

P1 导读

- 泛型编程和面向对象编程都会讲

P2 conversion function 转换函数

- ```
class Fraction
{
public:
 Fraction(int num, int den=1)
 : m_numerator(num), m_denominator(den) { }
 operator double() const {
 return (double) (m_numerator / m_denominator);
 }
private:
 int m_numerator; //分子
 int m_denominator; //分母
};
```
- 
- ```
Fraction f(3,5);
double d=4+f;    //调用operator double()將 f 轉為 0.6
```
- 转换函数，也就是重载了类型转换运算符，通常加const，无参数，无返回值，而且也没必要显式调用，隐式类型转换的时候，就会被自动调用转换函数
 - 这个目标转换类型可以是基本类型，也可以是前面出现过的自定义类型

P3 non-explicit-one-argument constructor

```

class Fraction
{
public:
    Fraction(int num, int den=1)
        : m_numerator(num), m_denominator(den) { }

    Fraction operator+(const Fraction& f) {
        return Fraction(.....);
    }
private:
    int m_numerator;
    int m_denominator;
};

```

```

Fraction f(3,5);
Fraction d2=f+4; //調用 non-explicit ctor 將 4 轉為 Fraction(4,1)
                //然後調用operator+

```

- 这里就是反向转换了，把4转换成Fraction类型

```

class Fraction
{
public:
    Fraction(int num, int den=1)
        : m_numerator(num), m_denominator(den) { }
    operator double() const {
        return (double)(m_numerator / m_denominator);
    }
    Fraction operator+(const Fraction& f) {
        return Fraction(.....);
    }
private:
    int m_numerator;
    int m_denominator;
};

```

```

Fraction f(3,5);
Fraction d2=f+4; //[Error] ambiguous

```

- 在两个并存，并且加的顺序改了之后，会产生ambiguous二义性，报错无法通过
- explicit-one-argument ctor 明确的单参数构造函数

```

class Fraction
{
public:
    explicit Fraction(int num, int den=1)
        : m_numerator(num), m_denominator(den) { }
    operator double() const {
        return (double)(m_numerator / m_denominator);
    }
    Fraction operator+(const Fraction& f) {
        return Fraction(.....);
    }
private:
    int m_numerator;
    int m_denominator;
};

```

```

Fraction f(3,5);
Fraction d2=f+4; //[Error] conversion from 'double' to 'Fraction' requested

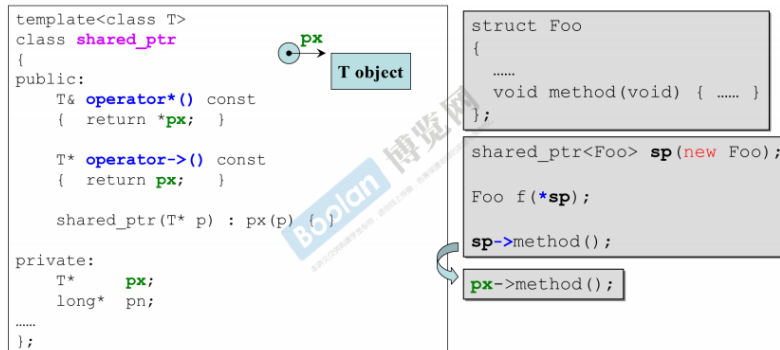
```

- 加上这个，4就不能随便转成4/1了。
- explicit关键字基本都用在构造函数前

P4 pointer-like classes

智能指针

- 一个类设计出来像指针，但是比指针更多一些功能



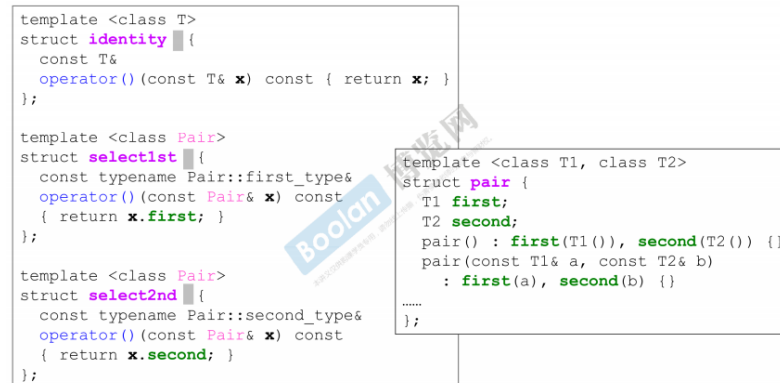
- 这里对指针的两个运算符重载，是语法规规定一定要这样写

迭代器

- 迭代器也是一种智能指针，迭代器在指针上包了一层，主要用来遍历容器

P5 function-like classes

- 设计一个class，让他的行为像一个函数
- 通常重载小括号运算符，通常都继承一些父类，这些父类大小为0，没有函数，只有一些类型的typedef，比如`unary_function`，`binary_function`等等

