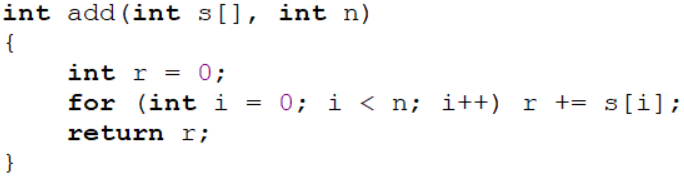
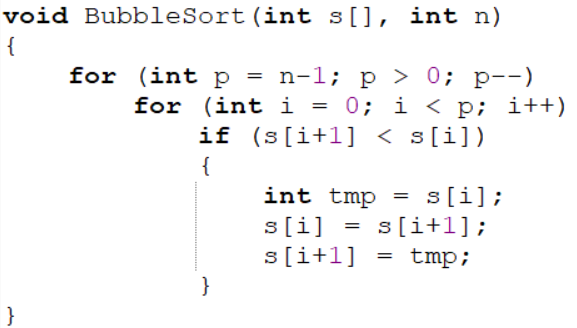
作业二

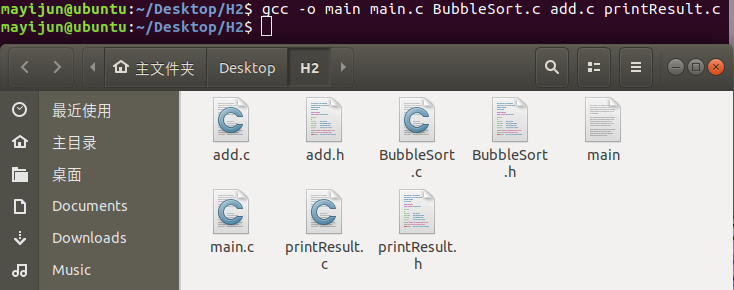
马逸君 17300180070

一、补全代码

如图所示。

二、使用不同的命令编译成可执行文件

（1）使用gcc一步生成可执行文件

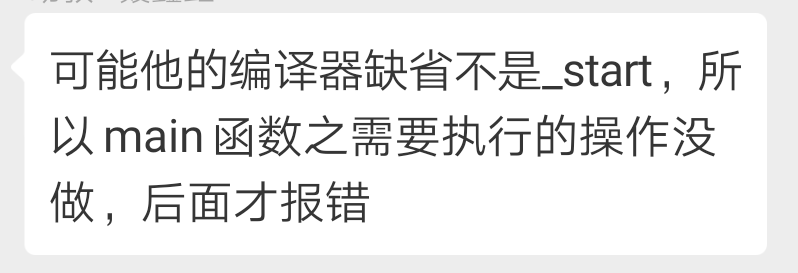
结果如示。

（2）先得到可重定位目标文件，再自行链接得到可执行目标文件

这一部分有两个值得讨论的问题：

其一是作业文档中要求加上的“-e main”参数。我在读题时就注意到了这个参数，并通过百度查到这个参数的作用是指定程序执行的入口点。所以当程序在第一次链接完成后执行出现段错误时，我马上去掉这个参数，程序恢复正常。

我和助教在群里讨论了这个问题，助教说，如果“-e main”不对的话，就改成“-e \_start”试试。我尝试，程序运行正常。说明程序有时候并不是以main作为入口的，可能包含一个\_start代码段，进行一些在执行main()函数之前的准备工作，然后再调用main()。

助教说，加上这个参数的原因是，有的链接器缺省不一定能找到正确的入口，所以需要时可以用这个参数显式指明入口。

其二是链接时文件名顺序的问题。之前的课上讲过，链接时，有包含关系的文件名必须满足：被包含的文件在包含文件的后面。所以我特意尝试了一下把main.o放到最后，想看看会报什么错误。但令我惊讶的是，没有报错。

