引言

在面向对象高级编程(上)中，侯捷老师分别从**基于对象**、**面向对象**，讲述了C++编程过程中相关的知识点、注意事项以及如何写好规范的代码。

1.**Object Based(基于对象)：**面对的是单一class的设计

- Class without pointer member(s) —— complex类
- Class with pointer member(s)　　——string类

2.**Object Oriented（面向对象）：**面对的是多重classes的设计。classes和classes之间的关系。

接下来，将依托这个分类类分别叙述知识点。

Ⅰ：Object Based(基于对象)：面对的是单一class的设计

在基于对象编程中，由两个例子贯穿始终。一个是：complex类，另一个是：string类。

一、Class without pointer member(s) —— complex类

complex(复数)类中，不含指针，用默认的析构函数即可，所以这就是本案例写了构造函数，但没有写析构函数的原因。

1.头文件与类的声明

在写一个带指针的类时，一定要特别小心！如果一个类不带指针，则多半可以不写析构函数。

头文件的防卫式声明

作用：防止同一个文件被包含多次

1）#ifndef

```
#ifndef  __COMPLEX__
#define   __COMPLEX__

... ... // 声明、定义语句

#endif
```

特点：

- 跨平台
- 可针对文件也可针对代码片段。
- 编译慢，有宏命名冲突的风险。

2）**#pragmaonce**

```
#pragmaonce

... ... // 声明、定义语句
```

特点：

- 不跨平台
- 只能针对文件
- 编译快，无宏命名冲突的风险。

2.构造函数

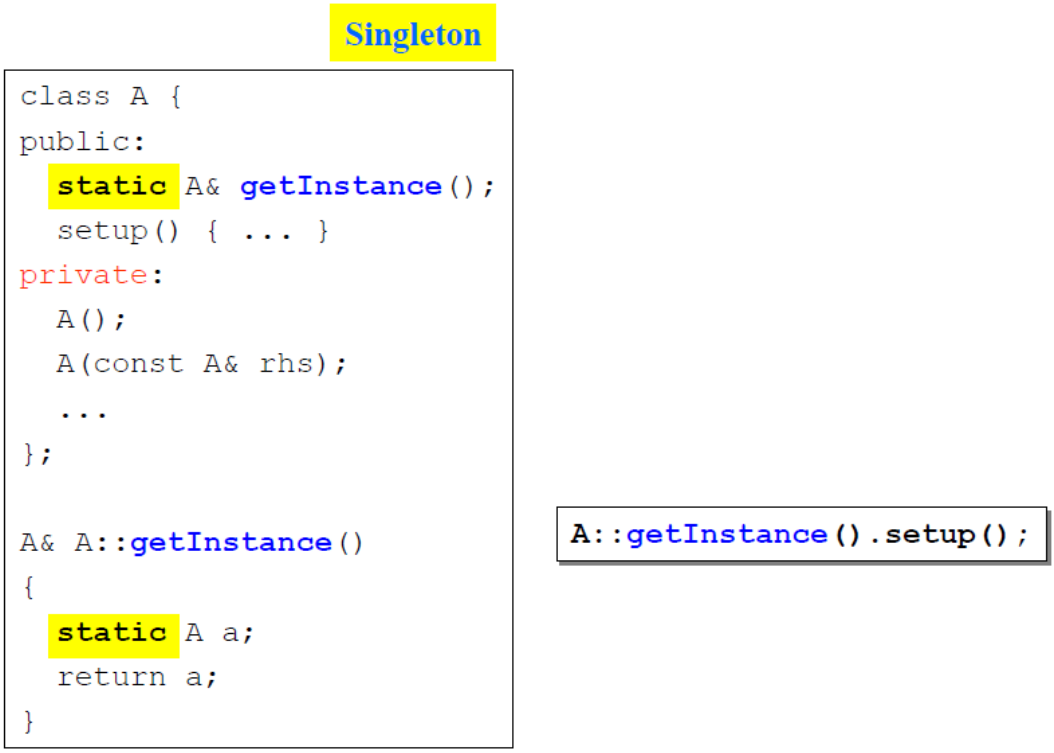
在class body 内定义的函数自动inline，在类外要加inline关键字。inline函数可以让编译变快，你可以试着把所有函数都定义inline，但编译器inline不inline就不一定了，换句话说，你只是提交了一份inline“申请”，如果inline的函数简单，编译器就给你通过”申请“。

函数重载常常发生在构造函数中。

如果有一个构造函数已经有默认值，可以重载其他的构造函数，但不能重载与它冲突的那一个。例如：

```
class complex{
public:
    complex(double r =0, double i =0)
    :re(r), im(i)
    {}
    complex():re(0), im(0){}  //ERROR!
    ....
private:
    double re, im;
};

//否则，该调用那个呢？有两个作用一样的构造函数。
complex c1;
complex c2();
```

构造函数放到private中，则外界不能创建该类的实例。在**单例Singleton模式**中就用到了这一点：



在一个类中，一定要将不**会改变数据**的成员声明为**const**。

3.参数传递与返回值

> 如果可以的话，参数传递与返回值的传递尽量**by reference**

参数传递的三种方式，设计类成员函数时，要提前考虑好那些函数的数据会改变，如果不改变请加上const。

- pass by value
- pass by reference
- pass by reference to const (推荐！)

返回值传递的时候，如果可以，建议使用return by reference。

那什么时候不能return by reference 呢?

首先考虑，如果一个函数操作得到一个结果B，那该结果放到什么位置上呢?