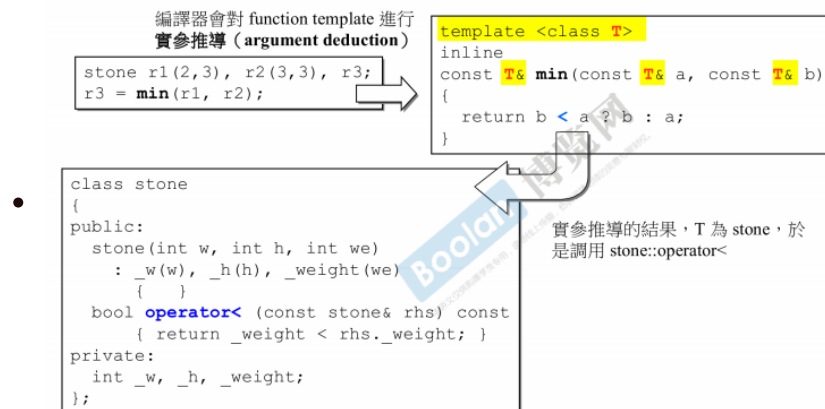


20

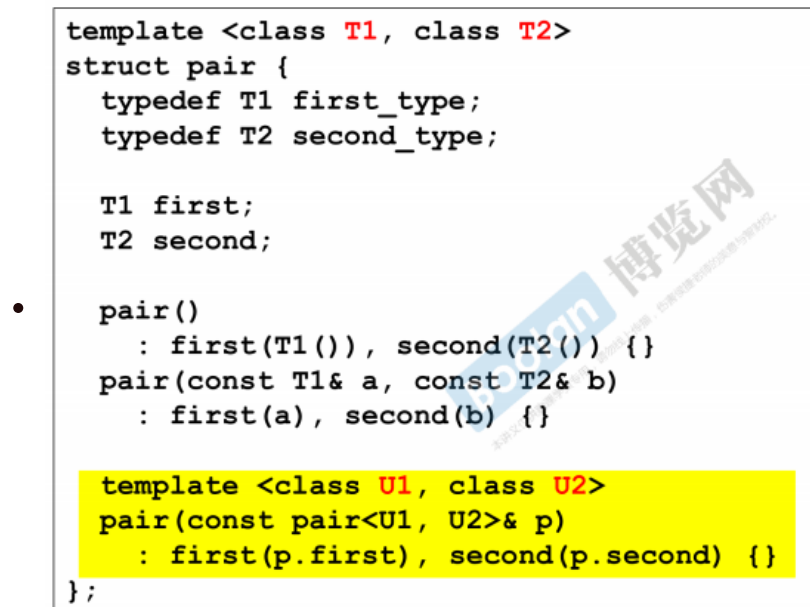
P6 namespace经验谈

P7 class template 类模板

P8 function template 函数模板

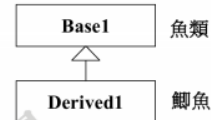


P9 member template 成员模板



- 黄色这一段在模板类里面，本身又是模板，所以叫成员模板
- T1 T2确定后，U1 U2还能变
- 这里的要求是初值p的U1能转型成T1，U2能转型成T2
- 向上转型up-cast
- 智能指针中也利用了这个成员模板

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



```
Base1* ptr = new Derived1; //up-cast

shared_ptr<Base1> sptr(new Derived1); //模擬 up-cast
```

P10 specialization 模板特化

- 从模板中抽出一些特殊的情况，给它开小灶，单独实现，调的时候优先调它

P11 partial specialization 模板偏特化

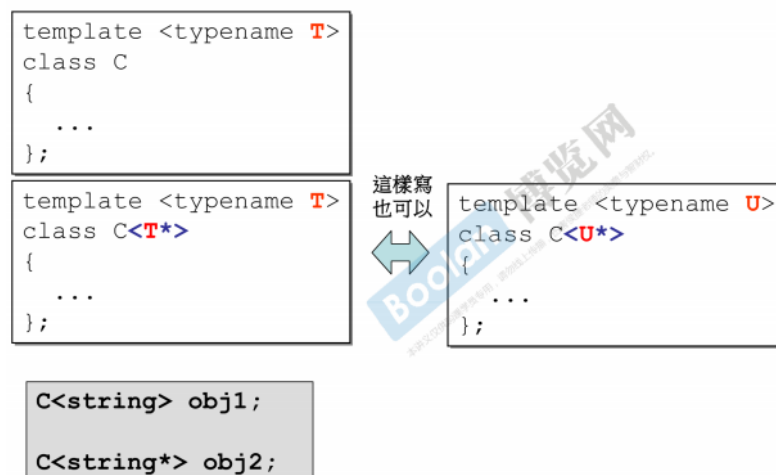
1. 个数的偏，绑定了一部分T

```
template<typename T, typename Alloc=.....>
class vector
{
...
};
```