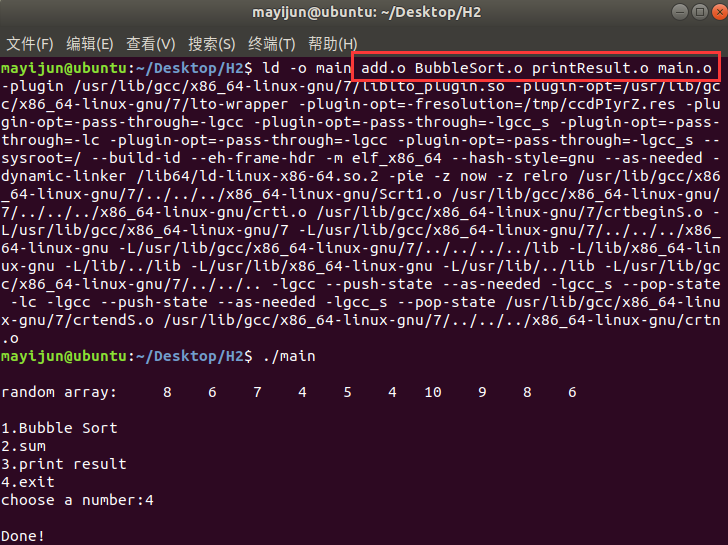
我到百度上查明了原因：

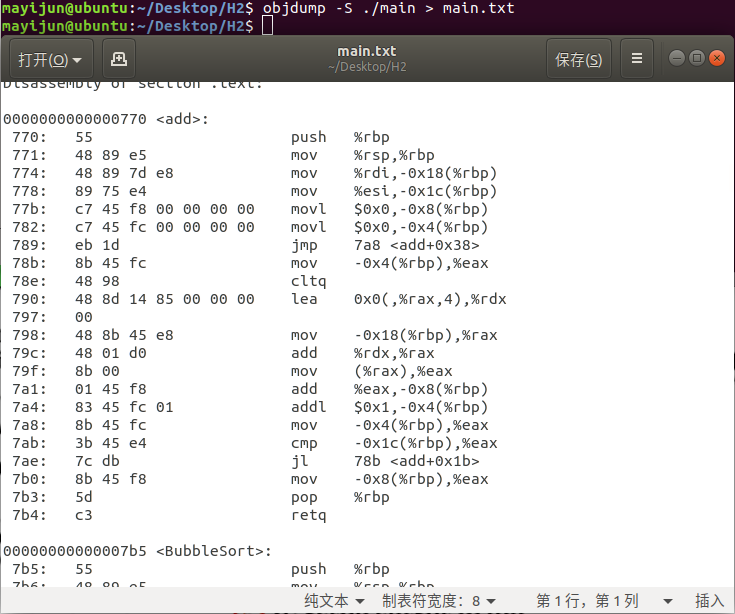
On Unix-like systems, the traditional behavior of compilers and linkers is to search for external functions **from left to right** in the object files specified on the command line. This means that the object file which contains the definition of a function should appear after any files which call that function.

……Most **current** compilers and linkers will search all object files, **regardless of order**, but since not all compilers do this it is best to follow the convention of ordering object files from left to right.

——*An introduction of gcc*

所以，在我的电脑上没有报错的原因是，现在的编译器和链接器大都非常人性化地加入了可以在前后所有文件里搜索符号定义的功能。但因为这一功能并非编译器和链接器规范的一部分，所以不是所有的编译器都支持，所以我们当前还是有必要按顺序书写文件名的。



三、反汇编

结果如示。

我用man objdump命令查阅了参数-S的含义：Display source code intermixed with disassembly, if possible. 意思应该是说把能确定源代码的内容显示成源代码的形式，不能的显示成反汇编指令的形式。

但是这样得到的反汇编结果中并没有显示源代码，按照它的说明，这应该表示没有可以确定的源代码。只有在编译时添加-g选项，反汇编时才会显示出一些函数名和循环体。百度和google上的资料很少，我只能自己脑补：可能反汇编生成源码与符号表有关。

必答题：

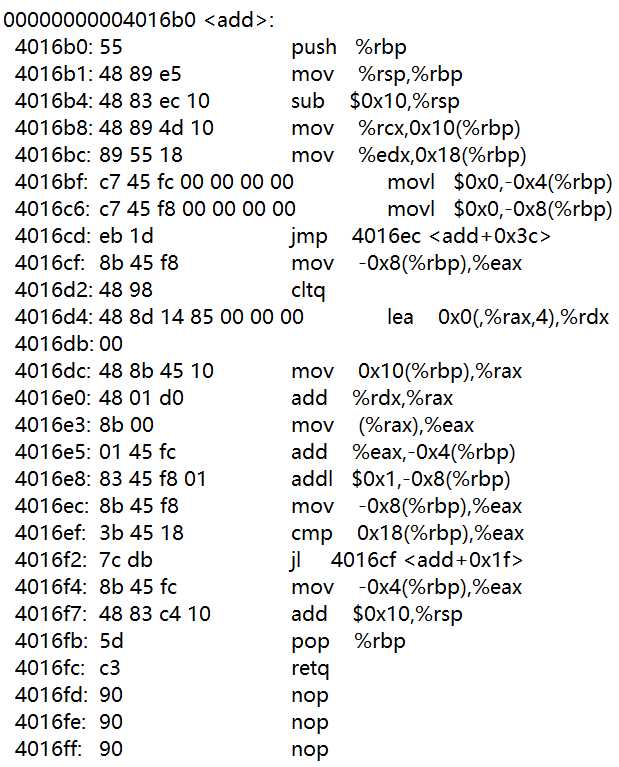
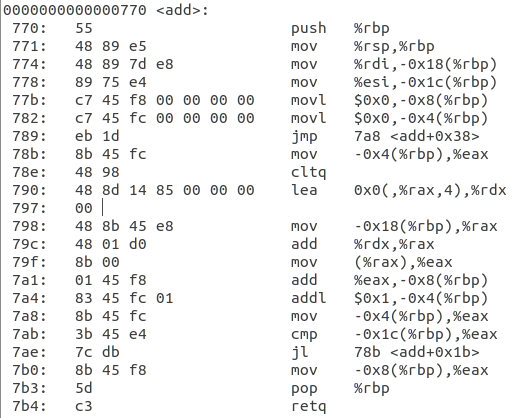
（1）分析同一个源程序在不同机器上生成的可执行目标代码是否相同。