- 情况一：在**该函数区域**，创建一个新的变量i，将B传给它。（不能return by reference）
- 情况二：将结果B传递给该函数已存在的一个变量。（可以return by reference）

对情况一，此时`return i`的话，返回的是创建的新变量i，但请记住，i是一个局部变量，它的生命周期仅在创建它的函数中。此时如果return by reference的话就会报错！

对情况二则没有此限制，如下，请留意**函数第一个参数并不是**`const reference`：

```
inline complex&            //可以return by reference
__doapl(complex* ths, const complex& r){
    ths->re += r.re;       //ths是一个已存在的变量
    ths->im += r.im;
    return *ths;
}

inline complex&
complex::operator += (const complex& r)
{
    return __doapl (this, r);
}
```

友元（friend）函数可以取得类的private中的数据，但不建议这么做，因为会破坏封装性。

但有一点请记住：**相同class内的各objects互为友元！**所以下面类中的函数取用private数据合法。

```
class complex
{
public:
  complex (double r = 0, double i = 0)
    : re (r), im (i)
  { }

  int func(const complex& param)
  { return param.re + param.im; }

private:
  double re, im;
};
```

```
{
  complex c1(2,1);
  complex c2;

  c2.func(c1);
}
```

4.操作符重载与临时对象

（一）操作符重载之成员函数

- 任何成员函数都有一个**隐藏的pointer（即this）**,操作符重载也不例外。这个pointer(this)就指向调用者。对双目运算符来说，调用者就是左边的那个。

operator overloading (操作符重載-1, 成員函數) this

```
inline complex&
__doapl(complex* ths, const complex& r)
{
  ths->re += r.re;
  ths->im += r.im;
  return *ths;
}

inline complex&
complex::operator += (const complex& r)
{
  return __doapl (this, r);
}
```

```
{
  complex c1(2,1);
  complex c2(5);
  c2 += c1;
}
```

```
inline complex&
complex::operator += (this, const complex& r)
{
  return __doapl (this, r);
}
```

- **传递者**无需知道**接收者**是以**reference形式**接收。
- "+="操作符的重载不能返回`void`类型是因为：用户有可能会进行**连加**操作。

```
inline complex&
__doapl(complex* ths, const complex& r)
{
  ...
  return *ths;
}

inline complex&
complex::operator += (const complex& r)
{
  return __doapl(this,r);
}
```

连加 不能void

可以是void

```
c3 += c2 += c1;
```

```
{
  complex c1(2,1);
  complex c2(5);
  c2 += c1;
}
```

return by reference语法分析
传递者无需知道接收者是以reference形式接受。

（二）操作符重载之非成员函数

operator overloading (操作符重載-2, 非成員函數) (無 this)

2-3 為了對付 client 的三種可能用法，這兒對應開發三個函數

```
inline complex
operator + (const complex& x, const complex& y)
{
  return complex (real (x) + real (y),
                  imag (x) + imag (y));
}

inline complex
operator + (const complex& x, double y)
{
  return complex (real (x) + y, imag (x));
}

inline complex
operator + (double x, const complex& y)
{
  return complex (x + real (y), imag (y));
}
```

```
{
  complex c1(2,1);
  complex c2;

  c2 = c1 + c2;
  c2 = c1 + 5;
  c2 = 7 + c1;
}
```

与之前的区别在于这种重载无this指针，它是**全域/局函数。**

（三）临时对象

仔细考虑一下，为什么**（二）**中的三种重载返回值不是by reference 而是by value?

这是因为**在函数中创建了一个临时对象！**故只能by value。

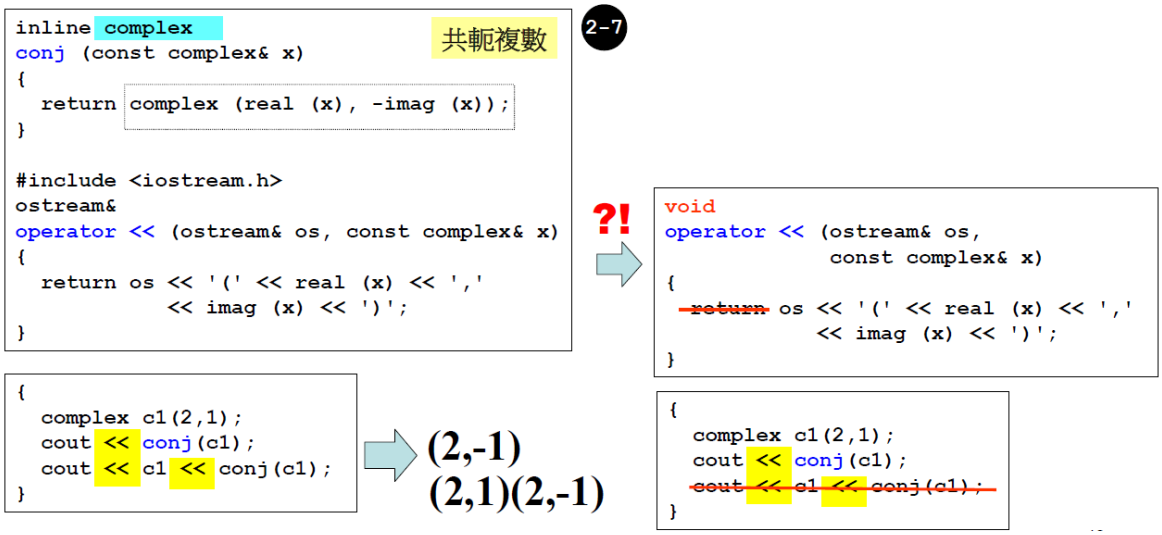
temp object (臨時對象) *typename* ();

2-3 下面這些函數絕不可 **return by reference**，因為，它們返回的必定是個 **local object.**

```
inline complex
operator + (const complex& x, const complex& y)
{
  return complex (real (x) + real (y),
                  imag (x) + imag (y));
}

inline complex
operator + (const complex& x, double y)
{
  return complex (real (x) + y, imag (x));
}

inline complex
operator + (double x, const complex& y)
{
  return complex (x + real (y), imag (y));
}
```

```
{
  int(7);

  complex c1(2,1);
  complex c2;
  complex();
  complex(4,5);

  cout << complex(2);
}
```

