

电子科技大学

实验报告

学生姓名： 侯晨 学 号： 202222081026 指导教师：

实验地点： 电子科技大学主楼 实验时间： 2023/11/8

一、实验室名称：

Linux 环境高级编程实验室

二、实验项目名称：

Linux 编程环境搭建与使用

三、实验学时：

4 学时

四、实验目的：

进行本次实验的目的是让学习者熟悉和掌握在 Linux 环境中进行软件开发所需的关键技能和工具。

五、实验内容：

VMWare 配置：

在这一部分，学习者学会了如何使用 VMWare 虚拟机软件来创建和配置虚拟机。虚拟机是一个隔离的环境，可以用于测试和学习不同的操作系统，而不影响主机系统。

APT 使用：

APT (Advanced Package Tool) 是一个在 **Debian** 和 **Ubuntu** 等 **Linux** 发行版上用于管理软件包的工具。学习者在这一部分了解了如何使用 **APT** 来安装、更新和卸载软件包，以及如何管理系统的依赖关系。

vi 高级使用：

vi 是一个在终端中使用的文本编辑器，具有强大的功能。在这一部分，学习者学习了 **vi** 的高级使用技巧，包括搜索、替换、复制粘贴等功能，以提高在命令行环境中编辑文本的效率。

Make 的使用：

make 是一个构建工具，用于管理程序的编译和链接过程。学习者在这一部分学会了编写 **Makefile** 文件，并使用 **make** 工具自动构建项目，以及如何管理源代码的依赖关系。

GDB 使用：

GDB (GNU Debugger) 是一个强大的调试工具，用于在程序运行时进行调试。学习者学习了如何使用 **GDB** 进行单步调试、查看变量、设置断点等操作，以帮助解决程序中的错误。

Google Test 使用：

Google Test 是一个用于进行 C++ 单元测试的框架。在这一部分，学习者学会了如何编写测试用例，使用 **Google Test** 的宏和断言进行测试，以及如何运行测试来验证代码的正确性。

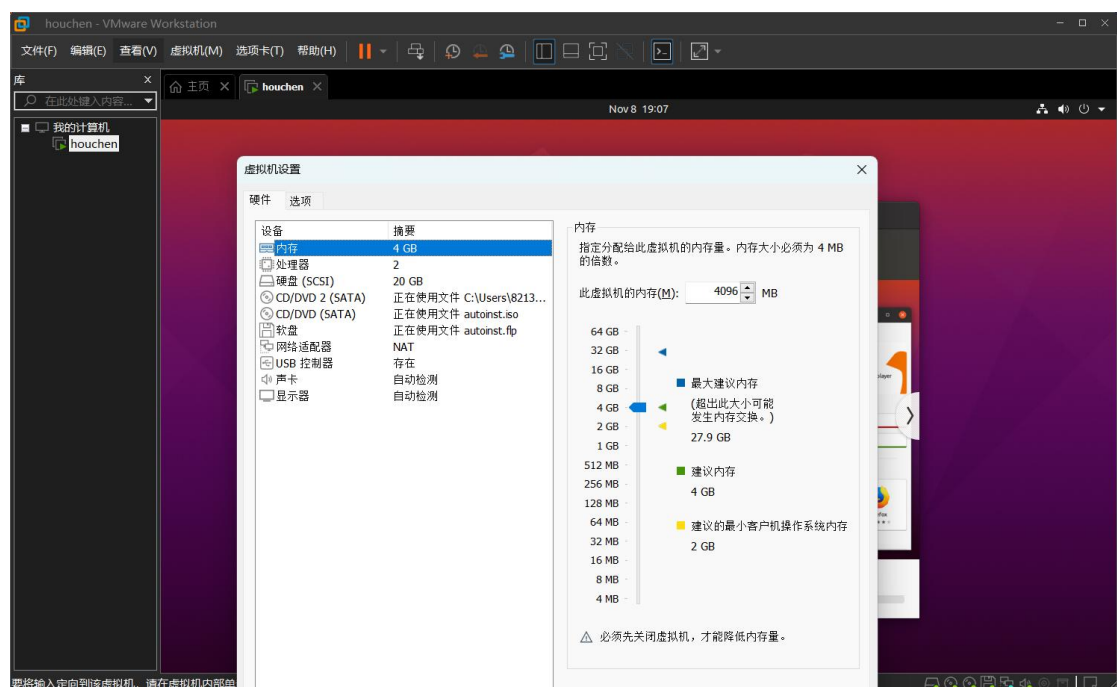
Valgrind 使用：

Valgrind 是一个用于检测内存泄漏和指针访问违规的工具。学习者在这一部分了解了如何使用 **Valgrind** 对程序进行内存分析，检测潜在的问题，以提高代码的质量和稳定性。

