引用和复制

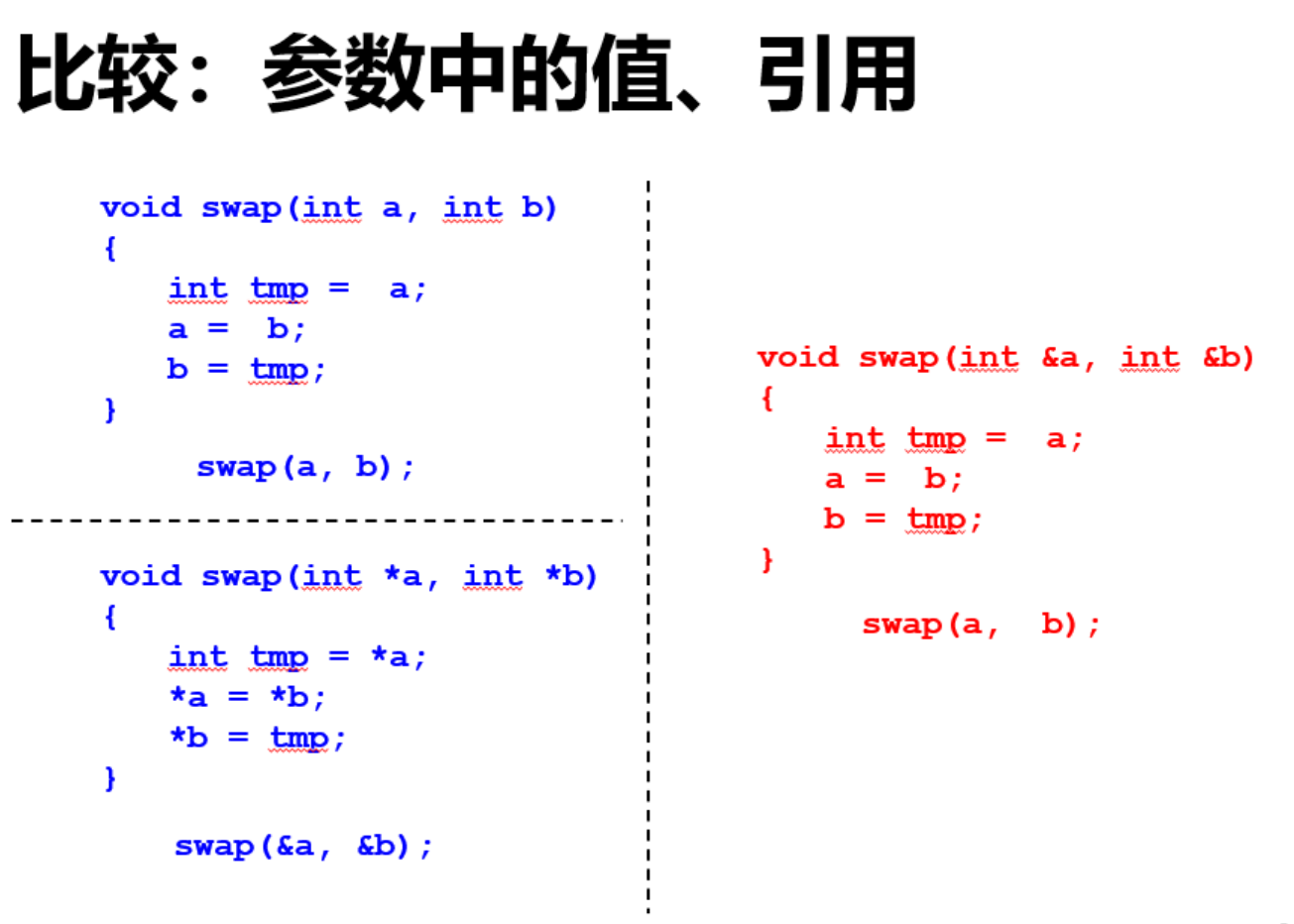
1. 引用

引用是变量的“别名”，例：`int v0; int & v1 = v0;` v1是变量v0的引用，它们在内存中是同一单元的两个不同名字。引用必须在**定义时进行初始化**（赋初值）。

函数参数可以是引用类型，表示函数的**形式参数与实际参数是同一个变量**，改变形参将改变实参。如调用以下函数将交换实参的值：

```
void swap(int& a, int& b){
    int tmp = b; b = a; a = tmp;
}
```

函数返回值可以是引用类型，但不得是函数的临时变量



6

反思：为什么需要引用?

