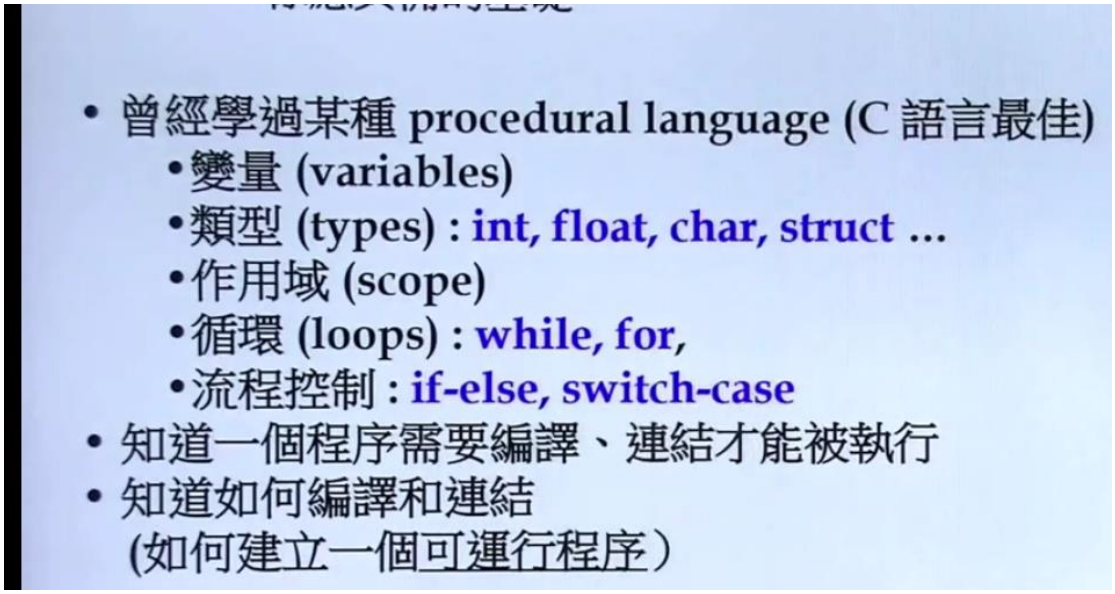
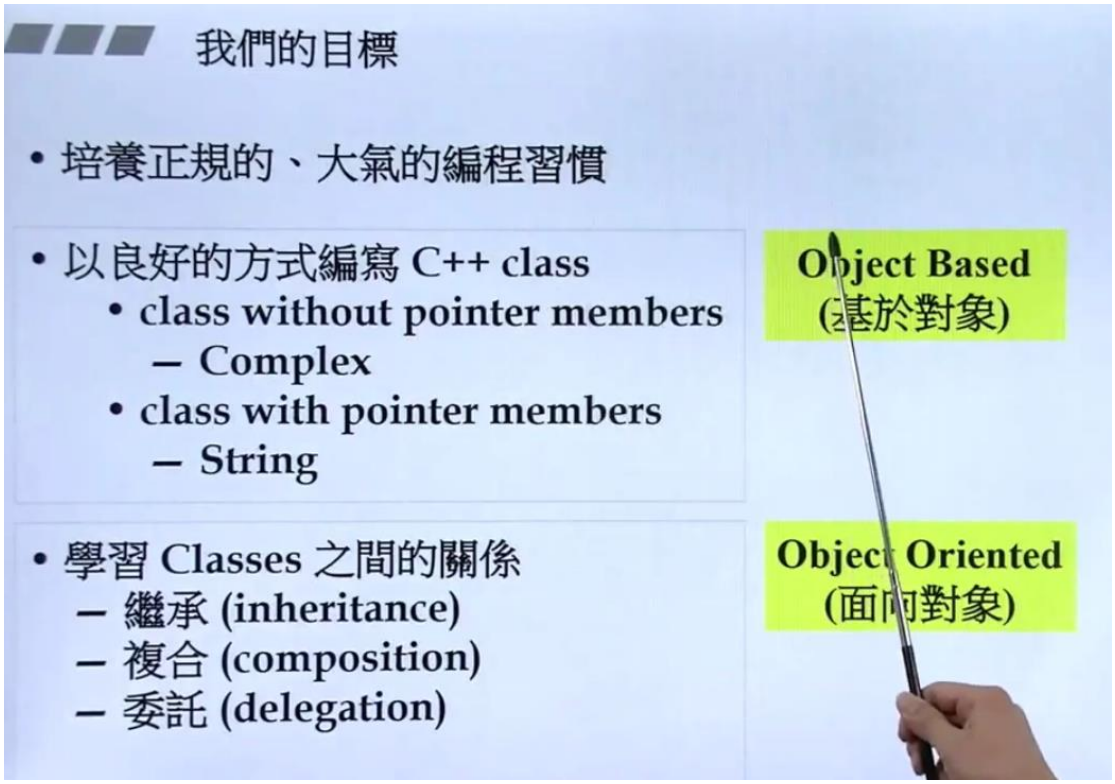
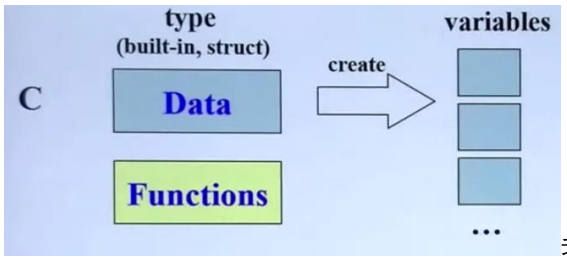
C++面向对象程序设计

基础:

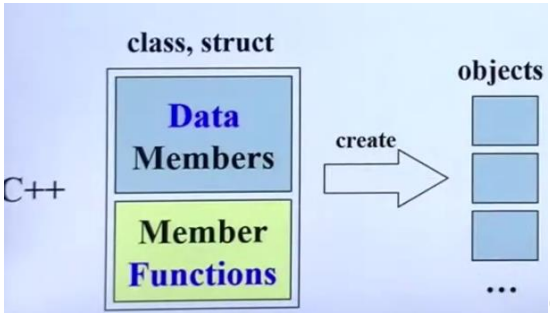




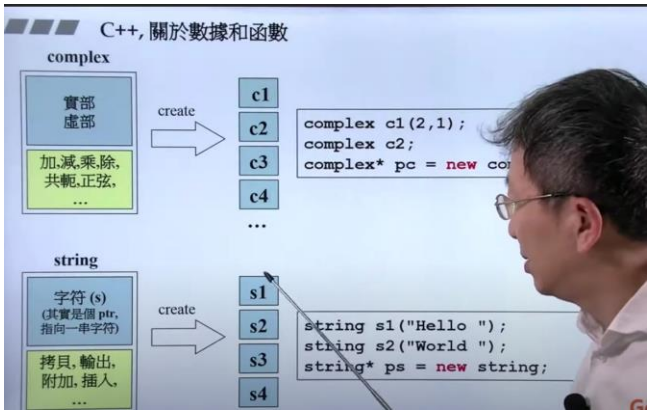
头文件与类的声明



我们通过数据类型(data type)来创建变量，并且调用函数(Functions)来使用这些变量，但是由于 C 语言没有足够的关键字来限制变量的使用范围，所有变量都是全局的，因此所有函数都可以使用它们。

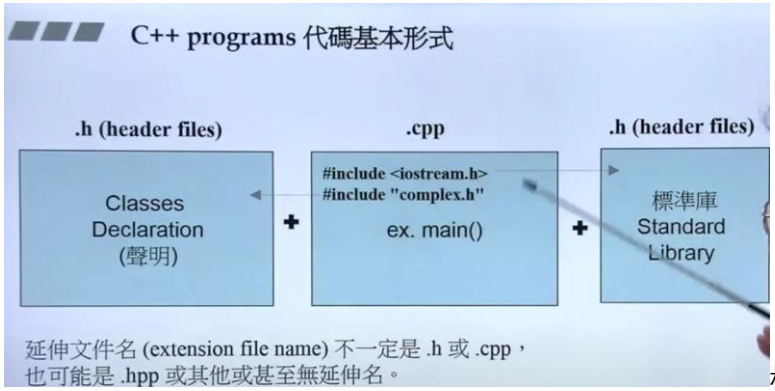


CPP 将数据和函数包在一起，只有一个类内的函数可以使用这些数据，我们想要使用这些类，就要以这种类型去创建对象。

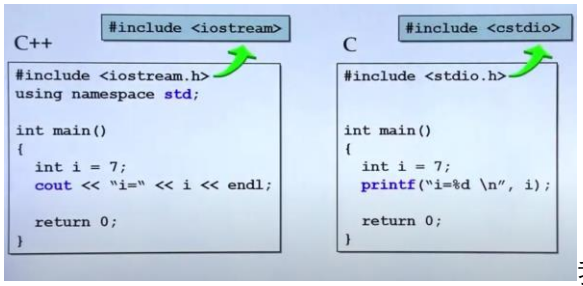


CPP 典型的数据类型分为有指针的和无指针的，分别以 complex(复数)和 string（字符串）为例。通过武指针创建的数据类型，内存内就有这个数，而有指针的内存存储的是一个指针（一个地址），大小相同。

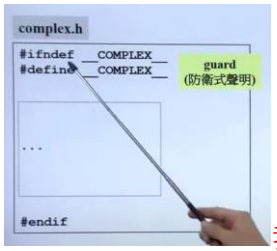
CPP 代码的基本形式



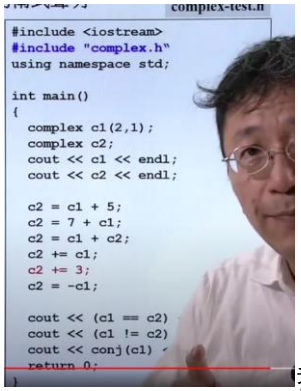
标准库以头文件.h 的形式呈现，以 include <>的形式引用，而且不必在意其位置。如果要引用自己编写的头文件，则要以 include “”的形式引用，而且我们需要知道其具体的位置才能成功引用。



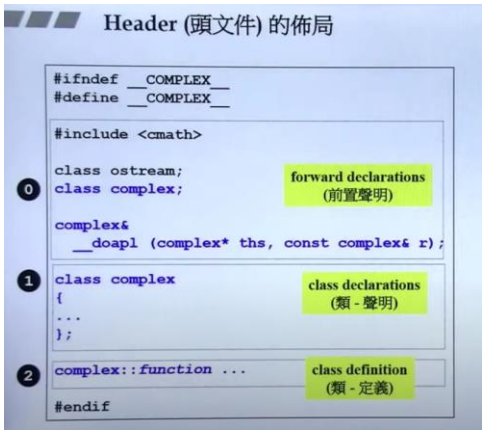
我们引用头文件 iostream.h（在一些平台上可以去掉.h 后缀），使用 std::cout 用于输出。

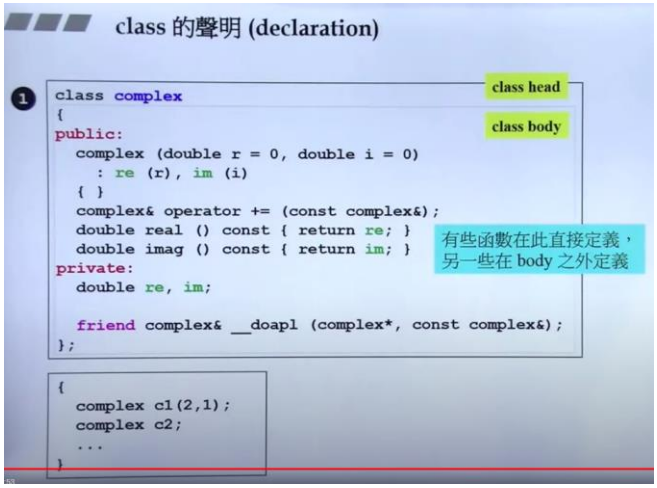


我们使用防卫式(guard)的声明，以便于引用者不必按照一定顺序来声明头文件，我们在头文件中写明(#ifndef _COMPLEX_)—如果没有定义过 COMPLEX，则将其定义出来(#define _COMPLEX_)。如果我们第一次引用该头文件，则我们会将其添加进来，而如果我们第二次引用该头文件，就因为 ifdef 判断不进入头文件，避免了重复引用。