六、实验步骤：

1. VMWare 网络配置

通过截图与文字，说明如何进行网络配置。需要囊括主要的操作步骤。注意：将主机名修改为你的姓名拼音，以杜绝抄袭。

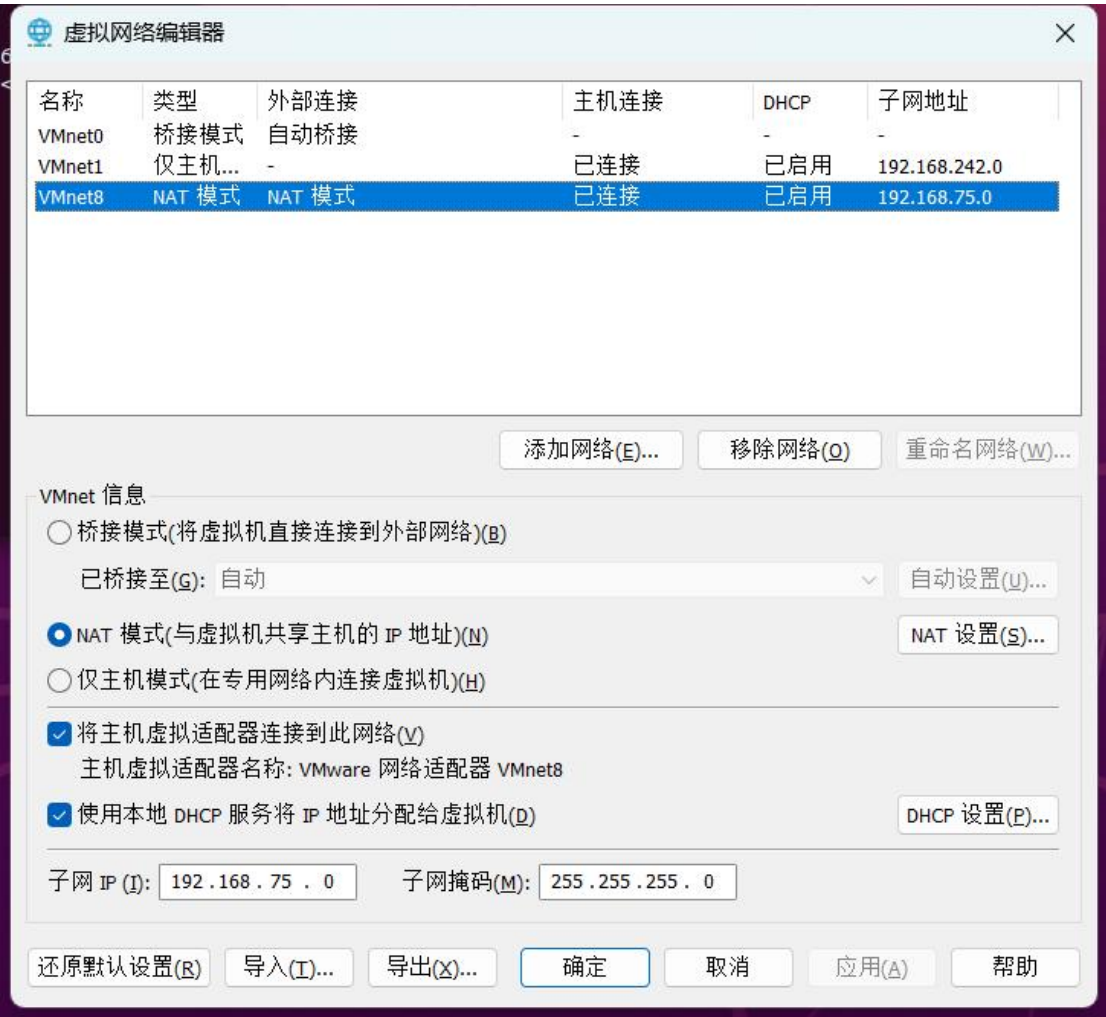


检 查 当 前 网 卡

```
Processing triggers for man-db (2.9.1-1) ...
hc202222081026@ubuntu:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.75.128 netmask 255.255.255.0 broadcast 192.168.75.255
    inet6 fe80::f220:f02f:995f:9853 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:33:de:1a txqueuelen 1000 (Ethernet)
    RX packets 2085 bytes 2602964 (2.6 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 970 bytes 87460 (87.4 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 235 bytes 20401 (20.4 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 235 bytes 20401 (20.4 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

确认当前网络配置



检查网络是否正常

```
hc202222081026@ubuntu:~$ ping www.baidu.com
PING www.a.shifen.com (14.119.104.189) 56(84) bytes of data:
64 bytes from 14.119.104.189 (14.119.104.189): icmp_seq=1 ttl=128 time=39.6 ms
64 bytes from 14.119.104.189 (14.119.104.189): icmp_seq=2 ttl=128 time=39.7 ms
64 bytes from 14.119.104.189 (14.119.104.189): icmp_seq=3 ttl=128 time=39.0 ms
```

2. APT 的使用

查找是否安装了 g++，安装 OpenSSH、build-essential。通过截图和文字，加以说明。

检查是否安装了 g++

