# Copy and Move

# Copying

- Create a new object from an existing one
    - For example, when calling a function

```
// Currency as pass-by-value argument
void func(Currency p) {
   cout << "X = " << p.dollars();
}
...
Currency bucks(100, 0);
func(bucks); // bucks is copied into p
```

Example: HowMany.cpp

## The copy constructor

- Copying is implemented by the copy constructor
- Has the unique signature

```
T::T(const T&);
```

- Call-by-reference is used for the explicit argument
- C++ builds a copy ctor for you if you don't provide one!
    - Copies each member variable
        - Good for numbers, objects, arrays

          – Copies each pointer
        - Data may become shared!

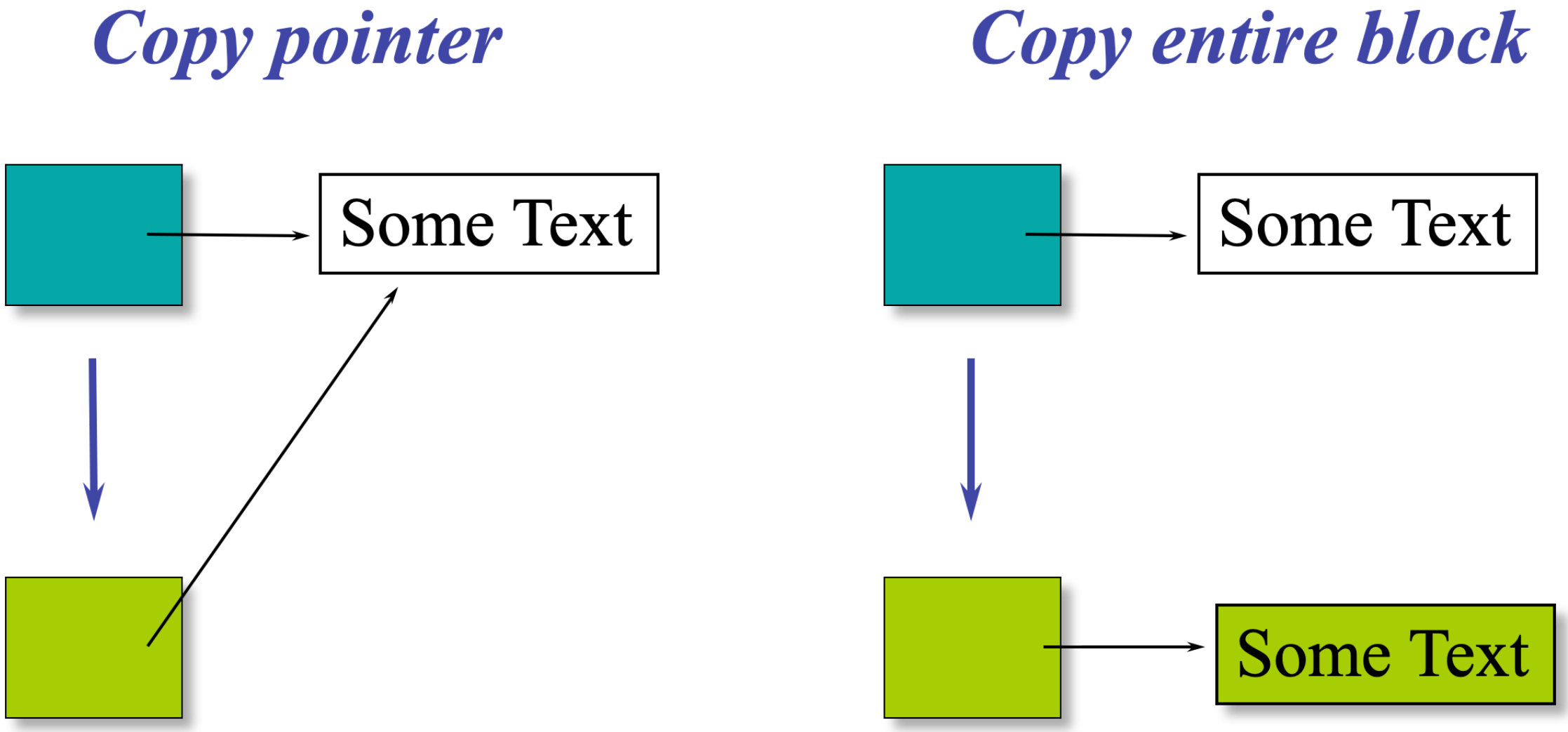
Example: HowMany2.cpp

## What if class contains pointers?

```
class Person {
public:
   Person(const char *s);
   ~Person();
   void print();
   // ... accessor functions
private:
   char *name;   // char * instead of string
   //... more info e.g. age, address, phone
};
```

