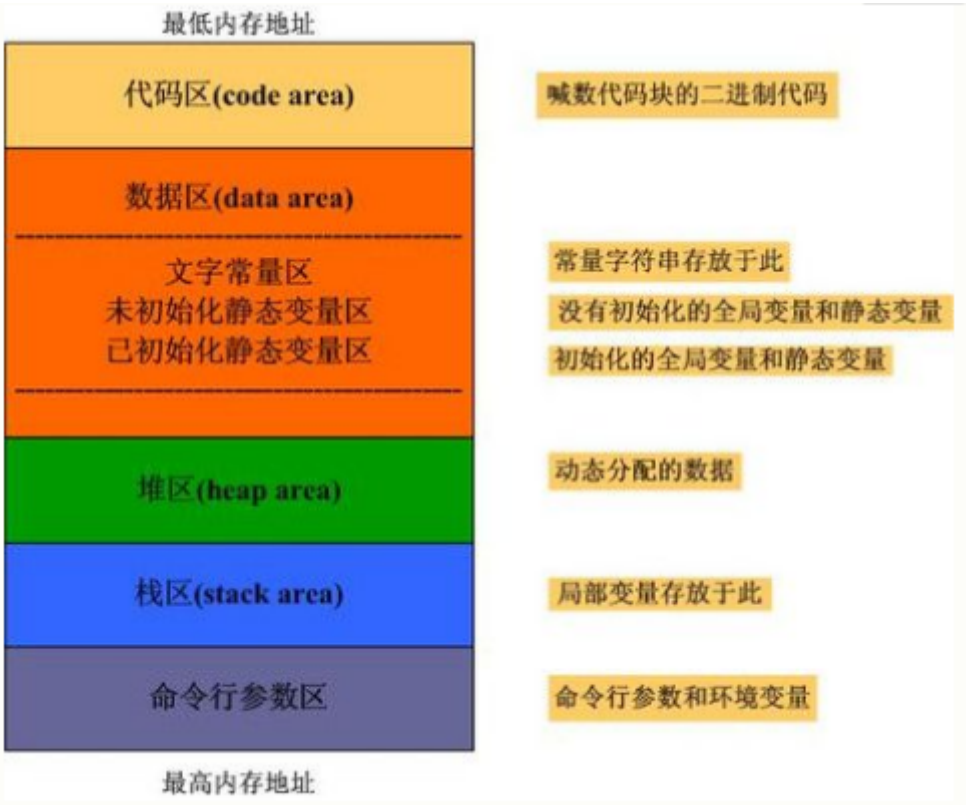
一、C++中动态内存管理：

1>、C/C++程序运行时：内存分布情况---必须是能够画出图解****



a、 栈又叫堆栈，非静态局部变量/函数参数/返回值等等，栈是向下增长的。

b、内存映射段是高效的I/O映射方式，用于装载一个共享的动态内存库。用户可使用系统接口创建共享共 享内存，做进程间通信。（Linux课程如果没学到这块，现在只需要了解一下）

c、. 堆用于程序运行时动态内存分配，堆是可以上增长的。

d、数据段--存储全局数据和静态数据。

e、代码段--可执行的代码/只读常量。

2>、malloc/calloc/realloc三个函数之间的区别　尤其喜欢问malloc****1、基本信息　2、底层实现原理

三者都是分配内存，都是stdlib.h库里的函数，但是也存在一些差异。

（1）malloc函数。其原型void *malloc(unsigned int num_bytes);

num_byte为要申请的空间大小，需要我们手动的去计算，如int *p = (int *)malloc(20*sizeof(int)),如果编译器默认int为4字节存储的话，那么计算结果是80Byte，一次申请一个80Byte的连续空间，并将空间基地址强制转换为int类型，赋值给指针p,此时申请的内存值是不确定的。

（2）calloc函数，其原型void *calloc(size_t n, size_t size);

其比malloc函数多一个参数，并不需要人为的计算空间的大小，比如如果他要申请20个int类型空间，会int *p = (int *)calloc(20, sizeof(int)) ,这样就省去了人为空间计算的麻烦。但这并不是他们之间最重要的区别，malloc申请后空间的值是随机的，并没有进行初始化，而calloc却在申请后，对空间逐一进行初始化，并设置值为0;