```
g++-arm-linux-gnueabihf
hc202222081026@ubuntu:/etc/apt$ sudo apt-cache policy g++
g++:
  Installed: (none)
  Candidate: 4:9.3.0-1ubuntu2
  Version table:
     4:9.3.0-1ubuntu2 500
                       500 http://us.archive.ubuntu.com/ubuntu focal/main
amd64 Packages
     4:5.3.1-1ubuntu1 500
                       500 http://mirrors.aliyun.com/ubuntu xenial/main a
md64 Packages
```

安装 openssh

```
hc202222081026@ubuntu:/etc/apt$ sudo apt-get install OpenSSH
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package OpenSSH
```

安装 build-essential

```
E: Unable to locate package openssh
hc202222081026@ubuntu:/etc/apt$ sudo apt-get install build
-essential
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  binutils binutils-common binutils-x86-64-linux-gnu
  cpp-9 dpkg-dev fakeroot g++ g++-9 gcc gcc-10-base
  gcc-9 gcc-9-base libalgorithm-diff-perl
  libalgorithm-diff-xs-perl libalgorithm-merge-perl
  libasan5 libatomic1 libbinutils libc-dev-bin libc6
  libc6-dbg libc6-dev libcc1-0 libcrypt-dev
  libctf-nobfd0 libctf0 libfakeroot libgcc-9-dev
```

3. Vi 的高级使用

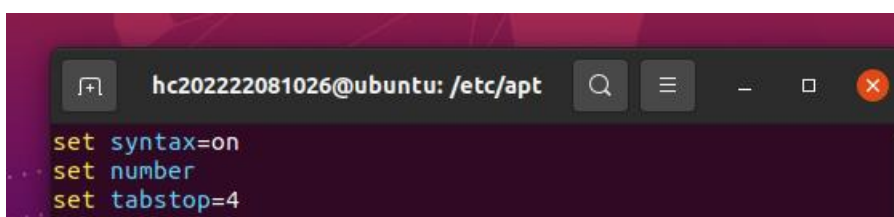
对.vimrc 文件进行分析，说明你自己定义的快捷方式。对 vi 的多

窗口截图，说明 vi 插件安装成功。

设置语法高亮

设置数字高亮

设置 tab 为 4 空格



The screenshot shows a terminal window with the title bar 'hc202222081026@ubuntu: /etc/apt'. Inside the terminal, the following commands are entered and executed:

```
set syntax=on
set number
set tabstop=4
```

多窗口


```
es terminal Nov 9 19:33
hc202222081026@ubuntu: ~
=====
" Netrw Directory Listing (netrw v165)
" /home/hc202222081026/Desktop/Myvim/Myvim/doc
" Sorted by name
" Sort sequence: [\/]$,\<core\%(\\.d\\+\\)\=\\>,.h$,.c$,.cpp$,\~\\=\\*$,*,.o$,
" Quick Help: <F1>:help -:go up dir D:delete R:rename s:sort-by x:specia
" =====
../
./
omnicppcomplete.txt*
taglist.txt*
winmanager.txt*
~
[No Name] [R0] 1,1 All
=====
" Netrw Directory Listing (netrw v165)
" /home/hc202222081026/Desktop/Myvim/Myvim/doc
" Sorted by name
" Sort sequence: [\/]$,\<core\%(\\.d\\+\\)\=\\>,.h$,.c$,.cpp$,\~\\=\\*$,*,.o$,
" Quick Help: <F1>:help -:go up dir D:delete R:rename s:sort-by x:specia
" =====
../
./
omnicppcomplete.txt*
taglist.txt*
winmanager.txt*
~
[No Name] [R0] 8,1 All
```

4. Make 的使用

自己编写一个程序，再编写其 Makefile 文件。注意，在 Makefile 文件对应的依赖关系树，必须是 3 层的完全二叉树。给出程序代码、Makefile 文件。

```
hc202222081026@ubuntu:~/Desktop/code/make$ make
g++ -o Adder.o -c Adder.cpp
g++ -o CAdder.o -c CAdder.cpp
g++ -o test Adder.o CAdder.o
rm *.o
hc202222081026@ubuntu:~/Desktop/code/make$ ./test
7
```

5. Gdb 的使用

自己编写一个多进程的程序，通过截图和文字，说明如何调试多

进程。

安装 gdb

```
hc202222081026@ubuntu:~$ sudo apt-get install gdb
[sudo] password for hc202222081026:
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

查看源文件代码

```
hc202222081026@ubuntu: ~/Desktop/code/GdbMultiProess
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...
(gdb) list
19         }
20     else
21     {
22         return pid;
23     }
24 }
25
26 void MyChild(void *pContext)
27 {
28     long long i = (long long)pContext;
```

打上断点，运行 发现没有线程创建


```

20     long long t = (long long)pContext,
(gdb) b 18
Breakpoint 1 at 0x1262: file test.cpp, line 18.
(gdb) run
Starting program: /home/hc202222081026/Desktop/code/GdbMultiProess/test
[Detaching after fork from child process 39594]
In Father
[Inferior 1 (process 39590) exited normally]
(gdb) info threads
No threads.

