### 1.重载

#### 1.1.函数重载

1.1.1. 编译器依靠传递的实参类型和个数确定重载函数的调用；

1.1.2. 编译器调用重载函数的规则(参数个数相同)：

(1)具有与实参类型相同的形参的重载函数。

(2)转换实参，将T转换为T&,或T&转换为T,或T转换为const T,然后寻找形

参类型匹配的重载函数(T为某数据类型)。

(3)实参进行标准的类型转换，如char→int→unsigned int,,然后寻找形参类型匹配的

重载函数。

(4)使用构造函数转换实参，使其与某一重载函数匹配。例如，如果类A有构造函数

A(int),那么it型的实参可以被编译器转换为类A的临时对象。

(5)使用类型转换函数进行转换，使其与某一重载函数匹配。

1.1.3.父类和子类声明一致的重载，不可能通过参数类型或数量确定调用函数，而是通过在父类的成员函数前加T:进行区分(其中T为父类名)；

#### 1.2.运算符重载

1.2.1运算符重载特点

(1)使用operator关键字对重载函数进行标识和定义。

(2)只能重载C++提供的标准运算符。

(3)参数个数固定。重载函数的参数个数与标准运算符保持一致。即对于一元运算符（前缀和后缀形式），重载函数有且只能有一个参数；对于二元运算符（中缀形式），重载函数有且只能有两个参数。

(4)针对类对象进行操作。即重载函数的参数至少有一个属于class类型。

1.2.2将运算符重载为友元和成员函数

由于重载运算符针对类对象进行操作，所以通常将运算符重载函数定义为某类的友元函数，或定义为类的成员函数。

1.2.3除了以常规形式调用重载的运算符外，还可以用operator关键字形式调用。

### 2.虚拟函数

#### 2.1静态联编与动态联编

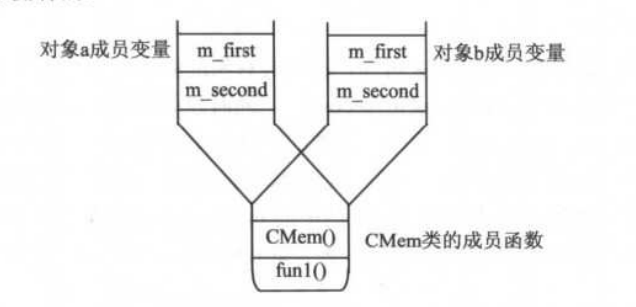
在C+编译时，对于常规的函数调用，编译器在函数的调用处插入函数的相对地址，程序运行时可以由函数的相对地址计算出函数的绝对地址，这样函数可以被正确调用。这种在编译时就确定函数地址的联编过程叫做静态联编。动态联编是指在程序编译时，编译器并不知道函数的相对地址，调用函数的相对地址只有在程序运行时才能确定。

#### 2.2引入虚拟函数

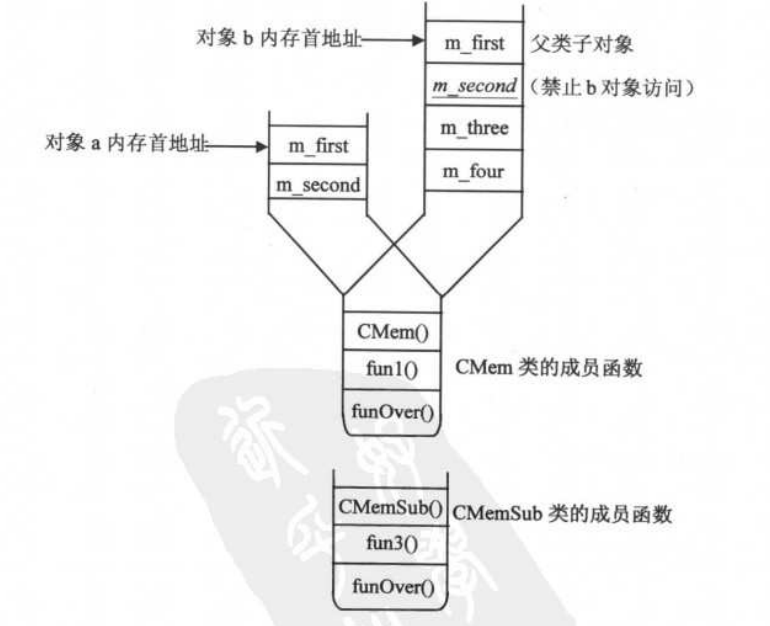
2.2.1重载虚拟函数要求所有函数声明完全一致，即函数名称和形参列表都一致。否则等同于普通函数的重载，不能实现动态联编。

