继承构造函数

- 类具有可派生性，派生类自动获得基类的成员变量和接口（虚函数和纯虚函数）
- 基类的构造函数也没有被继承，因此：

```
class A {
public:
    A(int i) {}
};

class B : public A {
public:
    B(int i): A(i), d(i) {}
private:
    int d;
};
```

- B的构造函数起到了传递参数给A的构造函数的作用：透传
- 如果A具有不只一个构造函数，B往往需要设计对应的多个透传

`using` 声明

- 派生类用 `using` 声明来使基类的成员函数成为自己的
  - 解决name hiding问题：非虚函数被 `using` 后成为派生类的函数
  - 解决构造函数重载问题

```cpp
class Base {
public:
    void f(double ) {
        cout << "double\n";
    }
};

class Derived : Base { //不是public继承
public:
    using Base::f;
    void f(int ) {
        cout << "int\n";
    }
};

int main()
{
    Derived d;
    d.f(4);
    d.f(4.5);
}
```

3-3.cpp

```
class A {
public:
    A(int i) { cout << "int\n"; }
    A(double d, int i) {}
    A(float f, char *s) {}
};

class B : A {
public:
    using A::A;
};

int main()
{
    B b(2);
}
```

- 继承构造函数是隐式声明的，如果没有用到就不产生代码

```
g++ 3-4.cpp --std=c++11
```

- 如果基类的函数具有默认参数值，`using`的派生类无法得到默认参数值，就必须转为多个重载的函数

```
class A {
public:
    A(int a=3, double b=2.4) {}
};
```

- 实际上可以被看作是：

```
A(int, double);
A(int);
A();
```

- 那么，被`using`之后就会产生相应的多个函数

- 多继承且多处继承构造函数时，可能出现的重载冲突只能通过主动定义派生类的构造函数来解决
- 不能声明继承构造函数的情况：
  - 基类的构造函数是私有的
  - 派生类是从基类中虚继承的
- 一旦声明了继承构造函数，就不会再生成自动默认构造函数了

委派构造函数

- 如果重载的多个版本的构造函数内要做相同的事情
  - 显然代码复制是设计不良的突出表现
  - 在构造函数内调用某个函数的问题是它发生在初始化之后

```
class Info {
public:
    Info() { InitRest();}   //  目标
    Info(int i) : Info() { tyep = i; }  // 委派
    Info(char e) : Info() { name=e; }
};
```

- 目标构造函数的执行先于委派构造函数
- 构造函数不能同时委派和使用初始化列表
    - 在构造函数内赋值不是最好的选择

```
class Info {
public:
    Info(): Info(1, 'a') {}    //  委派
    Info(int i) : Info(i, 'a') {}  // 委派
    Info(char e) : Info(1, e) {  }
private:
    Info(int i, char e): type(i), name(e) {}     //  目标
};
```

- 委派关系可以形成链接

```
class Info {
public:
    Info(): Info(1) {}    //  委派
    Info(int i) : Info(i, 'a') {}  // 目标&委派
    Info(char e) : Info(1, e) {  }
private:
    Info(int i, char e): type(i), name(e) {}    //  目标
};
```

- 但是要防止出现循环链接

浅拷贝vs深拷贝

- 浅拷贝：编译器自动产生的拷贝构造函数，会执行member-wise copy
- 当有成员变量是指针时，这种拷贝是有害的

```
class C {
public:
    C():i(new int(0)){
        cout << "none argument constructor called" << endl;
    }
    ~C(){
       cout << "destructor called" << endl;
        delete i;
    }
    int* i;
};

int main(){
    C c1;
    C c2 = c1;
    cout << *c1.i << endl;
    cout << *c2.i << endl;
}
```

- 所以必须编写自己的拷贝构造函数来实现深拷贝

```
class C {
public:
    C():i(new int(0)){
        cout << "none argument constructor called" << endl;
    }
    //增加此拷贝构造函数，根据传入的c，new一个新的int给i变量
    C(const C& c) :i(new int(*c.i)){
    }
    ~C(){
       cout << "destructor called" << endl;
        delete i;
    }
    int* i;
};
```

移动语义

- 拷贝函数中为指针成员分配新的内存再进行内容拷贝的方法有些时候不是必要的

3-18.cpp

```
//这是一个成员包含指针的类
class HasPtrMem {
public:
    HasPtrMem() : d(new int(0)) {
        cout << "Construct:" << ++n_cstr << endl;
    }

    HasPtrMem(const HasPtrMem& h) {
        cout << "Copy construct:" << ++n_cptr << endl;
    }

    ~HasPtrMem() {
        cout << "Destruct:" << ++n_dstr << endl;
    }

private:
    int* d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;

HasPtrMem GetTemp() {
    return HasPtrMem();//①
}

int main(){
    HasPtrMem m = GetTemp();//②
    return 0;
}
```