引用的特性：创建时必须初始化、初始化后便不能指向其他对象，不存在空引用

> The easiest way to think about a reference is as a fancy pointer. You never have to wonder whether it’s been initialized (the compiler enforces it) and how to dereference it (the compiler does it).

> ——《Thinking in C++》

参数中的常量和常量引用：

最小特权原则：给函数足够的权限去完成相应的任务，但不要给予他多余的权限。

在函数运行的过程中，我们有时并不希望他修改传入参数的值，只需要他能读取参数值并运算得到最终结果即可。

解决方案：在参数中使用常量/常量引用

```
void foo(const int &a, const int &b)
```

此时函数中仅能读取a和b的值，无法对a, b进行任何修改操作。

2. 拷贝构造函数

拷贝构造函数举例：

```
class Person {
    int id;
    ...
public:
    Person(const Person& src) { id = src.id;  ... }
    ...
};
```

即，拷贝构造函数是一种特殊的构造函数，它的参数是**同类对象的常量引用**，用参数对象的内容初始化当前对象。

拷贝构造函数被调用的三种常见情况：

1. 用一个类对象定义另一个新的类对象

`A a; A b(a);`

`A c = a;`

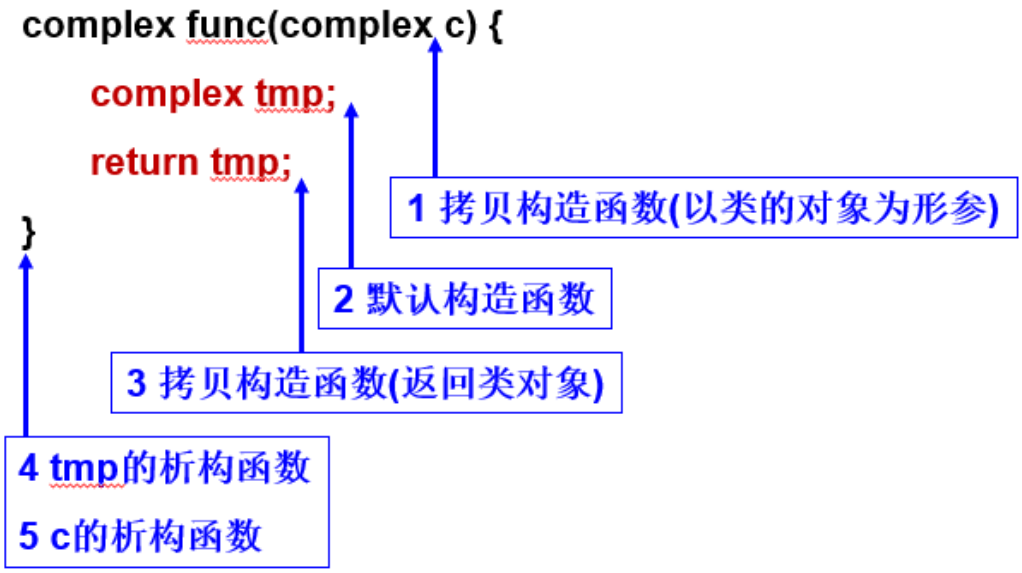
2. 函数调用时以类的对象为**形参**

Func(Test a)

3. 函数**返回**类对象

Test Func(void)

编译器会生成自动调用“拷贝构造函数”，在已有对象基础上生成新对象。



类的新对象被定义后，会调用构造函数或拷贝构造函数之一。如果调用拷贝构造函数，但没有给该类显式定义拷贝构造函数，编译器将**自动**合成，且采用**位拷贝(Bitcopy)**，即拷贝成员的**地址**而非内容。

【注意】位拷贝在遇到**指针**类型成员时可能会出错,会导致多个对象指向同一个地址。这样，如果类内含**指针**类型的成员时，不应使用默认的拷贝构造函数，因为有可能会“构造一次、析构两次”，导致指针对应的位置被重复删除。

