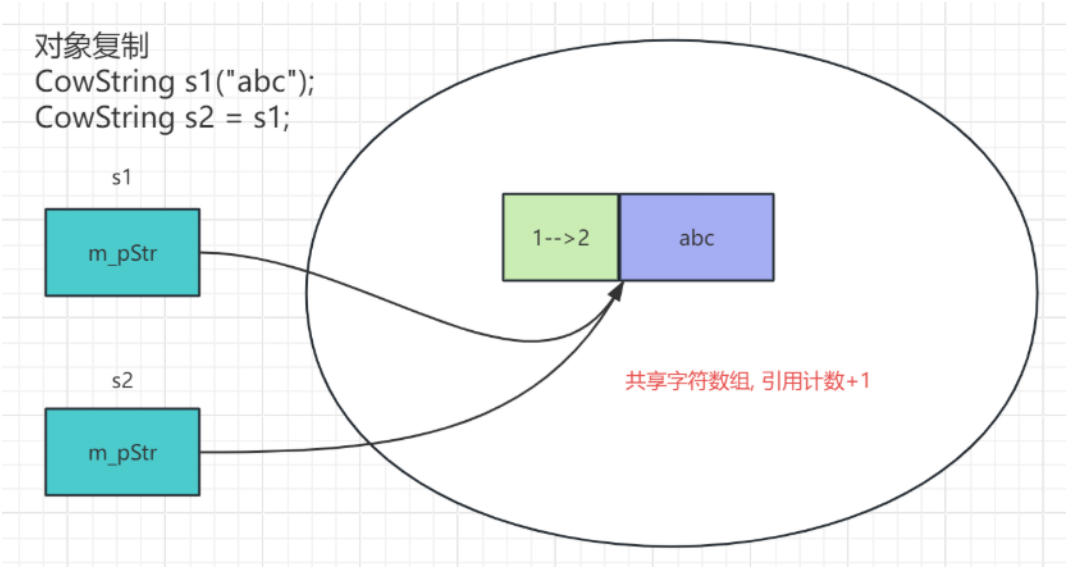
- **将引用计数保存在字符串内容前面**

引用计数保存到字符串内容的前面，方便访问。



8.2.3 对象复制

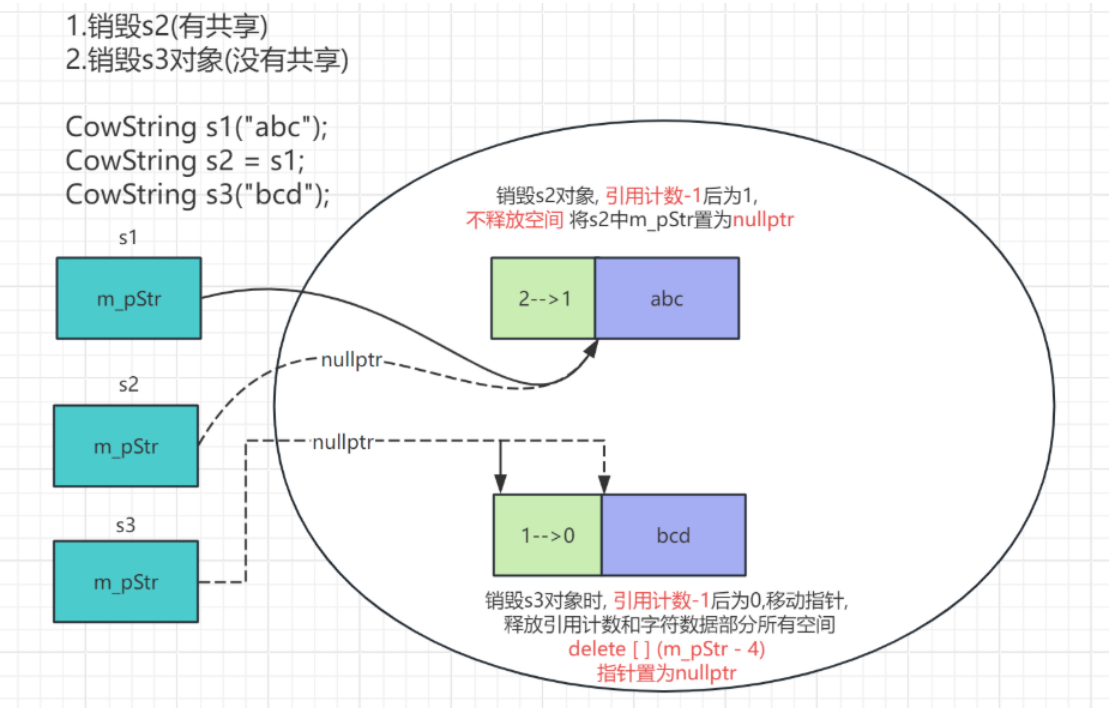
- 调用拷贝构造函数，在初始化列表中进行浅拷贝后将引用计数+1




```
// copy constructor
CowString::CowString(const CowString & cowSting)
: m_pStr(cowSting.m_pStr)
{
    cout << "copy constructor" << endl;
    // 引用计数+1
    incrementRefCount();
}
```