否

ISA：不同的架构支持的ISA不同，ISA不同则同一种指令的实现方式和功能可能不同。

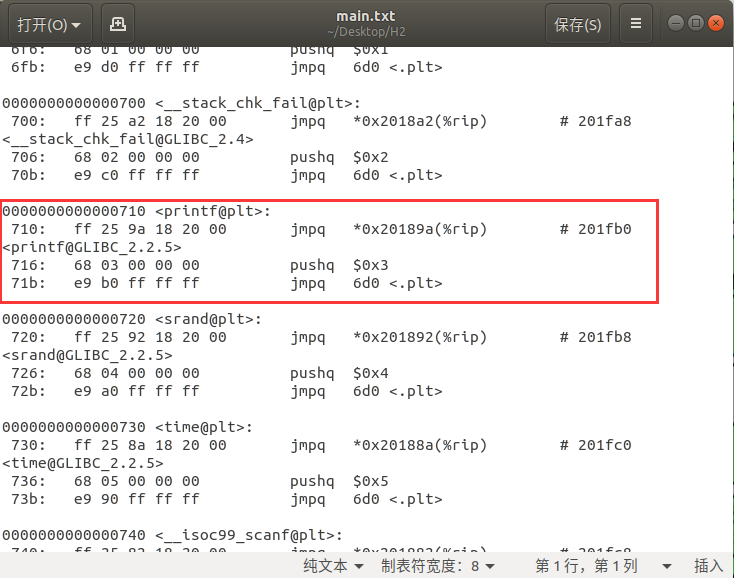
OS：不同OS可执行文件的格式、提供的系统库、对寄存器的使用方式等方面都有不同。

编译器：不同机器使用的编译器种类、版本、编译参数都可能不同，显然导致可执行目标代码的不同。

图：将源代码分别在Windows 10下和Ubuntu 18.04下编译得到可执行文件后反汇编的结果。

仅由<add>的反汇编内容可以得到的结论：两个系统在寄存器使用、栈结构方面都有不同。

另外，Windows下的反汇编文件大小为417KB，而Ubuntu则只有20.8KB。查看发现Windows的反汇编文件比Linux多了很多很多系统函数。说明两个系统在编译时提供的系统库不同。

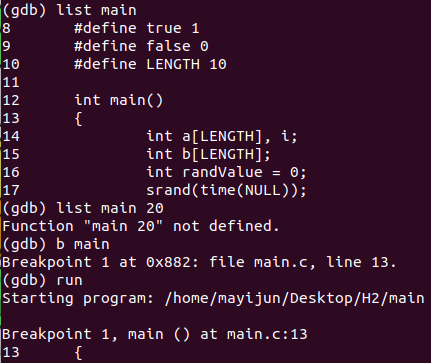
（2）在可执行目标文件的反汇编结果中，是否能找到printf()对应的机器代码段？

没有，只有相对应的全局偏移量表GOT条目和过程链接表PLT实现动态链接。（延迟绑定）

（3）为什么源程序文件的内容和可执行目标文件的内容完全不同？

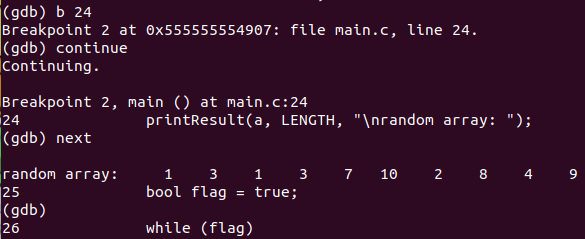
首先，语言不同。源程序文件一般是用高级语言写成的，而可执行目标文件则是机器语言，就算是其反汇编结果，也是以汇编语言的形式呈现的。