正因如此，正常情况下，应尽可能避免使用拷贝构造函数。解决方法：

（1）使用引用/常量**引用**传参数或**返回对象**；

（2）将拷贝构造函数声明为private；

（3）用delete关键字显式地让编译器不生成拷贝构造函数的**默认版本**。

3. 移动构造和右值引用

多数情况下，我们更需要对象的“移动”，而非对象的“拷贝”。c++11为此提供了一种新的构造函数，即移动构造函数。为理解移动构造函数的工作原理，首先要引入c++11的另一个新特性——右值引用。

首先，我们来区分**左值**和**右值**。左值：可以取地址、有名字的值，可出现在等号左右；右值：不能取地址、没有名字的值，常见于**常值**、**函数返回值**、**表达式**，只能出现在等号右侧。

例如：

```
int a = 1;
int b = func();
int c = a + b;
```

其中，a、b、c为左值，而1（常量）、func（函数返回值）、a+b的结果（表达式）为右值。

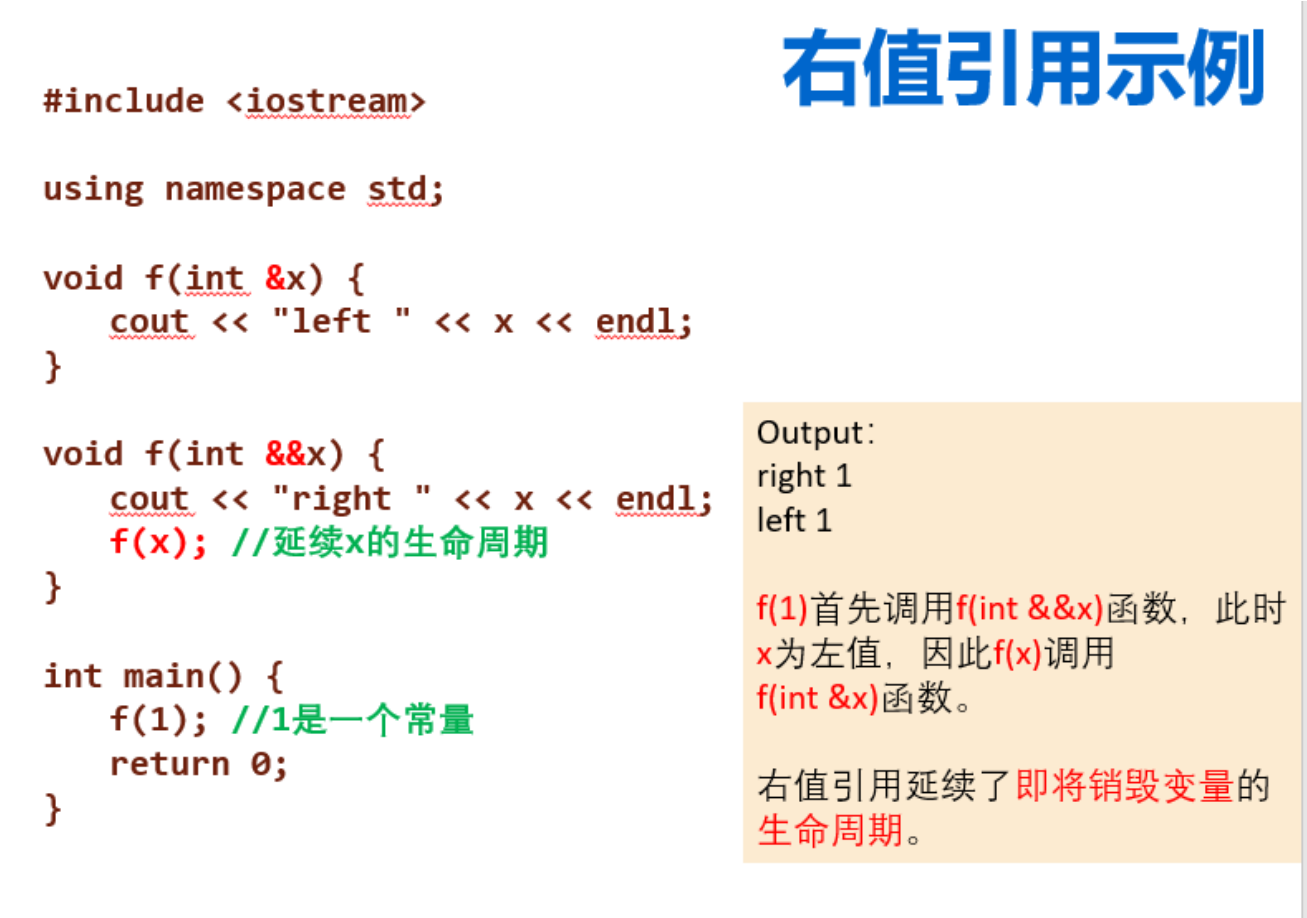
左值可以取地址，并且可以被&引用(左值引用)，右值不行：

```
int *d = &a;        ☺    int &d = a;       ☺
int *e = &(a + b); ☹    int &e = a + b;  ☹
```

