HW10

1

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。

1-1

先通过`gcc diamond_inherit.cpp -o 1`生成可执行文件1，再分别通过命令`nm 1 > nm.txt`和`nm 1 --demangle  > nm_de.txt`生成关于可执行文件1的符号表信息，查看两个符号表文件，可以看到，`nm.txt`部分

```
000000000000118c W _ZN1AC1Eiii
000000000000118c W _ZN1AC2Eiii
00000000000011c2 W _ZN1BC1Ev
00000000000011c2 W _ZN1BC2Ev
00000000000011fc W _ZN1CC1Ev
00000000000011fc W _ZN1CC2Ev
0000000000001236 W _ZN1DC1Ev
0000000000001236 W _ZN1DC2Ev
```

`nm_de.txt`部分

```
000000000000118c W A::A(int, int, int)
000000000000118c W A::A(int, int, int)
00000000000011c2 W B::B()
00000000000011c2 W B::B()
00000000000011fc W C::C()
00000000000011fc W C::C()
0000000000001236 W D::D()
0000000000001236 W D::D()
```

从上面可以看出名字改编后的对应关系

我在`cpp`代码中只创建了一个D类型的实例，个人猜测这里每个类都有两个符号对应是因为一个是声明的类，一个是实际创建的类

对于第一个`A::A(int, int, int)`，改编后的名字为`_ZN1AC1Eiii`，`_Z`为mangled符号的前缀，`N`为nested names的前缀，`1`表示名称A长度为1，后面`C1`为A的编号（用以区分两个不同的A），`E`为结束标识，`iii`表示参数类型为三个int类型

对于第一个`D::D`，改编后的名字为`_ZN1AC1Eiii`，`_Z`为mangled符号的前缀，`N`为nested names的前缀，`1`表示名称A长度为1，后面`C1`为D的编号（用以区分两个不同的D），`E`为结束标识

1-2

首先明确一点，成员函数被看作类作用域的全局函数,不在对象分配的空间里,只有虚函数才会在类对象里有一个指针,存放虚函数的地址等相关信息。原因如下

- 虚方法表是用来实现方法的动态绑定的。类中的虚方法可以被派生类重写，导致执行方式被派生类改变，因而出现动态绑定问题。
- 对于非虚方法，无论被其所在类的实例调用，还是被这个类的派生类的实例调用，方法的执行方式都不变，因而没有动态绑定的问题。
- 成员函数的地址，编译期就已确定，并静态绑定或动态的绑定在对应的对象上。对象调用成员函数时，早在编译期间，编译器就可以确定这些函数的地址，并通过传入this指针和其他参数，完成函数的调用，所以类中就没有必要存储成员函数的信息。

且存放虚方法表的指针放在对象分配空间的第一个域，用以产生其超类的视图

故以下主要研究虚方法的虚继承和非虚继承

- 采用`Visual Studio 2019 Command Prompt`的cl命令的`/d1reportSingleClassLayout`选项查看C++文件中对象内存分布情况

```
cl [filename].cpp /d1reportSingleClassLayout[className]
```

1. 虚方法的非虚继承

   每个类中都共同定义虚方法`VirtualFunction`，各自分别定义虚方法`display_N`（N为A、B、C、D）

   A:

```
class A size(16):
    +---
 0  | {vfptr}
 4  | m_data1
 8  | m_data2
12  | m_data3
    +---

A::$vftable@:
    | &A_meta
    |  0
 0  | &A::display_A
 1  | &A::VirtualFunction
```

   A中先是一个虚表指针，然后是其中定义的三个整型变量`m_data1 m_data2 m_data3`。

   虚方法表中依次包含了`display_A`和`VirtualFunction`

   B:

```
class B size(20):
    +---
 0  | +--- (base class A)
 0  | | {vfptr}
 4  | | m_data1
 8  | | m_data2
12  | | m_data3
    | +---
16  | m_b
```

```
    +---

B::$vftable@:
    | &B_meta
    |  0
 0  | &A::display_A
 1  | &B::VirtualFunction
 2  | &B::display_B
```

B继承了父类A，内存排布是先父类后子类，父类的分布和A相同，后面紧跟自身的整型成员变量`m_b`

虚方法表中依次包含了`display_A`（在类A中定义）`VirtualFunction`（在类B中定义）`display_B`（在类B中定义）

C:

```
class C size(20):
    +---
 0  | +--- (base class A)
 0  | | {vfptr}
 4  | | m_data1
 8  | | m_data2
12  | | m_data3
    | +---
16  | m_c
    +---

C::$vftable@:
    | &C_meta
    |  0
 0  | &A::display_A
 1  | &C::VirtualFunction
 2  | &C::display_C
```