```

在子进程上打上断点，运行 发现有线程创建了

```

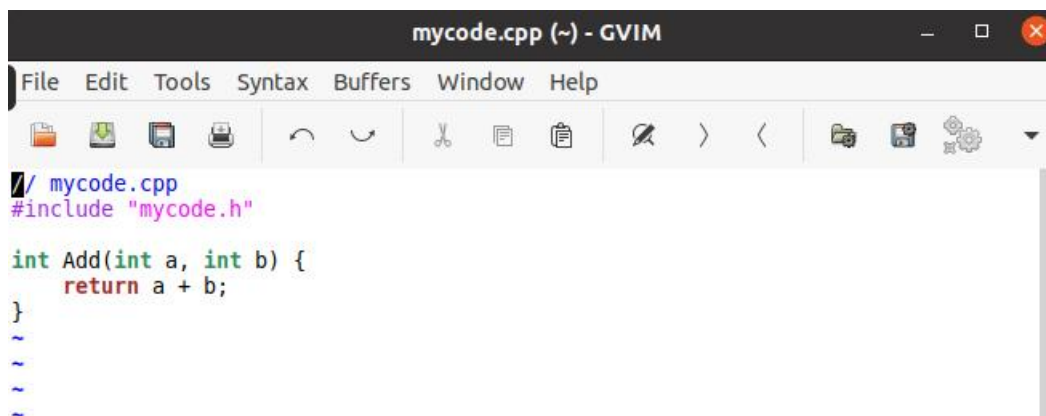
(gdb) b 6
Breakpoint 2 at 0x55555555229: file test.cpp, line 11.
(gdb) c
The program is not being run.
(gdb) info threads
No threads.
(gdb) r
Starting program: /home/hc202222081026/Desktop/code/GdbMultiProess/test

Breakpoint 2, CreateProcess (
    child=0x5555555553dd <__libc_csu_init+77>,
    pContext=0x2) at test.cpp:11
11     {
(gdb) info threads
  Id   Target Id         Frame
* 1    process 39595 "test" CreateProcess (
    child=0x5555555553dd <__libc_csu_init+77>,
    pContext=0x2) at test.cpp:11

```

6. Googletest 使用

自己编写一个程序，然后使用 googletest 对其进行测试。至少要包含三个测试用例。需要给出程序代码，测试代码，并进行适当分析。



```

mycode.cpp (~) - GVIM
File Edit Tools Syntax Buffers Window Help
[Icons]
// mycode.cpp
#include "mycode.h"

int Add(int a, int b) {
    return a + b;
}
~
~
~

```

```
mycode.h
1 // mycode.h
2 #ifndef MYCODE_H
3 #define MYCODE_H
4
5 int Add(int a, int b);
6
7 #endif // MYCODE_H

mycodetest.cpp
// mycodetest.cpp
#include "gtest/gtest.h"
#include "mycode.h" // 包含要测试的代码的头文件

TEST(AddTest, PositiveNumbers) {
    EXPECT_EQ(Add(2, 3), 5);
}

TEST(AddTest, NegativeNumbers) {
    EXPECT_EQ(Add(-2, -3), -5);
}

testmain.cpp
// testmain.cpp
#include "gtest/gtest.h"

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

成功运行 googletest 进行单元测试一

```
hc202222081026@ubuntu:~$ g++ -o testmain mycodetest.cpp mycode.cpp -lgtest -lgtest_main -pthread
hc202222081026@ubuntu:~$ ./testmain
Running main() from /home/hc202222081026/googletest/googletest/src/gtest_main.cc
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from AddTest
[ RUN    ] AddTest.PositiveNumbers
[      OK ] AddTest.PositiveNumbers (0 ms)
[ RUN    ] AddTest.NegativeNumbers
[      OK ] AddTest.NegativeNumbers (0 ms)
[-----] 2 tests from AddTest (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (0 ms total)
[ PASSED ] 2 tests.
```

修改源文件

```
// mycodetest.cpp
#include "gtest/gtest.h"
#include "mycode.h"

TEST(AddTest, NegativeInput) {
    // 使用 ASSERT_THROW 检查是否抛出异常
    ASSERT_THROW(Add(-1, 2), std::invalid_argument);
}
```

单元测试二测试成功

```
hc202222081026@ubuntu:~$ g++ -o testmain mycodetest.cpp mycode.cpp -lgtest -lgtest_main -pthread
hc202222081026@ubuntu:~$ ./testmain
Running main() from /home/hc202222081026/googletest/googletest/src/gtest_main.cc
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from AddTest
[ RUN     ] AddTest.NegativeInput
mycodetest.cpp:7: Failure
Expected: Add(-1, 2) throws an exception of type std::invalid_argument.
Actual: it throws nothing.