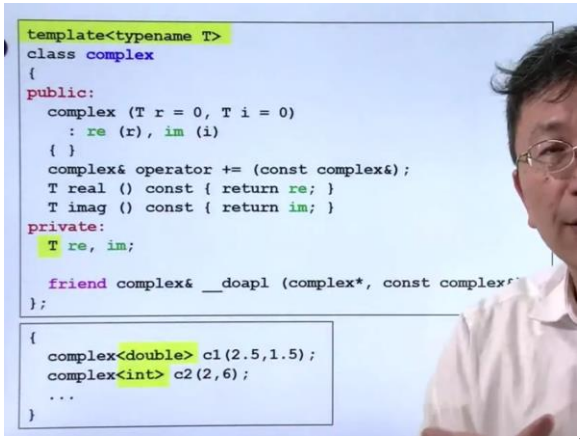
我们在测试文件中使用的方法和数据类型都是要在头文件中实现的。



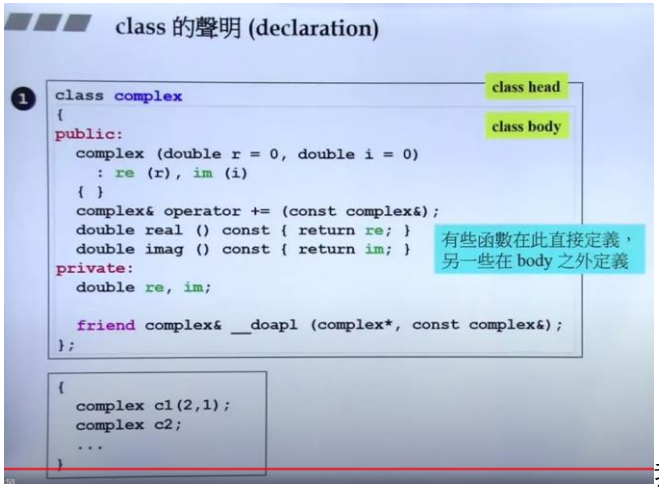


一个类要有 class head 和 class body，声明类的结构大概是如此的。

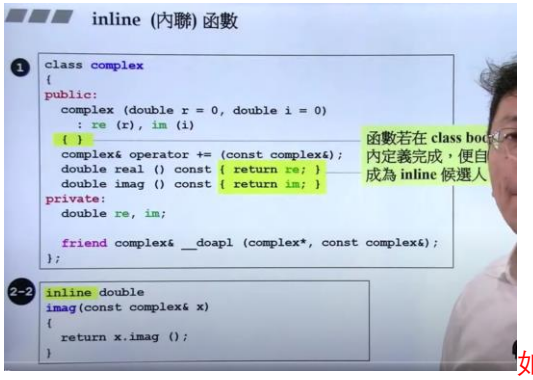
我们要定义一个复数的类，首先我们要声明该复数的实部和虚部，如果上面的 double re,im 我们将其声明为 double 值。而如果我们想要将 re 和 im 声明为 float 或者 int 类型时，我们就要只改变其类型，保留其他部分，重新声明一个类，这样是不合理的—我们想要声明不同类型的实部虚部就要去引入别的类。



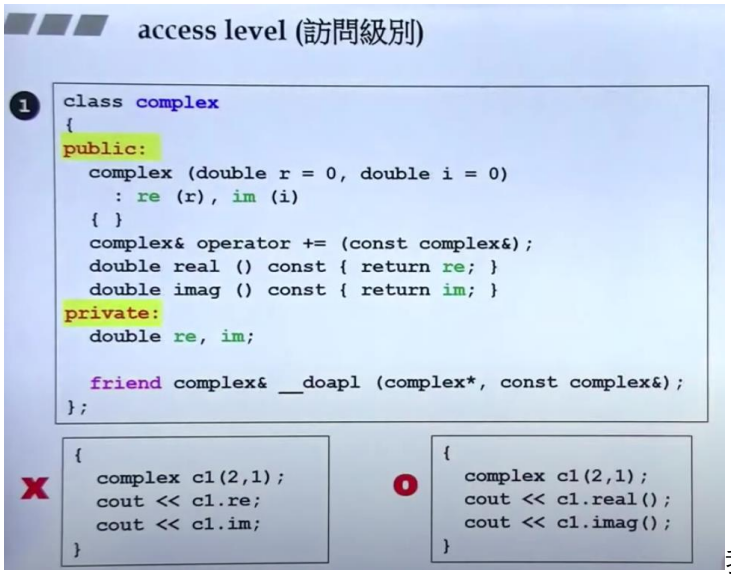
为解决这种需求，CPP 引入了 template<typename t>模板类型。我们想要不把类的变量的类型写死，而在用的时候再指定。



我们可以看到，在 complex 类里面，double real ()后有大括号，说明该方法在此声明并定义，而 complex&这一句后面没有大括号，则这个方法仅在此声明，而在 body 之外定义。



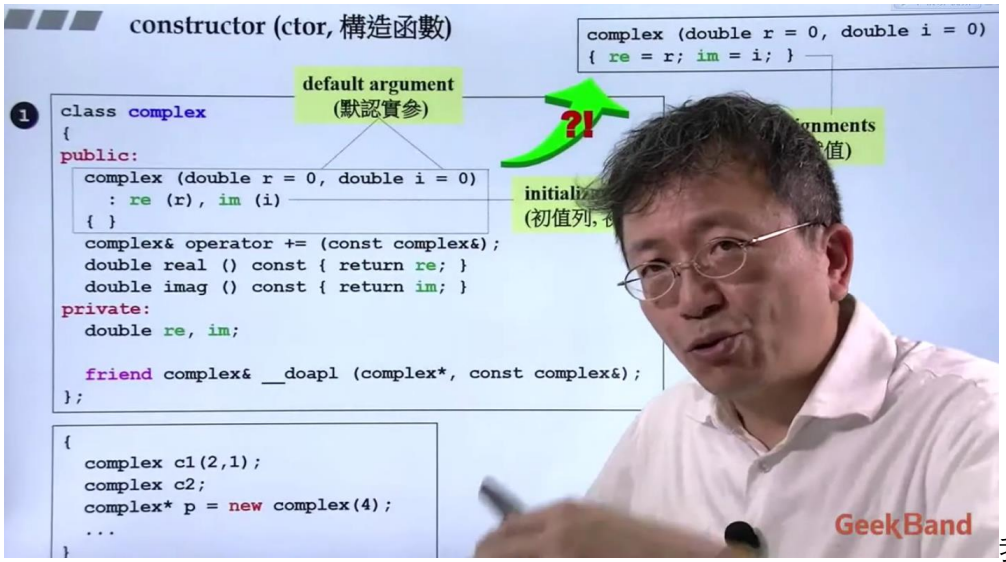
如果函数在 class body 内定义完成，则就称为 inline，而如果不是在 body 内定义，就不是 inline。如果函数是 inline function ,则函数运行较快。但是有些 function,即使写在 body 内，编译器也不会将其视为 inline function ,简单的将其分类就是函数的复杂程度。我们也可以为 body 外的 function 添加 inline 属性，而这样做之后，是否能够成为 inline function，还是要看编译器。



我们一般会设定类内的各个方法和变量的访问等级，如果添加了 private 属性，我们就没办法轻易的获取它，但是我们可以写出 public 的方法来暴露它。Public 和 private 是可以间接使用的，而不用聚集成两部

分。 数据都要放在 private 中。

构造函数



我们在创建对象的时候，会自动的调用构造函数，如上图的下方方框，创建一个有参数的对象、创建一个无参数的对象、创建一个可以修改的对象，其实都是在调用.h 的构造函数。

（1） 构造函数的名称必须要跟类的名称相同

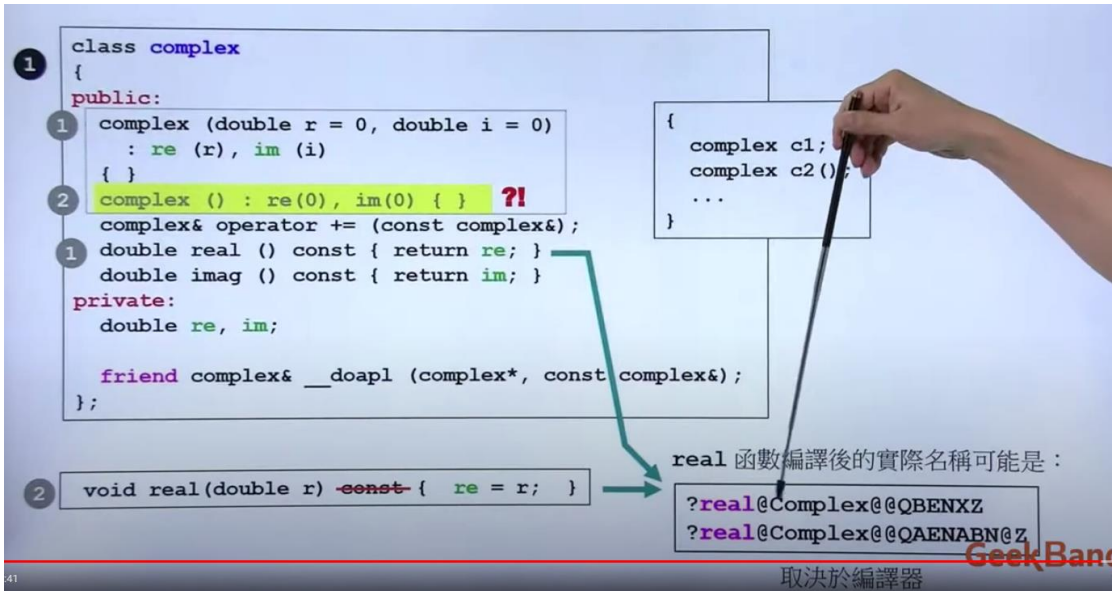
（2） 构造函数可以拥有参数

（3） 参数是可以有默认值(default argument)的，如果是第二种情况，没有指明参数，则就使用默认值作为参数值。这种默认值的特性其他函数也可以使用。

（4） 构造函数没有返回值类型

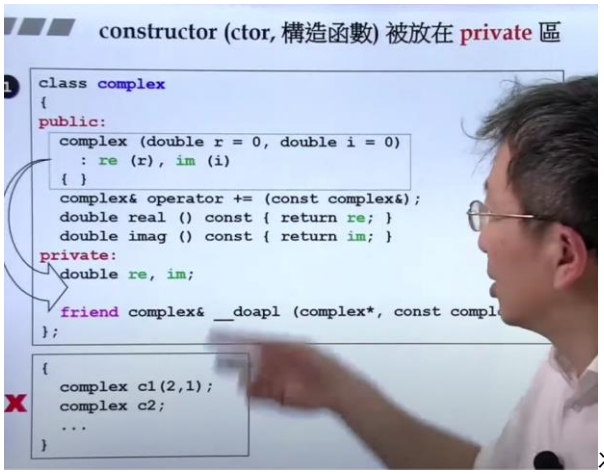
（5） 我们不必在构造函数的大括号内将传入的参数给到变量，而是使用初值列\初始列（initialization list）。那为何采用这种方法？一个变量的数值设定有两个阶段---初始化和赋值。而初值列的设定就是初始化的阶段，如果放到大括号内去赋值，则代表了我们放弃了初始化的阶段，则效率变得更低了。因此虽然结论相同，但过程是不一样的。