同类B，不再赘述

D:

```
class D size(44):
    +---
 0  | +--- (base class B)
 0  | | +--- (base class A)
 0  | | | {vfptr}
 4  | | | m_data1
 8  | | | m_data2
12  | | | m_data3
    | | +---
16  | | m_b
    | +---
20  | +--- (base class C)
20  | | +--- (base class A)
20  | | | {vfptr}
24  | | | m_data1
28  | | | m_data2
32  | | | m_data3
    | | +---
```

```
36  | | m_c
    | +---
40  | m_d
    +---

D::$vftable@B@:
    | &D_meta
    |  0
 0  | &A::display_A
 1  | &D::VirtualFunction
 2  | &B::display_B
 3  | &D::display_D

D::$vftable@C@:
    | -20
 0  | &A::display_A
 1  | &thunk: this-=20; goto D::VirtualFunction
 2  | &C::display_C
```

依次排布着继承的两个父类B、C，然后是自身的成员变量`m_d`。

B中包含了其成员变量`m_b`，以及A，A有一个0地址偏移的虚表指针，然后是成员变量`m_data1 m_data2 m_data3`；C的内存排布类似于B，且C中也有一份A

两个虚表，分别针对B和C，在&D_meta下方的数字是首地址偏移量，靠下面的虚表的那个-20表示指向这个虚表的虚指针的内存偏移，这正是C中的`vfptr`在D中的内存偏移。可以看到两个虚表中共包含了两个`display_A`（A中定义）、B和C各自的`display`（分别在B、C中定义）、`display_D`（D中定义），且在两个虚表中都包含了D中定义的`VirtualFunction`（因为父类B、C中也包含了`VirtualFUnction`），只是第一个表中直接给出了对应的地址，第二个表中通过goto间接给出了地址

2. 虚方法的虚继承

A：

```
class A size(16):
    +---
 0  | {vfptr}
 4  | m_data1
 8  | m_data2
12  | m_data3
    +---

A::$vftable@:
    | &A_meta
    |  0
 0  | &A::display_A
 1  | &A::VirtualFunction
```

和之前的A没什么区别，不再赘述

B:

```
class B size(32):
    +---
 0  | {vfptr}
 4  | {vbptr}
 8  | m_b
```

```
    +---
12  | (vtordisp for vbase A)
    +--- (virtual base A)
16  | {vfptr}
20  | m_data1
24  | m_data2
28  | m_data3
    +---

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

B::$vbtable@:
 0  | -4
 1  | 12 (Bd(B+4)A)

B::$vftable@A@:
    | -16
 0  | &A::display_A
 1  | &(vtordisp) B::VirtualFunction
```

这里发生了变化，原来是先排虚表指针与A成员变量，vfptr位于0地址偏移处，再排`m_b`；但这里的排布是先排B的虚表指针和成员变量`m_b`，再排A的虚表指针和三个成员变量

第一个vfptr是这个B对应的虚表指针，它指向vftable@B，包含了`display_B`（B中定义），第二个vbptr是这个B对应的虚表指针，它指向B的虚表vbtable，第三个vfptr是虚基类表A对应的虚指针，它指向vftable@A，其中包含了`display_A`（A中定义）、`VirtualFunction`（B中定义）

其中`vtordisp`的出现是因为派生类重写了虚基类的虚函数，且在派生类中定义了构造函数

C:

```
class C size(32):
    +---
 0  | {vfptr}
 4  | {vbptr}
 8  | m_c
    +---
12  | (vtordisp for vbase A)
    +--- (virtual base A)
16  | {vfptr}
20  | m_data1
24  | m_data2
28  | m_data3
    +---

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

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

C::$vftable@A@:
```

```
    | -16
 0  | &A::display_A
 1  | &(vtordisp) C::VirtualFunction
```