綁定

```
template<typename Alloc=.....>
class vector<bool, Alloc>
{
...
}
```

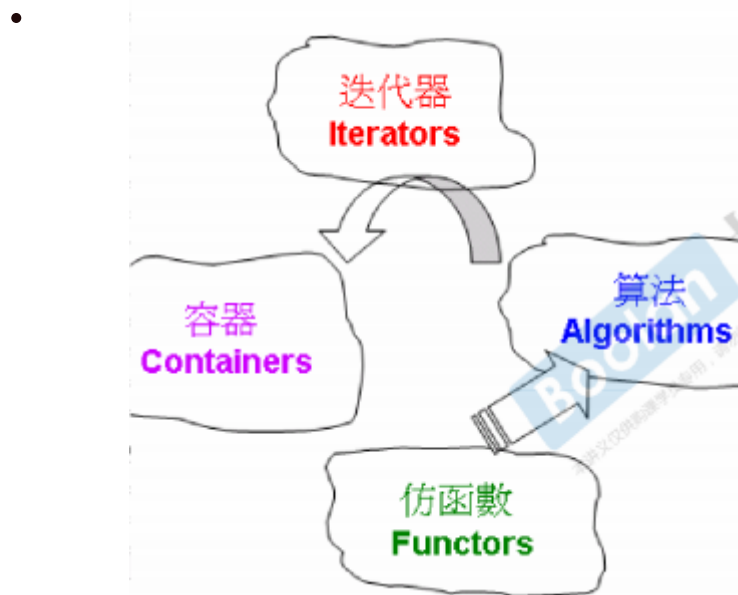
2. 范围的偏，不如从一个任意类型的T，到一个指针类型的T，也是一种偏特化



P12 template template parameter 模板模板参数

- 搞蛇皮啊，禁止套娃...
- 这一段模模糊糊...

P13 关于c++标准库



- 多写点小程序来测一下标准库的容器和算法
- 关于检查编译器支持c++哪个标准
 - 通过查看_cplusplus这个内置宏的值，199711就是c++98，201103就是c++11

- 可以在编译的时候，末尾加上-std=c++11或者-std=c++14，就可以支持到对应版本

P14 三个主题

variadic templates(since c++11) 不定模板参数

- 允许写任意个数的模板参数

auto(since c++11) 自动类型推导

- 要用auto之前，一定得先赋值。没赋值之前没法推导

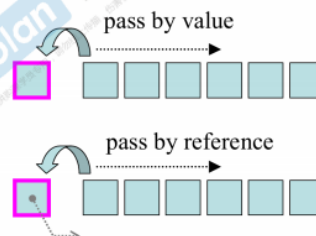
ranged-base for (since c++11)

```
for ( decl : coll ) {
    statement
}
```

```
for (int i : { 2, 3, 5, 7, 9, 13, 17, 19 }) {
    cout << i << endl;
}
```

- ```
vector<double> vec;
...
for (auto elem : vec) {
 cout << elem << endl;
}

for (auto& elem : vec) {
 elem *= 3;
}
```



- 这就有点像python了

## P15 reference 引用

---

```
int x = 0;
int& r = x;
int x2 = 5;
r = x2;
```

- 引用在赋初值之后就是别名了，等算一个const的pointer，再给它赋值，就等于给原本的x赋值了。但是编译器会把r和x变得一模一样，不管是地址还是大小，二者都相同。
- reference通常不用于声明变量，而是用于传参和传返回值

- ```

void func1(Cls* pobj) { pobj->xxx(); }
void func2(Cls obj) { obj.xxx(); }
void func3(Cls& obj) { obj.xxx(); }
.....
Cls obj;
func1(&obj); — 接口不同, 困擾
func2(obj); > 調用端接口相同, 很好
func3(obj); > 調用端接口相同, 很好

```

被調用端 寫法相同, 很好

reference 通常不用於聲明變量，而用於參數類型 (parameters type) 和返回類型 (return type) 的描述。

以下被視為 “same signature” (所以二者不能同時存在)：

```

double imag(const double& im) { ... }
double imag(const double im) { ... } // Ambiguity