不带指针的类多半不用写析构函数，

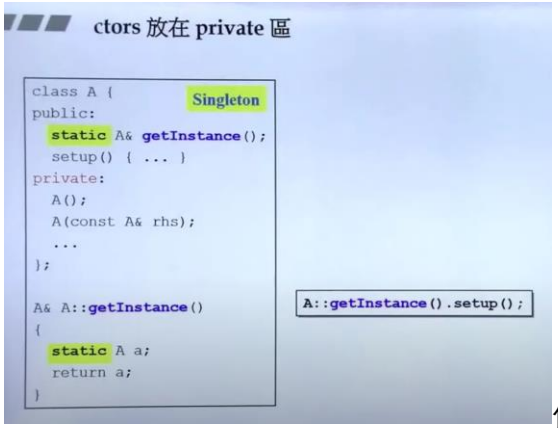


相同函数名称的函数有一个以上，就成为 overloading (重载)。上图中的第二组重载是可行的。

但是对于第一组重载，在创建没有参数的对像时，第一个函数和第二个函数都是能够被调用的，这就会产生编译错误，因此这样的在某种情况下起到相同作用的重载是不允许的。

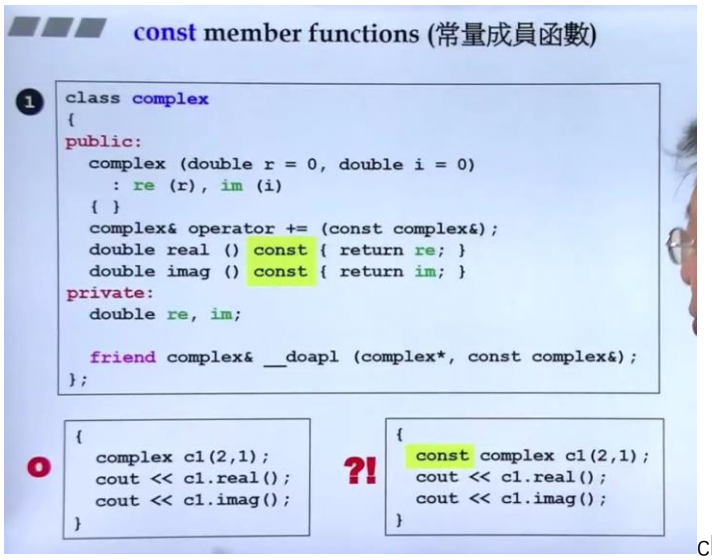


将构造函数设为 private，则不可以再通过该函数来创建对象。



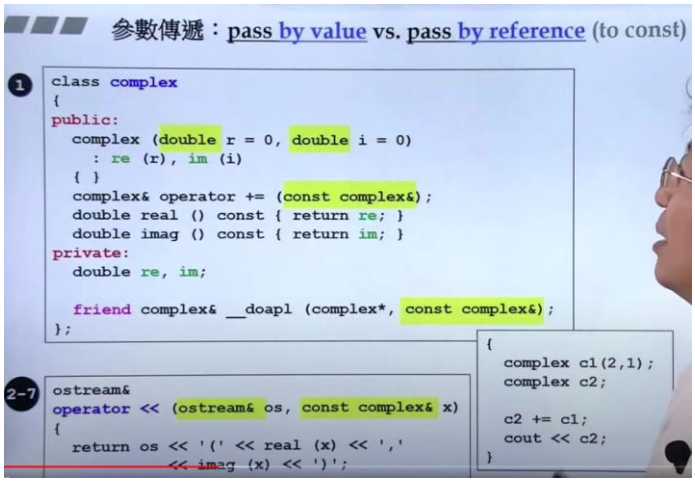
但是 singleton 的设计方法使用了 private 的构造函数，我们在 A 的一个方法里面创建对象，然后我们在外部通过这个方法来创建对象，也

就是说，确实是有需求将构造函数放在 private 内的。



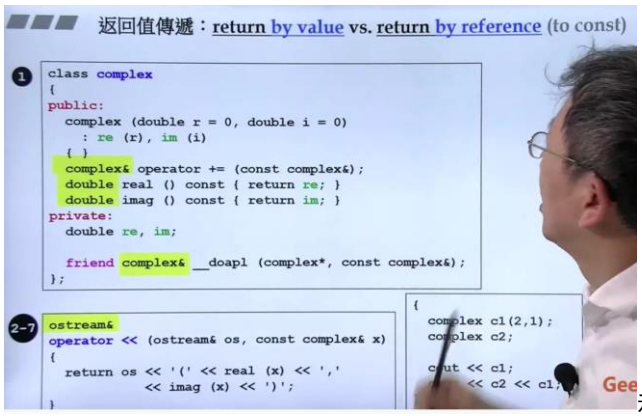
class 里面的函数分为会改变对象数据的和不会改变对象数据的两种。不会改变对象数据的，我们应该加上 const。

在对象和变量前加 const 前缀，就代表这个对象或变量不会是常量，不会发生改变，如果我们再调用这个对象的方法，但是调用的方法没有 const，则就会编译出错。使用者的本意是创建一个常量的对象，但是调用的方法(无 const)却有可能会改变这个对象，这是错误的，会出错的。但是创建无 const 的对象，是可以调用有 const 的方法的。

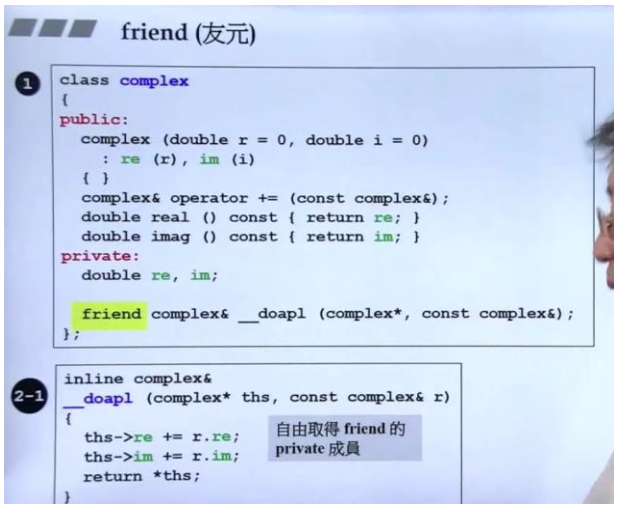


其中 complex 构造函数的参数是 pass by value，参数无特殊符号，而 complex&则是 pass by reference，参数有&特殊符号。Pass by value 就是将这个数据都传送过去，数据又多大，那就传送的多大。由于传送数据的大小是不确定的，有变化的，我们尽量使用引用（底层还是指针）传送数据。

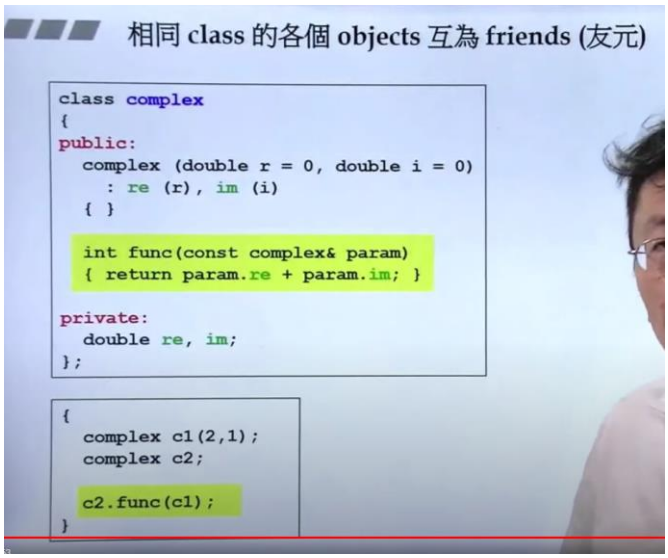
如下例中，complex 使用的是 pass by value，而下面的加法使用的是 pass by reference，速度更快。传递引用会导致使用引用改变原本的被引用值，有时候会导致我们不想要的结果，为避免这种情况，我们使用 const 来修饰引用，如果强行修改被 const 修饰的引用，编译就会出错。



和参数的传递一样，返回值的传递尽量也是 reference 的形式。



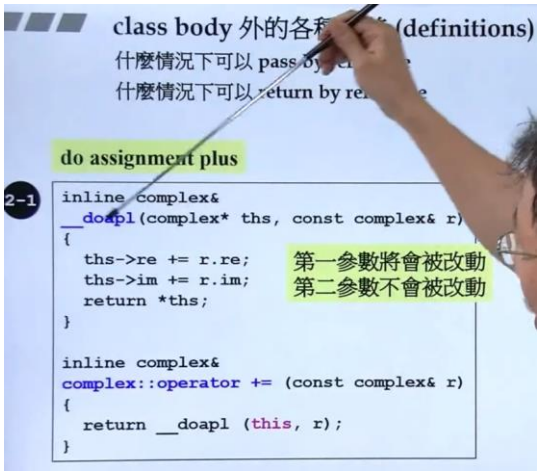
通过 friend 修饰符，我们定义的方法可以获取 private 的变量，如上图下例，_doapl 可以获取 ths 的 re 和 im，并且可以将其赋给另外一个 complex 对象的 re 和 im，而不用通过方法去获取。



我们可以再 complex 内定义一个 function 来获取实部和虚部的和，即使没有 friend 修饰，这样的话，我们可以通过 complex 创建的对象来获取 3 个值—实部、虚部、实部虚部和。这样似乎是打破了封装，但是其实并没有，我们可以这么认为---相同 class 内的各个 object 互为 friends，其实这个 function 和

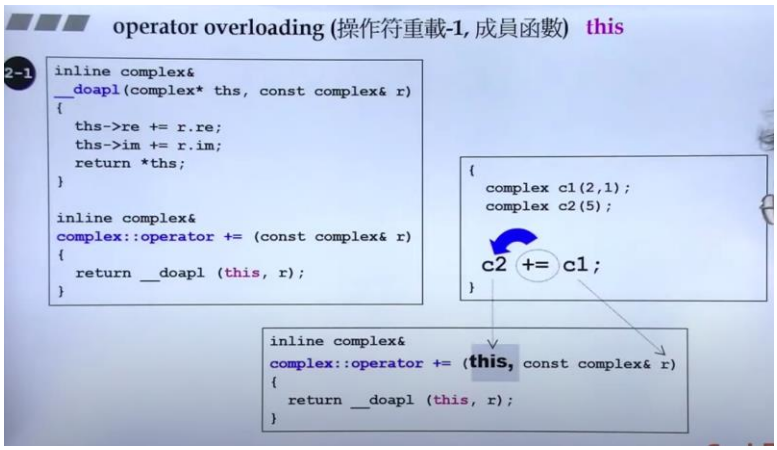
前面的获取实部虚部的方法性质是一样的。
做一下总结 ：

（1） 要写防卫式的声明
（2） 数据要用 private 修饰
（3） 对于不修改数据的方法和不能被修改的变量要写 const
（4） 返回值和参数尽量要 pass by reference 和 return by reference
（5） 构造函数要以初值列的方式设定参数值



来看这个方法体， ths.re+=r.re-----=ths.re=r.re+ths.re。因此第一参数 ths 会被改变，不用 const 修饰。而第二参数 r 实际上不会发生任何改变，因此我们用 const 去修饰。那最后的 retun 会返回到哪里？以上面的方法为例，返回的空间是已有的，我们将其返回到第一参数上。
而如果，是 ths.re+r.re，这两个对象的值都没有发生变化，那我们最后的结果就要有一个新的对象来承载。
也就是分两种情况，首先是必须在方法内开辟新对象来承载方法的结果的；其次是可以用已有的对象来承载结果的，如上面的例子。
如果是第一种情况，那方法返回的可能就是在方法内创建的新的对象，而这个对象在函数结束后就消失了，这就称为 local 的对象，这时候，我们就不能够返回 reference,因此此时 reference 指向的对象已经消失了，得不到我们想要的结果。

操作符重载



（成员函数—类内的函数—只有成员函数才有 this）这里方法参数里的 this 必须被隐藏，实现的功能是一样的，this 指代的就是调用这个函数的对象。即使 this 被隐藏，在方法体内还是可以使用 this。调用了+=之后，方法体再调用_doapl，实现复数的加法，但是在_doapl 内的+=应该如何实现？不会产生死循环吗？仔细看就可以知道，方法体内的+=的参数不再是复数了，也就是说，它的+=不再是调用参数为复数的+=了，这当然是另外的+=的重载。

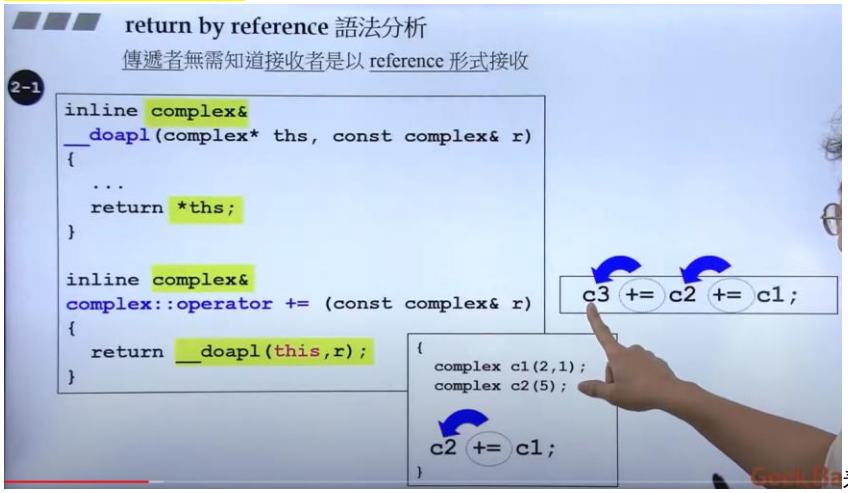
取地址运算符 &

& 是一元运算符，返回操作数的内存地址。例如，如果 var 是一个整型变量，则 &var 是它的地址。该运算符与其他一元运算符具有相同的优先级，在运算时它是从右向左顺序进行的。