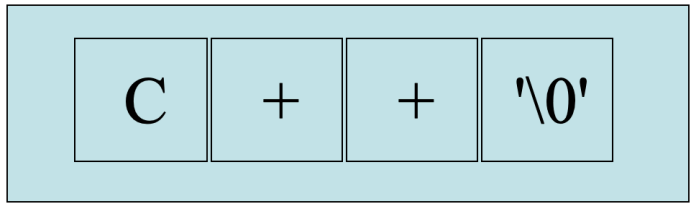
See: Person.h, Person.cpp

## Choices



## Character strings

- In C(and also in C++), a character string is
  - An array of characters
  - With a special terminator — `'\0'` or ASCII null
- The string `"C++"` is represented, in memory, by an array of four (4, count'em) characters



## Standard C library String fxns

- Declared in `<cstring>`

```
size_t strlen(const char *s);
```

- s is a null-terminated string
- returns the length of s
- length does not include the terminator!

```
char *strcpy (char *dest, const char *src);
```

- Copies src to dest stopping after the terminating null-character is copied. (src should be null-terminated!)
- dest should have enough memory space allocated to contain src string.
- Return Value: returns dest

## `Person (char*)` implementation

```
#include <cstring>      // #include <string.h>
using namespace std;
Person::Person( const char *s ) {
   name = new char[::strlen(s) + 1];
   ::strcpy(name, s); }
Person::~Person() {
   delete [] name;
}
// array delete
```

## Person copy constructor

- To Person.h declaration add copy ctor prototype:

```
Person( const Person& w );   // copy ctor
```

- To Person.cpp add copy ctor defintion:

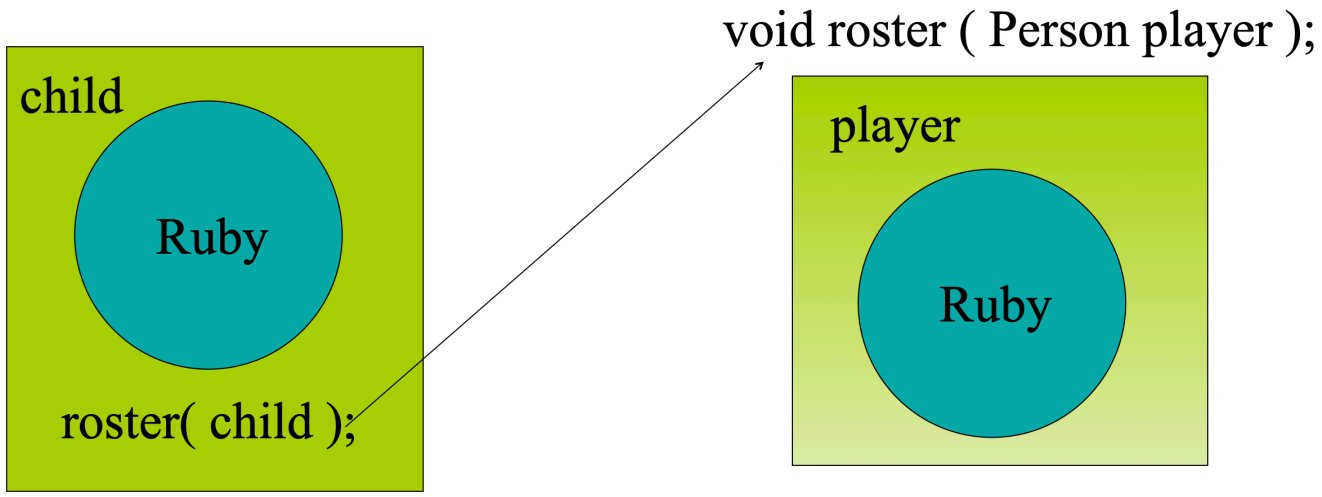
```
Person::Person( const Person& w ) {
   name = new char[::strlen(w.name) + 1];
   ::strcpy(name, w.name);
}
```

- No value returned
- Accesses `w.name` across client boundary
- The copy ctor initializes uninitialized memory