```

signature

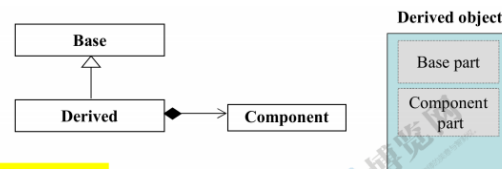
Q: const 是不是函數簽名的一部分?
A: Yes

P16 Object Model 对象模型

Inheritance关系下的构造和析构

Composition关系下的构造和析构

Inheritance+Composition关系下的构造和析构



構造由內而外

- Derived** 的構造函數首先調用 **Base** 的 default 構造函數，然後調用 **Component** 的 default 構造函數，然後才執行自己。

```

Derived::Derived(...): Base(), Component() { ... };

```

析構由外而內

Derived 的析構函數首先執行自己，然後調用 **Component** 的析構函數，然後調用 **Base** 的析構函數。

```

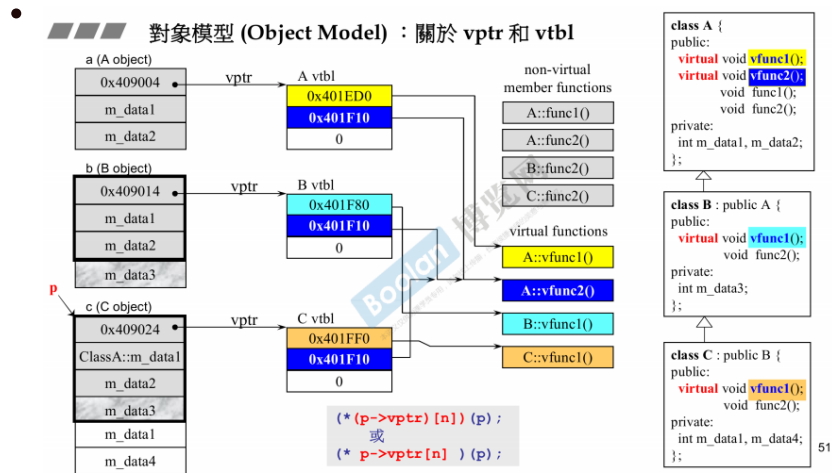
Derived::~Derived(...) { ... ~Component(), ~Base(); };

```

- 这里谁先后不好说，看具体编译器的实现

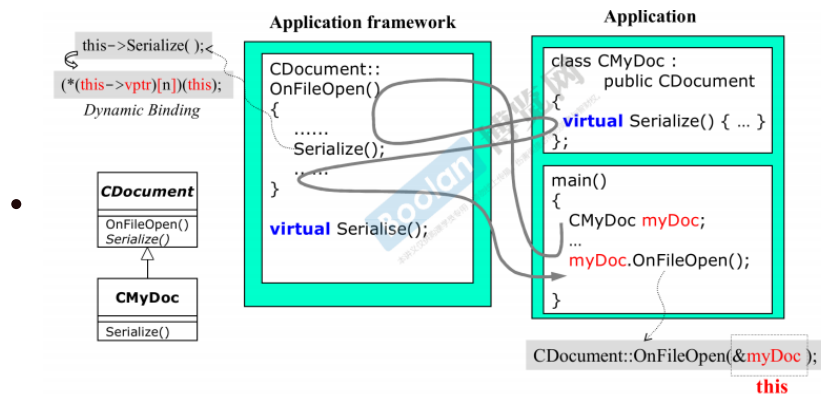
P17 对象模型 关于vptr和vtbl 虚指针和虚表

- 只要你的类里有虚函数，你的对象里就会多一个虚指针，指向一个虚表
- 继承，不止继承数据，也继承函数。而函数的占用内存大小不确定，所以继承函数，继承的不是函数的内存大小，而是函数的调用权



- 通过对象来调用虚函数时，不做静态绑定，而是动态绑定
 - 静态绑定就是函数被编译成call xxx, xxx是一个地址，是固定的
 - 动态绑定就是这个地址存在虚表里，还得找一趟
- 这里这个或，就没怎么懂，有哪个不行吗？感觉两个都对