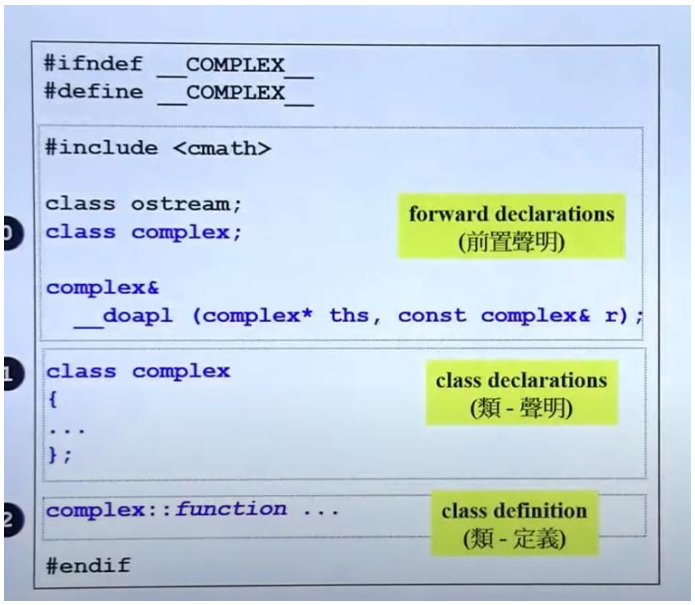
您可以把 & 运算符读作**"取地址运算符"**，这意味着，**&var** 读作"var 的地址"。

间接寻址运算符 *

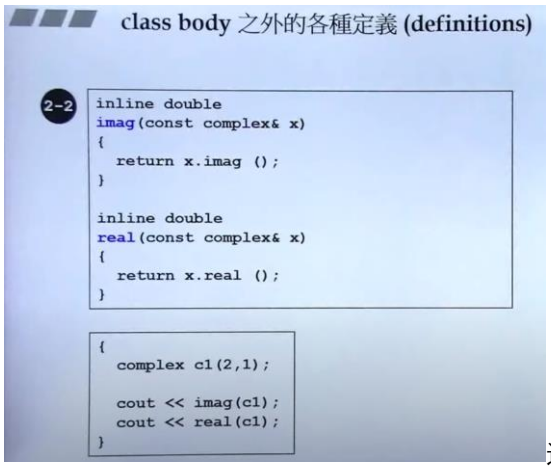
第二个运算符是间接寻址运算符 *，它是 & 运算符的补充。* 是一元运算符，返回操作数所指定地址的变量的值。



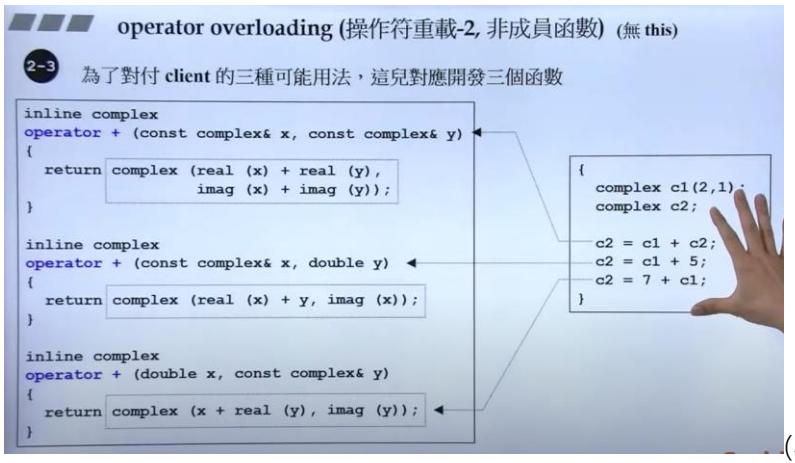
看下面的 c2 += c1 的例子。我们可以看到，我们操作的并非是 reference，而是对象，但是—传递者（使用者）无需知道接收者（被使用者）是以 reference 形式接收的，也就是说---使用者无所谓调用的方法的参数是 reference 还是 value，我们只需要一种形式的操作。同样的，return 的对象也无需判别方法体的返回值是 value 还是 reference。我们这样的操作，其实没有使用到返回值，也就是说，这时候+=的返回值是 void 也是没问题的，但是如果是上面的使用方法呢？这时候我们 c3 需要 c2+=c1 产生的对象，这时候，我们就必须用到返回值了。



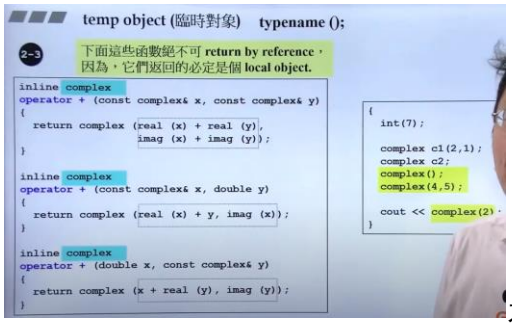
如果是 class 的 function,则需要 classname::function 来声明，只声明 function 则是 global function。



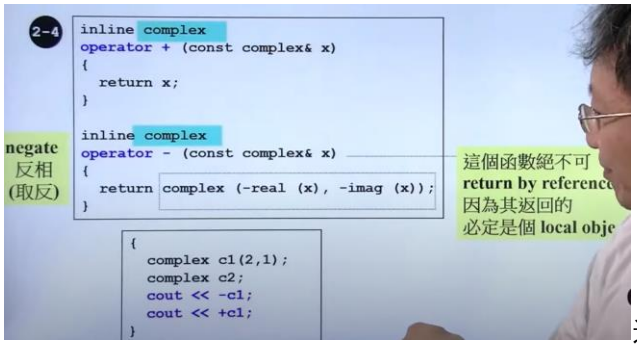
这里的两个方法体实现的方法很简单，我们使用 inline 来尝试加快它的速度，并且使用&来提高其效率。



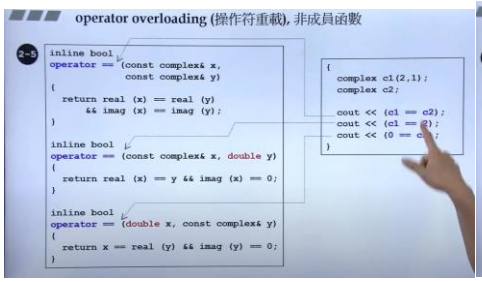
(非成员函数—class 外的方法—无 this)。此时是全局函数，无 this pointer。

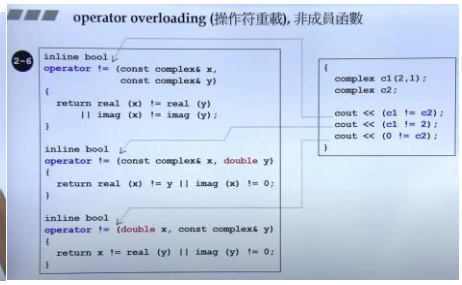


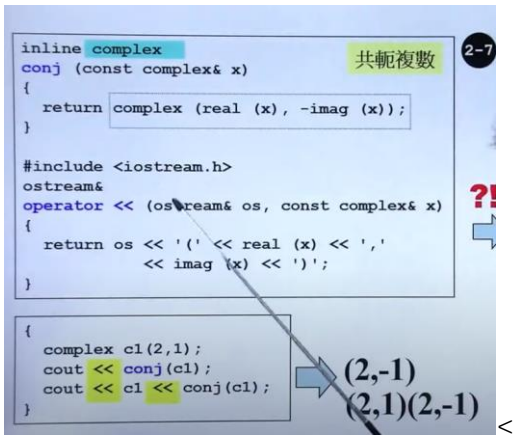
为何此时返回的是 value，而非 reference？这里的和之前不同的是—我们之前的是有能够承载结果的对象的，而现在，我们没有承载结果的对象，所以我们必须返回一个对象，用于盛放结果，相应的，我们也在方法体内看到—我们创建了一个 complex 对象，并且把结果的值赋给了它，如果此时我们传递 reference 出去，则在方法体执行完毕后，对象死亡，reference 就无法指定到这个对象了。我们这里用到了临时对象----用 typename ();来声明，这个对象不用给变量名，在像这样的情况下使用临对象是一个好选择。看右边的例子—我们在黄色的部分声明了临时对象，他们的生命周期在声明完毕后就结束了。临时对象绝对不能 return by reference



这里的+和-是代表正号和负号，那程序怎么分辨它们和加减呢？这其实很好理解，只要是一个参数的，那就是正负号。取反创建了一个新的对象而正号并没有，因此取反需要创建临时对象(temp object)，而正号则不必。







<<比较特殊，他只能写成全局的方法，第二个参数

不需要更改，我们将其用 const 修饰，而第一个参数不能添加 const，虽然看起来 cout 没有发生改变，但是其实在方法体内每次<<都改变了它的状态。 C++的操作符号都作用在左边。看第一种输出似乎不需要返回值，我们可以将返回值设为 void，但是如果是第二种输出，我们就需要返回值，首先运行左边的<<，通过返回值得到一个新的 ostream 的对象，然后再运行右边的<<，进行新的输出。

对没有指针的 class 做一个总结：

（1） 要使用初值列的方法来设定构造函数的参数

（2） 要考虑方法是否改变参数的值，决定是否要加 const

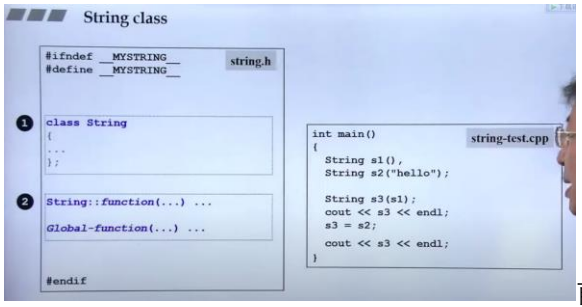
（3） 参数的传递尽量考虑 pass by reference,并且考虑参数是否会被改变，决定要不要加 const

（4） 数据要放在 private 中，除了 class 内部调用的不展示给外部的函数，函数一般使用 public

示例

见项目

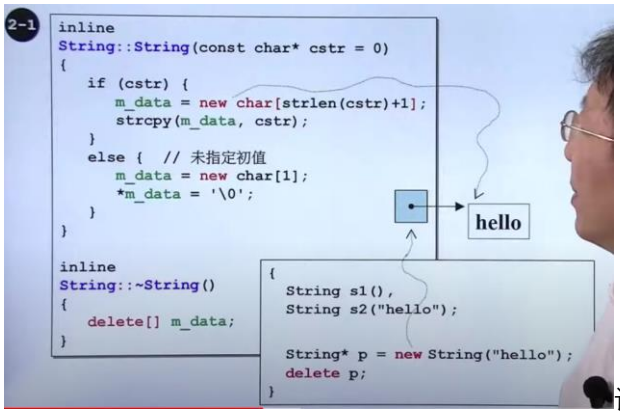
String Class



前面的 Complex 类虽然没有构建拷贝赋值和拷贝构造的函数，但是编译器会默认创建一个这样的动作，该动作会一位一位的拷贝对象，对于复数这样的数值型的对象是足够的，但是在 String 这样带有指针的类的操作有可能达不到我们想要的效果。

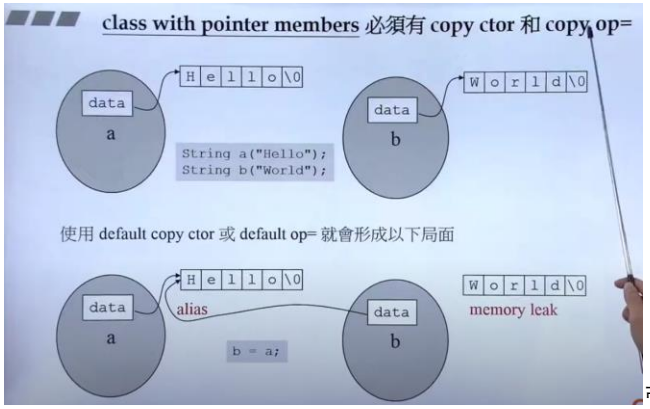
我们来看左边的第二个构造函数，我们可以看到，

这个构造函数接受的参数类型是 String 类型的指针，是他自己本身的类型，我们称这样的构造函数为拷贝构造；再看第三个构造函数，我们可以看到该函数接受的参数也是 String 类型的指针，我们称这种函数为拷贝赋值，只要我们的类带有指针，就一定要写出这两种类型的函数。第四个函数体~String()称析构函数，当对象死亡的时候，析构函数就会被调用起来。我们称这三个函数---拷贝构造，拷贝赋值，析构函数为 big three，三个特殊函数。



