# C++ 程序设计II 兼谈对象模型

## Conversion function - 转换函数 operator type()

```
// this type -> other type
class Fraction { // 分数类，分数可以被看成double
    public:
    Fraction(int num, int den = 1) : m_numerator(num), m_denominator(den) {}
    operator double() const // 最好加const
    {
        return ((double)m_numerator / m_denominator);
    } // 转换函数，不唯一，合理就行
    private:
        int m_denominator; // 分母
        int m_numerator; // 分子
};
// 使用
int main()
{
    Fraction f(3,5);
    double d = 4 + f; // 调用 operator double() 将 f 转换成 double
    cout << d << endl;// 4.6
}
```

## non-explicit-one-argument ctor

```
// other type -> this type
class Fraction { // 分数类
    public:
    // non-explicit-one-argument constructor
    Fraction(int num, int den = 1) : m_numerator(num), m_denominator(den) {}
    Fraction operator+(const Fraction& f)
    {
        return Fraction(...);
    }
    /* 当下面operator type() 函数同时存在时，会迷惑编译器，出错 ambiguous
    // 因为有两条路可行：
        1. 将 4 转Fraction
        2. 将 f 转double 在将 4.6 转Fraction
        从而产生歧义
    operator double() const // 最好加const
    {
        return ((double)m_numerator / m_denominator);
    } // 转换函数
```

```
    */
    private:
        int m_denominator; // 分母
        int m_numerator; // 分子
};
// 使用
int main()
{
    Fraction f(3,5);
    Fraction d = 4 + f; // 调用 non-explicit ctor 将 4 转换成 Fraction,然后调用operator+()
}
```

# explicit-one-argument ctor

```
// other type -> this type
class Fraction { // 分数类
    public:
    // explicit-one-argument constructor
    explicit Fraction(int num, int den = 1) : m_numerator(num), m_denominator(den) {} // 90% 的
    Fraction operator+(const Fraction& f)
    {
        return Fraction(...);
    }
    operator double() const // 最好加const
    {
        return ((double)m_numerator / m_denominator);
    } // 转换函数
    */
    private:
        int m_denominator; // 分母
        int m_numerator; // 分子
};
// 使用
int main()
{
    Fraction f(3,5);
    Fraction d = 4 + f; // [Error] conversion from double to Fraction requested, 4 不能自动转为
}
```

# 应用

```
template<class Alloc>
class vector<bool, Alloc> {
    public:
        typedef __bit_reference reference;
    protected:
        reference operator[](size_type n) {
            return *(begin() + defference_type(n));
        }
```

```
}
struct __bit_reference {
    unsigned int* p;
    unsigned int mask;
    ...
public:
    operator bool() { return !(!(*p & mask)); }
    ...
}
```

# pointer-like classes, 关于智能指针

> 一个 class 产生出来的对象像一个指针(do more then pointer, such as auto pointer)

```
template<class T>
class shared_ptr {
    public:
        // 固定写法
        T& operator*() const { return *px;}
        T* operator->() const { return px;}

        shared_ptr(T* p) : px(px) {}

    private:
        T*    px; // 指向 T 的指针
        long* pn;
        ...
};

struct Foo
{
    void method() {}
};
int main()
{
    shared_ptr<Foo> sp(new Foo); // 初始化sp为指向Foo的智能指针,即sp内部的px指向Foo
    Foo f(*sp); // *p 调用 operator*()
    sp -> method(); // 调用 method 函数 等价于 调用operator->() 再调用 px -> method();
}
```

# pointer-like classes, 关于迭代器(一种特别的智能指针)

```
// list 链表节点设计
template<class T>
class __list_node {
    void* prev;
    void* next;
```

```
    T data;
}

template<lass T, class Ref, class Ptr>
struct __list_iterator { // 链表的迭代器实现
    typedef __list_iterator<T, Ref, Ptr> self;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef __list_node<T>* link_type; // 指向链表节点的指针类型
    link_type node; // node 为指向链表节点的指针
    bool operator==(const self& x) const {return node == x.node;}
    bool operator!=(const self& x) const {return node != x.node;}
    // 本节重点
    reference operator*() const {return (*node).data;}
    pointer operator->() const {return &(operator*());}

    self& operator++() {node = (link_type)((*node).next); return *this;}//前置
    self operator++(int) {self tmp = *this; ++*this; return tmp;}//后置
    self& operator--() {node = (link_type)((*node).prev); return *this;}//前置
    self operator--(int) {self tmp = *this; --*this; return tmp;}//后置
};
// 使用
int main()
{
    list<Foo>::iterator it;
    *it; // 获得一个Foo对象
    it -> method();
    // 调用Foo::method()
    // 相当于 (*it).method();
    // 相当于 (&(*it)) -> method();
}
```