8.2.4 对象销毁

- 调用析构函数，将引用计数-1，然后进行判断，只有当引用计数为 0 时才进行空间的释放

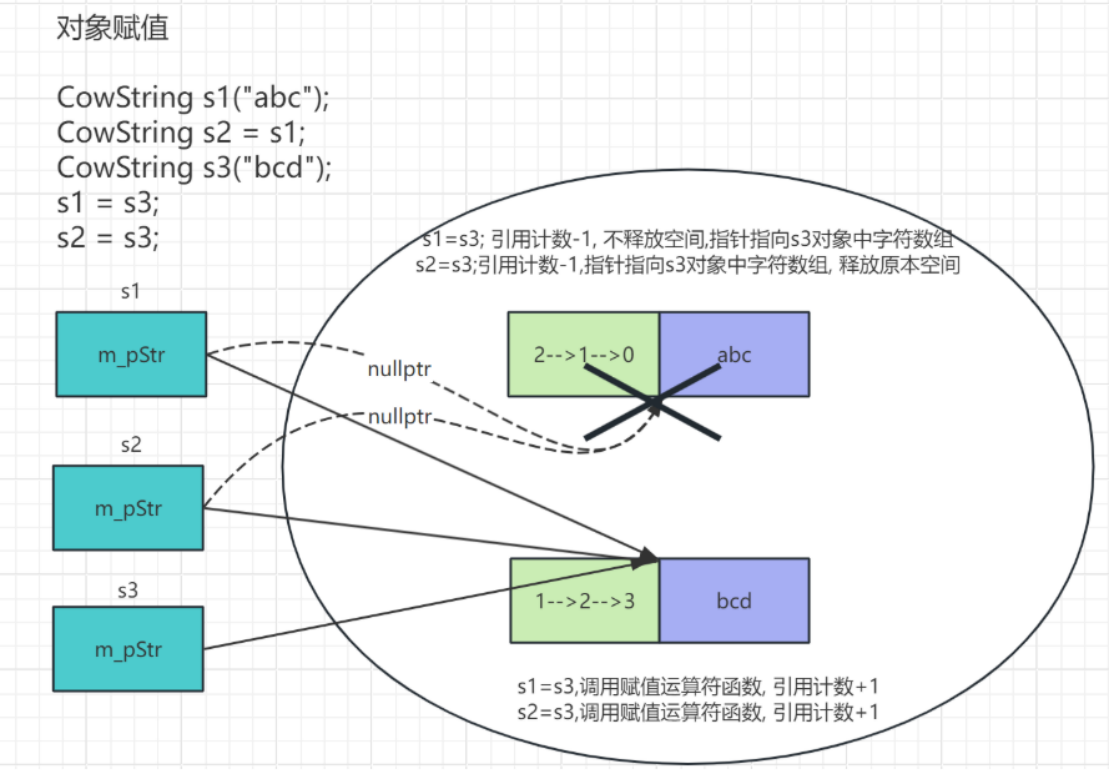



```
// destructor
CowString::~CowString()
{
    cout << "destructor" << endl;
    // 引用计数-1
    decrementRefCount();
    // 如果引用计数为 0 说明没有共享 回收空间
    if(getRefCount() == 0){
        // 回收引用计数+字符数据部分空间
        delete [] (m_pStr - 4);
        cout << "==>release heap" << endl;
    }
    m_pStr = nullptr;
}
```

8.2.5 对象赋值

（1）先将原本空间的引用计数-1，当引用计数为 0 时，才真正回收堆空间

（2）让自己的指针指向新的空间，并将新空间的引用计数+1



```
// operator =
CowString & CowString::operator=(const CowString & cowSting)
{
    // 自赋值判断
    if(this != &cowSting){
        // 引用计数-1
        decrementRefCount();
        // 判断引用计数是否为 0
        if(getRefCount() == 0){
            // 释放原本空间
            delete [] (m_pStr - 4);
            cout << "release heap" << endl;
        }
        m_pStr = nullptr;
        // 指针指向新空间
        m_pStr = cowSting.m_pStr;
        // 新空间引用计数+1
        incrementRefCount();
    }
    // 返回自身对象
    return *this;
}