设定字符串的方法有两种，将最后的一位设为 0，称为结束符号；或是最后不加 0，但是第一位就是代表字符串数组长度的值。

看上图，如果我们如下面第二种方法来创建对象，传入了一个指针，我们就调用相应的构造函数来创建对象，如果传入的指针的长度为 0，则不通过 if 检测进入 else，即使是 cstr 的长度为 0，我们也要创建一个长度为 1 的字符数组，用于存放结束符 0，这样我们就创建了一个空字符串。如果传入的 cstr 长度不为 0，程序进入 if，则我们创建一个长度为（cstr 长度+1）的字符数组，+1 的最后一位用于放结束符。然后我们将创建的数组拷贝到新创建的字符串对象上面去。没有指针的函数不需要带有析构函数，比如 complex 类就不需要释放内存。但是 String 这样带有指针的函数我们需要将指针不再指向的内存释放掉，否则会产生内存泄漏，实现这样的功能需要有析构函数~String()。我们使用 delete[]来删除这个对象。我们的方法内有指针，那我们多半要进行动态分配，带有指针的对象在死亡之前，析构函数会被调用，我们就要在析构函数内写出释放内存的操作用于释放动态分配的内存。

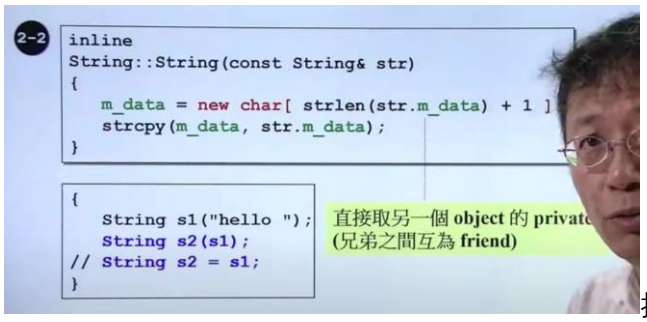


带有指针的 class 就必须有 copy ctor（拷贝构造）和 copy op=（拷贝赋值）

如上图所示，我们创建两个对象 a 和 b，就形成了上面的形式---a 和 b 里面的 data 其实只有一个指针，而指针指向的内存并不属于 a 和 b。

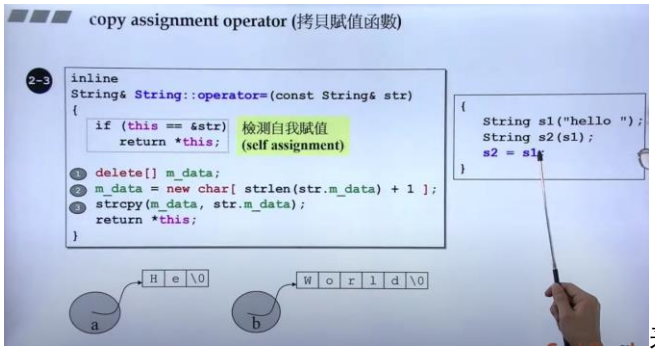
如果我们使用默认的 copy ctor 和 copy op=而不是自己重写，那我们将 b 赋值为 a 时，就会使得 b 内的 data 指针变成了 a 的 data 指针，两者指向了同一个内存区域，这样会导致两个问题---原来 b 指向的内存没有指针指向他了，也无法再有指针指向它了，这块内存是一个孤儿，造成了内存泄漏；a 和 b 指向同一块内存(这种情况称为别名，就是说一块内存有两个不同的指针，这两个指针都指向他)，如果我们通过 a 来更改这块内存的作用，那么指向它

的 b 也会受到影响，值也发生了改变了。这种 copy 称为浅拷贝。



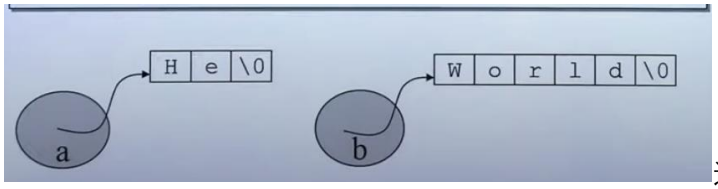
拷贝构造函数（拷贝—因为传入类型和方法类型是一样的；构造—方法的名称和类的名称相同）。

来看这个拷贝构造函数---我们传入 str 类型的指针，进入方法体之后，首先创建一个长度为 str 长度+1 的字符数组，然后我们将 str 的内容复制给当前对象的内容，经过这样的操作之后—b 指向的内存地址没有变化，但是内容发生了变化，也就是说，现在内存内有两块内容都是 hello 的内存了。这种操作我们称为深拷贝。在这里，str 可以直接获得 m_data，也就是直接取这个对象的 private data，这是因为兄弟之间互为友元。

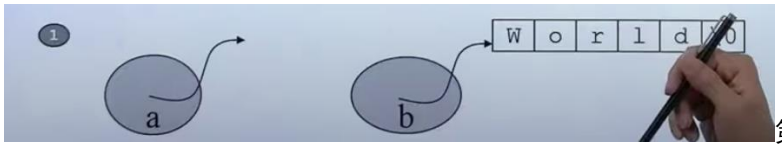


来看拷贝赋值函数。

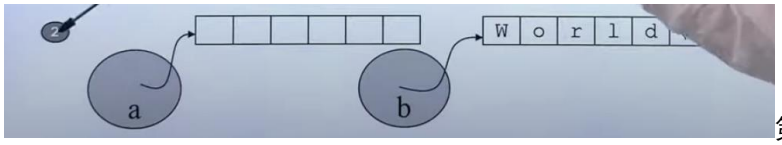
拷贝赋值函数的过程：先删除要改变内容的对象的内容，使其为空；再创建要复制的内容的长度+1 大小的字符数组；然后将被复制的值给到对象的字符数组，返回修改完毕的对象。



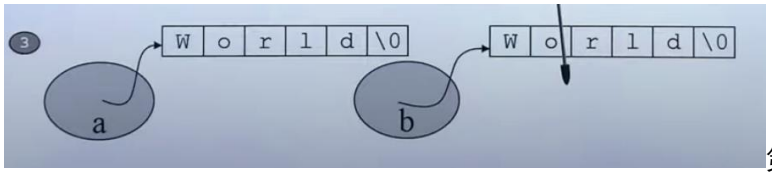
这是两个不同内容的对象，我们来还原一下拷贝赋值的过程。



第一步，删除内容

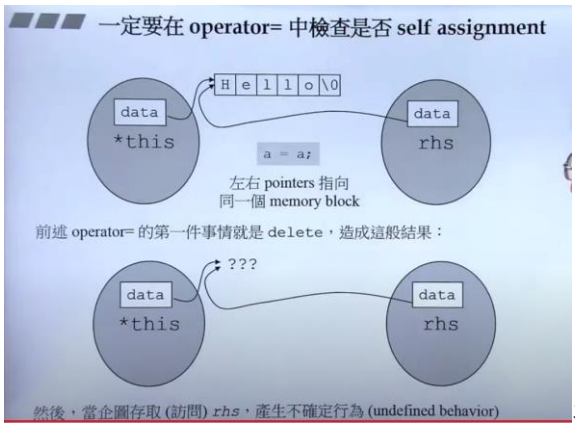


第二步，创建空数组

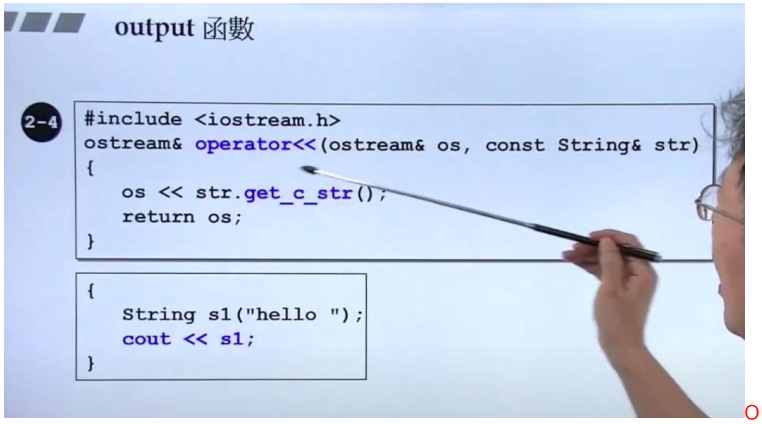


第三步，复制内容

在此之前，我们先做一个 self assignment，检测一下是否是自我赋值，如右边的例子，实际上 s2 和 s1 的内容是相同的（注意一下，经过前面的拷贝构造，这两个对象的指针是不同的），如果我们将 s1 的值给到 s2，再 self assignment 阶段，我们就能够检测出两者的内容相同，这样就可以不修改要操作的对象了，这样的可以提升效率。也可以避免特殊情况。

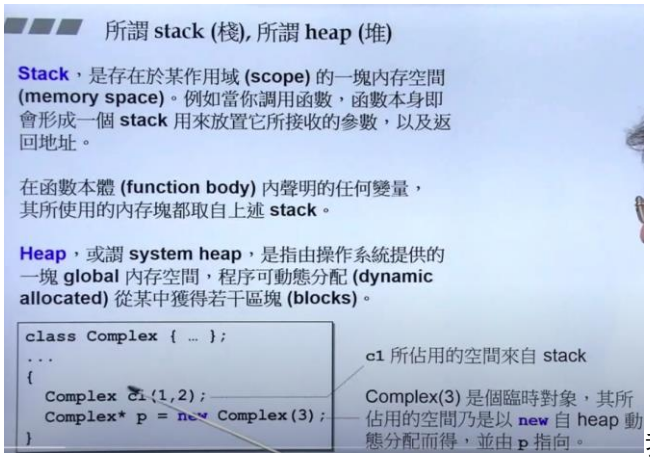


在这里，如果我们将 a 赋值给 a，这时候，传入拷贝赋值方法的两个参数 this 和 rhs 其实是同一个对象，如果没有自我赋值检测，我们执行第一步删除内容，下一步就会报错了，因为我们想要复制的内容被第一步删除了。这是不对的，自我赋值检测不仅仅是为了提升效率，也可以保证程序的鲁棒性。

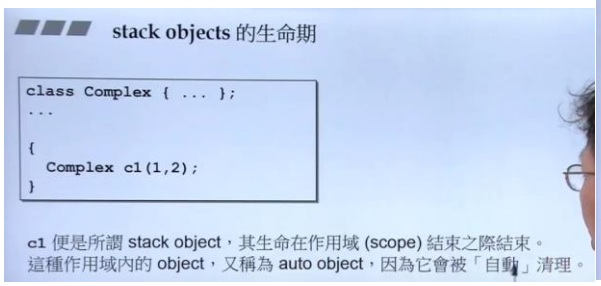


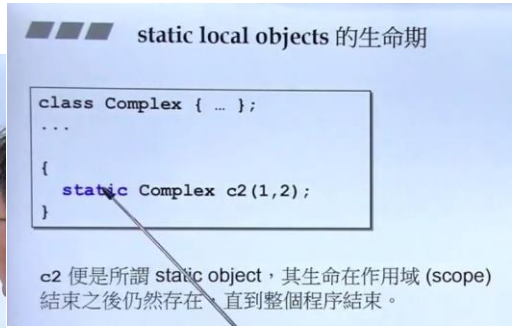
output 函数必须写成全局函数，如果写成 inline 的，就会导致我们调用时的语法为 str << cout，这不符合我们的习惯。

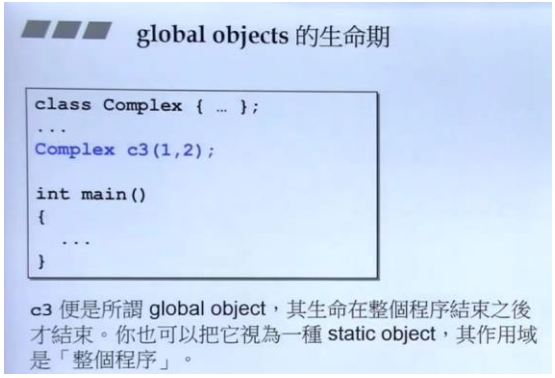
堆和栈

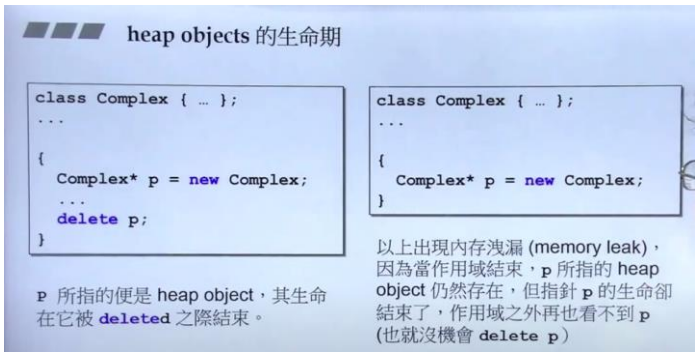


我们可以这么认为，方法体结束,Stack 自动消灭，不需要手动释放。另外，Stack 还存储 local object；而 new 创建的在 heap 内的对象，我们需要手动删除它。