#### 2.3虚拟函数的实现机制

2.3.1同一个类的所有对象共享同一个成员函数的地址空间，而每个对象有独立的成员变量地址空间。也可以这样说，成员函数是类拥有的，成员变量是类的对象拥有的。



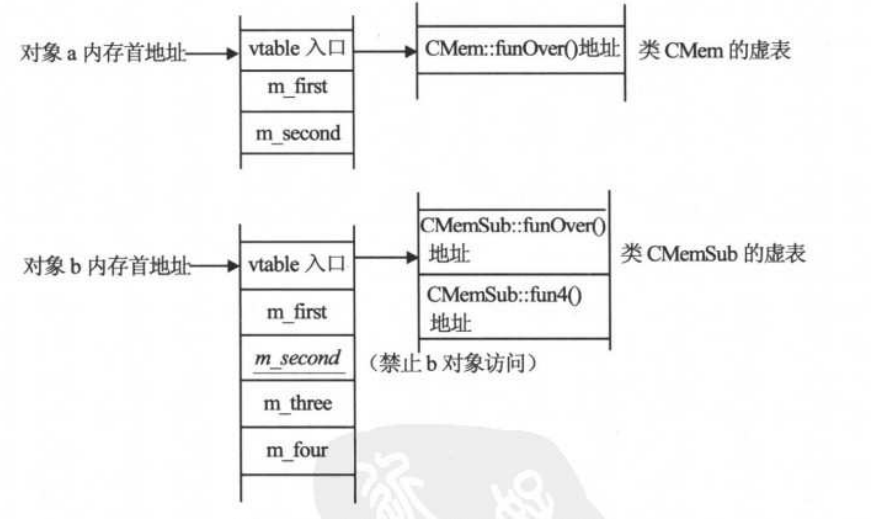
2.3.2派生类对象的内存区域包含了父类所有的成员变量空间,同时包含派生类定义的成员空间。其中，父类的成员变量空间被称为父类子对象，整个派生类实例的内存空间被称为全对象。但是由于编译器的限制，基类中那些没有被派生类继承的私有成员变量，派生类无法访问。对于从基类继承的成员函数，派生类和基类共享



2.3.3对于非虚拟函数的调用，编译器只根据数据类型翻译函数地址，判断调用的合法性。因为对象的内存地址空间中只包含成员变量，并不存储有关成员函数的信息，所以非虚拟函数的地址翻译过程与其对象的内存地址无关。如定义一个父类的指针pMem，不论pMem指向什么对象的地址，pMem调用成员函数永远是父类的，哪怕子类重载了。

2.3.4深入：虚拟函数的内存分配机制

因为虚拟函数的地址翻译取决于对象的内存地址，而不取决于数据类型（编译器对函数调用的合法性检查取决于数据类型）。原来，如果类中定义了虚拟函数，该类及其派生类就要生成一张虚拟函数表，即vtable.。而在类的对象地址空间中存储一个该虚表的入口，占4个字节，这个入口地址是在构造对象时由编译器写入的。



虚函数的工作机制，由于对象的内存空间中包含了虚表入口，编译器能够由这个入口找到恰当的虚函数，这个函数的地址不再由数据类型决定了。如定义一个父类的指针pMem，子类重载了父类的成员函数，pMem指向什么对象的地址，就调用什么对象的成员函数。

语句

pMem=&b;使pMem指向对象b的内存空间，调用pMem->-funOver(时，编译器得到了对象

b的vtable入口，并由这个入口找到CMemSub:fun40虚函数地址。

#### 2.4虚拟函数的应用

2.4.1如果类包含虚拟成员函数，则将此类的析构函数也定义为虚拟函数

这是很必要的，因为派生类对象往往由基类的指针引用，如果使用new操作符在堆中构造派生类对象，并将其地址赋给基类指针，那么最后要使用delete操作符删除这个基类指针（释放对象占用的堆栈）。这时如果析构函数不是虚拟的，派生类的析构函数不会被调用。造成内存泄漏。要解决这一问题很简单，只需在基类的析构函数前加virtual关键字即可，不必更改派生类的代码。