```

8.2.6operator[ ]重载实现

1. 判断引用计数是否大于 1，大于 1 说明共享
2. 原空间的引用计数-1
3. 深拷贝
4. 修改指针指向新空间
5. 初始化新空间的引用计数

```
char & CowString::operator[](size_t index)
{
 // 判断 index
 if(index >= size()){
      cout << "index is illegal!" <<endl;
      static char nullChar = '\0';
      return nullChar;
 }else{
      // 判断引用计数是否大于 1
      if(getRefCount() > 1){
           // 说明共享了 需要开辟新空间
           // 原来的引用计数-1
           decrementRefCount();
           char * temp = new char[4 + strlen(m_pStr)+1]() + 4;
           // 复制
           strcpy(temp, m_pStr);
           // 更改指向
           m_pStr = temp;
           // 初始化新空间引用计数
           initRefCount();
      }
      // 返回下标对应字符
      return m_pStr[index];
 }
}
```

注意 :重载[ ]运算符后发现虽然修改能正确实现了 ,但是读操作也会进行复制 ,也就是[ ]无法区分读写操作

8.2.7 使用内部代理类 CharProxy 区分读写操作

```
class CowString
{
public:
    // 代理类，用于延迟 COW 检查
    /* 代理类能区分读写操作主要是因为编译器
       会根据赋值运算符=左右两侧的内容来决定
       调用类型转换函数还是重载后的赋值运算符函数
    */
    class CharProxy
    {
    private:
        CowString &m_str;
        size_t m_index;

    public:
        CharProxy(CowString &str, size_t index)
            : m_str(str), m_index(index) {}

        // 读操作，使用类型转换函数
        operator char() const
        {
            return m_str.pstr[m_index];
        }

        // 写操作，触发 COW
        CharProxy &operator=(const char &c)
        {
            // 引用计数>1 则执行 COW
            if (m_str.getRefCount() > 1)
            {
                m_str.decrementRefCount();
                // 新开辟空间
                char *newStr = m_str.allocate(m_str.pstr);
```

- **读操作执行流程**

（1）通过 operator[ ]读取字符

```
char c = s1[0];  // 读操作
```

（2）s1[0]返回 CharProxy 对象

```
CharProxy operator[](size_t index) {
    return CharProxy(*this, index);
}
```

（3)由于 = 右侧是 CharProxy ,左侧是 char ,编译器会尝试将 Charproxy 转换成 char，由于 CharProxy 定义了 operator char( )，编译器会调用 operator char( )

```
operator char( ) const
{
    return m_str.pstr[m_index];
}
```

（4）最终直接返回字符，不触发 COW，引用计数保持不变

- **写操作执行流程**

（1）通过 operator[ ]写入字符

```
s1[0] = 'H';  // 写操作
```

（2）调用 operator[ ]，返回 CharProxy 代理对象

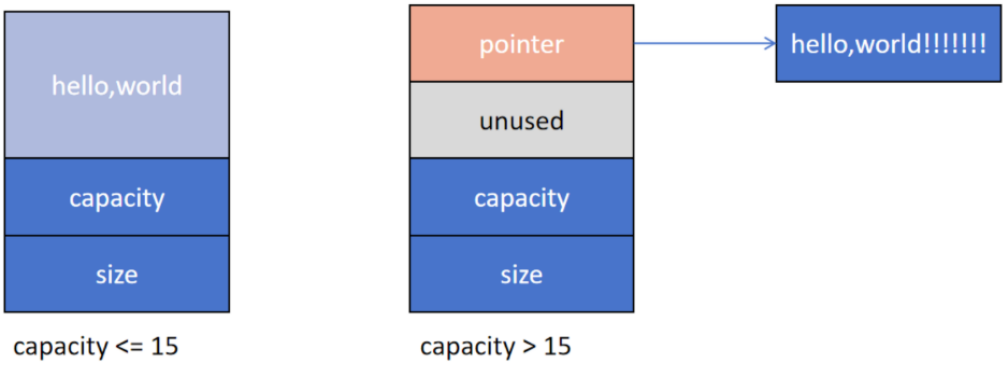
```
CharProxy operator[](size_t index) {
    return CharProxy(*this, index);
}
```