- 构造函数被调用1次，是在①处，第一次调用拷贝构造函数是在GetTemp return的时候，将①生成的变量拷贝构造出一个临时值，来当做GetTemp的返回
- 第二次拷贝构造函数是在②处。
- 同时就有了于此对应的三次析构函数的调用
- 例子里用的是一个int类型的指针，而如果该指针指向的是非常大的堆内存数据的话，那没拷贝过程就会非常耗时，而且由于整个行为是透明且正确的，分析问题时也不易察觉。

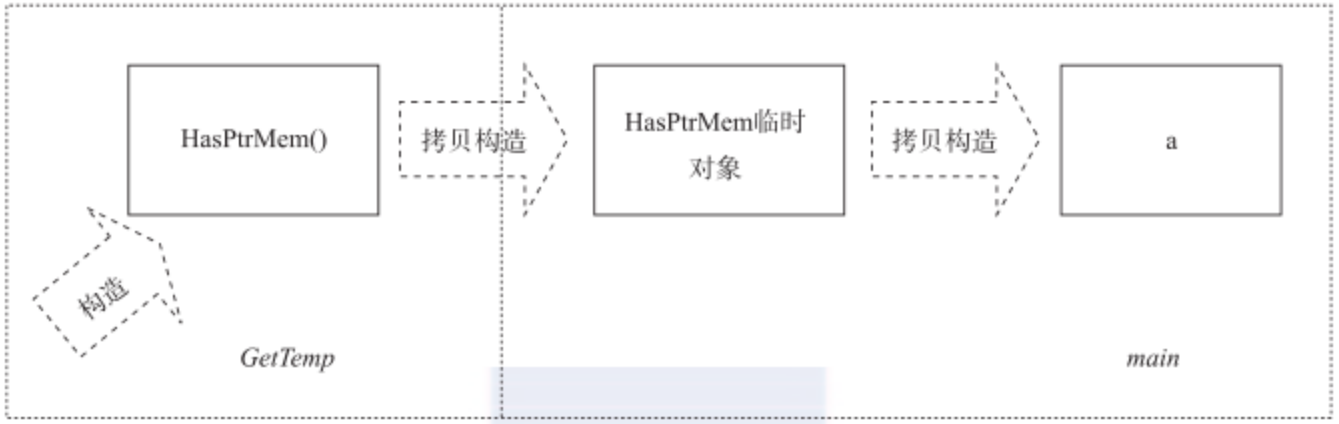

图 3-1　函数返回时的临时变量与拷贝

- 可以通过移动构造函数解决此问题

3-19.cpp

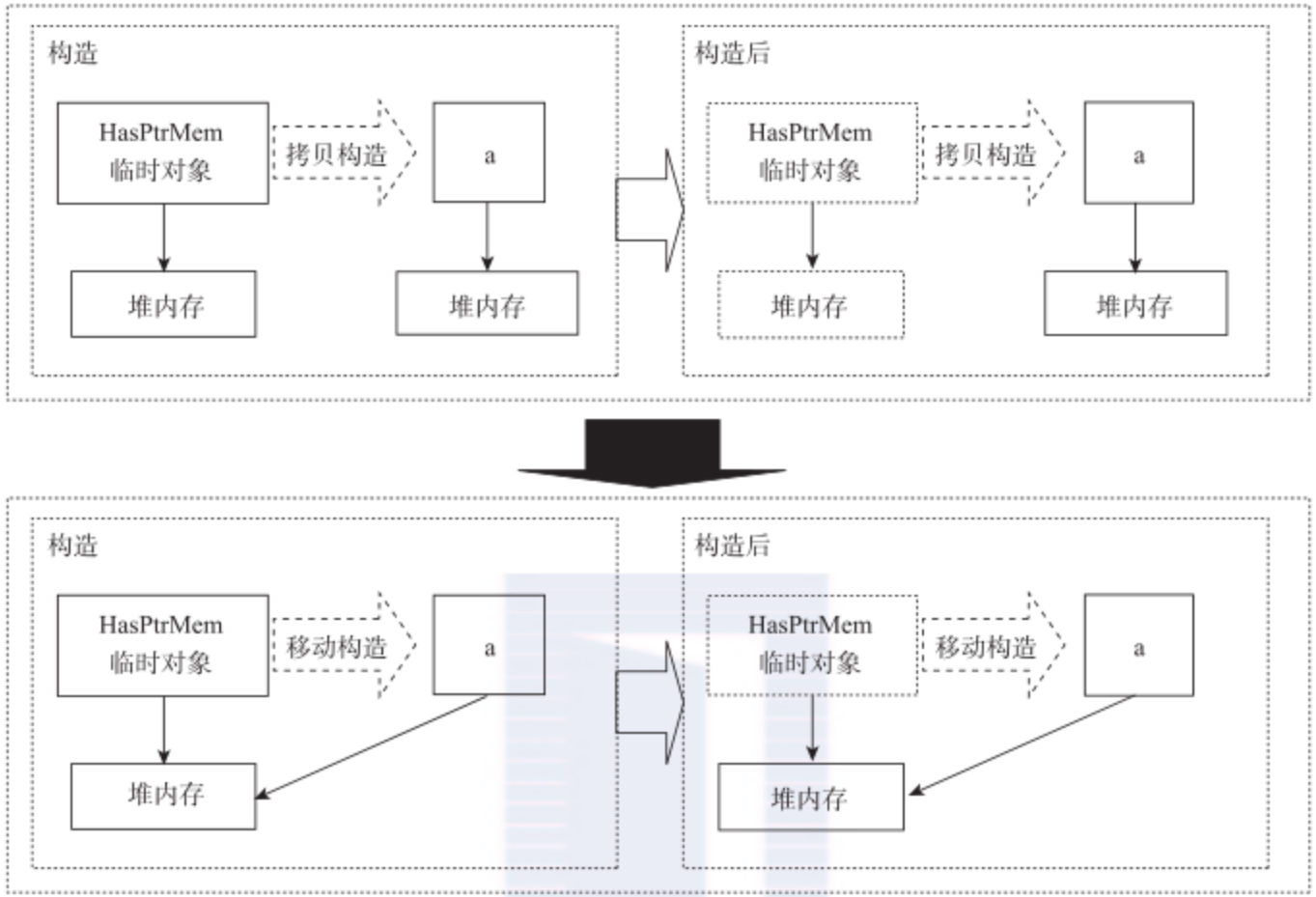

图 3-2　拷贝构造与移动构造

- 通过指针赋值的方式，将d的内存直接偷了过来，避免了拷贝构造函数的调用
- 注意③，这里需要对原来的d进行赋空值，因为在移动构造函数完成之后，临时对象会立即被析构，如果不改变d，那临时对象被析构时，因为偷来的d和原本的d指向同一块内存，会被释放，成为悬挂指针，会造成错误
- 为什么不用函数参数里带个指针或者引用当返回结果呢？不是性能的问题，而是代码编写效率及可读性不好

```
string *a;
int c = 1
int &b = c;
Calculate(GetTemp(),b);//最后一个参数用于返回结果
```

- 移动构造函数何时会被触发
  - 一旦用到的是个临时变量，那么移动构造语义就可以得到执行
  - 在C++中如何判断产生了临时对象？如何将其用于移动构造函数？是否只有临时变量可以用于移动构造?

左值、右值与右值引用

- 在赋值表达式中，出现在等号左边的就是“左值”，而在等号右边的，则称为“右值”
- 可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值