（四）千万不要把一些特殊的操作符重载写成员函数



5.complex类的完整实现

complex.h

```
#ifndef __MYCOMPLEX__
#define __MYCOMPLEX__

class complex;
complex&
  __doapl (complex* ths, const complex& r);
complex&
  __doami (complex* ths, const complex& r);
complex&
  __doaml (complex* ths, const complex& r);


class complex
{
public:
  complex (double r = 0, double i = 0): re (r), im (i) { }
  complex& operator += (const complex&);
  complex& operator -= (const complex&);
  complex& operator *= (const complex&);
  complex& operator /= (const complex&);
  double real () const { return re; }
  double imag () const { return im; }
private:
  double re, im;

  friend complex& __doapl (complex *, const complex&);
  friend complex& __doami (complex *, const complex&);
  friend complex& __doaml (complex *, const complex&);
};


inline complex&
__doapl (complex* ths, const complex& r)
{
  ths->re += r.re;
  ths->im += r.im;
```

```
  return *ths;
}

inline complex&
complex::operator += (const complex& r)
{
  return __doapl (this, r);
}

inline complex&
__doami (complex* ths, const complex& r)
{
  ths->re -= r.re;
  ths->im -= r.im;
  return *ths;
}

inline complex&
complex::operator -= (const complex& r)
{
  return __doami (this, r);
}

inline complex&
__doaml (complex* ths, const complex& r)
{
  double f = ths->re * r.re - ths->im * r.im;
  ths->im = ths->re * r.im + ths->im * r.re;
  ths->re = f;
  return *ths;
}

inline complex&
complex::operator *= (const complex& r)
{
  return __doaml (this, r);
}

inline double
imag (const complex& x)
{
  return x.imag ();
}

inline double
real (const complex& x)
{
  return x.real ();
}

inline complex
operator + (const complex& x, const complex& y)
{
  return complex (real (x) + real (y), imag (x) + imag (y));
}

inline complex
operator + (const complex& x, double y)
```

```
{
  return complex (real (x) + y, imag (x));
}

inline complex
operator + (double x, const complex& y)
{
  return complex (x + real (y), imag (y));
}

inline complex
operator - (const complex& x, const complex& y)
{
  return complex (real (x) - real (y), imag (x) - imag (y));
}

inline complex
operator - (const complex& x, double y)
{
  return complex (real (x) - y, imag (x));
}

inline complex
operator - (double x, const complex& y)
{
  return complex (x - real (y), - imag (y));
}

inline complex
operator * (const complex& x, const complex& y)
{
  return complex (real (x) * real (y) - imag (x) * imag (y),
                  real (x) * imag (y) + imag (x) * real (y));
}

inline complex
operator * (const complex& x, double y)
{
  return complex (real (x) * y, imag (x) * y);
}

inline complex
operator * (double x, const complex& y)
{
  return complex (x * real (y), x * imag (y));
}

complex
operator / (const complex& x, double y)
{
  return complex (real (x) / y, imag (x) / y);
}

inline complex
operator + (const complex& x)
{
  return x;
}
```

```
inline complex
operator - (const complex& x)
{
  return complex (-real (x), -imag (x));
}

inline bool
operator == (const complex& x, const complex& y)
{
  return real (x) == real (y) && imag (x) == imag (y);
}

inline bool
operator == (const complex& x, double y)
{
  return real (x) == y && imag (x) == 0;
}

inline bool
operator == (double x, const complex& y)
{
  return x == real (y) && imag (y) == 0;
}

inline bool
operator != (const complex& x, const complex& y)
{
  return real (x) != real (y) || imag (x) != imag (y);
}

inline bool
operator != (const complex& x, double y)
{
  return real (x) != y || imag (x) != 0;
}

inline bool
operator != (double x, const complex& y)
{
  return x != real (y) || imag (y) != 0;
}

#include <cmath>

inline complex
polar (double r, double t)
{
  return complex (r * cos (t), r * sin (t));
}

inline complex
conj (const complex& x)
{
  return complex (real (x), -imag (x));
}

inline double
```

```
norm (const complex& x)
{
  return real (x) * real (x) + imag (x) * imag (x);
}

#endif   //__MYCOMPLEX__
```

complex_text.cpp:

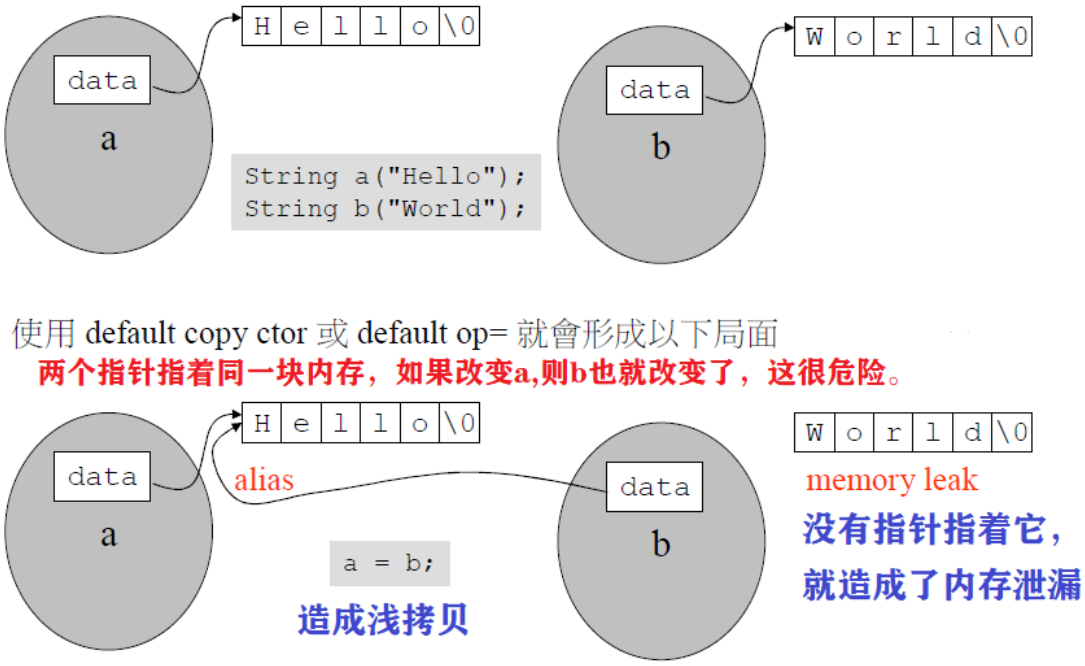
```
#include <iostream>
#include "complex.h"

using namespace std;

ostream&
operator << (ostream& os, const complex& x)
{
  return os << '(' << real (x) << ',' << imag (x) << ')';
}

int main()
{
  complex c1(2, 1);
  complex c2(4, 0);

  cout << c1 << endl;
  cout << c2 << endl;

  cout << c1+c2 << endl;
  cout << c1-c2 << endl;
  cout << c1*c2 << endl;
  cout << c1 / 2 << endl;

  cout << conj(c1) << endl;
  cout << norm(c1) << endl;
  cout << polar(10,4) << endl;

  cout << (c1 += c2) << endl;

  cout << (c1 == c2) << endl;
  cout << (c1 != c2) << endl;
  cout << +c2 << endl;
  cout << -c2 << endl;

  cout << (c2 - 2) << endl;
  cout << (5 + c2) << endl;

  return 0;
}
```