（3）由于 = 右侧是 char，左侧是 CharProxy，编译器会调用 operator = 运算符

```
CharProxy &operator=(const char &c) {
    // 检查引用计数 >1（是否共享数据）
    if (m_str.getRefCount() > 1) {
        // 1. 引用计数 -1
        m_str.decrementRefCount();
        // 2. 新分配内存并复制数据（COW）
        char *newStr = m_str.allocate(m_str.pstr);
        strcpy(newStr, m_str.pstr);
        // 3. 更新指针
        m_str.pstr = newStr;
        // 4. 初始化新数据的引用计数 =1
        m_str.initRefCount();
    }
    // 5. 修改字符
    m_str.pstr[m_index] = c;
    return *this;
}
```

8.3 短字符串优化（SSO）

- 当字符串的字符数小于等于 15 时，buffer 直接存放整个字符串
- 当字符串的字符数大于 15 时，buffer 存放的就是一个指针，指向堆空间



```
class string {
    union Buffer{
        char * _pointer;
        char _local[16];
    };

    size_t _size;
    size_t _capacity;
    Buffer _buffer;
};
```

四、 工具

1、输入输出流

1.1 流的四种状态（重点）

- goodbit 表示流处于有效状态，流只有在有效状态下才能正常使用

```
bool good() const        //流是 goodbit 状态，返回 true，否则返回 false
```

- eofbit 表示到达流结尾位置，流在正常输入输出情况下结束，会被置为 eofbit 状态

```
bool eof() const         //流是 eofbit 状态，返回 true，否则返回 false
```

- failbit 表示发生可恢复的错误 ,如期望读取一个 Int 数值 ,却读出一个字符串等错误。这种问题通常是可以修改的，流还可以继续使用

```
bool fail() const        //流是 failbit 状态，返回 true，否则返回 false
```

- badbit 表示发生系统级错误，通常情况下发生这种错误流就无法再使用了

```
bool bad() const          //流是 badbit 状态，返回 true，否则返回 false
```

1.2 标准输入输出流

1.2.1 标准输入流

- **istream 类定义了全局输入流对象 cin**，代表标准输入，从键盘获取数据，通过流提取符">>"从流中提取数据，">>"提取数据时会跳过流中的空格、tab 键、换行符等，只有在输入完数据再按回车键后才会将数据送入缓冲区
- **输入流从 failbit 状态恢复**

```
void test()
{
    int num = 10;
    cout << "执行输入操作前流的状态:" << endl;
    printStreamStatus(cin);

    cin >> num;
    cout << "执行输入操作后流的状态:" << endl;
    printStreamStatus(cin);

    if(!cin.good()){
        // 恢复流的状态
        cin.clear();
        // 清空缓冲区 后才能使用
        // 忽略当前输入流中的所有字符，直到遇到换行符为止. 这种方法适用于清空缓冲区，因为它可以跳过所有字符。
        cin.ignore(std::numberic_limits<std::streamsize>::max(), '\n');
        printStreamStatus(cin);
    }
    // 没有正常输入 num 被设置为 0
    cout << "num=" << num << endl;
    string line;
    cin >> line;
    cout << "line:" << line << endl;
}
```

- **缓冲区分类**

（1）全缓冲：填满缓冲区后才进行实际 I/O 操作，典型代表为对磁盘文件的读写

（2）行缓冲：只有当输入和输出中遇到换行符时，才执行真正的 I/O 操作，将缓冲区中的字符进行输出，典型代表为 cin

（3）不带缓冲：不进行缓冲，有多少数据就刷新多少，典型代表为 cerr

1.2.2 标准输出流

- **ostream 类定义了全局输出流对象 cout**，即标准输出，在缓冲区刷新时将数据输出到终端

1.2.3 标准错误流

- **ostream 类定义了全局输出流对象 cerr，即无缓冲标准错误流**

1.3 文件输入输出流

1.3.1 文件模式

1. **in**：以读取模式打开空间（默认用于 ifstream），如果文件不存在，打开失败
2. **out**：以写入模式打开文件（默认用于 ofstream），如果文件存在则覆盖，不存在则创建一个
3. **app**：追加模式，始终在文件末尾写入
4. **ate**：打开文件后，将文件指针定位到文件末尾
5. **trunc**：截断，若打开文件存在，清空内容并将文件大小置为 0
6. **binary**：二进制，读取或写入文件的数据为二进制

1.3.2 文件输入流

- **文件输入流对象的创建**

```
#include <fstream>
void test0(){
    ifstream ifs;
    ifs.open("test1.cc");

    ifstream ifs2("test2.cc");

    string filename = "test3.cc";
    ifstream ifs3(filename);
}
```