同B，不再赘述

D：

```
class D size(48):
    +---
 0  | +--- (base class B)
 0  | | {vfptr}
 4  | | {vbptr}
 8  | | m_b
    | +---
12  | +--- (base class C)
12  | | {vfptr}
16  | | {vbptr}
20  | | m_c
    | +---
24  | m_d
    +---
28  | (vtordisp for vbase A)
    +--- (virtual base A)
32  | {vfptr}
36  | m_data1
40  | m_data2
44  | m_data3
    +---

D::$vftable@B@:
    | &D_meta
    |  0
 0  | &B::display_B
 1  | &D::display_D

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

D::$vbtable@B@:
 0  | -4
 1  | 28 (Dd(B+4)A)

D::$vbtable@C@:
 0  | -4
 1  | 16 (Dd(C+4)A)

D::$vftable@A@:
    | -32
 0  | &A::display_A
 1  | &(vtordisp) D::VirtualFunction
```

可以看到虚继承在这里发挥了重要作用，D的内存布局中只有一份A，整体排布为B、C（均不包含A），然后是D中的成员变量`m_d`，然后是A。

共有5个虚指针。整体来看先排B的虚表指针和成员变量`m_b`，再排C的虚表指针和成员变量`m_c`，最后排A的虚表指针和三个成员变量。其余类似

3. 可以看到，虚继承的作用是减少了基类的重复，代价是增加了虚表指针的负担（需要更多的虚表指针）。
4. 总的来看，对于基类有虚函数的情况，每个子类都有虚表指针和虚表。
    - 如果是非虚继承，那么子类将父类的虚指针继承下来，并指向自身的虚表（发生在对象构造时）。有多少个虚函数，虚表里面的项就会有多少。多重（菱形）继承时，可能存在多个基类虚表与虚指针
    - 如果是虚继承，那么子类会有两份虚指针，一份指向自己的虚表，另一份指向基类的虚表，多重（菱形）继承时虚基表与虚基表指针有且只有一份

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：

inline.h

inline1.c

inline.c

用 gcc –S –m32 -O2 编译得到的`main`函数的汇编码如下（省去部分不重要的地方）

```
pushl    %ebp
movl    %esp, %ebp
andl    $-16, %esp
subl    $16, %esp       #这里只分配了16个字节

movl    $1, 8(%esp)

movl    $.LC0, 4(%esp)
movl    $1, (%esp)
call    __printf_chk

movl    $1, (%esp)
call    f1

leave
ret
```

用 gcc –S -m32 编译得到的`main`的汇编码如下

```
pushl    %ebp
movl    %esp, %ebp
andl    $-16, %esp
subl    $32, %esp

movl    $1, 28(%esp)

movl    28(%esp), %eax
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？

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。

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
```

关于gcc -O2的官方解释：更加优化。GCC执行几乎所有受支持的优化，这些优化不涉及空间速度的权衡。与-O相比，该选项增加了编译时间和生成代码的性能

2-1

`inline`的特征：将短小的函数定义为内联函数，在编译时，编译器会把内联函数的代码副本放置在每个调用该函数的地方，从而减少函数调用、返回所带来的开销。编译器会对inline函数做类型、安全检查，更加安全可靠。但内联本质上是以增加空间的消耗为代价的。

阶段：语义分析阶段

从题目中给出的代码难以看出这点，因为对f()的调用没有产生实际效果（被认为是死代码），直接被删掉了，故在汇编码中看不到内联的展开，我重写了一个`add.c`，从生成的汇编码`add.s`中可以看出其中的函数`add(int a, int b)`在主函数的调用处被展开（通过gcc -S -O2进行优化）

```
main:
.LFB1:
    .cfi_startproc
    endbr64
    movl    c(%rip), %eax
    addl    d(%rip), %eax
    ret
    .cfi_endproc
```

2-2

- 栈上空间分配的优化，减少无用空间的分配。相较于未优化的28字节，优化过后程序只在栈上分配了16个字节
- 对内联函数f()的优化，由于函数f()所进行的操作对`inline.c`中后续的操作没有任何影响（被认为是死代码），故直接将其删除
- 对函数printf调用的优化，用`__printf_chk`代替对`printf`的调用，所需的参数更少
- 函数f1()调用前传参方式的优化，原本需要先将1从`28(%esp)`取出到eax寄存器，再进行传参

  ```
  movl    28(%esp), %eax
  movl    %eax, (%esp)
  call    f1
  ```

  优化后直接将常数1当作参数传入，减少了寄存器的存取操作

  ```
  movl    $1, (%esp)
  call    f1
  ```

2-3

链接阶段产生错误

`inline.c`和`inline1.c`中都生成了f()的代码。虽然`inline.c`和`inline1.c`都内联了f()，所以函数f()实际上是没用到的。但两份 f()的代码还是会引起链接冲突。用`static`修饰可以保证其作用域只在编译单元内，这就避免了冲突