那么，右值引用是什么呢?

虽然右值无法取地址，但可以被&&引用(右值引用)，例如`int && e = a + b;`，其中`a+b`就是右值。 右值引用是无法绑定左值的，例如`int && e = a;`就不可以。

右值引用有啥用呢？其实就是延续即将销毁的变量的生命周期。例如下图中，本来1就是一个右值，但是进入函数`void f(int &&x)`之后，它就变成了左值。



使用右值引用作为参数的构造函数叫做**移动构造函数**。

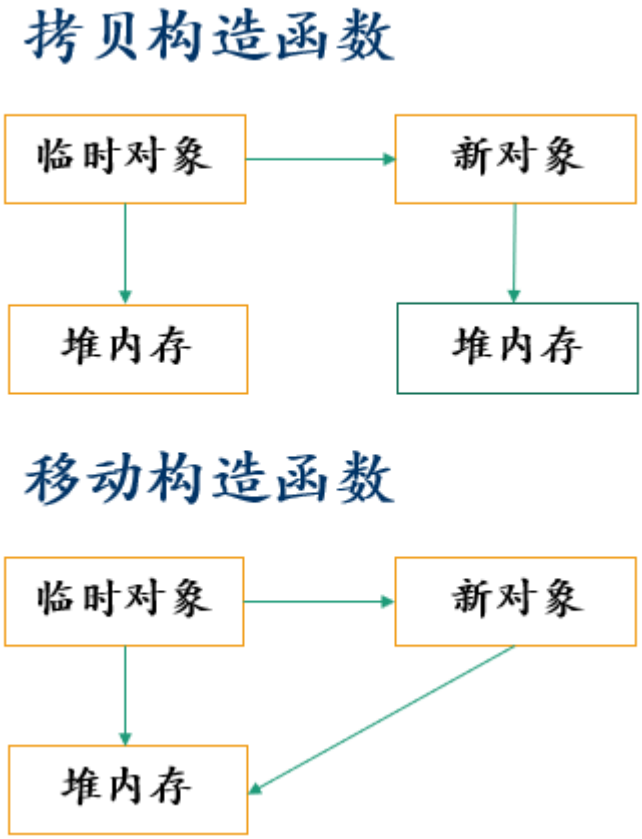
回忆一下**拷贝构造函数**的形式是:

```
ClassName(ClassName&);
ClassName(const ClassName&);
```

那么，**移动构造函数**的形式就是:

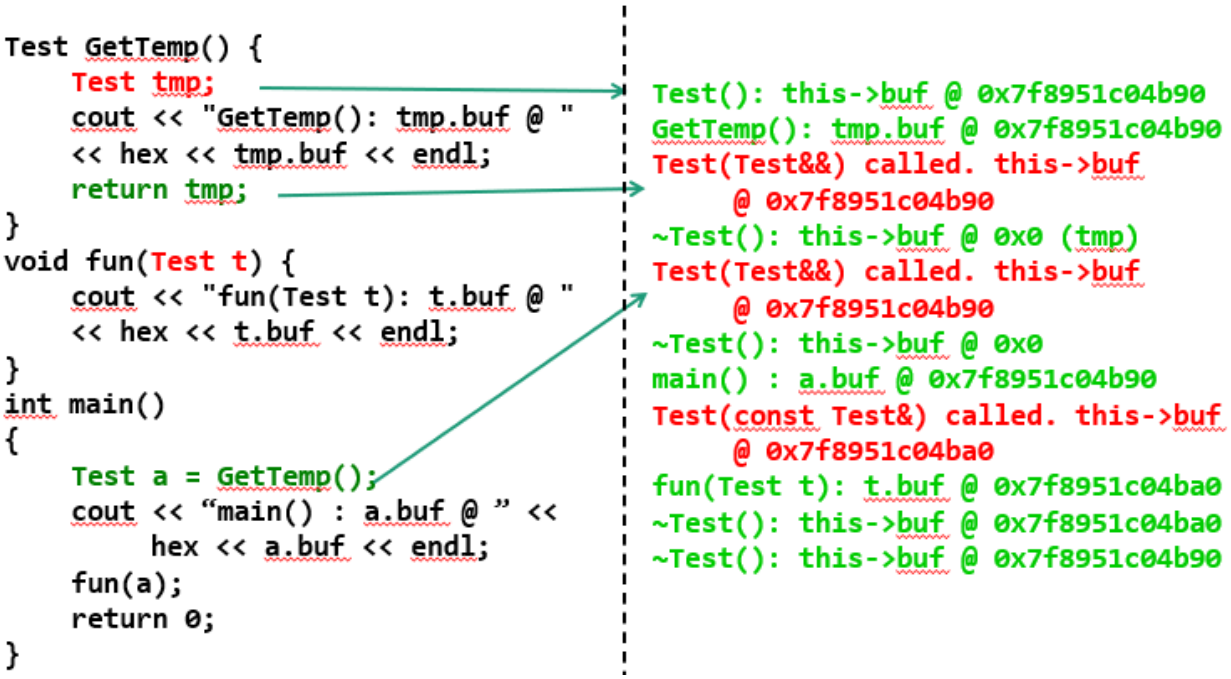
```
ClassName(ClassName&&);
```

它用来**偷“临时变量”中的资源**（如内存）。临时变量被编译器设置为**常量**形式，所以使用“拷贝构造”函数是不行的，无法将资源“偷”出来。基于“右值引用”定义的“移动构造”函数却被语言支持接受临时变量，且能“偷”出其中的资源，避免**频繁的拷贝。**



移动构造函数与拷贝构造函数的一个最主要的特征差别就是类中堆内存是**重新开辟**并拷贝，还是直接将指针指向那块地址。对于一些**即将析构的临时类**，移动构造函数**直接利用了原来临时对象中的堆内存**，新的对象无需开辟内存，临时对象无需释放内存，从而大大提高计算效率。