#### 2.5纯虚拟函数

形式：void virtual BuildValues(char Statement)=0;

定义了纯虚函数的类称为抽象基类。抽象基类节省了内存空间，但不能用来实例化对象。其派生类必须重载所有的纯虚函数，否则产生编译错误。由于抽象基类不能实例化对象，只能通过类名限定的方法调用它的常规虚函数。

### 3. 静态成员和类模板

#### 3.1静态成员

3.1.1C++中的静态成员变量

类的静态成员变量不同于普通意义的静态变量，有其特殊性。类的静态成员变量既有类成员的特性又有全局变量的特性，但其实质是全局变量。静态成员变量在全局范围内定义并初始化，在类中进行声明，它的可访问权限受类的控制，即只有声明为公有成员才能被外部访问。在外部访问静态成员时可以不通过对象，加类的限定符即可。

私有（或保护）静态成员变量用于对象之间通信，公有静态成员变量不仅可用于对象之间通信，还可供外部访问。这样，外部可以直接通过类名标识访问该类公有静态成员，获得该类对象的信息，就像访问全局变量一样。

全局动态变量可以代替类的公有静态成员变量，但如果这个全局变量专门为存储一类对象的某个状态或属性而定义，应将它定义为该类的公有静态成员，这样便于代码阅读，并减少名字冲突。

3.1.2静态成员函数

(1)C++类的静态成员函数与其静态成员变量的意义也相近，既有类成员的特性又有全局函数的特性。它的可访问权限受类的控制，即只有声明为公有成员才能被外部访问。在外部访问静态成员函数时可以不通过对象，加类的限定符即可。静态成员函数的声明和定义没有什么特别，只要在成员函数的声明前加static关键字即可。

(2) 由于不用实例化类的对象，就可以调用类的静态成员函数（公有），所以编译器对静态成员函数进行了必要的约束。

a.不可访问该类中非静态成员（包括变量和函数）。

b.不可定义为虚拟的。

c.不可加const关键字。

d.静态成员函数调用时，不传递this指针。所以hs指针在函数体内不可用：当类的重载运算符函数定义为静态时，其参数列表同定义为友元函数时一样。

(3)公有的静态成员函数可用于访问私有静态成员变量，可以通过类名标识符如全局函数一样被外部访问。私有或保护的静态成员函数很少使用。如果一个全局函数的操作只与某个类相关，并且没有访问该类的非静态成员，那么将它定义为该类的公有静态成员函数，这样便于代码阅读，并减少名字冲突。

#### 3.2类模板

3.2.1类模板的定义

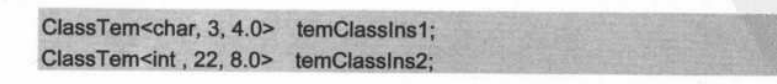
(1)定义类模板的语法是：

template<模板形参列表>class模板名

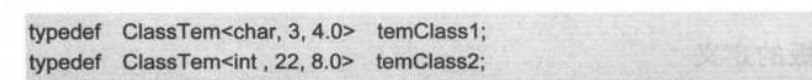
{

∥成员变量、成员函数、友元函数和内嵌类的声明或定义

};

如：

可改写为：



那么以后就可以将temClass1和temClass.2作为类名使用了。

(2)使用类模板，实例化生成模板类时，编译器首先用传递的实参替换模板定义的形参，生成模板类的定义，然后进行类型检查等编译活动。

### 4. 多重继承和内嵌类

#### 4.1继承方式与访问权限

4.1.1派生类的继承权限



#### 4.2多重继承和虚拟基类

4.2.1多重继承的二义性

（1）在调用父类的相应成员可以在前面加上类标识限制；

（2）在父类没有共同双亲的情况下，一般在子类中将名字冲突的成员重新定义，以此来解决二义性问题。这样，由多重继承的子类再派生的子类，或类的实例化使用者就不必为成员的二义性烦恼了。