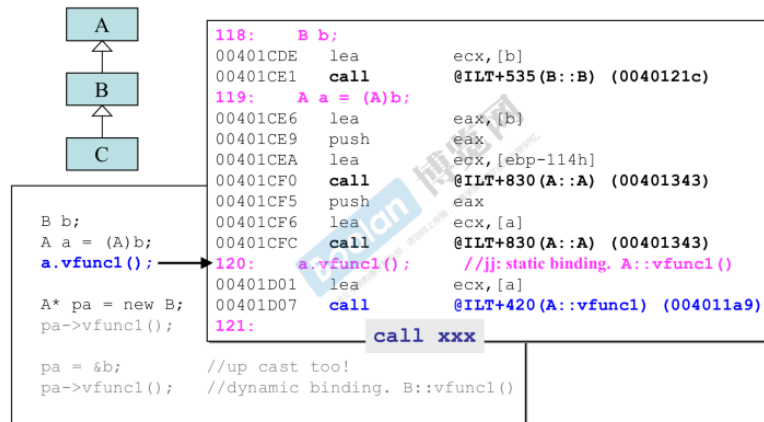
P18 对象模型 关于this



- 重点是这个this，左上角那两句就是动态绑定

P19 关于Dynamic Binding 动态绑定

- 这里通过对象来调用函数，属于静态绑定
-



- 下面通过指针来调用函数，属于动态绑定



- 这里的汇编，dword ptr就是一个2字节的指针，mov a b就是把b的值赋给a
- mov eax,dword ptr[pa];
- 就是把指针pa所指地址的值取出来赋给eax

P20 谈谈const

- const放在函数参数后面，是const member functions常量成员函数，一般的全局函数不能这样放

谈谈 const

當成員函數的 const 和 non-const 版本同時存在，const object 只會 (只能) 調用 const 版本，non-const object 只會 (只能) 調用 non-const 版本。

	const object (data members 不得變動)	non-const object (data members 可變動)
const member functions (保證不更改 data members)	✓	✓
non-const member functions (不保證 data members 不變)	✗	✓

class template std::basic_string<...> 有如下兩個 member functions:

```
charT
operator[] (size_type pos) const
{ ..... /* 不必考慮 COW */ }
```

```
reference
operator[] (size_type pos)
{ ..... /* 必須考慮 COW */ }
```

COW : Copy On Write

```
const String str("hello world");
str.print();
```

如果當初設計 string::print() 時未指明 const，那麼上行便是經由const object 調用non-const member function，會出錯。此非吾人所願

non-const member functions 可調用 const member functions，反之則不行，會引發：(VC) error C2662: cannot convert 'this' pointer from 'const class X' to 'class X &'. Conversion loses qualifiers

58

- const object不能调用non-const member functions
- const也属于函数签名的一部分
- 标准库的string使用了引用计数法，可以共享字符串的内容

P21 关于new delete

- new: 先分配memory, 再调用ctor
- delete: 先调用dtor, 再调用memory

P22 重载::operator new ::operator delete ::operator new[] ::operator delete[]

-
- The diagram shows the internal workings of `new` and `delete` for a class `Foo`. It includes a `try` block for `new` (allocating memory and casting to `Foo*`), a `delete` call, and a `Foo` class definition with `operator new` and `operator delete` methods. Arrows indicate the flow of memory allocation and deallocation.
- ```
Foo* p = new Foo;
```

  
...  

```
delete p;
```
  - ```
try {  
    1 void* mem = operator new(sizeof(Foo));  
    p = static_cast<Foo*>(mem);  
    2 p->Foo::Foo();  
}
```
 - ```
1 p->~Foo();
2 operator delete(p);
```
  - ```
class Foo {  
    public:  
        per-class allocator  
        void* operator new(size_t);  
        void operator delete(void*, size_t);  
        // ...  
};
```
- 可以通过重载这些, 接管内存的malloc和free
 - 重载带[]的new和delete, 就是对象数组, 内存空间大小就是对象数量*对象大小+一个表示对象数量的int值

P23 示例

-
- The diagram illustrates the use of global `operator new` and `operator delete` functions. It shows a code block with `Foo* pf = new Foo;` and `delete pf;`, and another block with `Foo* pf = ::new Foo;` and `::delete pf;`. A green arrow points from the global operators to the `::new` and `::delete` calls. A note indicates that if there are no members, global operators are used.
- ```
Foo* pf = new Foo;
```

```
delete pf;
```

  
下面强制採 globals  

```
Foo* pf = ::new Foo;
```

```
::delete pf;
```

  
若無 members 就調用 globals  

```
void* ::operator new(size_t);
```

```
void ::operator delete(void*);
```
  - 通过加::, 可以绕过现有的成员函数, 直接调用全局的new和delete

## P24 重载new(), delete()

- 可以重载class member operator new(), 前提是每个版本的重载都必须有独特的参数列, 并且第一参数必须是size\_t, 其余参数以new所指定的placement arguments为初值

## P25 basic\_string 使用new(extra)扩充申请量

- 为了在字符串里添加引用计数的功能，使用了new(extra)来实现