二、Class with pointer member(s) ——string类

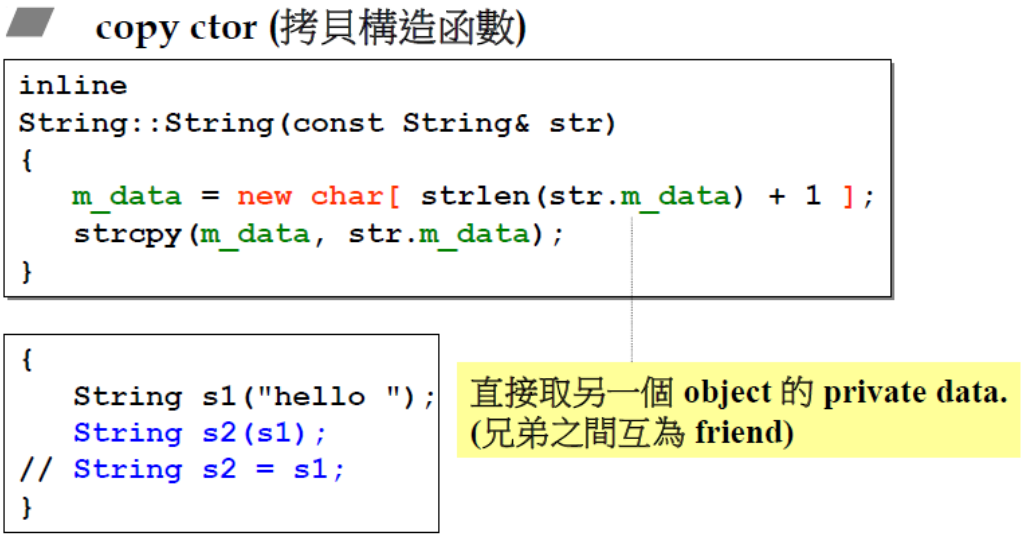
该string类，内含指针，所以要自己写析构函数，不能使用默认析构函数。

6.三大构造函数：拷贝构造、拷贝赋值、析构

1）class with pointer members 必须有拷贝构造和拷贝赋值，否则就会**造成浅拷贝。**



2）为了避免浅拷贝，所以要把指针所指的内容也要拷贝过来了，这叫**深拷贝。**



3）拷贝赋值的经典**四步曲**

以 `s1 = s2` 为例(s1、s2是两个字符串)：

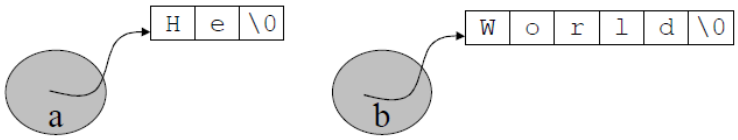
- 第一步：**检测自我赋值。**（否则有可能导致未定义情况）
- 第二步：清理掉s1的数据。
- 第三步：为s1分配一块与s2一样大的内存空间
- 第四步：将s2拷贝到s1中。

copy assignment operator (拷貝賦值函數)

```
inline
String& String::operator=(const String& str)
{
   if (this == &str)
      return *this;

1  delete[] m_data;
2  m_data = new char[ strlen(str.m_data) + 1 ];
3  strcpy(m_data, str.m_data);
   return *this;
}
```

檢測自我賦值
(self assignment)

```
{
    String s1("hello ");
    String s2(s1);
    s2 = s1;
}
```



53

7.堆、栈与内存管理

这部分内容建议看视频，视频比文字要来的清晰的多。这里只对截取一些概念、特征做一下说明。

（一）Stack(栈)

概念：是存在于某作用域 (scope) 的一块内存空间(memory space)。例如当你调用函数，函数本身即会形成一个stack 用來放置它所接收的参数，以及返回地址。
在函数本体 (function body) 內声明的任何变量，其所使用的內存块都取自上述 stack。

（二）heap(堆)

概念：或謂 system heap，是指由操作系統提供的一塊 global 內存空間，程序可動態分配 (dynamic allocated) 從某中獲得若干區塊 (blocks)。

```
class Complex { … };
...
{
  Complex c1(1,2);
  Complex* p = new Complex(3);
}
```

c1 所佔用的空間來自 stack

Complex(3) 是個臨時對象，其所佔用的空間乃是以 new 自 heap 動態分配而得，並由 p 指向。

（三）生命周期

1）stack objects 的生命期

```
class Complex { ... };
...

{
  Complex c1(1,2);
}
```

c1 便是所謂 stack object，其生命在作用域 (scope) 結束之際結束。
這種作用域內的 object，又稱為 auto object，因為它會被「自動」清理。