# function-like classes, 所谓 仿函数 (像函数的类)

> () 函数调用操作符 function call
>
> 一般只要看到class内重载了()操作符，那他的用意就是想要变成一个function,其构造的对象称为函数对象
>
> 标准库中，仿函数会继承一些奇特的基类如：unary_function , binary_function

```
// 大小为零
template<class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};



template<class T>
struct identity : public unary_function<T, T>{
    const T&
```

```
    operator() (const T& x) const {return x;} // 重载了函数调用操作符
};

template<class Pair> // 源代码暗示你给他一个 pair
struct select1st : public unary_function<Pair, typname Pair::first_type> {
    const typename Pair::first_type&
    operator() (const Pair& x) const {return x.first;}
};

template<class Pair>
struct select2nd : public unary_function<Pair, typname Pair::second_type> {
    const typename Pair::second_type&
    operator() (const Pair& x) const {return x.second;}
};

template<class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair() : first(T1()), second(T2()) {} // 无参初始化
    pair(const T1& a, const T2& b) : first(a), second(b) {} // 传参初始化
};

int main()
{
    pair<char,int> p('a',97);
    char ch = select1st(p);
    int in  = select2nd(p);
}
```

# namespase 经验谈

> 避免命名冲突

```
using namespace std;
// ------------------------------------------
#include <iostream>
#include <memory> // shared_ptr

namespace jj01 {
    void test_member_template() { ... }
} // namespace

// ------------------------------------------

#include <iostream>
#include <list>
namespase jj02 {
    template<typename T>
    using Lst = list<T,allocator<T>>;
```

```
    void test_template_param() {...}
} // namespace

// ------------------------------------------
// 可以将所有测试函数放在一个main里面执行
int main(int argc, char** argv) {
    jj01::test_member_template();
    jj02::test_template_param();
}
```

# class template, 模板类(泛型编程)

```
template<class T>
class Class {
    T a;
public:
    T getA() const { return a;}
};
```

# function template, 函数模板

> 编译器会对 function template 进行 实参推导 (augument deduction)

```
class stone {
public:
    stone (int w, int h, int we)
    : _w(w), _h(h), _weight(we) {}
    bool operator< (const stone& ths) const {return _weight < rhs._weight;}
private:
    int _w, _h, _weight;
};

template<class T>
inline
const T& min(const T& a, const T& b)
{
    return b < a ? b : a; // 实参推导的结果，T 为stone，于是调用 stone::operator<()
}
```

# member template, 成员模板(令构造函数更有弹性)

```
template<class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
```

```
    T1 first;
    T2 second;

    pair() : first(T1()), second(T2()) {} // 无参初始化
    pair(const T1& a, const T2& b) : first(a), second(b) {} // 传参初始化

    // 成员模板(是成员又是模板)
    template<class U1, class U2>
    pair(const pair<U1, U2>& p) // 构造函数 U1,U2 必须(可转型为)继承于 T1,T2
    : first(p.first), second(p.second) {}
};
```

### 例子:

```
class Base1 {};
class Derived1 {};

class Base2 {};
class Derived2 {};

pair<Derived1, Derived2> p;
pair<Base1, Base2> p2(p); // 可以

pair<Base1, Base2> b;
pair<Derived1, Derived2> b2(b); // 不可以



template<typename _Tp>
class shared_ptr : public __shared_ptr<_Tp> {
    template<typename _Tp>
    explicit shared_ptr(_Tp1* __p)
    : __shared_ptr<_Tp>(__p) {}
};

Base1* ptr = new Derived1; // up-cast 向上造型
shared_ptr<Base1> sptr(new  Derived1); // 模拟up-cast
```

## specialization, 模板特化

```
// 泛化
template <class Key> // 绑定
struct hash { };

// 特化
template <>
struct hash<char> {
    size_t operator() (char x) const { return x;}
};
```

```
template <>
struct hash<int> {
    size_t operator() (int x) const { return x;}
}

int main()
{
    cout << hash<int> () (1000); // 临时对象
}
```

