1

阅读C++:94—类继承（菱形继承、虚继承(virtual虚基类)），请自行编写简单的具有菱形继承的C++程序（B和C从A继承，D从B、C继承）

1）练习用nm和demangle的方法分析程序中的2个名字的改编；

2）试分析在虚继承和非虚继承下的对象布局和方法表布局，并给出你的分析方法。

注：请将相关源码、汇编码以及解答文件打包提交。

(1)

- 首先`g++ demangle.cpp -o demangle`生成可执行文件，然后`nm demangle > nm1.txt`以及`nm -C demangle > nm2.txt`分别生成mangled符号表信息以及demangled符号表信息。
- ```
  //demangled name
  00000000000012ae W A::getMa()
  0000000000001292 W A::A(int)
  0000000000001292 W A::A(int)
  00000000000012c2 W B::B(int, int)
  00000000000012c2 W B::B(int, int)
  00000000000012f6 W C::C(int, int)
  00000000000012f6 W C::C(int, int)
  0000000000001384 W D::func()
  000000000000132a W D::D(int, int, int, int)
  000000000000132a W D::D(int, int, int, int)
  ```
- ```
  //mangled name
  00000000000012ae W _ZN1A5getMaEv
  0000000000001292 W _ZN1AC1Ei
  0000000000001292 W _ZN1AC2Ei
  00000000000012c2 W _ZN1BC1Eii
  00000000000012c2 W _ZN1BC2Eii
  00000000000012f6 W _ZN1CC1Eii
  00000000000012f6 W _ZN1CC2Eii
  0000000000001384 W _ZN1D4funcEv
  000000000000132a W _ZN1DC1Eiiii
  000000000000132a W _ZN1DC2Eiiii
  ```
- 参考课上ppt所讲来对名字对应关系分析
  - `A::getMa()`对应`_ZN1A5getMaEv`： _Z为mangled符号前缀；N为nested names的前缀，后跟<length,id>对，以E作结束标志。1A表示作用域A及其名称长度，5getMa表示函数名getMa及其名称长度；最后v表示没有参数，为void
  - `D::D(int, int, int, int)`对应`_ZN1DC1Eiiii`以及`_ZN1DC2Eiiii` 同样分析，1D为构造函数的名称及长度，C1/C2为编号（区分不同的D对象），iiii对应4个int型的参数。
  - 其余名字分析基本类似，这里按题目要求选取两个名字做了分析。
  - 对于issue里提出的构造函数都有两个的情况，如

    ```
    0000000000001292 W _ZN1AC1Ei
    0000000000001292 W _ZN1AC2Ei
    ```

个人猜测是因为在main函数里创建对象时两个符号分别对应一个声明的类，一个实际创建的实例。

(2)

- 查阅资料发现使用`Visual Studio`的`Developer Command Prompt for VS 2019`中提供的命令`cl [filename].cpp /d1reportSingleClassLayout[className]`能看到对象内存分布情况。
- 本题目主要考察对于虚函数在虚继承和非虚继承下的差异，所以考虑A里实现FuncA，B里实现FuncB，C里实现FuncC，并且B,C,D都重写了各自父类的虚函数。
- 注意到在非虚函数的情况下，C++的处理是静态多态，或静态链接：函数调用在程序执行前就准备好了（如函数地址等），称为早绑定。而添加了virtual关键字的函数，则会因为对象实例的运行时类型而动态绑定，这需要虚函数表来辅助寻址。再进一步，多态性质要求子类要能够有效的产生父类对象的视图。

非虚继承

`1.cpp`

```
class A size(8):
        +---
 0      | {vfptr}
 4      | a
        +---

A::$vftable@:
        | &A_meta
        |  0
 0      | &A::FuncA

A::FuncA this adjustor: 0
```

A中主要是虚表指针以及成员变量a，虚方法表里则包含虚函数FuncA()

```
class B size(12):
        +---
 0      | +--- (base class A)
 0      | | {vfptr}
 4      | | a
        | +---
 8      | b
        +---

B::$vftable@:
        | &B_meta
        |  0
 0      | &B::FuncA
 1      | &B::FuncB

B::FuncA this adjustor: 0
B::FuncB this adjustor: 0
```

正如书上所讲，类B对象b的A视图包含两部分：b的前一部分域(虚函数表指针，变量a)以及b引用的虚方法表的前一部分（FuncA函数地址）。在A视图之后才是分别紧跟B中变量b以及虚函数表中的函数FuncB()

```
class C size(12):
        +---
 0      | +--- (base class A)
 0      | | {vfptr}
 4      | | a
        | +---
 8      | c
        +---

C::$vftable@:
        | &C_meta
        |  0
 0      | &C::FuncA
 1      | &C::FuncC

C::FuncA this adjustor: 0
C::FuncC this adjustor: 0
```