- **文件读取**

逐词读取：使用>>运算符，默认以换行符，空格作为间隔符

```
// 默认以换行符,空格作为间隔符
// 一次读取一个字符串
string word;
// 只要 ifs 是 goodbit 状态就会一直读取
while(ifs >> word){
        cout << word << endl;
}
// 规范操作，使用完之后关闭流
ifs.close();
```

按行读取：使用 String 类的 getline 函数

```
void test4()
{
        // 使用 string 中的 getline
        using std::string;
        ifstream ifs("aa.txt");
        string line;
        // eof 状态结束循环
        while(std::getline(ifs,line)){
                cout << line << endl;
        }

        // 关闭流
        ifs.close();
}
```

1.3.3 文件输出流

- **文件输出流对象的创建**

```
string filename = "test3.cc";
ofstream ofs3(filename);

string line("hello,world!\n");
ofs << line;

ofs.close();
```

以追加模式创建

```
string filename = "test3.cc";
ofstream ofs3(filename，std::ios::app);
```

1.4 字符串输入输出流

1.4.1 字符串输入流

- **将字符串的内容传输给字符串输入流对象，再通过这个对象进行对字符串的处理**
- **字符串输入流通常用来处理字符串内容，比如读取配置文件，通常以空格作为分隔符**

```
//myserver.conf
ip 192.168.0.0
port 8888
dir ~HaiBao/53th/day06

//readConf.cc
void readConfig(const string & filename){
	ifstream ifs(filename);
	if(!ifs.good()){
		cout << "open file fail!" << endl;
		return;
	}

	string line;
	string key, value;
	while(getline(ifs,line)){
		istringstream iss(line);
		iss >> key >> value;
		cout << key << " -----> " << value << endl;
	}
}

void test0(){
	readConfig("myserver.conf");
}
```

1.5 字符串输出流

- **用途是将各种类型的数据转换成字符串类型**

```
void test0(){
    int num = 123, num2 = 456;
    ostringstream oss;
    //把所有的内容都传给了字符串输出流对象
    oss << "num = " << num << " , num2 = " << num2 << endl;
    // str()用于返回构造好的字符串
    cout << oss.str() << endl;
}
```

注意：将字符串、int 型数据、字符串、int 型数据统统传给了字符串输出流对象，存在其缓冲区中，利用它的 str()函数，全部转为 string 类型并完成拼接

2、移动语义

2.1 右值存储位置

- **在内存中存储**：当右值是很大的对象或者编译器决定这样做更高效时可能会在内存中分配空间
- **存在于寄存器中**：对于简单的右值，如基本数据类型，编译器可能会选择将其存储于寄存器中，减少内存访问的需要，加快程序的执行速度

2.2 右值引用

- 右值引用不能绑定到左值，但是可以绑定到右值，也就是右值引用可以识别出右值
- **右值引用若是一个有名字的变量，那么此时右值引用是一个左值，若没有名字，那么此时右值引用是一个右值（比如直接作为函数返回值）**

```
int gNum = 10;

int && func(){
    return std::move(gNum);
}

void test1(){
    // &func();   //无法取址，说明返回的右值引用本身也是一个右值
    int && ref = func();
    &ref;   //可以取址，此时 ref 是一个右值引用，其本身是左值
}
```

2.3std::move 函数

- **显式的将左值转换为右值，本质上是一个强制转换**
- **当将一个左值转换为右值后，如果利用右值引用绑定 std::move 的返回值，并进行修改操作，那么原来的左值对象也会随之修改，可能无法正常工作了，必须要重新赋值才可以继续使用**

```
void test() {
    int a = 1;
    &(std::move(a)); //error，左值转成了右值

    String s1("hello");
    s1.print();
    //如果经历了 std::move 的强转后没有进行修改操作,之后 s1 对象还是可以正常使用的,
    std::move(s1);//没有使用返回值
    s1.print();

    //调用移动赋值运算符函数,在移动赋值运算符函数中形参 String && rhs = std::move(s1)
    // rhs.m_pstr = nullptr; 进行了修改操作
    // 会使 s1 对象本身的 m_pstr 变成空指针 再进行 print 操作时可能有问题
    String s2("abc");
    s2 = std::move(s1);
    s1.print();// 这里程序中断
    s2.print();
}
```

注意 :std::move 的本质是在底层做了类型转换来标记一个对象为右值(它本身并不移动数据或资源，只是为移动操作提供条件使得移动构造函数和移动赋值运算符函数能够被调用)