2）static local objects 的生命期

```
class Complex { … };
...

{
  static Complex c2(1,2);
}
```

c2 便是所謂 static object，其生命在作用域 (scope)
結束之後仍然存在，直到整個程序結束。

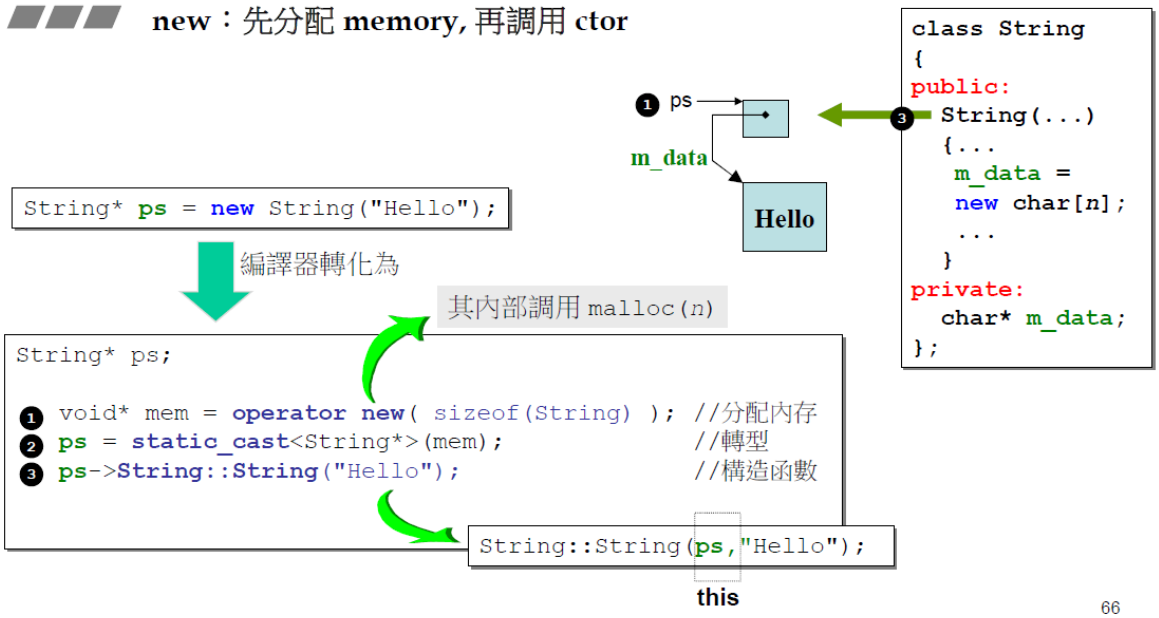
3）global objects 的生命期

```
class Complex { … };
...
Complex c3(1,2);

int main()
{
  ...
}
```

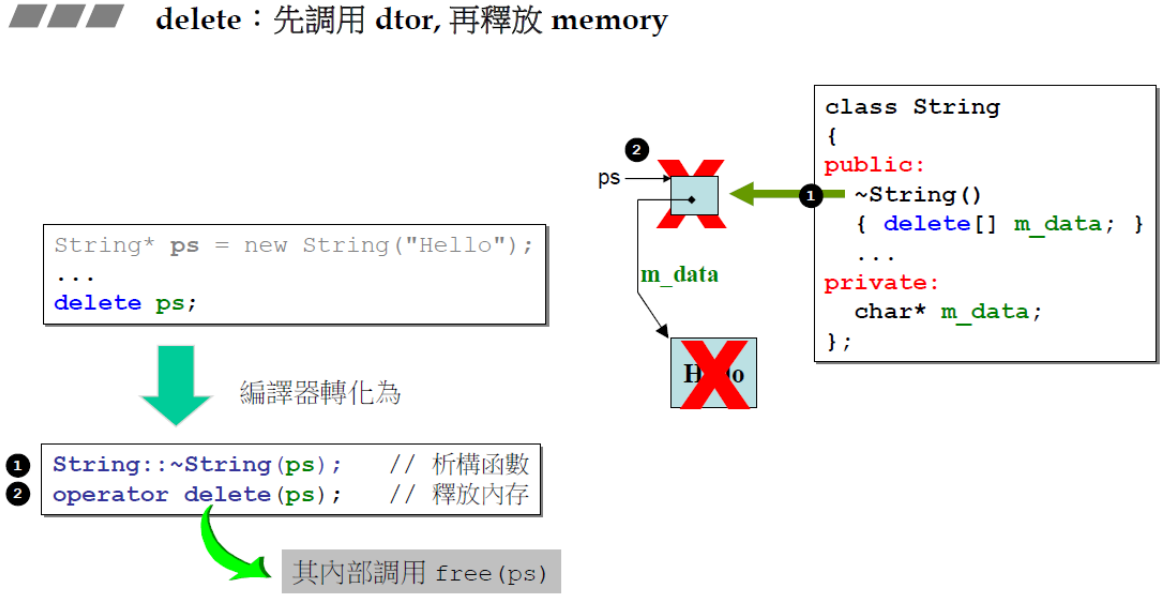
c3 便是所謂 global object，其生命在整個程序結束之後
才結束。你也可以把它視為一種 static object，其作用域
是「整個程序」。