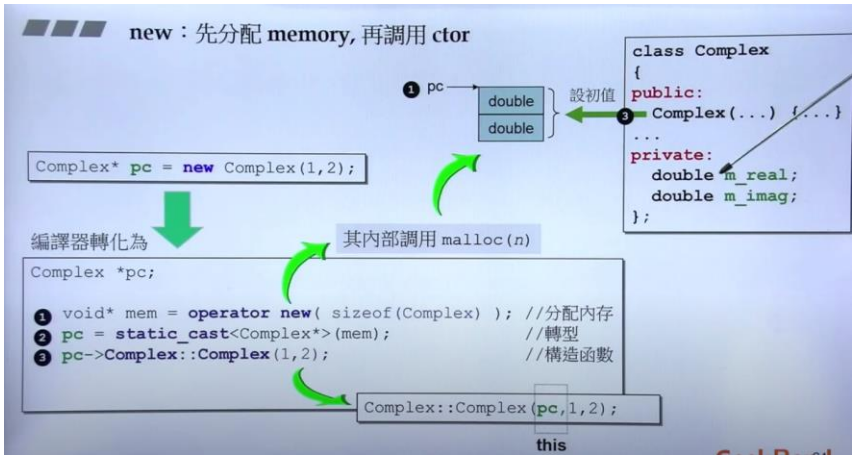


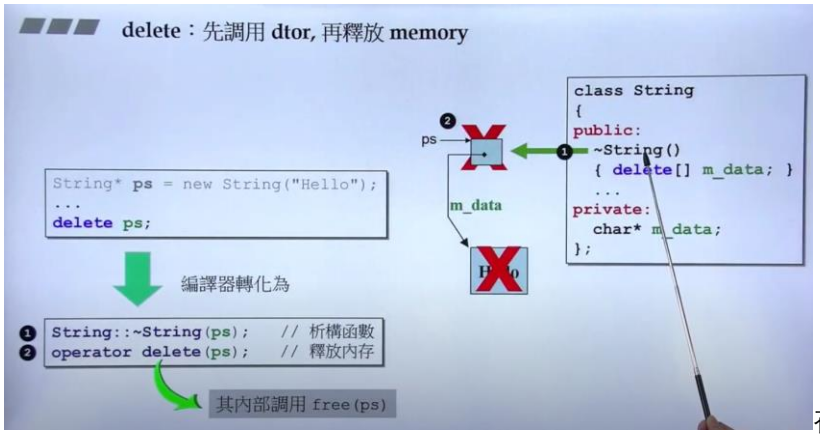




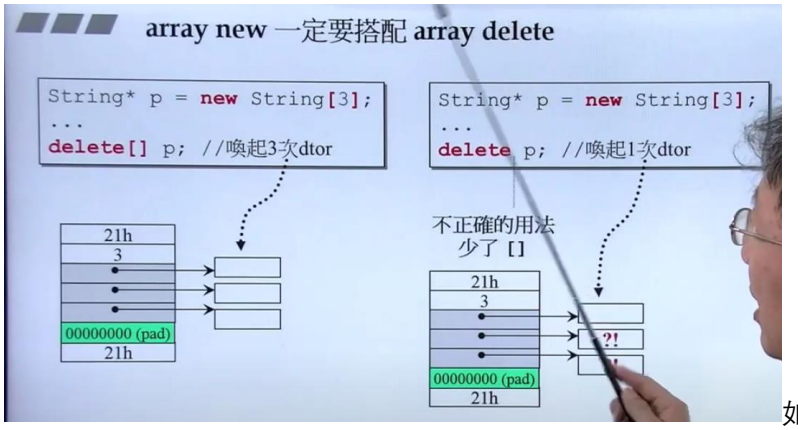
内存泄漏是指---我们无法再对一块内存做操作—删除，修改等等，这样导致了内存的浪费。比如上面的例子—指针在作用域

之后消失，但是他指向的内存还存在，这就造成了内存泄漏。



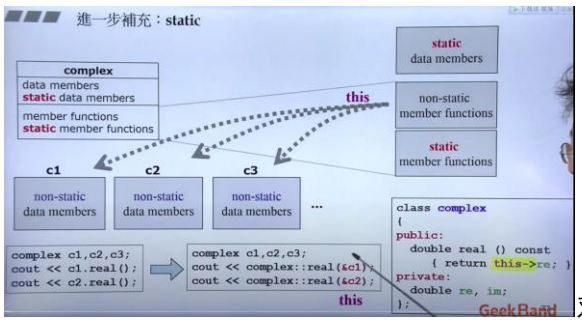


在这里的第一步，析构函数将字符串指针指向的内容 m_data 杀死，第二步是将 ps 也就是指针释放掉。

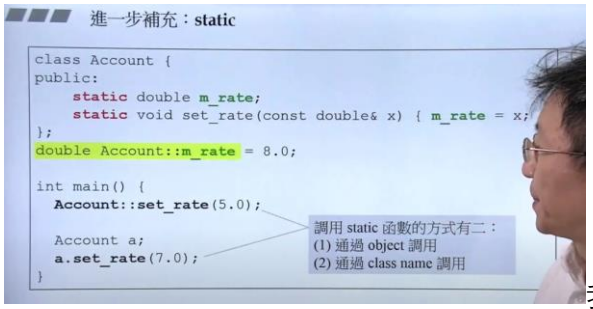


如果调用的不是 delete[]，则编译器认为删除的并非数组，只调用一次析构函数，导致删除只有一个内存空间。

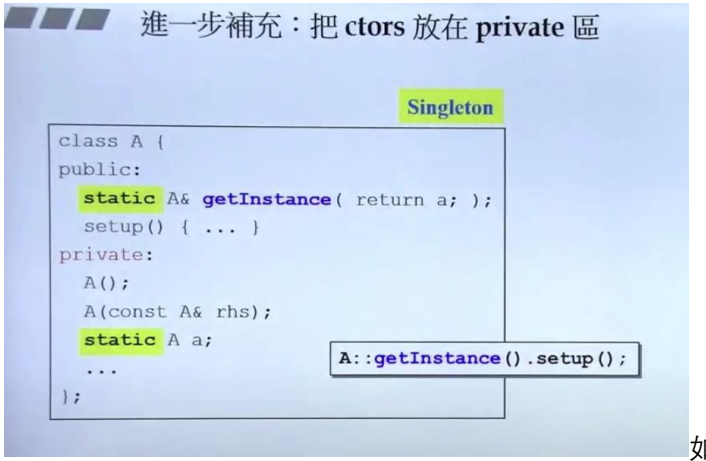
补充



对于无 static 关键字的对象，我们需要 this pointer 才能够获取到对象的数据，而加上 static 关键字后，对象和数据就脱离开来了。由于没有对象相关联，因此这样的静态的数据只有一份，那什么情况下我们需要静态的数据呢？当所有对象都使用同一份数据时，我们就可以使用 static 来修饰这个数据，比如银行的利率，所有用户都使用该利率，这跟对象没有关系，因此我们可以将利率这一组数据设为 static。那什么时候使用 static 修饰的函数呢？static 的函数和普通的函数不同在于没有 this pointer，正因为没有 this pointer，我们就没有办法像普通的函数那样去处理对象的数据，他只能够用于处理静态的数据，

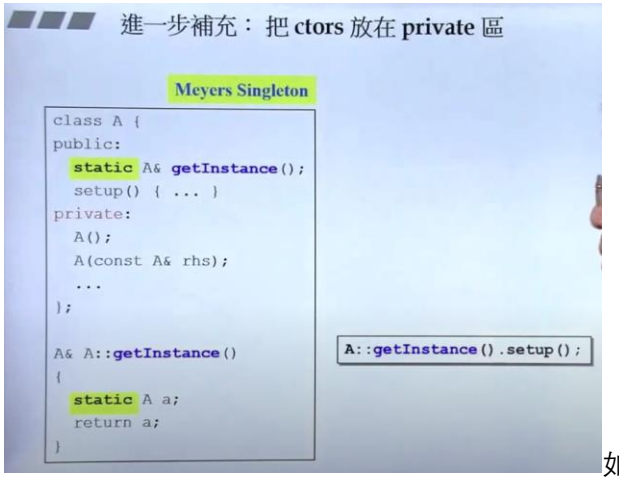


我们在类内声明一个 double 值，然后我们就必须在类外定义它（分配内存），在这里定义的时候是否赋值都是可以的，但黄色的部分是必须的。需要注意的是如果我们使用函数来调静态方法，那对象的地址 this pointer 是不会被隐藏然后传入，这跟非静态的方法是不同的，静态的方法目的一般是修改静态的数据，我们可以看到，即使是通过了对象调用静态函数修改了静态数据，但是这个静态数据是全局的，是不为这个对象所独有的，因此这种调用方法和使用类名调用的性质和作用是相同的。

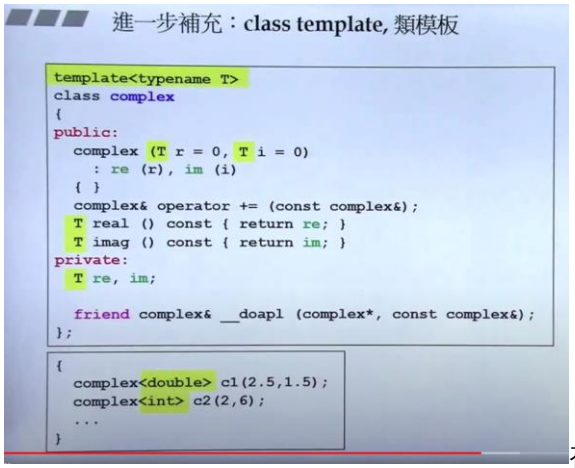


如果我们想要我们的类只能有一个对象，那 singleton 的设计方法是高效的，我们将类的构造函数、拷贝构造都放在 private 里，

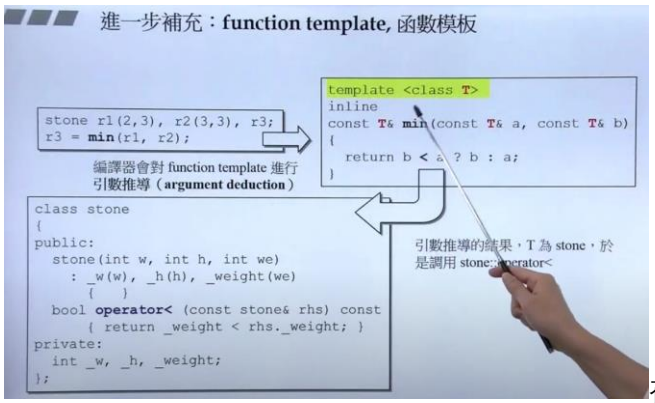
然后在 private 里面就创建一个对象，这样的话，我们就没有办法像往常一样来创建对象了，然后，我们在 public 里写一个函数用于返回类的对象，我们就通过这个方法来调用属于这个类的其他 public 的方法，如上图调用 setup()。但这样的写法会使得 a 一直存在，浪费内存。



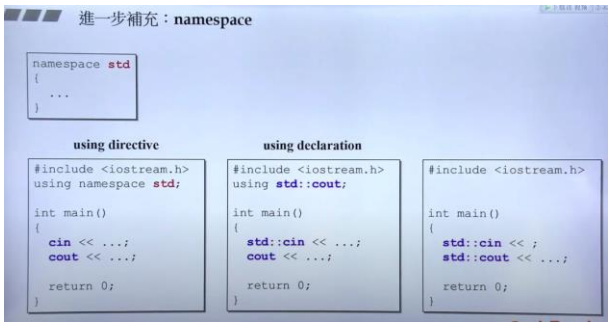
如果我们使用左边这样的方法，将创建对象放在函数内，这样只有在调用该函数时，对象才会被创建，函数执行完毕，对象死亡，这样对于内存管理是有利的。虽然在外部调用时的形式是一样的，但确实内部发生了变化。