3、智能指针

3.1auto_ptr

- auto_ptr 可以进行复制和赋值，但是有隐患，因此被废弃掉了

```
auto_ptr<int> ap2(ap);
cout << "*ap2:" << *ap2 << endl; //ok
cout << "*ap:" << *ap << endl;   //error
//auto_ptr 底层的原理是移交管理权，所以 ap 底层指针已经被置空了,所以对 ap 解引用导致段错误
```

3.2unique_ptr（重要）

1. **不允许复制或赋值，具有对象语义**
2. **独享所有权的智能指针**

- unique_ptr 通过独占资源的方式，确保只有一个指针指向资源，并负责在其生命周期结束时自动释放资源，避免显式的内存管理

```
void test0(){
    unique_ptr<int> up(new int(10));
    cout << "*up:" << *up << endl;
    // get()-->返回指向被管理对象的指针
    cout << "up.get(): " << up.get() << endl;

    cout << endl;
    //独享所有权的智能指针，对托管的空间独立拥有
    //拷贝构造已经被删除
    unique_ptr<int> up2 = up;//复制操作  error

    //赋值运算符函数也被删除
    unique_ptr<int> up3(new int(20));
    up3 = up;//赋值操作  error
}
```

3. **作为容器元素时只能传递右值属性的 unique_ptr，由于容器默认是值传递，若传入左值形态的 unique_ptr，会进行复制操作，而 unique_ptr 不允许复制**

- 使用 std::move 将左值变为右值

- 直接使用 unique_ptr 的构造函数，创建匿名对象（临时对象），构建右值

```
vector<unique_ptr<Point>> vec;
unique_ptr<Point> up4(new Point(10,20));
//up4 是一个左值
//将 up4 这个对象作为参数传给了 push_back 函数，会调用拷贝构造
//但是 unique_ptr 的拷贝构造已经删除了
//所以这样写会报错
vec.push_back(up4);   //error

vec.push_back(std::move(up4));   //ok
vec.push_back(unique_ptr<Point>(new Point(1,3))); //ok
```

3.3shared_ptr（重要）

- 某些场景下我们需要多个指针共享一个资源，需要保证资源再最后一个使用者释放后才被销毁，shared_ptr 使用引用计数的方式，思想与 COW 类似，是强引用的智能指针

1. **共享所有权的智能指针**

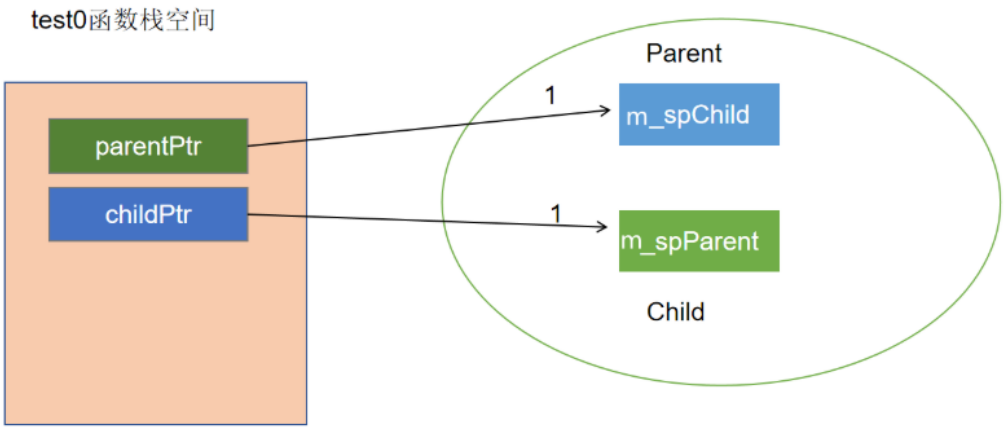
   使用引用计数记录对象的个数
2. **可以进行复制或赋值，具备值语义**
3. **可以作为容器的元素**