[ FAILED ] AddTest.NegativeInput (0 ms)
[-----] 1 test from AddTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] AddTest.NegativeInput

1 FAILED TEST
```

修改源代码

```
mycodetest.cpp + (~) - GVIM1
File Edit Tools Syntax Buffers Window Help
[Icons]

// mycodetest.cpp
#include "gtest/gtest.h"
#include "mycode.h"

TEST(AddTest, CustomOutput) {
    // 使用 EXPECT_EQ 来检查输出是否符合预期
    testing::internal::CaptureStdout(); // 捕获标准输出
    std::cout << Add(3, 4); // 调用函数
    std::string output = testing::internal::GetCapturedStdout(); // 获取标准输出
    EXPECT_EQ(output, "7\n"); // 检查输出是否符合预期
}

~
~
~
```

单元测试三测试成功:

```
hc202222081026@ubuntu:~$ g++ -o testmain mycodetest.cpp mycode.cpp -lgtest -lgtest_main -pthread
hc202222081026@ubuntu:~$ ./testmain
Running main() from /home/hc202222081026/googletest/googletest/src/gtest_main.cc
[====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from AddTest
[ RUN ] AddTest.CustomOutput
mycodetest.cpp:10: Failure
Expected equality of these values:
  output
    Which is: "7"
  "7\n"

[  FAILED  ] AddTest.CustomOutput (0 ms)
[-----] 1 test from AddTest (0 ms total)

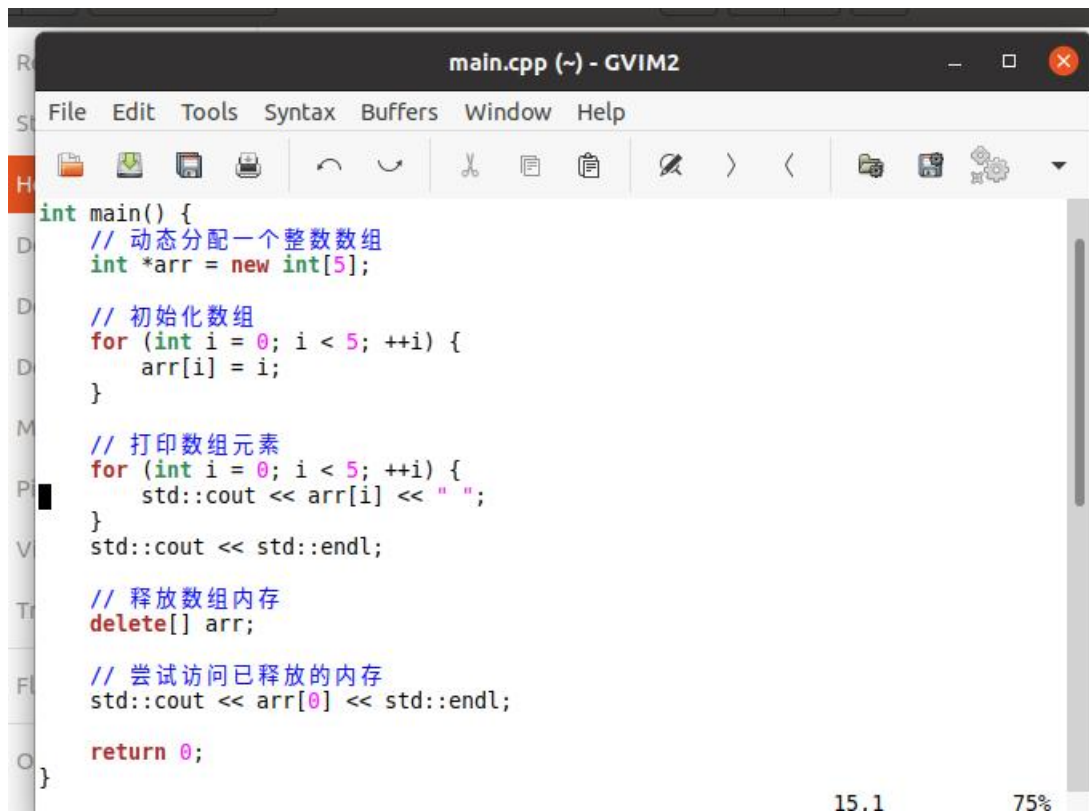
[-----] Global test environment tear-down
[====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] AddTest.CustomOutput