C类与B类分析一致

```
class D size(28):
        +---
 0      | +--- (base class B)
 0      | | +--- (base class A)
 0      | | | {vfptr}
 4      | | | a
        | | +---
 8      | | b
        | +---
12      | +--- (base class C)
12      | | +--- (base class A)
12      | | | {vfptr}
16      | | | a
        | | +---
20      | | c
        | +---
24      | d
        +---

D::$vftable@B@:
        | &D_meta
        |  0
 0      | &D::FuncA
 1      | &D::FuncB

D::$vftable@C@:
        | -12
 0      | &thunk: this-=12; goto D::FuncA
 1      | &D::FuncC

D::FuncA this adjustor: 0
D::FuncB this adjustor: 0
D::FuncC this adjustor: 12
```

由于是非虚继承，所以C++里实现是独立的多重继承结构，A类对象在D中存在两个副本。并且内存排布上则是依次为B类、C类的内存排布（与前面B，C的内存排布模式一致），最后才是D类自己的成员变量d。再看到两个虚函数表，因为D类对象需要能产生B类对象视图、C类对象视图，所以需要有各自的表，这也与书上对方法表的分布式存储一致(书p412图12.10),即对于`vftable@B`其中应该是装有FuncA和FuncB的指针，并且均是D中的覆盖实现。同理`vftable@C`里有FuncA，FuncC的指针，也是D中的覆盖实现。但是注意到分布式存储的好处在于，对于B和C视图里相同实现的方法(如FuncA)，是共用一个地址，C的表里通过goto跳转到B视图里的FuncA处。

此外，还应该注意到，`D::$vftable@C@`下方的-12表示指向这个虚表的虚指针vptr在D的内存布局里的偏移。

而C视图下对于FuncC的引用则通过`D::FuncC this adjustor: 12`来找到对应的vptr

虚继承

`2.cpp`

```
class A size(8):
        +---
 0      | {vfptr}
 4      | a
        +---

A::$vftable@:
        | &A_meta
        |  0
 0      | &A::FuncA

A::FuncA this adjustor: 0
```

A是虚基类，与前面无区别。

```
class B size(20):
        +---
 0      | {vfptr}
 4      | {vbptr}
 8      | b
        +---
        +--- (virtual base A)
12      | {vfptr}
16      | a
        +---

B::$vftable@B@:
        | &B_meta
        |  0
 0      | &B::FuncB

B::$vbtable@:
 0      | -4
 1      | 8 (Bd(B+4)A)

B::$vftable@A@:
        | -12
 0      | &B::FuncA
```

```
B::FuncA this adjustor: 12
B::FuncB this adjustor: 0
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
               A      12       4       4 0
```

C++里的虚继承的实现方式是共享的重复继承，所以在内存排布上发生了变化，A类函数的重写与B中新的虚函数是用两个虚函数表分开存放的。从上到下依次为B中函数的vfptr，vbptr, B中成员变量b，A中函数的vfptr，A中成员变量a。这里需要特别说明vbtable的作用：存放B的内存分布里两个虚表指针相对于vbptr的偏移。这个额外的开销的好处在于能通过vbtable快速找到对应的A视图和B视图的起始位置（即vfptr的位置）

```
class C size(20):
        +---
 0      | {vfptr}
 4      | {vbptr}
 8      | c
        +---
        +--- (virtual base A)
12      | {vfptr}
16      | a
        +---

C::$vftable@C@:
        | &C_meta
        |  0
 0      | &C::FuncC

C::$vbtable@:
 0      | -4
 1      | 8 (Cd(C+4)A)

C::$vftable@A@:
        | -12
 0      | &C::FuncA

C::FuncA this adjustor: 12
C::FuncC this adjustor: 0
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
               A      12       4       4 0
```

C的分析同B

```
class D size(36):
        +---
 0      | +--- (base class B)
 0      | | {vfptr}
 4      | | {vbptr}
 8      | | b
        | +---
12      | +--- (base class C)
12      | | {vfptr}
16      | | {vbptr}
20      | | c
        | +---
24      | d
        +---
```

```
        +--- (virtual base A)
28      | {vfptr}
32      | a
        +---

D::$vftable@B@:
        | &D_meta
        |  0
 0      | &D::FuncB

D::$vftable@C@:
        | -12
 0      | &D::FuncC

D::$vbtable@B@:
 0      | -4
 1      | 24 (Dd(B+4)A)

D::$vbtable@C@:
 0      | -4
 1      | 12 (Dd(C+4)A)

D::$vftable@A@:
        | -28
 0      | &D::FuncA

D::FuncA this adjustor: 28
D::FuncB this adjustor: 0
D::FuncC this adjustor: 12
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
               A      28       4       4 0
```