# partial specialization, 模板偏特化 -- 个数上的偏

```
// 泛化
template <typename T, typename Alloc = ...>
class vector
{
    pass
};
// 偏特化
template <typename T, typename Alloc=...>
class vector<bool, Alloc> // 绑定第一个模板参数，必须从左到右依次指定，不可跳跃
{
    pass
};

// ------------------------------------------
// 范围上的偏特化
template <typename T>
class C
{
    pass
};

template <typename T>
class C<T*>
{
    pass
};

C<string> obj1;
C<string*> obj2;
```

# template template parameter, 模板模板参数

```
 // Container为模板模板参数
template <typename T, template<typename T> class Container >
class XCls
{
```

```
private:
    Container<T> c;
public:
    pass
};

template <typename T>
using Lst = list<T, allocator<T>>;

XCls<string, list> my1st1; // Error list本身是模板，未定义
XCls<string, Lst> my1st2; // 可以



template<typename T, template<typename T> class SmatPtr>
class XCls
{
private:
    SmatPtr<T> sp;
public:
    XCls() : sp(new T) {}
};

XCls<string, shared_ptr> p1; // ok
XCls<string, unique_ptr> p2; // no
XCls<string, weak_ptr> p3; // no
XCls<string, auto_ptr> p4; // ok
```

## 这个不是 template template parameter

```
template<typename T, class Sequence = deque<T>>
class stack {
    pass
protected:
    Sequence c;
}

stack<int> s1;
stack<int, list<int> > s2; // 已经指定list<int>
```

# 关于 C++ 标准库

> Iterator 迭代器
>
> Container 容器
>
> Algorithm 算法
>
> Functors 仿函数

查看标准库，并实验每一个已经实现的标准库功能。

# C++11 新特性

## variadic templates 数量不定的模板参数 (since C++11)

```
template <typename T, typename... Types> // ... 也是语法的一部分
void print (const T& firstArg, const Types&... args)
{
    cout << firstArg << endl; // << 须重载
    print(args...); // 递归调用，每次分为firstArg和后面 Types... (其他的多个)
}

// 调用
print(7.5, "hello", bitset<16>(337), 42);
/*
7.5
hello
0000000101111001
42
*/

// 通过 sizeof(args...) 可以得到后面一包是几个
```

## auto 语法糖(since c++11)

```
list<string> c;
auto ite = find(c.begin(), c.end(), target);
// 让编译器自动帮你推断类型，但前提是你得有assign语句，不然编译器不知道怎么推
auto it; // 不能这样写，编译器不知道it是什么类型
it = find(c.begin(), c.end(),target); // 错误
```

## range-base for 语法糖(since C++11)

```
for(auto i : container) // 逐个取出，copy到 i 上 (pass by value)
{
    statement
}
for(auto &i : container) // pass by reference 更改了container 中的值
{
    statement
}
```

```
for(int i : {0,1,2,3,4,5,6,7,8,9}) // 新语法
{
    cout << i << endl;
}
```

## reference 引用(代表),实际上是指针实现的

```
int x = 0;
int* p = &x; // p pointer to x
int& r = x;  // r reference to x , r 代表 x, r 从一而终，不能再代表其它对象了
int x2 = 5;

// 编译器制造的假象
cout << (sizeof(r) == sizeof(x)) << endl; // true 但其实 r 是一个指针
cout << (&r == &x) << endl;               // true 假象

r = x2;      // r 不能从新代表其它对象，只是把 r 代表的 x 的值变成 x2 的值罢了
int& r2 = r; // r2 reference to r (r2 代表 r, 亦相当于代表 x)

// 不管是内置类型还是自定义类型都符合上述结论
```

> reference 通常不用于声明变量，而用于参数类型(parameter type)和返回类型(return type)的描述.

```
void func1(T *pobj) {pobj -> xxx();}
void func2(T pobj) {pobj.xxx();} // 须拷贝，传递较慢
void func3(T& pobj) {pobj.xxx();}

T obj;
func1(&obj);// 调用接口不同，困扰
func2(obj); // 调用接口相同，很好
func3(obj);

// 特别注意，写函数重载时，以下两种(same signature)不能并存
double imag(const double& im);
double imag(const double im); // Ambiguous
```

> Q : const 是不是函数签名的一部分，即是否能作为重载的指标?
>
> A : 是!