   作为容器元素时，既能传递左值，又能传递右值
4. **具备移动语义，即有移动构造和移动赋值函数**

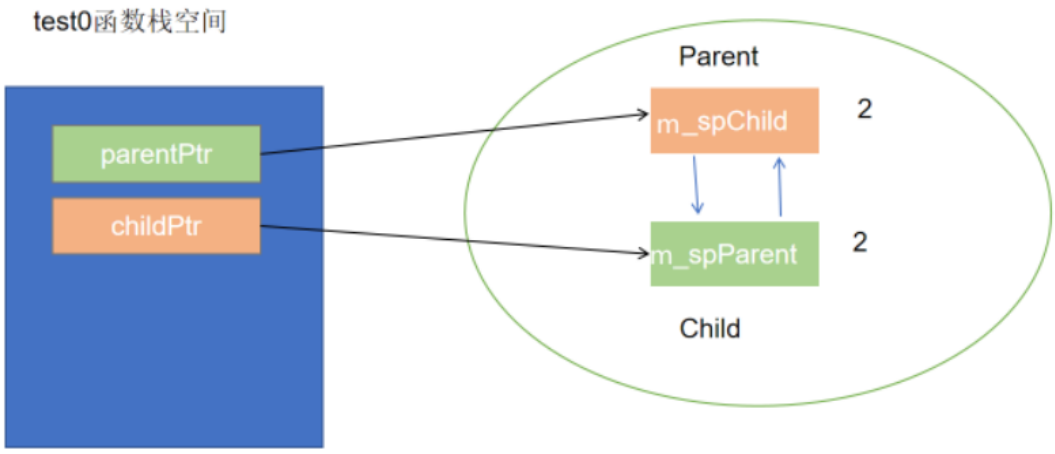
```
shared_ptr<int> sp(new int(10));
// sp.refCount = 1
cout << "sp.use_count(): " << sp.use_count() << endl;

 cout << "执行复制操作" << endl;
shared_ptr<int> sp2 = sp;
// sp.refCount=sp2.refCount = 2
cout << "sp.use_count(): " << sp.use_count() << endl;
cout << "sp2.use_count(): " << sp2.use_count() << endl;

cout << "执行赋值操作" << endl;
shared_ptr<int> sp3(new int(30));
sp3 = sp;
// sp.refCount=sp2.refCount=sp3.refCount = 3
cout << "sp.use_count(): " << sp.use_count() << endl;
cout << "sp2.use_count(): " << sp2.use_count() << endl;
cout << "sp3.use_count(): " << sp3.use_count() << endl;

cout << "作为容器元素" << endl;
vector<shared_ptr<int>> vec;
vec.push_back(sp);                          // 传入左值
vec.push_back(std::move(sp2));          // 传入右值
```

5. shared_ptr 存在循环引用问题

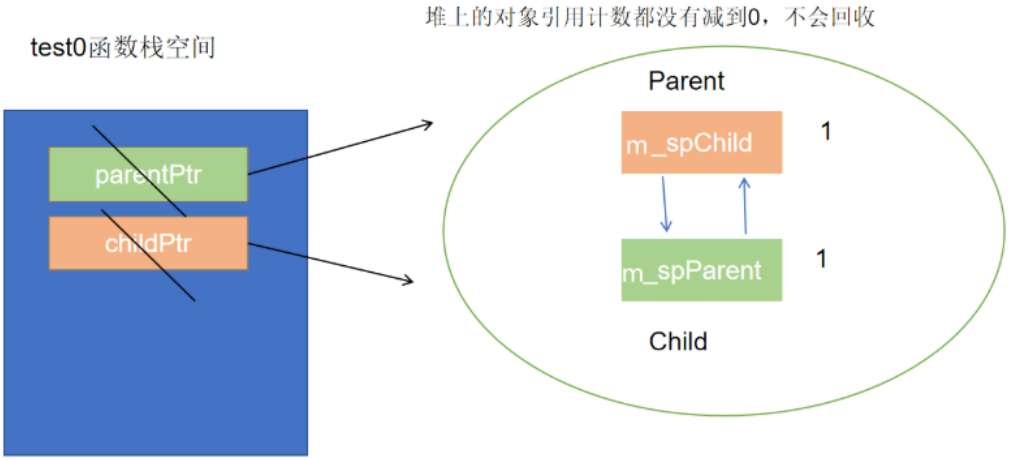
```
shared_ptr<Parent> parentPtr(new Parent());
shared_ptr<Child> childPtr(new Child());
//获取到的引用计数都是 1
cout << "parentPtr.use_count():" << parentPtr.use_count() << endl;
cout << "childPtr.use_count():" << childPtr.use_count() << endl;
```




```
parentPtr->spChild = childPtr;
childPtr->spParent = parentPtr;
//获取到的引用计数都是 2
cout << "parentPtr.use_count():" << parentPtr.use_count() << endl;
cout << "childPtr.use_count():" << childPtr.use_count() << endl;
```



- **因为 childPtr 和 parentPtr 会先后销毁，但是堆上的 Parent 对象和 Child 对象的引用计数都变成了 1，而不会减到 0，所以没有回收**