因为虚继承的存在，所以D的内存布局里只有一份A， 内存排布为B、C，D中成员变量d，A。共有3个vfptr以及2个vbptr。并且在B视图(C视图同理)下都有vbptr，里面存放的是B视图下的FuncB以及FuncA各自对应的vfptr相对于vbptr位置的偏移。由此可见，虚继承实现里减少了虚基类重复继承时空间占用（只有一份副本），但是相应增加了一些虚函数表及指针来寻址。

2

下面是关于内联inline的程序及其在x86/Linux下用gcc编译的示例

汇编码如下。gcc -O2: 更加优化，GCC执行几乎所有受支持的优化。

```
;-O2
pushl   %ebp
movl    %esp, %ebp
andl    $-16, %esp
subl    $16, %esp
movl    $1, 8(%esp);a的存储空间
movl    $.LC0, 4(%esp)
movl    $1, (%esp)
call    __printf_chk
movl    $1, (%esp)
call    f1
leave
ret
```

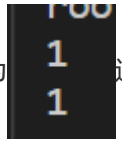
```
pushl   %ebp
movl    %esp, %ebp
andl    $-16, %esp
subl    $32, %esp
movl    $1, 28(%esp)
movl    28(%esp), %eax;进行参数入栈
movl    %eax, (%esp)
call    f
movl    $.LC0, %eax
movl    28(%esp), %edx
movl    %edx, 4(%esp)
movl    %eax, (%esp)
call        printf
movl    28(%esp), %eax
movl    %eax, (%esp)
call    f1
leave
ret
```

(1)

试从产生的汇编代码总结gcc处理inline的特征，内联是在编译的什么阶段被处理?

- 资料查阅：GCC的static inline定义很容易理解：你可以把它认为是一个static的函数，加上了inline的属性。这个函数大部分表现和普通的static函数一样，只不过在调用这种函数的时候，gcc会在其调用处将其汇编码展开编译而不为这个函数生成独立的汇编码（某些特殊情况除外）

  static函数：函数的定义和声明默认情况下是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。c语言里的静态函数会被自动分配在一个一直使用的存储区，直到退出应用程序实例，避免了调用函数时压栈出栈，速度快很多。

- gcc处理inline特征：在其调用处将其汇编码展开编译而不为这个函数生成独立的汇编码，从而减少函数调用、返回带来的开销。从执行结果来看，生成的可执行文件输出为 

这与实际相符，因为f()和f1()的调用并不会影响main里对printf的输出。

  此外，还需注意到，实际上在给出的-O2的汇编码里，并没有f(a)的调用痕迹，这是因为在inline函数在汇编码里被展开后，分析发现其代码并没有实际效果，认为是死代码，在优化时被gcc删除了。

- 内联是在语义分析阶段被处理的。

(2)

试说明编译器进行了哪些优化而得到带`-O2`选项生成的汇编码。

- 栈上空间分配的优化：优化前栈上分配空间`subl $32, %esp`32字节，优化后`subl $16, %esp`为16字节。
- 对参数压栈代码的优化

  优化前需要把调用printf（对f1函数的调用同理）所需的字符串和a的值先分别放到eax和edx寄存器，然后再入栈

  ```
  movl    $.LC0, %eax
  movl    28(%esp), %edx
  movl    %edx, 4(%esp)
  movl    %eax, (%esp)
  ```

  优化后则直接一步到位。

  ```
  movl    $.LC0, 4(%esp)
  movl    $1, (%esp)
  ```

- 对内联函数的优化

  先做内联展开，然后分析发现内联函数是死代码，所以直接删除。

(3)

如果将`inline.h`第1行的`static`去掉，执行`gcc inline.c inline1.c`，产生如下错误，试说明原因，并指出这是在编译的哪个阶段产生的错误。

```
/tmp/ccv6AbOz.o: In function `f':
inline1.c:(.text+0x0): multiple definition of `f'
/tmp/cce1rioR.o:inline.c:(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

- 链接阶段产生错误（由错误信息ld）
- 在预处理阶段，inline.c和inline1.c都把inline.h的内容展开到程序正文。尽管都做了内联，但是在链接时会由于两个文件都有f(a)的定义而引起冲突。用static修饰时则f函数只在声明他的文件当中可见，不能被其他文件所用，由此避免了冲突。