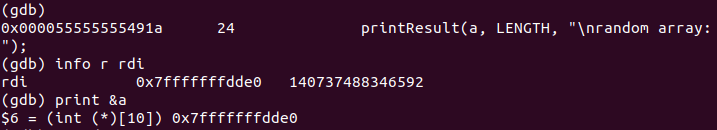
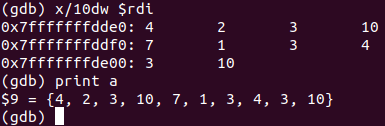
其次，会经过复杂的处理步骤。预处理生成的文本文件就已经与源程序不同，编译时除了语言不同还可能由编译器进行一些优化，最后链接时还需要把多个分散的源文件中的相同节合并，文件结构更改。

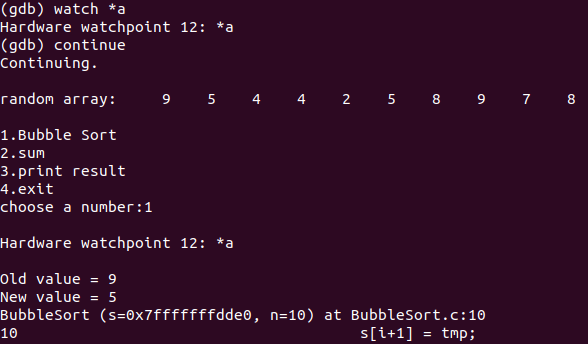
最后，即使是用源程序编译的结果和可执行目标文件反汇编的结果对比，还是有所不同。链接时会发生重定位，改变了指令和数据的地址；在数值表示形式和指令助记符后缀等方面也稍有不同。

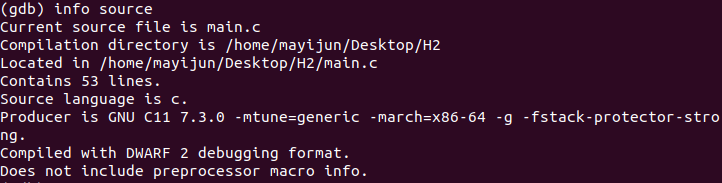
四、使用gdb中的各类命令进行调试

首先我尝试了list命令，它的功能是打印指定行或指定函数入口处（缺省为当前行）前后若干（本机上为10）行源码。然后我在main()入口处设置断点，开始单步调试。

执行到生成随机数组的循环体时，因为这一段程序不需要调试，所以我在循环体结束的后一行设置了断点，并通过continue指令使程序运行至该断点处暂停。

接下来我试用了x、info和print命令。我运行到调用函数的一个过程中，并在适当的时候读取了rdi寄存器的值，发现它的值正是函数第一个参数的值，验证了x86-64中函数调用的第一个参数是通过rdi寄存器来传送的。然后我又用x命令直接读了rdi指向的地址的内容，与理论值一致。

然后是watch命令。我给a[0]这个地址加上了监视点，并运行了冒泡排序，成功监测到了该地址的内容的变化。

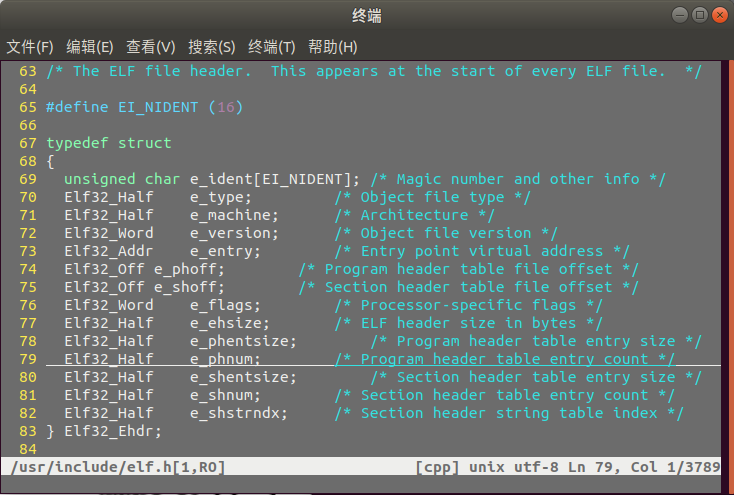
最后我调用了info source命令，系统打印了文件名、路径、编译路径、行数、语言、编译器、宏定义等信息。