1 FAILED TEST
```

7. Valgrind 使用

自己编写一个程序，然后使用 `valgrind` 检测其有无内存泄露、指针访问违规。需要给出程序代码，截图和文字说明。

编 写 源 代 码 :



```
main.cpp (~) - GVIM2
File Edit Tools Syntax Buffers Window Help

int main() {
    // 动态分配一个整数数组
    int *arr = new int[5];

    // 初始化数组
    for (int i = 0; i < 5; ++i) {
        arr[i] = i;
    }

    // 打印数组元素
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

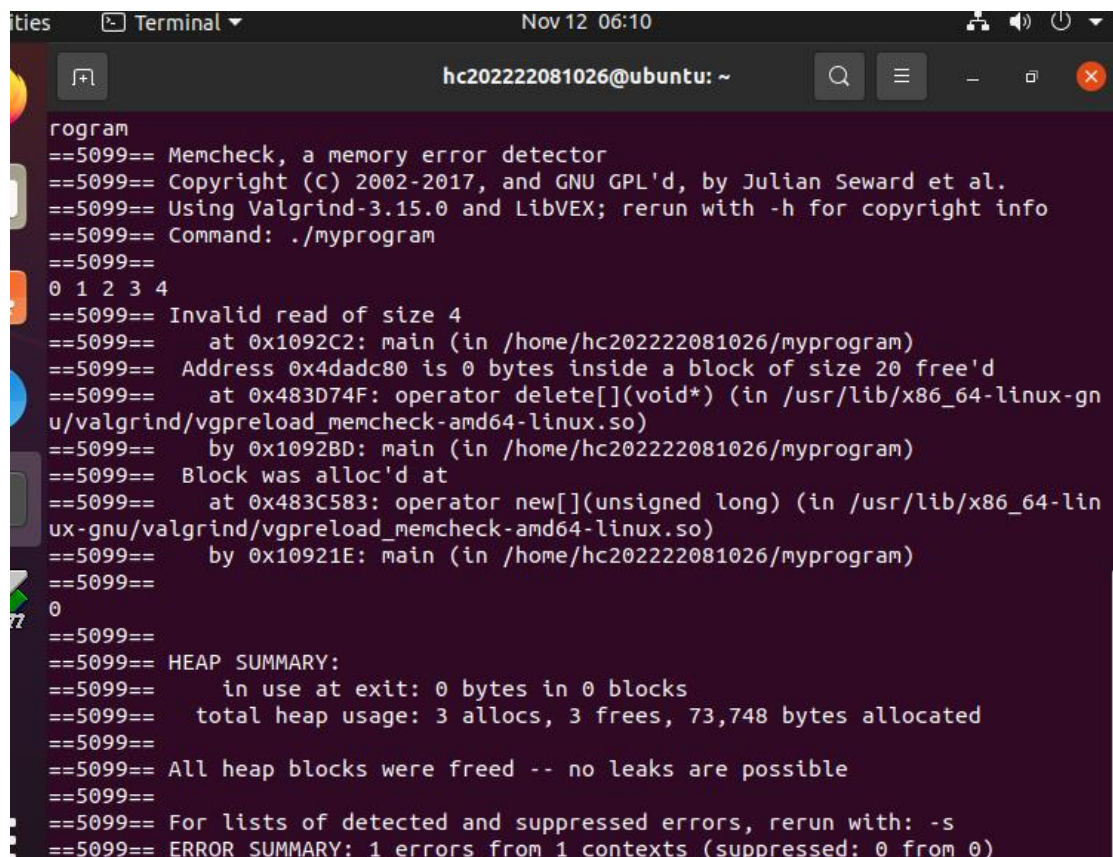
    // 释放数组内存
    delete[] arr;

    // 尝试访问已释放的内存
    std::cout << arr[0] << std::endl;

    return 0;
}
```

15.1 75%

进行内存泄露测试，测试成功



```
hc202222081026@ubuntu: ~
program
==5099== Memcheck, a memory error detector
==5099== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5099== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5099== Command: ./myprogram
==5099==
0 1 2 3 4
==5099== Invalid read of size 4
==5099==    at 0x1092C2: main (in /home/hc202222081026/myprogram)
==5099==   Address 0x4dadcd80 is 0 bytes inside a block of size 20 free'd
==5099==    at 0x483D74F: operator delete[](void*) (in /usr/lib/x86_64-linux-gn
u/valgrind/vgpreload_memcheck-amd64-linux.so)
==5099==   by 0x1092BD: main (in /home/hc202222081026/myprogram)
==5099==   Block was alloc'd at
==5099==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-lin
ux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5099==   by 0x10921E: main (in /home/hc202222081026/myprogram)
==5099==
0
==5099==
==5099== HEAP SUMMARY:
==5099==   in use at exit: 0 bytes in 0 blocks
==5099==   total heap usage: 3 allocs, 3 frees, 73,748 bytes allocated
==5099==
==5099== All heap blocks were freed -- no leaks are possible
==5099==
==5099== For lists of detected and suppressed errors, rerun with: -s
==5099== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