在这里，我们使用 template<typename T>来声明 T 是未定的类型，我们就需要在使用该类创建对象时使用<>声明变量的类型。

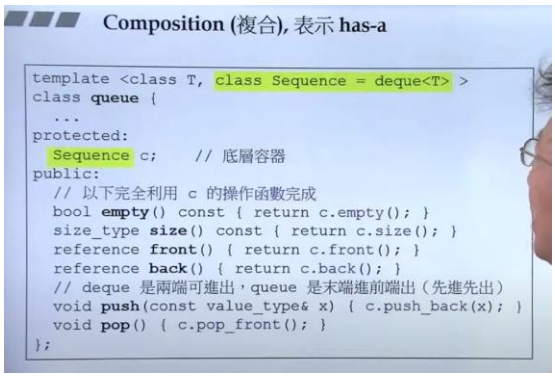


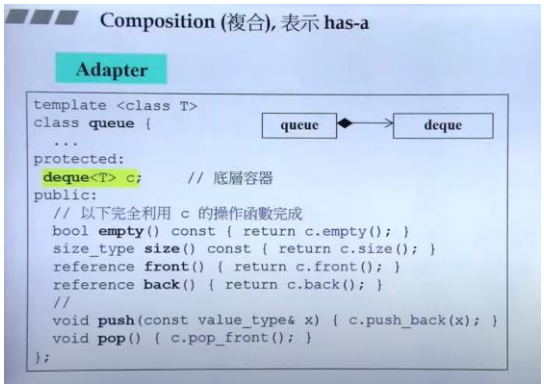
在这里的 min 函数我们使用函数模板<class T>来定义它，这样无论是什么样的对象我们都能够自适应的进行比较。比如我们对 stone 类型的函数进行<比较，这样就会自动调用 stone::operator<。与类模板不同，我们不需要使用<>来指明类型，而编译器会自动地对函数模板进行实参推导，然后对于 min 内的<进行操作符重载，定位到使用的 stone 类型的比较函数中，



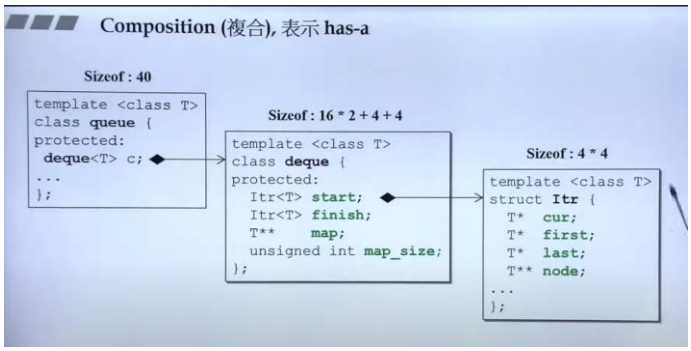
面向对象的编程

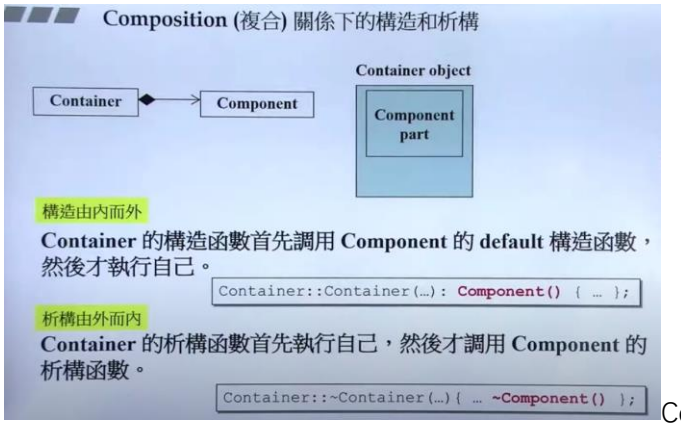
Inheritance(继承)，Composition(复合),Delegation(委托)



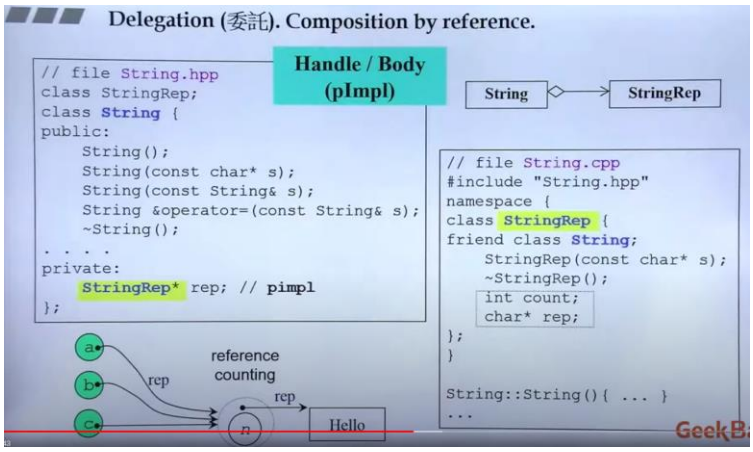


我们用图来表示关系：黑色菱形的一端就是容器，表示容纳了另外一个。
上右图的 pop 出栈的函数是使用了 deque 中的正向出栈 pop_front 函数，实际上只是对其进行了换名，这种设计模式我们称为 Adapter—改造。



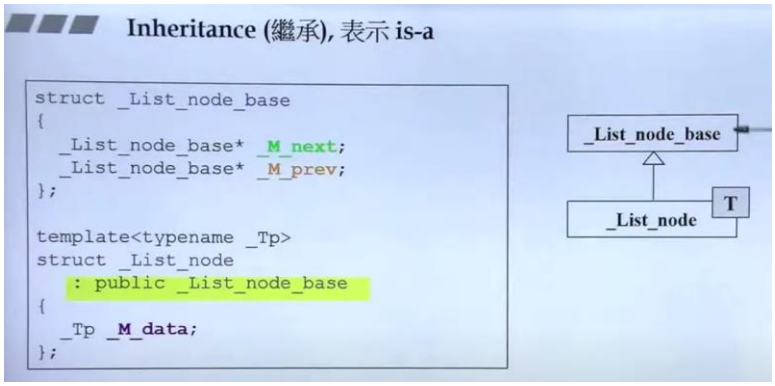


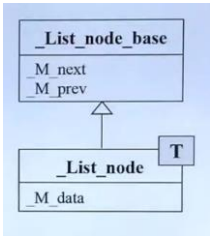
Composition 的情况下我们应该如何进行构造和析构？构造应该由内而外，如上图，我们在 Container 的构造函数中，首先使用初值列来调用默认的构造函数用于构造内部，然后再执行{}构造外部。而析构应该由外而内，应该先将 Container 析构，然后再析构 Component。



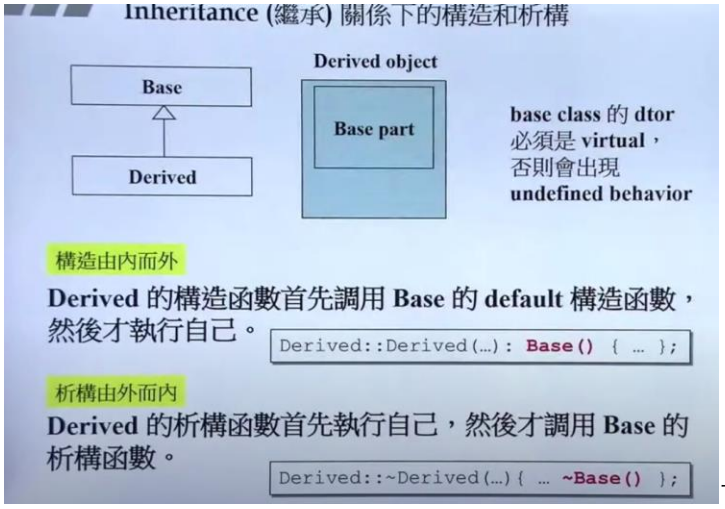
与 Composition 不同，Delegation(Composition by reference)使用的是别的类的指针，这种关系我们使用的是空的菱形来表示。与 Composition 的生命周期不同，Composition 的对象在我们自己的对象创建时就被创建出来了，而 Delegation 的对象则不然，我们在使用时才创建他。这种设计方法非常流行，我们将接口放在左边，但实际的动作实现在右边，当我们使用时就调用相应的右边的方法，这种设计方法称为 pimpl(pointer to implementation 或 Handle/Body)。

下面的图表示了实现的过程，我们有三个字符串 a,b,c 他们指向了同一个 reference counting，reference counting 内有两个内容:n—引用计数，rep—引用指针，而这个 rep 指向了 hello 这块内存，也就是说，我们有三个对象 abc,指向了同一块内存，我们称为共享，这样节省了内存，但是---这样会导致修改一个对象的内容就会使得引用该内容的对象都会发生改变，这不是我们所乐见的，因此，我们使用 copy on right 方法，如果我们要修改共享内存的其中一个对象的内容，我们就单独 copy 这个内容给这个对象，这样这个对象就有了单独的内存空间用于修改内容，其他的对象不变，还是指向原来的内存区域。

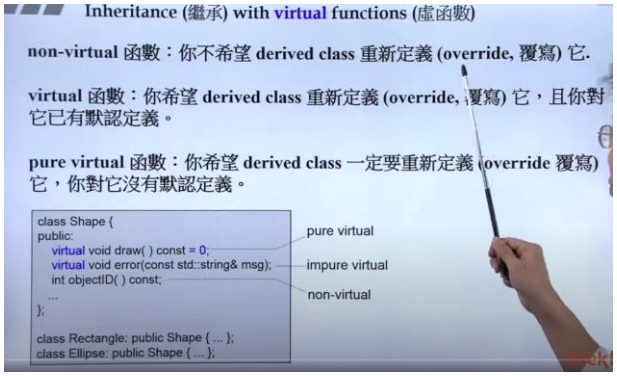


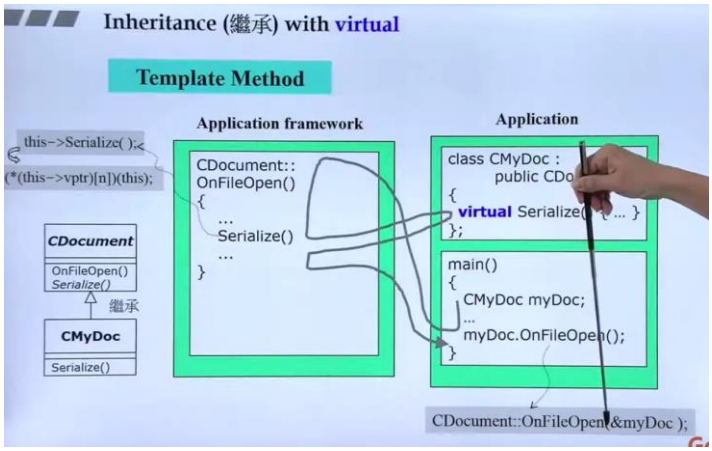


图中的三角箭头表示继承的关系，其中指向的是父类，发出的是子类。C++的继承分为三类 public、private 和 protected，但是最常用的就是 public，使用该继承就是在表达一种信息—is a.