## When are copy ctors called?(1)

- During call by value

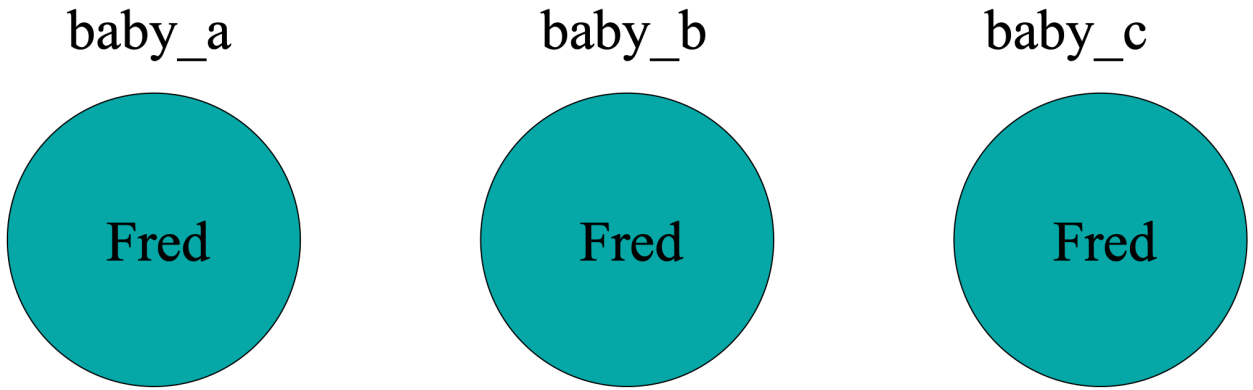
```
void roster( Person );  // declare function
Person child( "Ruby" ); // create object
roster( child );        // call function
```



## When are copy ctors called?(2)

- During initialization

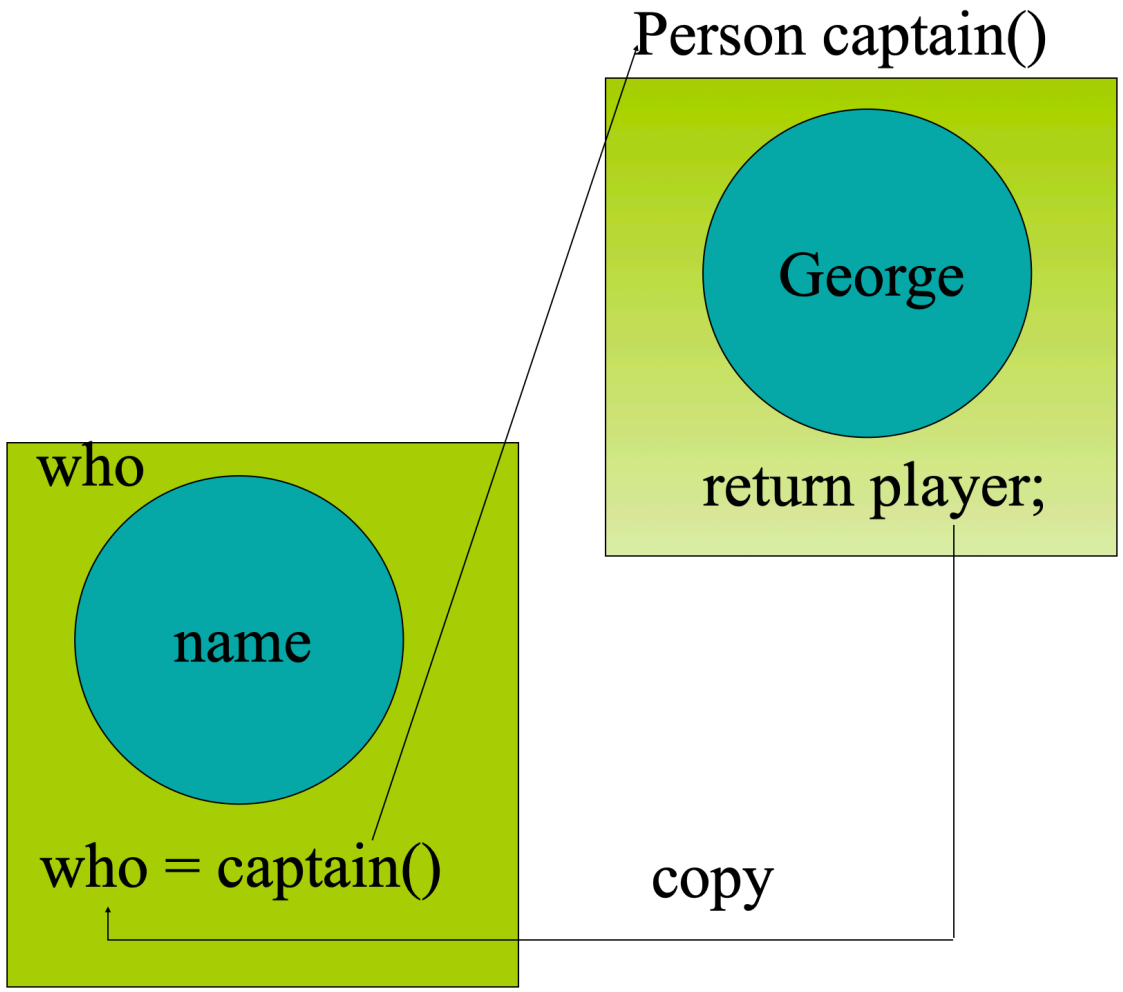
```
Person baby_a("Fred");
// these use the copy ctor
Person baby_b = baby_a;   // not an assignment
Person baby_c( baby_a );  // not an assignment
```



