C++中的虚基类是一种用于解决菱形继承（diamond inheritance）问题的特性，该问题通常出现在多继承的情况下，当一个类继承自两个或多个类，这些基类又继承自同一个虚基类时。

下面是关于虚基类背后的实现机制的总结：

1.菱形继承问题：考虑如下继承关系，其中类D继承自B和C，而B和C都继承自A，形成了一个菱形继承结构。

A

/ \

B C

\ /

D

在这种情况下，如果A类中有成员变量或虚函数，它们可能被继承类D多次，导致二义性和冗余。

2.解决方法：为了解决菱形继承问题，C++引入了虚基类的概念。虚基类是在多继承中的基类前加上virtual关键字的类。在上面的例子中，A可以被定义为虚基类，即：

class A

{

public:

// ...

};

通过将A定义为虚基类，只会有一个A的实例被共享，而不是多个实例。这意味着在类D中只有一个A子对象。

3.实现机制：虚基类的实现涉及到在派生类中对虚基类子对象的管理。在内部，C++使用虚基类表（virtual base table，简称vtable）来实现虚基类。该表记录了虚基类子对象的偏移量等信息，以便在访问虚基类成员时能够正确定位。

4.访问虚基类成员：当访问虚基类成员时，编译器会使用虚基类表来确定如何定位虚基类的成员。这确保了在派生类中只有一个虚基类子对象被访问。

5.构造函数和析构函数：在虚基类的构造函数和析构函数中需要特殊处理，以确保虚基类的构造和析构正确执行。通常，在最派生类的构造函数初始化列表中调用虚基类的构造函数，以避免重复构造。在析构函数中，虚基类的析构函数会被自动调用，确保资源正确释放。

总之，虚基类是C++中用于解决菱形继承问题的关键工具。它通过虚基类表的机制来确保只有一个虚基类子对象被共享，从而避免了二义性和冗余。这在多继承的复杂继承关系中非常有用。虚基类的使用需要谨慎，特别是在构造和析构函数中，以确保正确的对象构造和销毁顺序。

**第十四章**

C++提供了几种重用代码的手段。第13章介绍的公有继承能够建立is-a关系，这样派生类可以重用基类的代码。私有继承和保护继承也使得能够重用基类的代码，但建立的是has-a关系。使用私有继承时基类的公有成员和保护成员将成为派生类的私有成员：使用保护继承时，基类的公有成员和保护成员将成为派生类的保护成员。无论使用哪种继承，基类的公有接口都将成为派生类的内部接口。这有时候被称为继承实现，但并不继承接口，因为派生类对象不能显式地使用基类的接口。因此，不能将派生对象看作是一种基类对象。由于这个原因，在不进行显式类型转换的情况下，基类指针或引用将不能指向派生类对象。还可以通过开发包含对象成员的类来重用类代码。这种方法被称为包含、层次化或组合，它建立的也是has-a关系。与私有继承和保护继承相比，包含更容易实现和使用，所以通常优先采用这种方式。然而私有继承和保护继承比包含有一些不同的功能。例如，继承允许派生类访问基类的保护成员：还允许派生类重新定义从基类那里继承的虚函数。因为包含不是继承，所以通过包含来重用类代码时，不能使用这些功能。另一方面，如果需要使用某个类的几个对象，则用包含更适合。例如，State类可以包含一组County对象。

多重继承（MI）使得能够在类设计中重用多个类的代码。私有MI或保护MI建立has-a关系，而公有MI建立is-a关系。MI会带来一些问题，即多次定义同一个名称，继承多个基类对象。可以使用类限定符来解决名称二义性的问题，使用虚基类来避免继承多个基类对象的问题。但使用虚基类后，就需要为编写构造函数初始化列表以及解决二义性问题引入新的规则。

类模板使得能够创建通用的类设计，其中类型(通常是成员类型)由类型参数表示。典型的模板如下:

template <class T>

class Ic

{

Tv;

...

public:

Ic(const T & val) : v(val) { }

...

};

其中，T 是类型参数，用作以后将指定的实际类型的占位符（这个参数可以是任意有效的C++名称但通常使用T和Type）。在这种环境下，也可以使用typename代替class:

template <typename T> // same as template <class T>

class Rev {...} ;

类定义（实例化）在声明类对象并指定特定类型时生成。例如，下面的声明导致编译器生成类声明用声明中的实际类型short替换模板中的所有类型参数T:

class Ic<short> sic; // implicit instantiation

这里，类名为Ic<short>，而不是Ic。Ic<short>称为模板具体化。具体地说，这是一个隐式实例化使用关键字template声明类的特定具体化时，将发生显式实例化:

template class IC<int>; // explicit instantiation

在这种情况下，编译器将使用通用模板生成一个int具体化——Ic<int>，虽然尚未请求这个类的对象。

可以提供显式具体化——覆盖模板定义的具体类声明。方法是以 template<>打头，然后是模板类名称，再加上尖括号（其中包含要具体化的类型）。例如，为字符指针提供专用Ic类的代码如下:

template < > class Ic<char \*>.

{

char \* str;

...

public:

Ic(const char \* s) : str(s) { }

...

};

这样，下面这样的声明将为 chic 使用专用定义，而不是通用模板:

class Ic<char \*> chic;

类模板可以指定多个泛型，也可以有非类型参数:

template <class T，class TT，int n>

class Pals {...};

下面的声明将生成一个隐式实例化，用double代替T，用string代替TT，用6代替n:

Pals<double，string，6> mix;

类模板还可以包含本身就是模板的参数:

template < template <typename T> class CL,typename U, int z>

class Trophy {...};

其中z是一个int值，U为类型名，CL为一个使用 template<typename,T>声明的类模板。

类模板可以被部分具体化:

template <class T> Pals<T，T，10> {...};

template <class T，class TT> Pals<T，TT，100> {...};

template <class T，int n> Pals <T，T\*，n> {...};.

第一个声明为两个类型相同，且n的值为6的情况创建了一个具体化。同样，第二个声明为n等于100的情况创建一个具体化；第三个声明为第二个类型是指向第一个类型的指针的情况创建了一个具体化。

模板类可用作其他类、结构和模板的成员。

所有这些机制的目的都是为了让程序员能够重用经过测试的代码，而不用手工复制它们。这样可以简化编程工作，提供程序的可靠性。