3.4weak_ptr

- **是 shared_ptr 的一个补充，专门为解决循环引用的问题而诞生，是一个弱引用的智能指针，使用 weak_ptr 进行复制或赋值时，不会导致引用计数加 1**
- **weak_ptr 只知道所托管的对象是否存活，没有对托管资源的管理权和访问权，若要访问，必须要提升为 shared_ptr 才行**
- **初始化**

```
1    weak_ptr<int> wp;//无参的方式创建 weak_ptr
2
3    //也可以利用 shared_ptr 创建 weak_ptr
4    weak_ptr<int> wp2(sp);
```

- **判断关联的空间是否还在**

1. 可以使用 use_count 函数，如果 use_count 的返回值大于 0，表明关联的空间还在
2. 将 weak_ptr 提升为 shared_ptr

```
shared_ptr<int> sp(new int(10));
weak_ptr<int> wp;//无参的方式创建 weak_ptr
wp = sp;//赋值

// 将 weak_ptr 提升为 shared_ptr
shared_ptr<int> sp2 = wp.lock();
if(sp2){
cout << "提升成功" << endl;
cout << *sp2 << endl;
}else{
cout << "提升失败，托管的空间已经被销毁" << endl;
}
```

3. 使用 expired 函数

```
bool flag = wp.expired();
if(flag){
cout << "托管的空间已经被销毁" << endl;
}else{
cout << "托管的空间还在" << endl;
}
```

3.5 删除器

- 如果管理的是普通的资源，不需要写出删除器，就使用默认的删除器即可，只有针对 FILE 或者 socket 这一类创建的资源，才需要改写删除器，使用 fclose 之类的函数

1. **unique_ptr 自定义删除器**

- 创建一个代表删除器的 struct，定义 operator()函数

```
struct FILECloser{
void operator()(FILE * fp){
   if(fp){
        fclose(fp);
        cout << "fclose(fp)" << endl;
   }
}
};
```

- 创建 unique_ptr 接管文件资源时，删除器参数使用我们自定义的删除器

```
void test1(){
string msg = "hello,world\n";
// 这里直接传 FILECloser 类即可
unique_ptr<FILE，FILECloser> up(fopen("res2.txt","a+"));
//get 函数可以从智能指针中获取到裸指针
fwrite(msg.c_str(),1,msg.size(),up.get());
}
```

2. shared_ptr 自定义删除器

```
void test1(){
    string msg = "hello,world\n";
    //在 unique_ptr 的模板参数中加入删除器类
    unique_ptr<FILE，FILECloser> up(fopen("res2.txt","a+"));
    fwrite(msg.c_str(),1,msg.size(),up.get());
}


void test2(){
    string msg = "hello,world\n";
    FILECloser fc;
    //在 shared_ptr 的构造函数参数中加入删除器对象
    shared_ptr<FILE> sp(fopen("res3.txt","a+"),fc);
    fwrite(msg.c_str(),1,msg.size(),sp.get());
}
```

3.6 智能指针的误用

- **原因都是因为将一个原生裸指针交给了不同的智能指针进行托管，而造成尝试对一个对象销毁两次**

1. **unique_ptr 的误用**

```
void test0(){
//需要人为注意避免
Point * pt = new Point(1,2);
unique_ptr<Point> up(pt);
unique_ptr<Point> up2(pt);
}

void test1(){
unique_ptr<Point> up(new Point(1,2));
unique_ptr<Point> up2(new Point(1,2));
//让两个 unique_ptr 对象托管了同一片空间
up.reset(up2.get());
}
```

2. shared_ptr 的误用

- 使用不同的智能指针托管同一片堆空间,只能通过 shared_ptr 开放的接口——拷贝构造、赋值运算符函数
- 如果是用裸指针的形式将一片资源交给不同的智能指针对象管理，即使是 shared_ptr 也是不行的

```
void test2(){
Point * pt = new Point(10,20);
shared_ptr<Point> sp(pt);
shared_ptr<Point> sp2(pt);
}

void test3(){
//使用不同的智能指针托管同一片堆空间
shared_ptr<Point> sp(new Point(1,2));
shared_ptr<Point> sp2(new Point(1,2));
sp.reset(sp2.get());
}
```

五、 模板

- 定义：模板是一种实现代码重用的工具，将数据类型作为参数，在编译时根据模板传入的参数类型生成具体的模板函数，以减少代码冗余，提高开发效率，由于模板在编译时进行类型安全检查，还能增强类型安全性，避免运行时错误，模板是一种泛型编程
- 分类，模板分为函数模板和类模板