## When are copy ctors called?(3)

- During function return

```
Person captain()  {
   Person player("George");
   return player;
}
...
Person who("");
...
```



## Copies and overhead

- Compilers can "optimize out" copies when safe!
- Programmers need to
  - Program for "dumb" compilers
  - Be ready to look for optimizations

## Example

```
Person copy_func( char *who ) {
    Person local( who );
    local.print();
    return local; // copy ctor called!
}
Person nocopy_func( char *who ) {
    return Person( who );
} // no copy needed!
```

## Constructions vs. assignment

- Every object is constructed once
- Every object should be destroyed once
  - Failure to invoke `delete()`
  - Invoking `delete()` more than once
- Once an object is constructed, it can be the target of many assignment operations

# Person: string name

- What if the name was a `string` (and not a `char*`)

```
#include <string>
class Person {
public:
   Person( const string& );
   ~Person();
   void print();
   // ... other accessor fxns ...
private:
   string name;   // embedded object (composition)
   // ... other data members...
};
```

**Person: string name...**

- In the default copy ctor, the compiler recursively calls the copy ctors for all member objects (and base classes).
- default is memberwise initialization

Example: DefaultCopyConstructor.cpp

## Copy ctor guidelines

- No pointers, no need for a copy ctor
- In general, be explicit
  - Create your own copy ctor -- don't rely on the default
- If you don't need one declare a private copy ctor
  - prevents creation of a default copy constructor
  - generates a compiler error if try to pass-by-value - don't need a defintion

Example: NoCopyConstruction.cpp

# types of function parameters and return value

## way in

- a new object is to be created in f

```
void f(Student i);
```

- better with const if no intend to modify the object

```
void f(Student *p);
```

- better with const if no intend to modify the object

```
void f(Student& i);
```

## way out

- a new object is to be created at returning

```
Student f();
```

- what should it points to?

```
Student* f();
```

- what should it refers to?

```
Student& f();
```

## hard decision

```
char *foo() {
    char *p;
    p = new char[10];
    strcpy(p, "something");
    return p;
}
void bar() {
    char *p = foo();
    printf("%s", p);
    delete p;
}
```

## tips

- Pass in an object if you want to store it
- Pass in a const pointer or reference if you want to get the values
- Pass in a pointer or reference if you want to do something to it
- Pass out an object if you create it in the function
- Pass out pointer or reference of the passed in only
- Never new something and return the pointer

# 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;
```

- 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
}
```

- 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。

- 可以通过移动构造函数解决此问题

3-19.cpp

- 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？

# 移动拷贝函数

参数为右值引用的拷贝构造函数

Example: DynamicArray.cpp

## `std::move()`

```
vector<int> v1{1, 2, 3, 4};
vector<int> v2 = v1; // 此时调用用复制构造函数，v2是v1的副本vector<int> v3 = std::move(v1); // 此时调用用移动构造函数
```

- 通过 `std::move` 将 `v1` 转化为右值，从而激发 `v3` 的移动构造函数，实现移动语义

## `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& |
| 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

# 对象初始化的形式

```
//小括号初始化
string str("hello");

//等号初始化
string str = "hello";

//大括号初始化
struct Studnet
{
    char *name;
    int age;
};
Studnet s = {"dablelv", 18};//Plain of Data类型对象
Studnet sArr[] = {{"dablelv", 18}, {"tommy", 19}}; //POD数组
```

## 列表初始化

```
class Test
{
    int a;
    int b;
public:
    Test(int i, int j);
};
Test t{0, 0};                   //C++11 only, 相当于 Test t(0,0);
Test *pT = new Test{1, 2};      //C++11 only, 相当于 Test* pT=new Test(1,2);
int *a = new int[3]{1, 2, 0}; //C++11 only
```

## 容器初始化

```
// C++11 container initializer
vector<string> vs={ "first", "second", "third"};
map<string,string> singers ={ {"Lady Gaga", "+1 (212) 555-7890"},{"Beyonce Knowles", "+1 (212) 555-0987"}};
```

## std::swap()

```
void swap(T& a, T& b) {
    T tmp{a}; // 调用用拷贝构造函数
    a = b; // 拷贝赋值运算符
    b = tmp; // 拷贝赋值运算符
}
```

```
void swap(T& a, T& b) {
    T temp{std::move(a)};
    a = std::move(b);
    b = std::move(tmp);
}
```

## using function

- The derived class is able to “using” functions of its parent class

```
class Base {
public:
   void f()
};
class Child : public Base {
public:
   using Base::f;
   void f(int i) {}
};
```

# 6个默认行为函数

- 默认构造
- 拷贝构造
- 移动构造
- 拷贝赋值
- 移动赋值
- 虚析构

## What we've learned today?

- Copy constructor
- Move constructor