七、总结及心得体会：

实验对 **Valgrind** 体会最深刻。

心得体会：

谨慎使用动态内存：

通过本次实验，我进一步认识到动态内存的使用需要谨慎。及时释放动态分配的内存是一项重要的任务，否则容易导致内存泄漏。使用 **delete** 后要确保不再访问已释放的内存。

调试工具的重要性：

Valgrind 类似的调试工具在软件开发中的重要性不可忽视。它们可以提供对程序运行时行为的深入洞察，帮助找出潜在的问题，提高代码的质量。

学会阅读 **Valgrind** 输出：

Valgrind 输出的信息可能比较详细，但学会阅读并理解这些信息是非常重要的。了解 **Valgrind** 的输出将有助于定位问题的根本原因，从而更快地进行修复。

持续集成和测试：

在实际项目中，可以考虑将 **Valgrind** 集成到持续集成和测试中，以确保每次代码变更都能够及时进行内存检查。这有助于捕捉和修复问

题的速度更快，保证软件质量。

八、对本实验过程及方法、手段的改进建议：

虽然在本实验中我们使用了命令行方式，但 Valgrind 也有一些 GUI 工具，如 KCacheGrind 和 Valgrind Analyzer。在实验中可以简要介绍这些工具，以便学习者了解如何通过图形化界面更方便地分析 Valgrind 输出

报告评分：

指导教师签字：

电子科技大学

实验报告

学生姓名：侯晨 学号：202222081026 指导教师：丁旭阳

实验地点：电子科技大学 实验时间：2023/11/15

一、实验室名称：

Linux 环境高级编程实验室

二、实验项目名称：

对象序列化实验

三、实验学时：

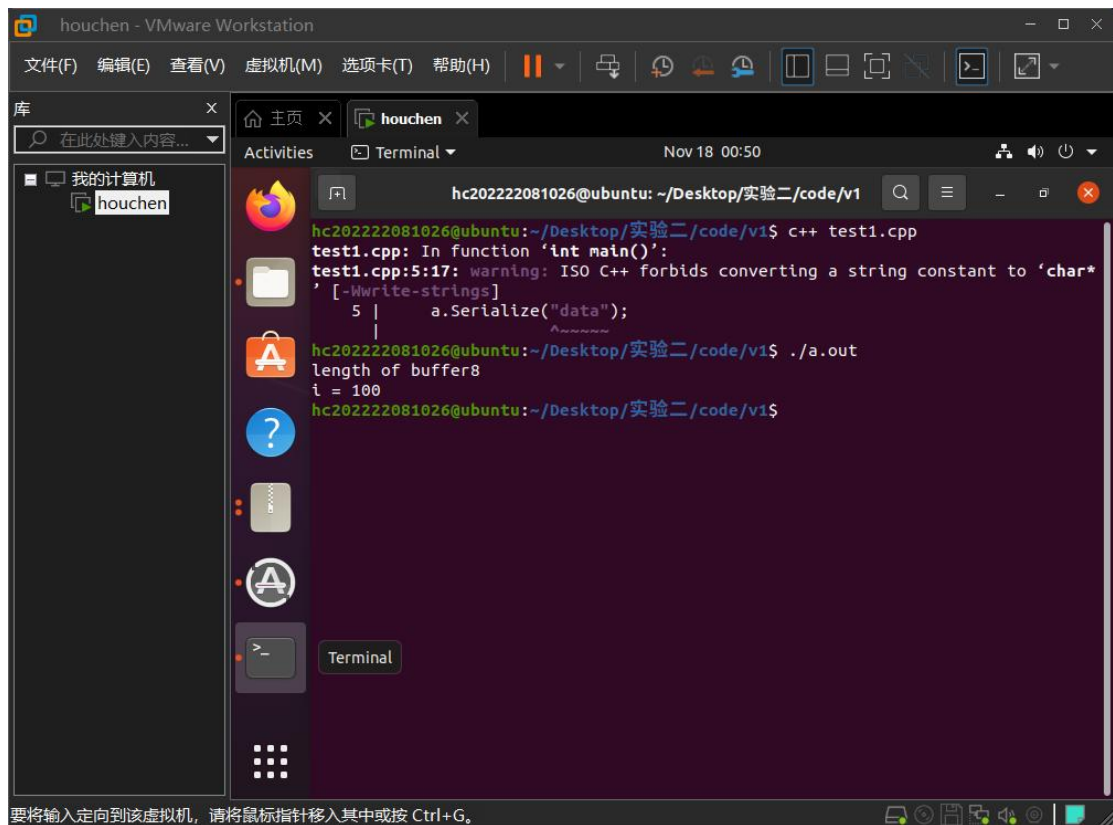
4 学时

四、实验目的：

需要说明为什么要进行本次实验

五、实验内容：

- 版本 1：将一个类的一个对象序列化到文件



```
#include "A.h"

int main(){
    A a(100);
    a.Serialize("data");
    A b = A::Deserialize("data");
    b.f();
    return 0;
}
```