# Object Model 对象模型

## Part I 的承接

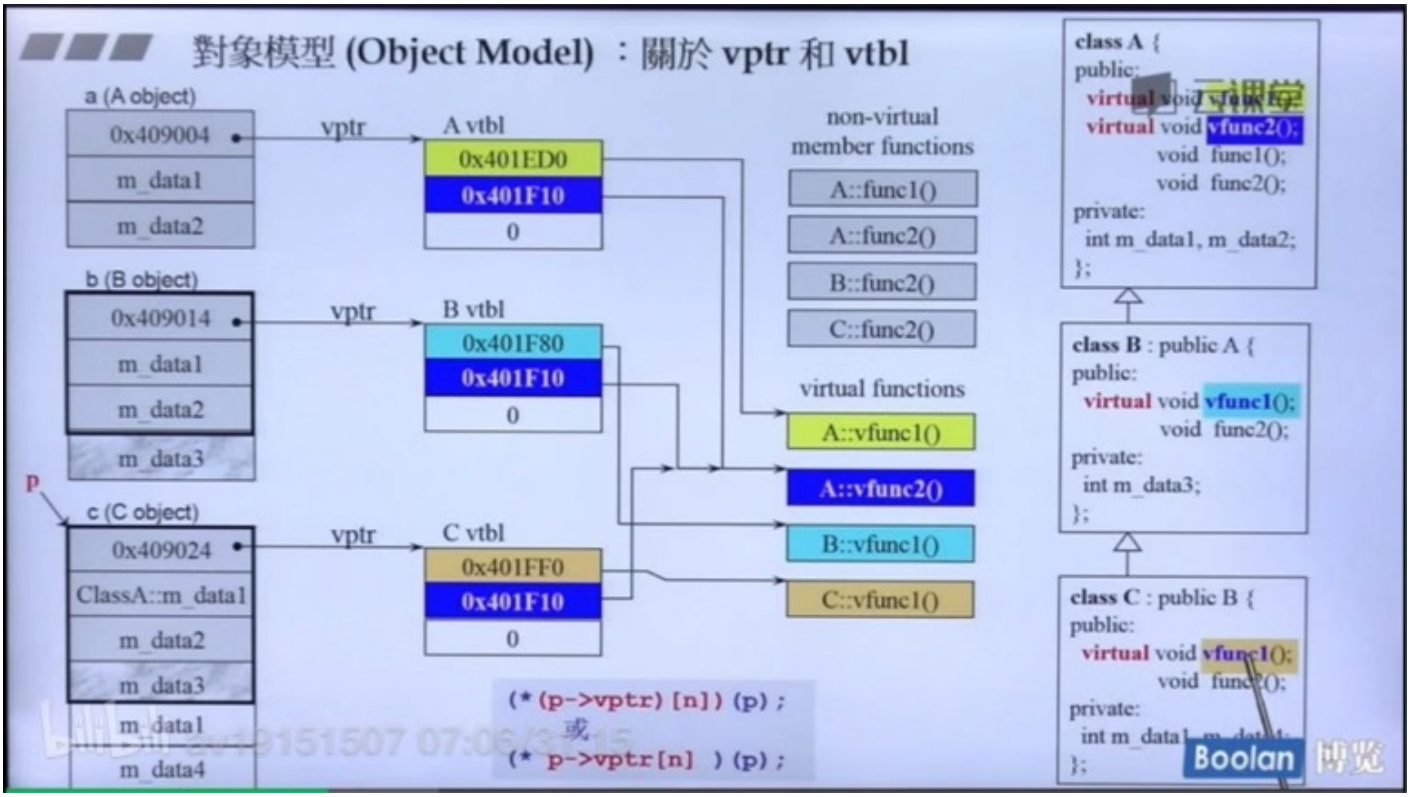
> Inheritance 继承： 构造由内而外，析构由外而内
>
> Composition 复合： 构造由内而外，析构由外而内
>
> Inheritance+Composition： 构造由内而外，析构由外而内
>
> > Derived::Derived(...) : Base(),Component() {...}
> >
> > Derived::~Derived() {... Component(); Base()} // 与构造相反