（3）realloc函数和上面两个有本质的区别，其原型void realloc(void *ptr, size_t new_Size)

用于对动态内存进行扩容(及已申请的动态空间不够使用，需要进行空间扩容操作)，ptr为指向原来空间基址的指针， new_size为接下来需要扩充容量的大小。

3>、malloc/free和new/delete的区别*****

malloc/free和new/delete的共同点是：都是从堆上申请空间，并且需要用户手动释放。

不同的地方是：

1. malloc和free是函数，new和delete是操作符
2. malloc申请的空间不会初始化，new可以初始化
3. malloc申请空间时，需要手动计算空间大小并传递，new只需在其后跟上空间的类型即可
4. malloc的返回值为void*, 在使用时必须强转，new不需要，因为new后跟的是空间的类型
5. malloc申请空间失败时，返回的是NULL，因此使用时必须判空，new不需要，但是new需要捕获异常
6. 申请自定义类型对象时，malloc/free只会开辟空间，不会调用构造函数与析构函数，而new在申请空间 后会调用构造函数完成对象的初始化，delete在释放空间前会调用析构函数完成空间中资源的清理

4>、new/delete和new[]/delete[]

new　T：1、调用void* operator new(size_t size)

{

malloc循环申请---->申请成功，直接返回

申请失败---提供空间不足的应对措施(用户)

提供：循环申请

未提供：抛异常

}

2、调用T的构造函数完成控件的初始化

delete：

1、调用T的析构函数释放对象中的资源

2、void operator delete()释放对象的空间

new T[N]:

1、申请空间：void* operator new[](size_t size)--->void *operator new(size_t size)--->循环采用malloc申请

2、构造N个对象：调用N构造函数初始化对象

delete[]

1、释放N个对象中的资源：调用N次析构函数

2、释放N个对象的空间：void operator delete[](void* p)--->void operator delete(void* p)--->free

定位new表达式：对已经存在的空间进行初始化

new(p) T(参数):

1、调用void* operator new(size_t size,void* where)

```
{
    return where;      不需要真正申请空间，直接将空间返回
}
```

2、调用T类型的构造函数

二、模板

1、什么是模板？什么是泛型编程？

模板：就是编译器生成代码用的模子。模板又分为函数模板和类模板。

泛型编程：编写与类型无关的通用代码，是代码复用的一种手段。模板是泛型编程的基础。

2、函数模板的实例化

a>隐式实例化

编译器如果检测到用户对函数模板进行实例化：

1、在工程中找---是否存在处理具体类型的Add函数

找到：直接调用，不需要模板生成

未找到：继续2

2、在工程中找---是否存在Add类型的函数模板

找到:

a、推演实参的类型

b、结合模板，生成处理具体类型的函数

c、调用生成的处理具体类型的函数

未找到：编译失败

注意：在对模板进行隐式实例化期间，不会进行隐式类型转化

b>显示实例化

Add<int>(1, 2);
Add <double>(1, 2.0)；1int型隐式转化为double转化成
Add<int>(1, "1234"); // 编译报错

如果是显式实例化，相当于已经明确将模板中T的类型具体化,编译器不需要再进行参数类型推演，直接根据<>中的类型生成代码。
注意：如果实参类型与<>中类型不匹配时，编译器可能会进行隐式类型转化
转化成功: 生成代码，编译通过
转化失败:报错

3、函数模板的原理
模板的编译:
1.在实例化之前:编译器只是对模板进行简单的语法检测--比如:模板参数列表写的释放有问题
不会生成处理具体类型代码
2.在实例化之后:

非类型的模板参数
模板参数列表中的类型分为:
1.类型参数:类型不具体---class T
2.非类型的模板参数--int, size
template<class T,size_t N>
class array
{};
注意:
1. N在模板中是一个常量
2、浮点数、类对象以及字符串是不允许作为非类型模板参数的。

4.类模板的特化
1.什么是特化:就是对模板中的类型参数进行特殊话的处理
模板大部分情况可能都可以正常处理，但是对于有些类型的处理可能就是一个错误
template<class T>
const T& Max(const T& left const T& right)
{
return left > right? left : right;