例子：



第一个移动构造：把tmp的内容交给了 GetTemp的返回值。GetTemp返回值 占用了 tmp的内存。

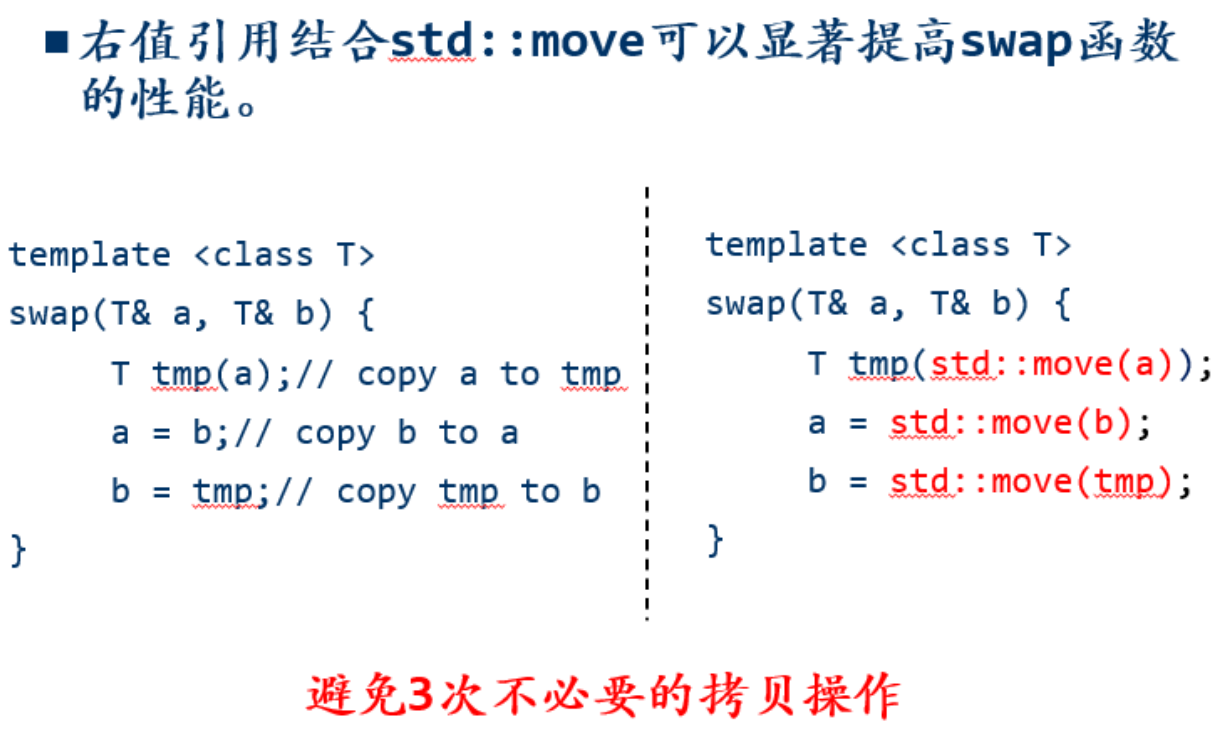
第二个移动构造：Test a=GetTemp(); a又占用了GetTemp()的内存。

std::move

那么，如果有一个不需要的左值，如何调用移动构造函数呢？我们可以使用std::move函数解引用，将左值转化为右值。亦即，将变量和变量值分离，变量转化为未初始化变量，变量值处于“无主”状态。例如：

```
Test a;
Test b = std::move(a) //a will not be used
```

上面的结果是，a变为一个**没有赋值**的Test类型变量，b“鸠占鹊巢”地霸占了a原来的值。



4. 赋值运算符

已定义的对象之间相互赋值，在C++中是通过调用对象的“赋值运算符函数”来实现的：

```
ClassName& operator= (const ClassName& right) {
   if (this != &right) {// 避免自己赋值给自己
        // 将right对象中的内容复制到当前对象中...
        this->buf = right.buf;
        cout << "operator=(const Test&) called.\n"
    }
   return *this;
}
```

应用：

```
ClassName a;
ClassName b;
a = b;
```

注意这个差别`ClassName a = b;`

5. 类型转换

当编译器发现表达式和函数调用所需的数据类型和实际类型不同时，便会进行自动类型转换。

自动类型转换可通过定义特定的**转换运算符**和**构造函数**来完成：

1. 在源类中定义“目标类型转换运算符”

```
class Src {
public:
  Src() { cout << "Src::Src()" << endl; }
  operator Dst() const {
    cout << "Src::operator Dst() called" << endl;
    return Dst();
  }
};
```

```
class Src; // 前置类型声明，因为在Dst中要用到Src类
class Dst {
public:
  Dst() { cout << "Dst::Dst()" << endl; }
  Dst(const Src& s) {
    cout << "Dst::Dst(const Src&)" << endl;
  }
};
```

2.在目标类中定义“源类对象作参数的构造函数”

以上两种方法任选一种，下面的代码都可以正确运行：

```
void Func(Dst d) { }
int main()
{
  Src s;
  Dst d1(s);  //显式类型转换

  Dst d2 = s; //隐式类型转换
  Func(s);   //隐式类型转换
  return 0;
}
```

如果用**explicit**修饰类型转换运算符或类型转换构造函数，则相应的类型转换必须显式地进行，则上面的代码 `Dst d2 = s`，`Func(s)` 就会报错。

除自动类型转换外，在有必要的时候还可以进行**强制类型转换。static_cast**，类似于C风格的强制转换。无条件转换，静态类型转换。之前的示例修改为：

```
int main()
{
  Src s;
  Dst d1(s);

  Dst d2 = static_cast<Dst>(s);
  Func(static_cast<Dst>(s));
  return 0;
}
```