（四）new 与 delete的工作流程（这里以string类为例，原视频中还讲了complex类）

1）**new:**先分配内存，再调用构造函数



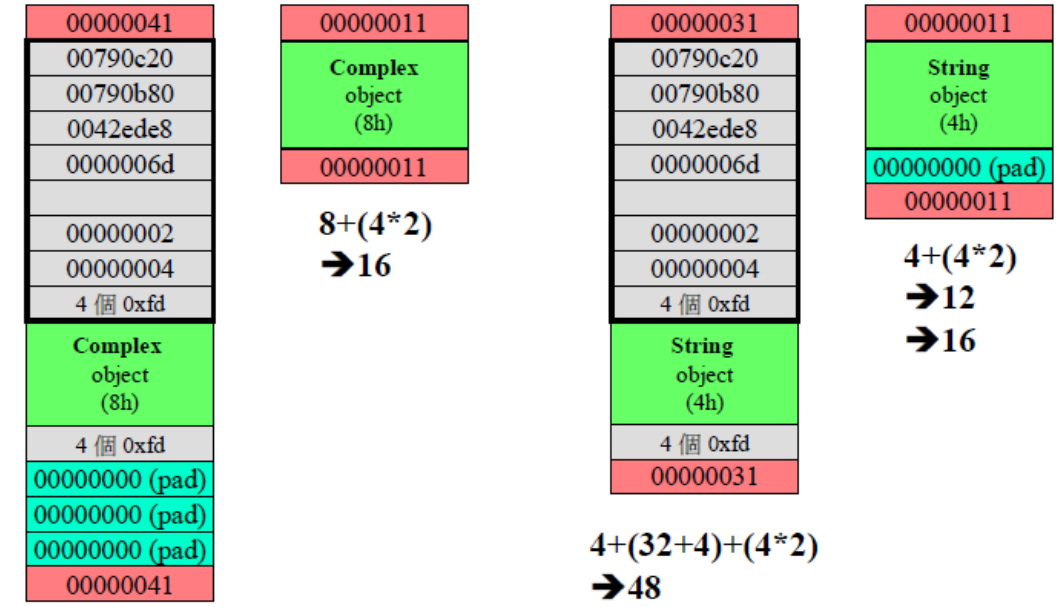
2）**delete**：先调用析构函数，在释放内存



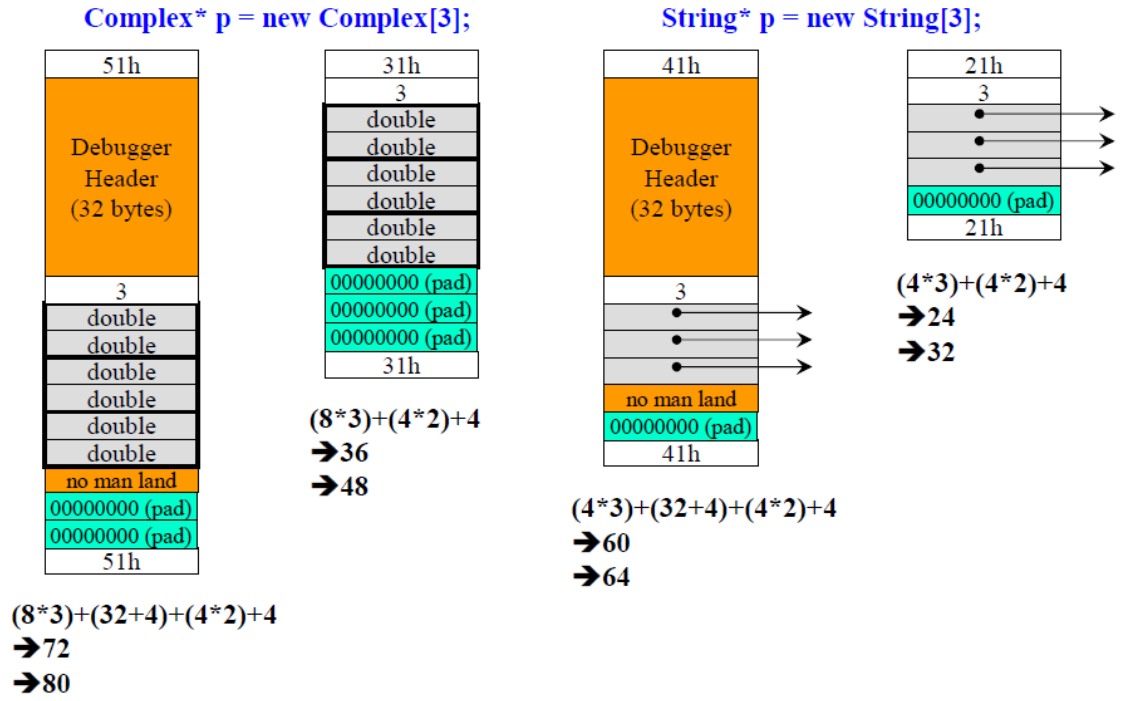
（五）动态分配所得的内存块(memory block), in VC

这部分侯捷老师深入了编译底层进行了讲解，视频中侯捷老师提到：目前市面上的书籍资料都没有如此详细的对内存块的剖析。所以如果想了解，请移步原视频观看。

動態分配所得的內存塊 (memory block), in VC



動態分配所得的 array



8.String类的完整实现

VS编译器在这里可能会遇到一个问题：

```
' strcpy': This function or variable may be unsafe. Consider using strcpy_s instead. To disable deprecation, use_CRT_SECURE_NO_WARNINGS. See online help for details.
```

这是在使用头文件#include中的strcpy()和strcat()函数时出现了一个错误(ctime函数也会报这个错)

可能的原因：因为这些C库函数很多没有内部检查，微软担心这些函数可能造成栈溢出，所以改写了这些函数，并在原来的函数名字后加上_s以和C库函数区分。

解决办法：找到**项目属性**，点击**C/C++**里的**预处理器**，对预处理器进行编辑，在里面加入：`_CRT_SECURE_NO_WARNINGS`即可解决问题。

原代码如下：

string.h

```
#pragma once
#include <cstring>
#include <iostream>
class String {
public:
    String(const char* cstr = 0);
    String(const String& str);
    String& operator=(const String& str);
    ~String();
    char* get_c_str() const { return m_data; }
private:
    char* m_data;
};

inline
String::String(const char* cstr)
{
    if (cstr) {
        m_data = new char[strlen(cstr) + 1];
        strcpy(m_data, cstr);
    }
    else {
        m_data = new char[1];
        *m_data = '\0';
    }
}

inline
String::~String()
{
    delete[] m_data;
}

inline
String& String::operator= (const String& str)
{
    if (this == &str)
        return *this;

    delete[] m_data;
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
    return *this;
}

inline
```

```
String::String(const String& str)
{
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
}

std::ostream& operator << (std::ostream& os, const String& str)
{
    os << str.get_c_str();
    return os;
}
```

string_text.cpp

```
#include "string.h"
#include <iostream>

using namespace std;

int main()
{
  String s1("hello");
  String s2("world");

  String s3(s2);
  cout << s3 << endl;

  s3 = s1;
  cout << s3 << endl;
  cout << s2 << endl;
  cout << s1 << endl;
}
```

9.拓展补充：staic、类模板、函数模板及其他

（一）static

- 静态成员函数只能操作静态数据。
- 静态成员函数**没有this指针。**
- **静态数据一定要在类外进行定义。**
- 调用静态函数的方法有两种：（1）通过对象调用；（2）通过类名调用。

```
class Account {
public:
    static double m_rate;
    static void set_rate(const double& x) { m_rate = x; }
};
double Account::m_rate = 8.0;

int main() {
  Account::set_rate(5.0);

  Account a;
  a.set_rate(7.0);
}
```

調用 static 函數的方式有二：
(1) 通過 object 調用
(2) 通過 class name 調用

除此之外，还需要知晓：**一个函数中static的东西，只有当该静态的东西被调用的时候，它才会被创建**，且离开该函数作用域后它依然存在。（下面的单例模式中，就用到了这一点。）