}

2.特化的分类

全特化:将模板参数列表中所有类型具体化

```
template<class T1, class T2>
class Data
{
public:
Data() {cout<<"Data<T1, T2>" <<endl;}
private:
T1 _d1;
T2 _d2;
};
 template<>
 class Data<int, char>
 {
 public:
 Data() {cout<<"Data<int, char>" <<endl;}
 private:
 T1 _d1;
 T2 _d2;
 };
```

偏特化:

1.部分特化:将模板参数列表中部分参数具体化

```
template<class T1, class T2>
class Data
{
public:
Data() {cout<<"Data<T1, T2>" <<endl;}
private:
T1 _d1;
T2 _d2;
};

 // 将第二个参数特化为int
 template <class T1>
 class Data<T1, int>
 {
 public:
```

```
Data() {cout<<"Data<T1, int>" <<endl;}
private:
T1 _d1;
int _d2;
};
```

2.让模板参数列表中的类型限制更加严格

//两个参数偏特化为指针类型

```
template <typename T1, typename T2>
class Data <T1*, T2*>
{
public:
Data() {cout<<"Data<T1*, T2*>" <<endl;}
private:
T1 _d1;
T2 _d2;
}
```

5.类型萃取---实现方式

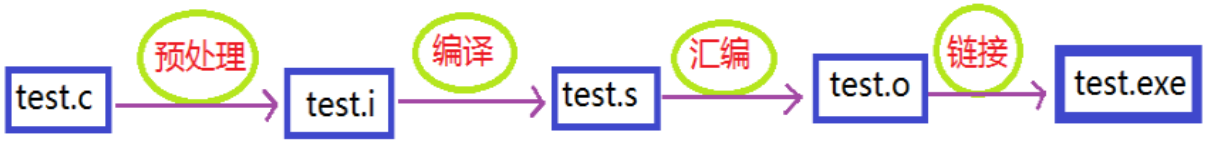
原理:就是对类模板特化的一种应用　只是调高代码运行效率的一种手段

6.分离编译

1.什么是分离编译?

一个工程中，可能有多个源文件，每个源文件是单独编译的，头文件是不参与编逢(在预处理阶段已经展开)

2.程序写完到能够正常运行，需要经历那些阶段：预处理、编译、汇编、链接 *****



- 预处理（Preprocessing）

1、　宏替换

2　　、头文件包含

3　　、条件编译的选择

- 编译（Compilation）

将预处理完的文件进行词法分析、语法分析、语义分析及优化后，生成相应的 .s 汇编代码。

- 汇编（Assemble）

将编译完的汇编代码翻译成机器码，并生成可重定位目标程序的 .o 目标文件。

- 链接（Linking）

通过链接器 ld 将目标文件和库文件链接在一起，最后生成可执行文件（executable file）。

3.模板不支持分离编译?

模板的编译：

1.在实例化之前:编译器只是对模板进行简单的语法检测--比如:模板参数列表写的释放有问题

不会生成处理具体类型代码

2.在实例化之后：就会生成具体类型的代码

解决方式:

1.按照其他文件的使用方式，在模板的定义文件中进行实例化

2.将模板的声明和实现放在一个.hpp文件中

7、模板的优缺点

【优点】

1. 模板复用了代码，节省资源，更快的迭代开发，C++的标准模板库(STL)因此而产生

2. 增强了代码的灵活性

【缺陷】

1. 模板会导致代码膨胀问题，也会导致编译时间变长

2. 出现模板编译错误时，错误信息非常凌乱，不易定位错误

三、在线OJ的输入和输出

输出:要仔细看题目的输出要求

输入:

OJ算法:接口类型OJ和IO类型OJ

算法的接口已经提供好，只需要直接进行编码

IO类型的OJ:需要用户自己接受测试用例--- -定要循环输入

两种:

情况一：三个整形输入: while(cin>>a>>b> >c){...}

情况二:整行的输入：一行中有多个单词，找出长度最长的单词while(getline(cin, s))

四、继承和多态****

继承:

1.概念

继承可以提高代码复用

在保持原有类特性的基础上进行扩展

体现出一种层次结构

2.继承权限

public、protected、 private

类成员/继承方式	public继承	protected继承	private继承
基类的public成员	派生类的public成员	派生类的protected成员	派生类的private成员
基类的protected成员	派生类的protected成员	派生类的protected成员	派生类的private成员
基类的private成员	在派生类中不可见	在派生类中不可见	在派生类中不可见

三种不同继承方式下:基类不同访问权限的成员在子类的访问权限或者可见性

默认继承权限: class--private　struct---> public

在C++中， class和struct的区别? ****

1.类和对象默认访问权

2. 继承权限

3. 模板参数列表

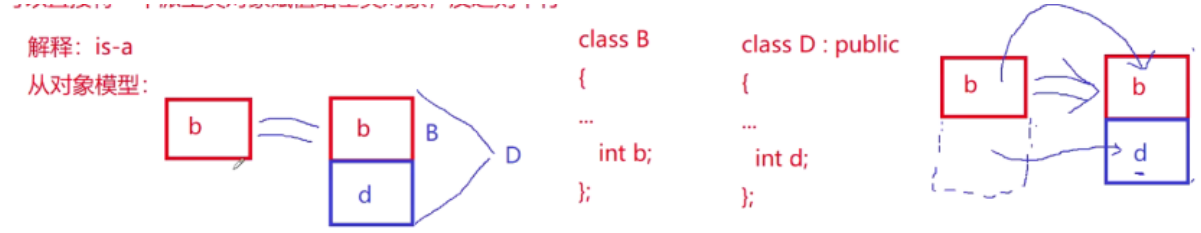
4以上三个之外几乎没有什么区别

3.赋值兼容规则----一定是public继承

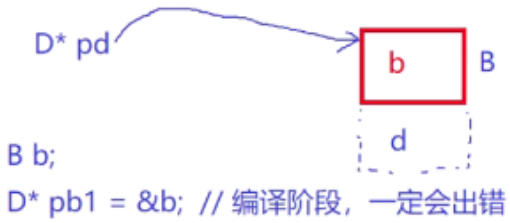
如果是public的继承方式---子类和基类是is-a 可以将子类看成是一个基类的对象，

使用:在类外时，所有用到基类对象的位置都可以用一个子类对象来进行代替

>>可以直接将一个派生类对象赋值给基类对象，反之则不行



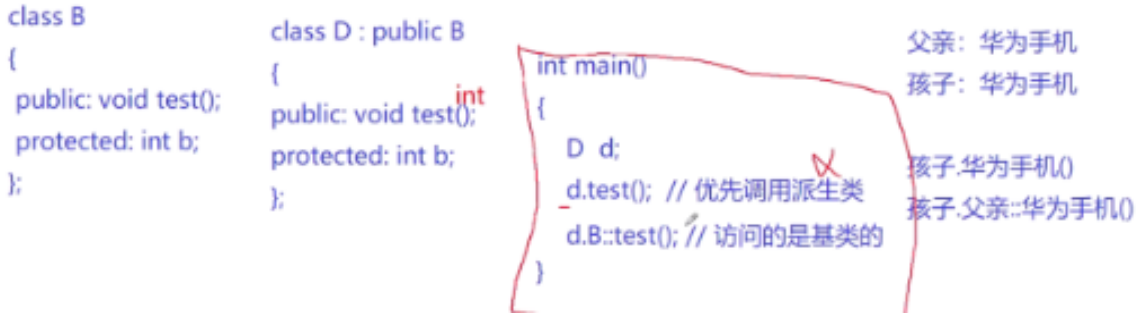
>>可以让基类的指针或引用指向子类的对象，反之则不行，但是可以强制类型转化(问题:可能会有安全性问题)



4.继承作用域

子类和基类属于不同的作用域。不论是什么继承方式，基类中私有的成员在子类中都是不可见

同名隐藏：基类和派生类中可能会存在相同名称的成员，当子类对象去访问相同名称的成员时，优先访问到的是派生类自己的



同名隐藏:

成员变量:只要基类和派生类成员变量名字相同，与类型是否相同无关

成员函数:只要基类和派生类成员函数的名字相同，与函数的原型是否相同无关

注意:派生类中test(),test(int), 两个函数不是函数重载，原因:作用域不一样

派生类不能直接访问基类同名的成员，如果硬要访问，B::test(); 使用不是很方便

或者说可能会忘记添加B::,导致一些错误

不建议:在基类和派生类中出现同名成员　例外:多态

5.继承体系中:构造和析构的规则

1>.如果基类没有定义构造函数，派生类是否提供构造函数都可以

2>.如果基类的构造函数是缺省的构造函数(无参构造函数||带有全缺省的构造函数)，派生类的构造函数释放提供都可以

此时:编译器会为派生类生成一个默认的构造函数(无参),并且会在派生类构造函数初始化列表显式调用基类的构造函数已完成基类部分成员的初始化

3>.如果基类具有非缺省的构造函数(带有参数的构造函数)，此时派生类必须显式提供自己的构造函数，并且必须在其构造函数初始化列表的位置显式调用基类的构造函数。

class B

{

```
    public: B(int a){}
};

//没有显式定义构造函数，一定编译出错
class D:public B{};
```

原因

1.如果一个类没有显式定义自己的构造函数,编译器将会生成一个默认的无参构造函数

2.编译器必须要在派生类构造函数初始化列表显式调用基类的构造函数---问题:基类的构造函数具有参数，派生列在调用时必须要传参，但是编译器不知道应该传递什么参数而导致无法调用而引起编译失败

```
class D:public B
{
public D(int){} // 没有显式调用基类的构造函数
}//在基类中无法找到无参的构造函数而引起编译失败

//正确写法:
class D : public B
{
  public: D(int b): B(b){}
}
```

4>.继承体系中:派生类对象构造和析构时--构造和析构的次序

```
class B
{
  public:
    B(){}
    ~B(){}
};

class D:public B
{
   public:
     D():B(){}
     ~D(){}
```

```
};

void TestFunc()
{
    D d;
   //创建那个类的对象，编译器就会调用那个类的构造函数
   //析构那个类的对象，编译器就会调用那个类的析构函数
}
```

函数调用次序: D()--->初始化列表位置调用B(),并执行完成-->执行派生类构造函数的函数体()

打印:先去打印B()---->打印D()

析构:~D()

```
    {
        //先释放派生类自己的资源
        //编译器会在派生类析构函数最后一条调用语句之后插入以下代码
        call ~B();
     }
```

6、那些成员可以被子类继承：

1>基类中普通的成员函数和成员变量都会别继承的子类中----代码复用

2>静态成员变量---会被子类继承，并且在整个继承体系中只有一份

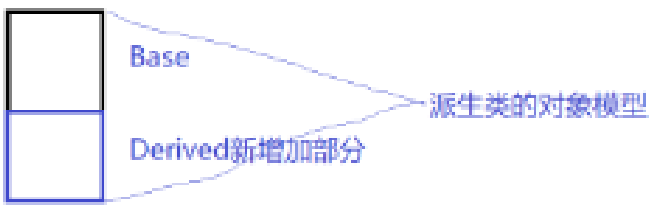
3>友元：因为友元不是类的成员，因此友元函数不能被子类继承

4>构造函数、拷贝构造函数、赋值运算符重载、析构函数是否被子类继承?

7、不同继承方式下派生类的对象模型

对象模型：对象中各个成员变量在内存中的存储方式

单继承：



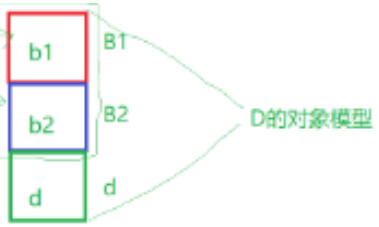
多继承：一个类可以有多个基类(类似：自己又亲爹。也有干爹)

注意：每个基类前都必须增加继承权限，否则就是默认的继承权限

```
class B1
{
public:
   int b1;
};
```

```
class B2
{
public:
   int b2;
};
```

```
class D : public B1, public B2
{
public: int d;
};
```