# 关于 vptr 和 vtbl (虚指针和虚表)

> 只要类中有虚函数，其对象在内存中就会多一根指针（指向虚表）
>
> 父类有虚函数，子类对象也一定拥有该指针

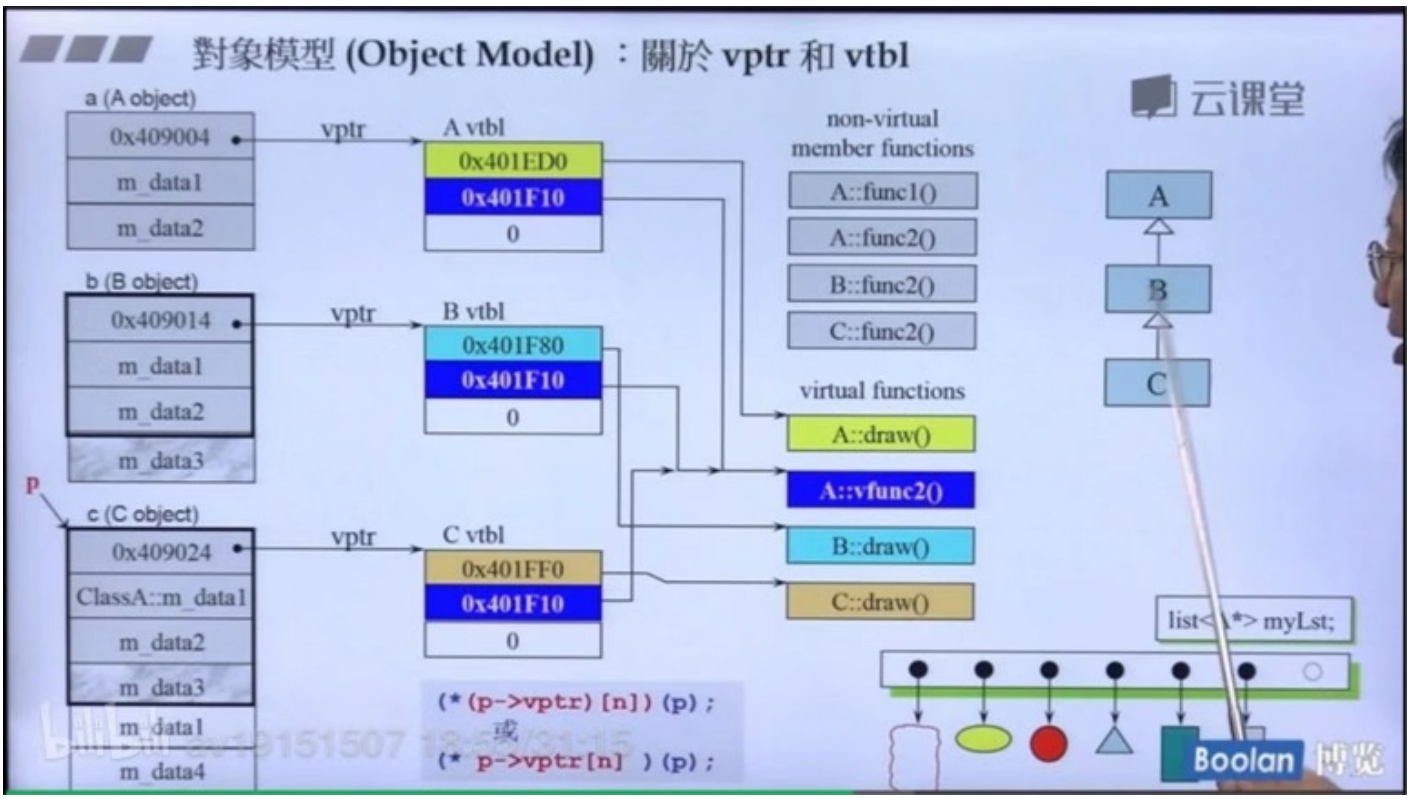
```
class A {
public:
    virtual void vfunc1();
    virtual void vfunc2();
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};
class B : public A {
public:
    virtual void vfunc1(); // override
    void func2();
private:
    int m_data3;
};
class C : public B {
public:
    virtual void vfunc1(); // override
    void func2();
private:
    int m_data1, m_data4;
};
```

## 内存模型： 动态绑定




```
C *p = new C;
(*p).vfunc2();
// 相当于,其中 n 表示虚表中的第 n 个虚函数
(*(p->vptr)[n])(p);
(* p->vptr[n] )(p);
```

## 应用：PPT 图形类(多态的应用)



## 总结:

C++ 编译器看到一个函数调用，会有两个考量（静态绑定，动态绑定)

1. 静态绑定是被编译成: CALL xxxx(func address)。
2. 但如果符合某些条件就会动态绑定：
    - 通过指针调用
    - 指针向上转型 up-cast
    - 所调用的是虚函数(virtual func)

      动态绑定的形式：虚机制

      多态：指针具有很多的类型（型态）

---

# 关于 this pointer