（二）把构造函数放到private区域

Meyers Singleton 单例模式

```
class A {
public:
  static A& getInstance();
  setup() { ... }
private:
  A(); (默认)构造函数放置private区，意味着外界无法创建A对象。
  A(const A& rhs);
  ...
};

A& A::getInstance()
{
  static A a; 将唯一的静态对象放置类外的接口函数中，这
  return a;   样的好处在于：没人调用它，它就不创建。
}
```

该静态函数是外界的唯一接口，外界只能通过该函数得到唯一的实例对象A a。

```
A::getInstance().setup();
```

相当于：a.setup();

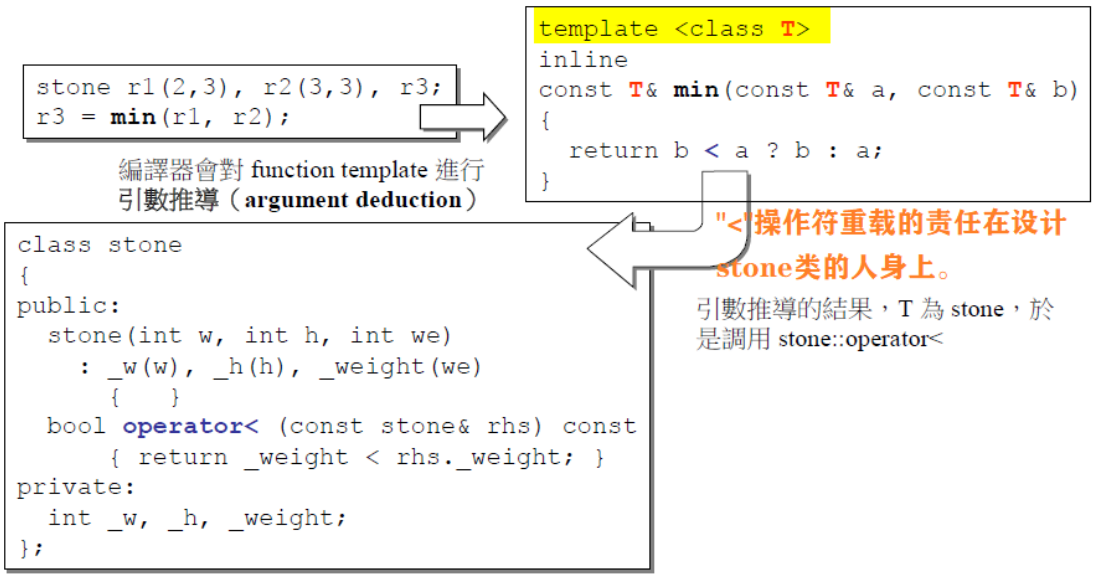
(三) cout

(四) class template类模板

```
template<typename T>
class complex
{
public:
  complex (T r = 0, T i = 0)
    : re (r), im (i)
  { }
  complex& operator += (const complex&);
  T real () const { return re; }
  T imag () const { return im; }
private:
  T re, im;

  friend complex& __doapl (complex*, const complex&);
};
```

```
{
  complex<double> c1(2.5,1.5);
  complex<int> c2(2,6);
  ...
}
```

（五）函数模板



Ⅱ：Object Oriented（面向对象）：面对的是多重classes的设计

即：**Object Oriented Programming, Object Oriented Design**（OOP，OOD）

面对对象编程中，牢固掌握以下"三把大刀"就足够应付绝大多数情况。

- **Inheritance (继承)** —— is - a
- **Composition (复合)** —— has-a
- **Delegation (委託)**

10.复合、委托与继承

(一) Composition (复合)，表示has-a

(1) Adapter(改造)：A类若复合B类，如果有需要，则A类可以使用B类中的东西进行改造。

```
template <class T>
class queue {
  ...
protected:
  deque<T> c;      // 底層容器
public:
  // 以下完全利用 c 的操作函數完成
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference front() { return c.front(); }
  reference back() { return c.back(); }
  //
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_front(); }
};
```

queue → deque

Adapter

(2) 一个复合类的大小 = 该类数据大小 + 该类中复合类的大小

Sizeof : 40

```
template <class T>
class queue {
protected:
  deque<T> c;
...
};
```

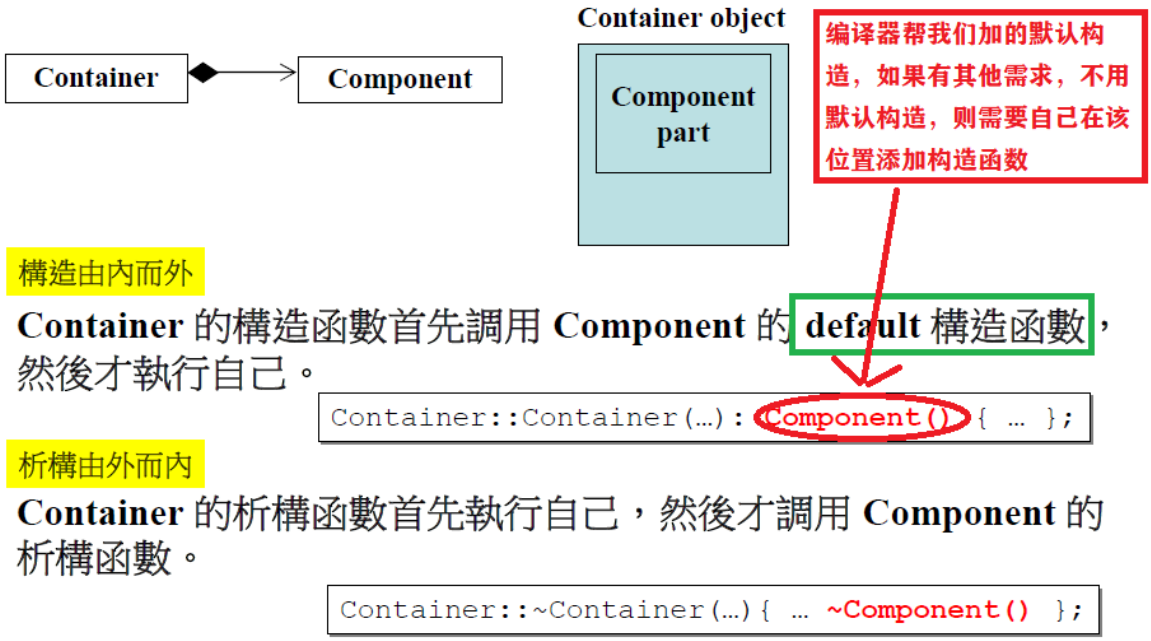
Sizeof : 16 * 2 + 4 + 4

```
template <class T>
class deque {
protected:
  Itr<T> start;
  Itr<T> finish;
  T**    map;
  unsigned int map_size;
};
```