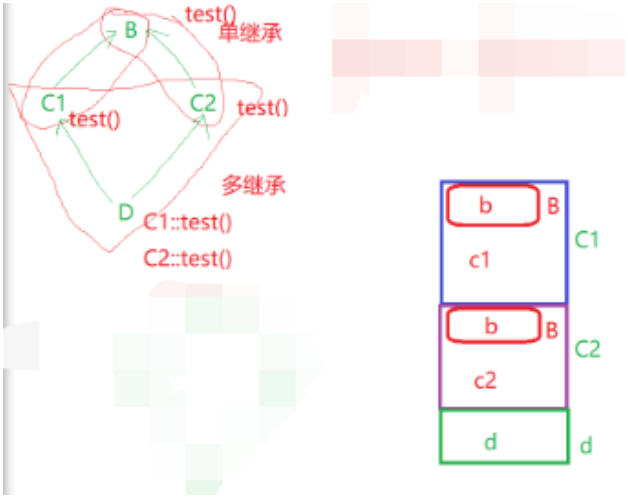
菱形继承：单继承+多继承复合起来

```
class B
{
public:int b;
};
```

```
class C1:public B
{
public: int c1;
};
```

```
class C2:public B
{
public: int c2;
};
```

```
class D:public C1, public C2
{
public: int d;
};
```



1>必须要会计算派生类的大小

2>能够画出派生类对象模型图解

3>派生类对象中将最顶层基类中成员存储了两份

4>缺陷：

```
D d;
d.c1=1;//没有任何问题
d.b=2;//将会编译失败
```

1、b在整个派生类模型中有两份，C1、C2；如果通过派生类对象之间访问最顶层基类中的成员（成员 变量、成员函数），编译器不知道是应该访问C1基类中继承的b，还是应该访问从C2基类中继

承的b，即：菱形继承的二义性问题

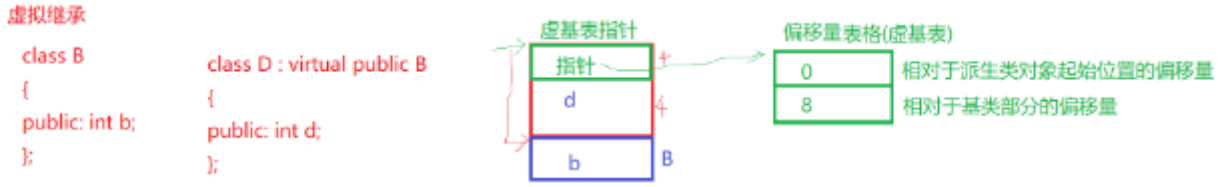
2、浪费空间

----关于菱形继承二义性问题的解决：

1.从表层去解决--最顶层成员在派生类中有两份，直接通过派生类对象访问时最终会造成派生类对象不知道应该访问那一个，使访问明确化 d.C1::b= 1; d.C2:test();

代码可以通过编译,但是最顶层成员变量在派生类对象中仍旧是有多份---浪费空间

2.从核心上解决---如果能够将最顶层成员变量在派生类对象中只存储一份---二义性问题解决、空间浪费解决：采用菱形虚拟继承的方式解决



虚拟继承和普通继承方式:

1.派生类的对象模型是倒立:派生类部分在上,基类部分在下

2.虚拟继承的对象模型中多了4个字节:保存虚基表指针--->偏移量表格(虚基表)之

3.通过派生类对象访问基类成员的不同:

普通继承方式:直接访问

菱形继承:

d.b= 1;

mov eax,dword ptr [d]_ // 取对象前4个字节中的内容---拿到了虚基表的地址

mov ecx,dword ptr [eax+4]//获取虚基表指向向后偏移4个字节之后的空间中的内容:即获取相对于基类部分的偏移量

mov dword ptr dlecx].1//赋值:将d对象起始地址向后偏移ecx(8)个字节,即基类中的成员b

4.普通的继承方式:编译器可能会为派生类生成默认的构造函数

虚拟继承方式:编译器-定会为派生类生成默认的构造函数---原因: 因为在创建派生类对象时，编译器

必须要将虚基表指针填写在对象前

4个字节，而该步操作必须在创建

对象期间完成；因此:该步骤不能

在构造函数中完成

类和阶段:

语法---如果一个类没有显式定义自己的构造函数,编译器将会生成一份默认的无参构造函数

实际情况:编译器可能没有严格按照语法去做，构造函数是否会一定会生成---结论: 不一定

生成的条件:如果编译器感觉自己需要，就会生成一份默认的构造函数

以下4种情况，编译器一定会生成默认的构造函数

1.类和对象阶段:

如果A类定义了无参的构造函数或者全缺省的构造函数

B类没有显式定义构造函数,但是B类中包含了一个A类的对象，编译器一定会给B类生成一份默认的构造函数

2.继承体系中:

如果基类显式定义无参的构造函数或者全缺省的构造函数，派生类没有显式定义构造函数，编译器一定会给派生类生成一份默认的构造函数

目的:为了调用基类构造函数以完成基类部分成员的初始化

3.虚拟继承中:

编译器一定会为派生类生成默认构造函数

目的:在构造派生类对象时，需要将虚基表的地址填写在对象的前4个字节中

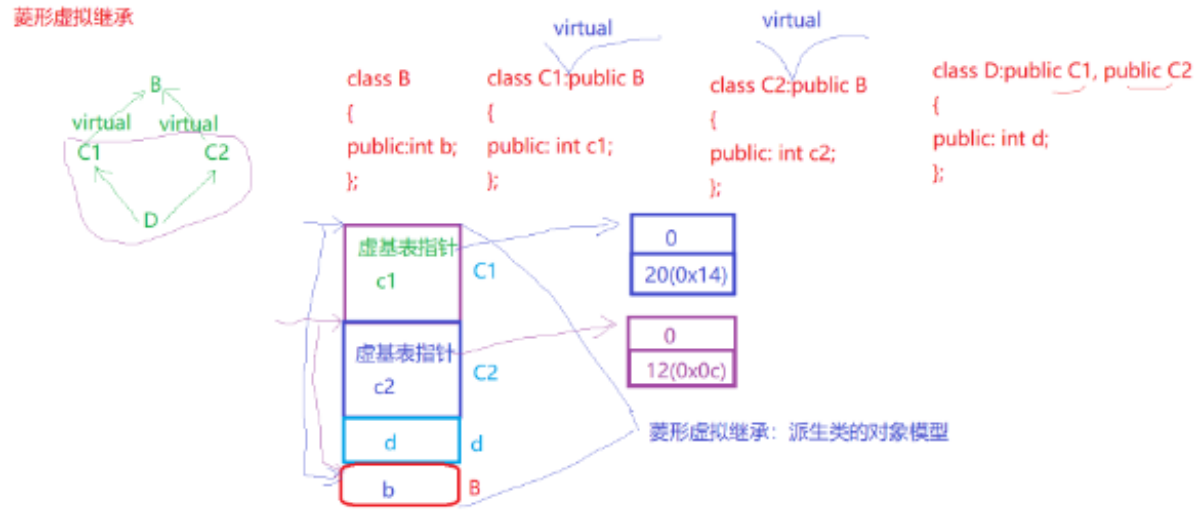
4.包含有虚函数的类:

编译器-定会为派生类生成默认构造函数

目的:在构造派生类对象时，需要将虚表的地址填写在对象的前4个字节中

面试常问问题:你平时是怎么学习的?都看过那些书?

菱形虚拟继承:



因为：菱形虚拟继承中，最顶层基类B的成员部分在派生类对象中只存储了一份，因此就不会存在二义性问题

```
D d;
d.b=1;//可以通过编译
C1& c1=d;
C2& c2=d;
c1.b=2;
```

c2.b=3;//c1和c2访问的是派生类对象中的同一个b

通过虚基表中的偏移量访问最顶层B类中的成员
d.b= 1;
mov　　eax,dword ptr [d]_ // 取对象前4个字节中的内容---拿到了虚基表的地址
mov　　ecx,dword ptr [eax+4]//获取虚基表指向向后偏移4个字节之后的空间中的内容:即获取相对于基类部分的偏移量
mov　　dword ptr dlecx].1//赋值:将d对象起始地址向后偏移ecx(8)个字节,即基类中的成员b