- 在C++11中，右值是由两个概念构成的，一个是将亡值（xvalue，eXpiring Value），另一个则是纯右值（prvalue，Pure Rvalue）

纯右值 vs 将亡值

- 纯右值就是C++98标准中右值的概念，讲的是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值就是一个纯右值。一些运算表达式，比如`1 + 3`产生的临时变量值，也是纯右值。而不跟对象关联的字面量值，比如：`2`、`'c'`、`true`，也是纯右值。此外，类型转换函数的返回值、lambda表达式等，也都是右值
- 将亡值则是C++11新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用`T&&`的函数返回值、`std::move`的返回值，或者转换为`T&&`的类型转换函数的返回值

右值引用

- 右值引用就是对一个右值进行引用的类型。事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。通常情况下，我们只能是从右值表达式获得其引用。比如：

```
T && a = ReturnRvalue();
```

- 这个表达式中，假设`ReturnRvalue`返回一个右值，我们就声明了一个名为`a`的右值引用，其值等于`ReturnRvalue`函数返回的临时变量的值。
- 右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名

```
T && a = ReturnRvalue();
```

- `ReturnRvalue` 函数返回的右值在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命期将与右值引用类型变量 `a` 的生命期一样。只要 `a` 还“活着”，该右值临时量将会一直“存活”下去
- 所以相比于以下语句的声明方式：

```
T b = ReturnRvalue();
```