4.2.2使用虚拟基类解决二义性问题

在父类存在共同双亲的情况下，可以采用更好的方式解决类成员的二义性问题，那就是虚拟基类。

虚拟基类的作用是：如果在一个派生类的继承结构中，存在两个同样的虚拟基类，那么该类的实例化对象内存中只存在一个虚拟基类的子对象。

为了保证虚拟基类只被构造一次，虚拟基类的构造函数不能按照传统的方式调用，即不能被直接子类调用，而是由当前最低层次类的构造函数调用，该类被称为最派生类。若最派生类构造函数初始化列表中没有显式地调用虚拟基类的构造函数，会调用虚拟基类的默认构造函数。

#### 4.3内嵌类与类合成

4.3.1内嵌类的应用价值

内嵌类被封装在包容类中，外部不能对它直接访问，也避免了和它产生名字冲突。可以说，内嵌类对于外部是不可见的。同时，内嵌类虽然是包容类的成员，但二者的访问控制是完全独立的，不存在任何耦合，其中一方的定义发生变化，不影响另一方。

针对内嵌类的以上特点，可以用它封装一个数据处理的用户接口部分，使数据处理与数据的表示分离。内嵌类就像一个工具，用户通过内嵌类实现对包容类数据的处理。内嵌类的定义可以根据用户的需求进行修改、增加或删除，这对包容类没有什么影响：包容类可以根据处理对象的不同，封装不同的数据集，这种代码变动也不会惊动内嵌类。

4.3.2类合成

如果一个类包含一个内嵌类，那么也要包含该内嵌类的对象作为成员变量，否则内嵌类是毫无意义的。一个类中包含另一个类的对象作为数据成员变量，叫做类的合成。

如果一个类和另一个类在客观语义上是服务员与服务对象的关系，可以考虑采用内嵌类的方式设计类结构。但如果这个服务员同时为多个不同的类服务，就不宜采用内嵌类了。这时可以将服务员类定义为全局类型，只要在每个服务对象类中定义一个该类型的成员变量，服务对象类即可享受到服务员类提供的所有服务。这就是类合成的作用。

#### 4.4类继承与类合成的应用

4.4.1构造初始化和赋值操作

如果一个类合成了其他类，即包含其他类的对象，首先要保证该对象在构造函数初始化列表中被正确构造。一般要使被合成类的所有构造函数都有被调用的机会。

在使用继承时也是如此，要恰当地调用父类的构造函数进行初始化，也要保证父类所有构造函数有机会被调用。

在书写初始化列表时，要注意系统定义的隐藏构造函数，即以该类的对象为参数的构造函数

ClassName:ClassName(const ClassName&a)；如果该类有若干父类，该构造函数的初始化列表中将调用父类的这个隐藏构造函数。所以，如果重新定义了这个构造函数，不要忘记在初始化列表中调用父类的构造函数。

当类中定义了指针类型的成员时，一定要亲手重载“=”运算符，不要依赖系统的定义。

在类继承与类合成的情况下，重载赋值运算符时，一定要对基类或合成类进行处理。

4.4.2继承与合成的选择

如果对象之间在客观语义上是一种一般和特殊的关系，如学生→初中生，应该使用公有继承。即用父类描述一般性，子类描述特殊性。这种关系又称为ISA关系。如果类结构的层次较多，为抽象出多层子类的共性，也可以使用抽象基类，类层次之间可采用公有派生。

对于私有继承，一般为了代码重用，父子之间既无明显的一般与特殊关系，也无明显的服务员与服务对象关系。如列表与树，树可以私有派生于列表：又如点与线，线可以私有派生于点。因为以点为处理对象的过程并不适用于线，点定义的成员函数也不能直接用来操作线，所以不能采用公有继承，不能将点的成员直接暴露给用户。

如果对象之间是服务员与服务对象的关系，如绘画和画笔，应采用类合成方式，这种关系被称为USEA:如果是包含与被包含的关系，如硬盘与电脑，也应采用类合成方式，这种关系被称为HASA。在某些情况下，可以用私有继承代替类合成。

### 5.MFC类结构

5.1