Sizeof : 4 * 4

```
template <class T>
struct Itr {
  T*  cur;
  T*  first;
  T*  last;
  T** node;
...
};
```

(3) Composition**复合关系下的构造和析构**



(4) 在**复合**中，类和其复合的类是**同时创建**的。

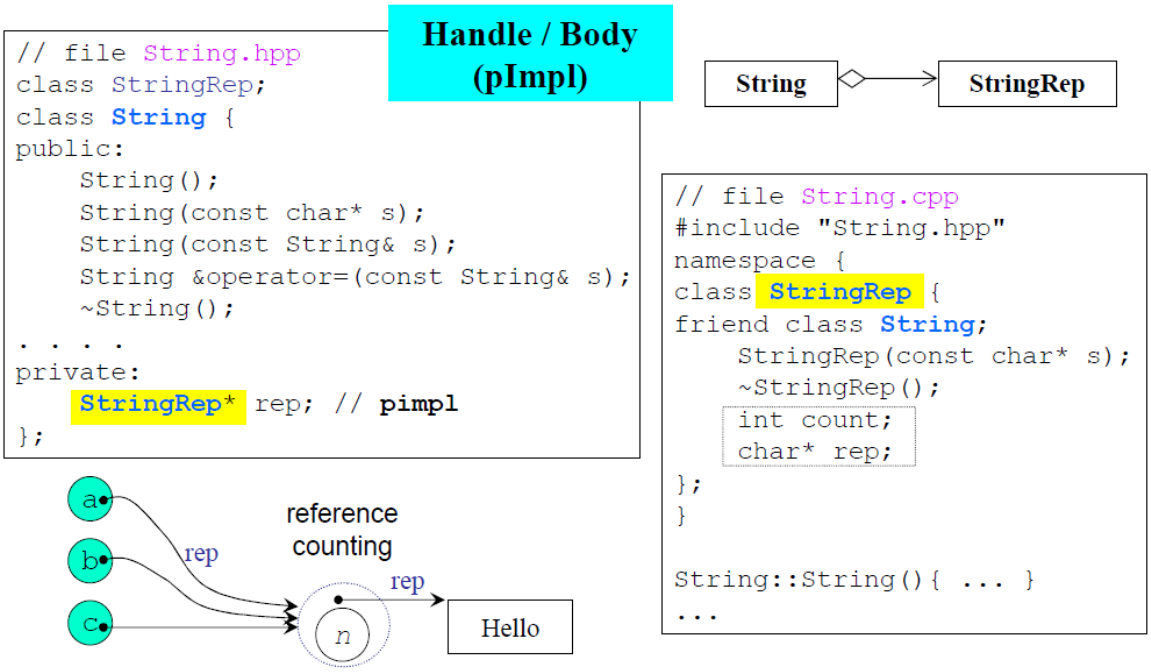
(二) Delegation(委托)，或者称作：Composition by reference

通俗的讲，委托就是我拜托/委托别的类，来帮助我实现一些东西。我只创建一个指针，指向我委托的那个类，让我的功能，都在我委托的那个类中实现。

委托其实和复合的功能很像，其实这就是对不同的实现分配到不同术语，你只需简单的记住，A类内含一个指针指向另一个类B(该类中实现了A的功能)就可以称作委托。

还有一点要注意，委托和复合他们中的类创建的时间不一样：在**复合**中，类和其复合的类是**同时创建**的；而在**委托**中，我委托的那个类创建的时间我不清楚，反正一定比A类晚，即**不同步创建**。

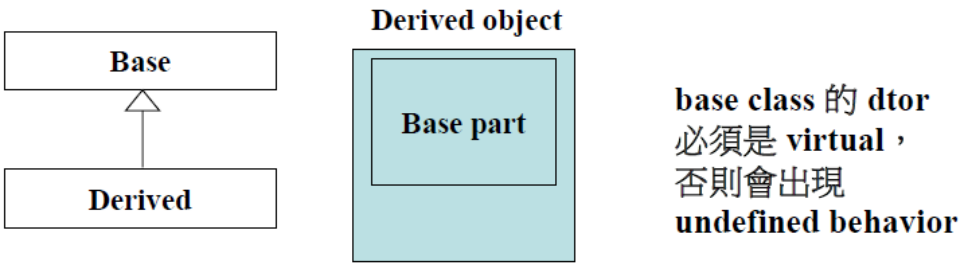
Handle/Body (pimpl) 手法，其实就是一种委托，在Handle中创建一个指针作为接口指向另一个类Body，在Body中实现Handle的功能。



(三) Inheritance (继承), 表示is-a

不同于其他语言，C++中的继承除了public继承外，还有private、protect继承，其中public最重要，用的最多。

Inheritance继承关系下的构造和析构，其实和复合关系下的构造和析构很像：



構造由內而外

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

```
Derived::Derived(…): Base() { … };
```

析構由外而內

Derived 的析構函數首先執行自己，然後才調用 Base 的析構函數。

```
Derived::~Derived(…){ … ~Base() };
```

11.最后：虚函数与多态、委托相关设计

这部分内容主要讲了一些设计模式的内容，即如何利用复合、委托、继承设计出一个良好的类。建议看一下原视频，这里就不再进行叙述。

- non-virtual 函数：你不希望derived class 重新定义(override, 覆写) 它.
- virtual 函数：你希望derived class 重新定义(override, 覆写) 它，且你對它已有默认定义。
- pure virtual 函数：你希望derived class 一定要重新定义(override 覆写)
  它，你对它沒有默认定义。