- 右值引用变量声明就会少一次对象的析构及一次对象的构造。因为 `a` 是右值引用，直接绑定了 `ReturnRvalue()` 返回的临时量，而 `b` 只是由临时值构造而成的，而临时量在表达式结束后会析构因应就会多一次析构和构造的开销

- 能够声明右值引用 `a` 的前提是 `ReturnRvalue` 返回的是一个右值。通常情况下，右值引用是不能够绑定到任何的左值的。比如下面的表达式就是无法通过编译的。

```
int c;
int && d = c;
```

- 相对地，在C++98标准中就已经出现的左值引用是否可以绑定到右值（由右值进行初始化）呢？比如：

```
T & e = ReturnRvalue();
const T & f = ReturnRvalue();
```

- `e` 的初始化会导致编译时错误，而 `f` 则不会

```
T & e = ReturnRvalue();
const T & f = ReturnRvalue();
```

- 在常量左值引用在C++98标准中开始就是个“万能”的引用类型
- 可以接受非常量左值、常量左值、右值对其进行初始化
- 而且在使用右值对其初始化的时候，常量左值引用还可以像右值引用一样将右值的生命期延长
- 相比于右值引用所引用的右值，常量左值所引用的右值在它的“余生”中只能是只读的
- 相对地，非常量左值只能接受非常量左值对其进行初始化

`std::move()`

- C++11中，<utility>中提供了函数`std::move`，功能是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，用于移动语义
- 被转化的左值，其生命期并没有随着左右值的转化而改变

3-21.cpp

- 在编写移动构造函数的时候，应该总是使用`std::move`转换拥有形如堆内存、文件句柄的等资源的成员为右值，这样一来，如果成员支持移动构造的话，就可以实现其移动语义，即使成员没有移动构造函数，也会调用拷贝构造，因为不会引起大的问题

- 移动语义一定是要改变临时变量的值
- `Moveable c(move(a));` 这样的语句。这里的 `a` 本来是一个左值变量，通过 `std::move` 将其转换为右值。这样一来， `a.i` 就被 `c` 的移动构造函数设置为指针空值。由于 `a` 的生命期实际要到所在的函数结束才结束，那么随后对表达式 `*a.i` 进行计算的时候，就会发生严重的运行时错误

- 在C++11中，拷贝/移动改造函数有以下3个版本：

```
T Object(T&)
T Object(const T&)
T Object(T&&)
```

- 其中常量左值引用的版本是一个拷贝构造函数版本，右值引用参数的是一个移动构造函数版本
- 默认情况下，编译器会为程序员隐式地生成一个移动构造函数，但是如果声明了自定义的拷贝构造函数、拷贝赋值函数、移动构造函数、析构函数中的一个或者多个，编译器都不会再生成默认版本
- 所以在C++11中，拷贝构造函数、拷贝赋值函数、移动构造函数和移动赋值函数必须同时提供，或者同时不提供，只声明其中一种的话，类都仅能实现一种语义。

完美转发

- 完美转发(perfect forwarding)，是指在模板函数中，完全依照模板的参数类型将参数传递给模板中调用的另外一个函数

```
template <typename T>
void IamForwarding (T t)  {
    IrunCodeActually(t);
}
```

- 因为使用最基本类型转发，会在传参的时候产生一次额外的临时对象拷贝
- 所以通常需要的是一个引用类型，就不会有拷贝的开销。其次需要考虑函数对类型的接受能力，因为目标函数可能需要既接受左值引用，又接受右值引用，如果转发函数只能接受其中的一部分，也不完美

- 考虑类型：

```
typedef const A T;
typedef T& TR;
TR& v = 1;
```

- 在C++11中引入了一条所谓“引用折叠”的新语言规则

TR的类型定义	声明v的类型	v的实际类型
T&	TR	A&
T&	TR&	A&
T&	TR&&	A&
T&&	TR	A&&
T&&	TR&	A&

- 于是转发函数为：

```
template <typename T>
void IamForwarding (T&& t)  {
    IrunCodeActually(static_cast<T&&>(t));
}
```

对于传入的左值引用：

```
void IamForwarding (X& && t)  {
    IrunCodeActually(static_cast<X& &&>(t));
}
```

折叠后是：

```
void IamForwarding (X& t)  {
    IrunCodeActually(static_cast<X&>(t));
}
```

对于传入的右值引用：

```
void IamForwarding (X&& && t)  {
    IrunCodeActually(static_cast<X&& &&>(t));
}
```

折叠后是：

```
void IamForwarding (X&& t)  {
    IrunCodeActually(static_cast<X&&>(t));
}
```

3-24.cpp