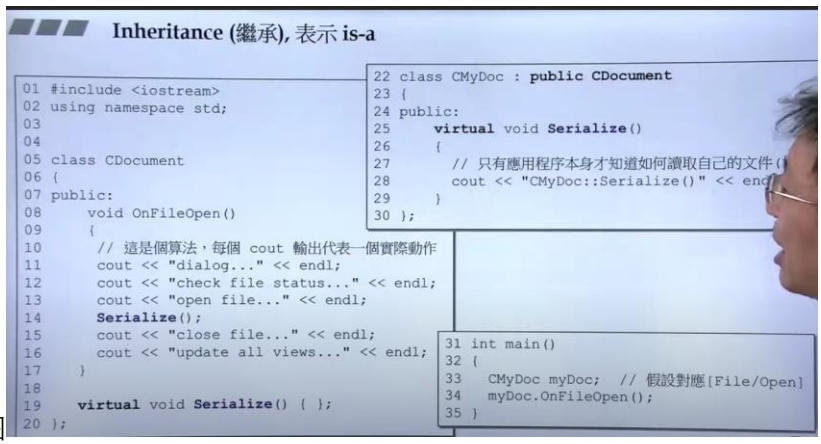
可能成为父类的类，析构函数就要设为 virtual。

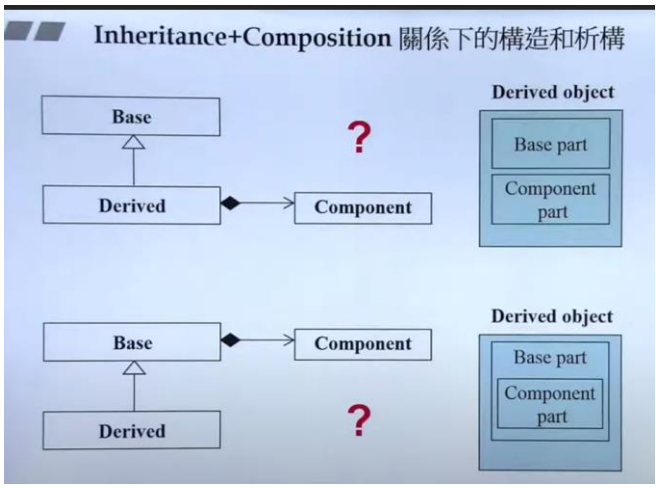


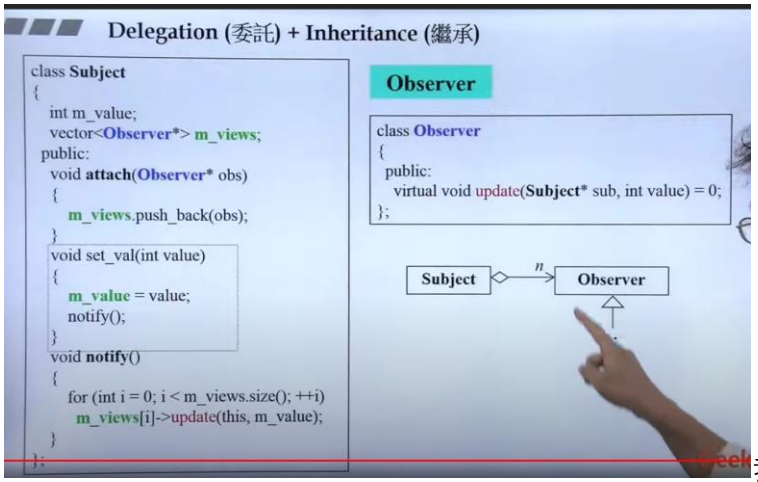


我们举一个例子，这里CDocument 父类已经写好了大部分的接口，但是我们将父类中的 Serialize()设为虚函数(纯虚或虚函数皆可)。然后我们写自己的类 CMyDoc，继承 CDocument，而我们创建子类的对象时，我们就可以直接调用子类中没有定义但是父类中有定义的函数，比如 OnFileOpen()。这样的设计将 Serialize()延缓到子类中去进行，这种做法称为 Template Method。我们来看一下到底是怎么样调用到子类的 Serialize()的，我们使用子类对象 myDoc 调用父类的函数 OnFileOpen(),编译器传入的 this pointer 其实就是 myDoc，这样在执行父类的方法时运行到 Serialize()时，类似于重载一样的操作，因为对象是子类的对象，那这个方法执行的就是子类的方法—virtual Serialize。

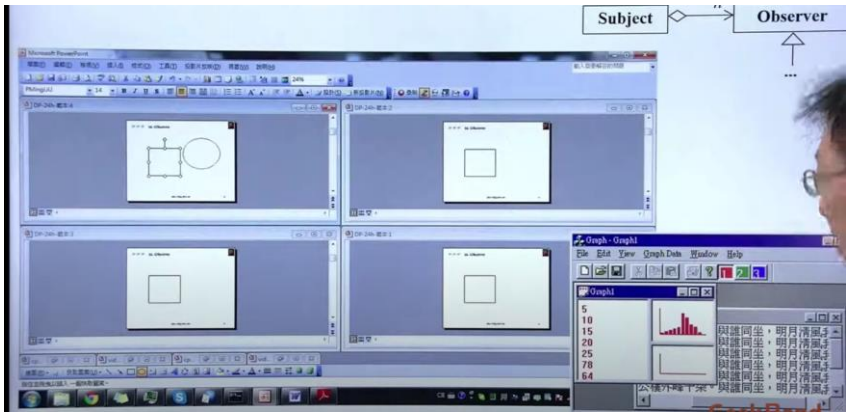
具体的写法如图



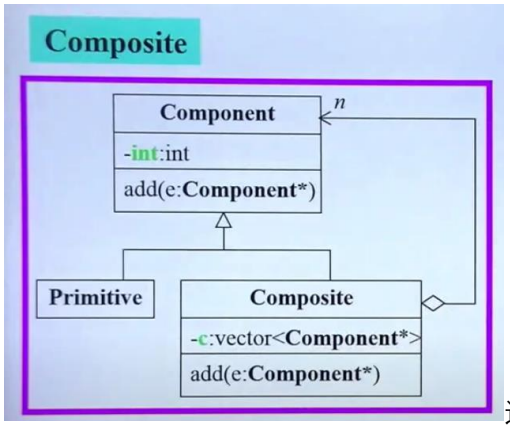




我们使用 Delegation 方式来创建指针的数组容器指向 Observer,而 Observer 类作为父类可以被继承，而基于 Observer 的子类创建的对象其实都是一种 Observer，那他们都可以放进 Delegation 创建的 Observer 向量中，比如我们有三个不同子类的 Observer 再看这同一份数据，它的表现形式可以被子类重新定义为想要的形式。

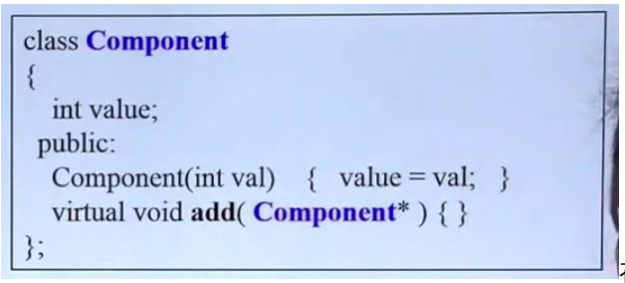


比如左边是相同的 Observer 有四个，那他们就是同一个 Observer 的不同对象，右边的则是不同的 Observer 对象有三个，那他们就是属于不同的 Observer 子类。

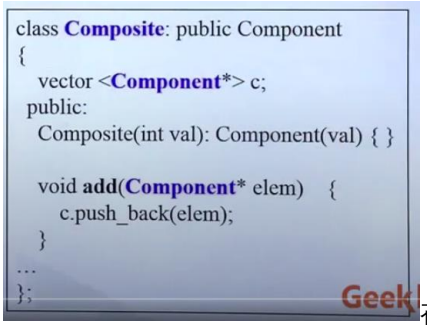


这里我们以文件系统 FileSytem 为例，创建类的关系。先看框图，其中 Primitive(基本、基础)为 File 类，还包括 Composite(组合)，而 Composite 应该可以容纳很多 Primitive，就是他们的组合产物。而如果我们想要创造一种容器，可以用于盛放 Primitive 和 Composite，那我们就想到创建这两个类的父类 Component，那么这两个子类与 Component 就是 is-a 的关系，那创建的父类的容器都可以盛放这两个子类。需要

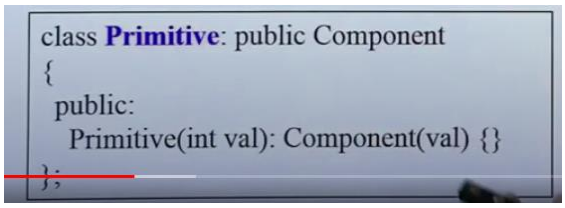
注意的是，在容器内的东西大小必须是一样的，如果我们直接盛放内容，不会通过编译，而盛放的如果是指针，那他们就满足了大小一致的要求。我们为 Composite 创建一个函数名为 add，那我们即想要能够添加 Primitive 对象，也想要能够添加 Composite 对象，那我们就将这个方法的参数设为这两者的父类对象指针 Component,，这样的话两者的指针都可以添加进来。

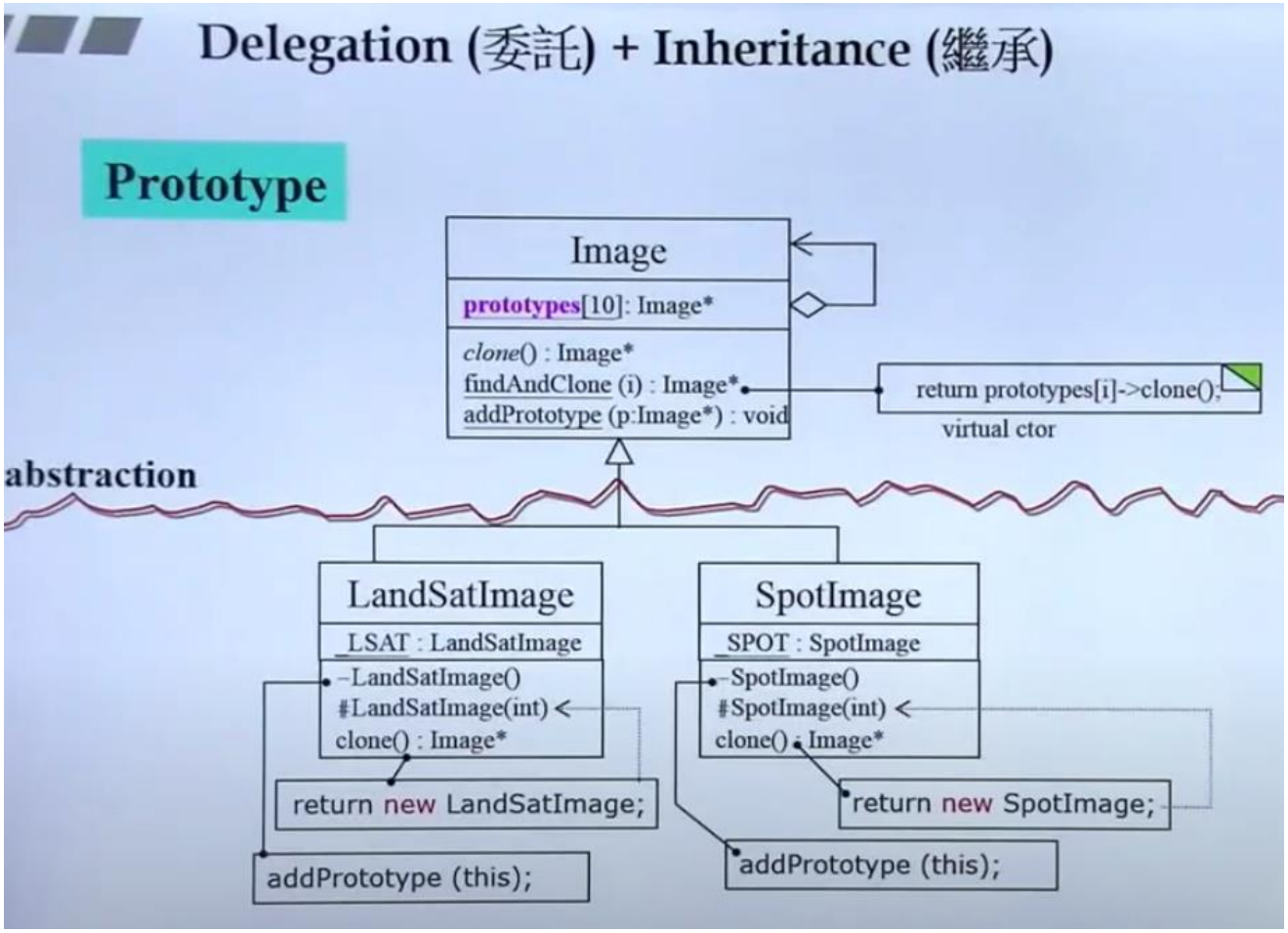


在这里我们写 Component 内的 add 函数时，不能够将其定义为纯虚函数=0，因为基础类 Primitive 没有使用这个继承来的函数，编译不会通过，因此我们将其仅仅定义为虚函数，而在 Composite 中再去重写这个函数。



在这里，Composite 使用了 Delegation。使用了继承之后，我们也要注意构造函数的写法。





我们创建了一个父类，但是我们不知道以后基于这个父类创建的子类的名称，这种情况应该怎么处理？

这种图，如果有下划线，比如 LSAT，那我们就认为他是静态的。画图和我们平常的代码恰好是相反的，比如 LandSatImage 的第二行，这行的意思是，static LandSatImage LSAT;创建一个自身的静态对象，对象类型是 LandSatImage，对象名是 LSAT。如果每一个子类都这样写，那他们就自己创造了自己，就是所谓的原型。而父类应该准备一个空间将子类的原型放入，这样父类就可以观察到子类，就可以对其进行处理。
子类创造起静态的自己时，会调用其构造函数，在 LandSatImage 里，其构造函数为私有的，在图中表示为—LandSatImage，前面的符号就代表私有的，#表示 prootected，即使构造函数是 private 的，内部也可以随意调用它创建对象。然后我们调用 addPrototype(this)；这是继承自父类的函数，将对象传递到父类的容器里，这样父类里就有了子类的原型。而 clone 函数可以 new 子类自己，这样父类就可以通过传递上去的原型调用 clone 函数来创建子类对象了，如果没有原型，就不能通过子类对象来调用 clone 函数了。那如果我们将 clone 设为静态函数，即使没有对象不也可以调用 clone 函数了吗？其实不然，因为调用静态函数的方法有两种：通过对象调用，这种已经排除；通过类名调用，而我们在写的情况就是---不知道未来以这个框架为父类的子类的名称，那这种调用方法是不可行的。