五、实现readelf –h main.o的功能

代码见提交的rh.cpp文件。

查阅资料知，对每一台Linux机器，其ELF文件的结构定义都位于/usr/include/elf.h中。

打开该文件：

这样一来，我们马上就得知了ELF头位于文件开头位置，也知道了在本机上它的详细结构定义。在网上的相关资料的提示下，我还在elf.h文件中找到了各种数值对应的含义，这样便可以像readelf -h命令一样输出完整的信息（而不只是ELF头的数据）了。

接下来只要找到直接从.o文件里读原始数据的方法。

我的第一反应是用vim打开了main.o，显示乱码，显然ELF编码不支持用vim读取。

然后我使用了freopen()来读main.o，但因为ELF文件里的数据是连续的，前后无空格，scanf()只有对%c才能做到前后无空格的时候正常读取（%d、%lu等都不行），所以我一开始的做法是，用%c读入若干个连续的字节，利用union转换成其它数据类型。

后来我想到fread()命令可以指定读入的字节数，在参考了网上的代码中fread()的使用之后，把读入部分改成了fread()直接实现，不再需要繁杂的调用函数、union类型转换过程。

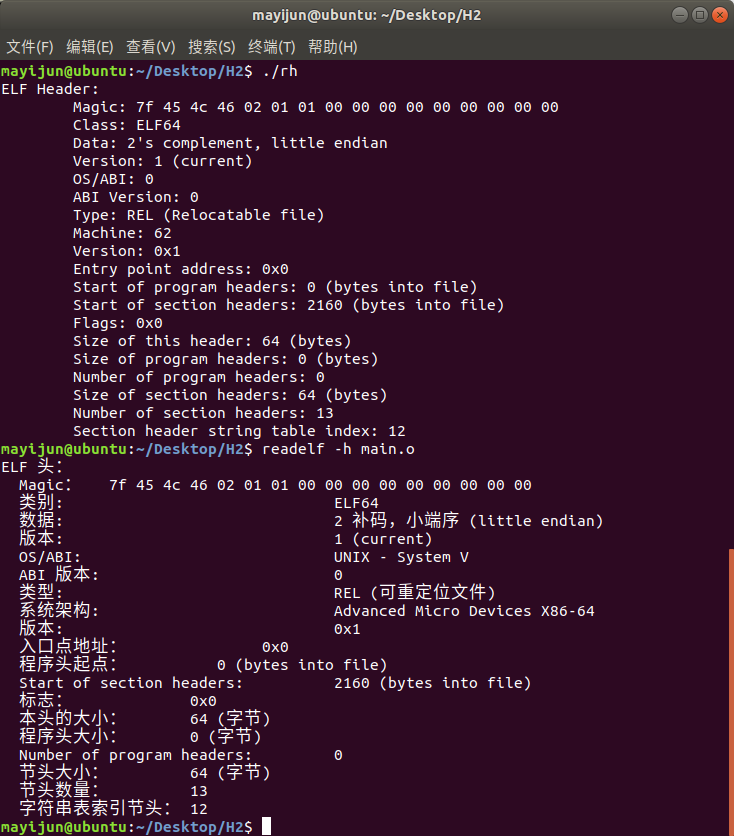
程序运行效果如图。

不考虑中文和英文的不同，只有一个地方我的程序尚未实现：从OS/ABI和系统架构对应的数据，读出其对应的完整名称。这是因为，这些名称都是在elf.h中用宏定义出来的（如#define EM\_X86\_64 62），其解析较困难。

我在百度上和谷歌上无法查到readelf命令的工作方式。而我自己的想法也只有两种办法：

其一是把这些宏名在自己的rh.cpp中打一个字符串常量表，用下标对应输出。但因为elf.h中一共定义了200多种Architecture，这种方法的效率是很低的，也毫无技术含量。

其二是写一个字符串解析程序，把elf.h中的宏名和宏值读出来，然后再建立反映射关系。这种方法的通用性强（对不同的elf.h都可以使用，而前一种方法必须修改源码才能用），我猜想readelf可能就是这么工作的。但因为这种方法的实现极其复杂，也不是我们这门课程主要关心的内容，所以我就不再深入探究了。



参考资料：

[1] <https://blog.csdn.net/wangpengqi/article/details/8279839>

使用LD链接时候文件的顺序 wangpengqi 2012.12.10

[2] <https://blog.csdn.net/king_cpp_py/article/details/80334086>

ELF文件头结构 js0huang 2018.5.16

[3] <https://blog.csdn.net/bufanq/article/details/51530875>

编程读取ELF文件头 bufanq 2016.5.29