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 using namespace std ;
6 class A{
7
8     public:
9         A(){
10             i = 0;
11
12
13         A(int j){
14             i = j;
15         }
16
17         int Serialize(char* pBuffer){
18             long length = sizeof(this);
19             cout << "length of buffer" << length << endl;
20             ofstream *o = new ofstream(pBuffer);
21             o->write((char*)this, length);
22             o->close();
23             delete o;
24             return 1;
25         }
26
27         static A Deserialize(const char* pBuffer){
28
29             static A Deserialize(const char* pBuffer){
30                 char buf[1000];
31                 ifstream is(pBuffer);
32                 is >> buf;
33                 A *a = (A*) buf;
34                 is.close();
35                 return (*a);
36             }
37
38             void f(){
39                 cout << "i = " << this->i << endl;
40             }
41
42             private:
43                 int i;
44 };

```

- 版本 2：将一个类的多个对象序列化到文件

```

hc202222081026@ubuntu: ~/Desktop/实验二/code/v2
c202222081026@ubuntu:~/Desktop/实验二/code/v2$ c++ test2.cpp
c202222081026@ubuntu:~/Desktop/实验二/code/v2$ ./a.out
= 100
= 200
c202222081026@ubuntu:~/Desktop/实验二/code/v2$

```

```
// 将A指针的vector序列化到文件
static int Serialize(const char* pFilePath, vector<A *>& v) {
    FILE* fp = fopen(pFilePath, "w+"); // 以读写方式打开文件

    // 遍历A指针的vector并将每个A对象的内容写入文件
    for(int i = 0; i < v.size(); i++) {
        fwrite(v[i], sizeof(int), 1, fp);
    }

    fclose(fp);
    return 1;
}

// 从文件反序列化A指针的vector
static int Deserialize(const char* pFilePath, vector<A *>& v) {
    FILE *fp = fopen(pFilePath, "r"); // 以读方式打开文件

    // 遍历A指针的vector并从文件中读取每个A对象的内容
    for(int i = 0; i < v.size(); i++) {
        fread(v[i], sizeof(int), 1, fp);
    }

    fclose(fp);
    return 1;
}

};
```

```
using namespace std;

int main() {
    A a1(100);
    A a2(200);

    vector<A *> v1;
    v1.push_back(&a1);
    v1.push_back(&a2);

    // 将A对象指针的vector序列化到文件
    SerializerForAs::Serialize("data2", v1);

    A a3;
    A a4;

    vector<A *> v2;
    v2.push_back(&a3);
    v2.push_back(&a4);

    // 从文件反序列化A对象指针的vector
    SerializerForAs::Deserialize("data2", v2);

    // 调用A对象的函数打印i的值
    a3.f();
    a4.f();
}
```



```

int Serialize(char* pBuffer) {
    long length = sizeof(this);
    ofstream *o = new ofstream(pBuffer);

    // 将A对象的内容写入文件
    o->write((char*)this, length);
    o->close();
    delete o;

    return 1;
}

// 从文件反序列化A对象
static A Deserialize(const char* pBuffer) {
    char buf[1000];
    ifstream is(pBuffer);

    // 从文件中读取A对象的内容
    is >> buf;
    A *a = (A*) buf;

    is.close();
    return (*a);
}

// 打印i的值
void f() {
    cout << "i = " << this->i << endl;
}

// 获取对象的大小
int GetObjSize(unsigned int * pObjSize) {
    *pObjSize = 4;
    return 1;
}

```

- 版本 3：将两个类的多个对象序列化到文件

```

hc202222081026@ubuntu:~/Desktop/实验二/code$ cd v3
hc202222081026@ubuntu:~/Desktop/实验二/code/v3$ c++ test3.cpp
test3.cpp: In member function 'bool Serializer::Serialize(const char*, std::vector<Serialized>&)':
test3.cpp:97:9: warning: control reaches end of non-void function [-Wreturn-type]
   97 |         }
      |         ^
hc202222081026@ubuntu:~/Desktop/实验二/code/v3$ ./a.out
i = 2
i = 3 , j = 4
i = 4 , j = 5
i = 5

```

```

class Serializer {
public:
    // 将Serialized结构体的vector序列化到文件
    bool Serialize(const char *pFilePath, vector<Serialized> &v) {
        FILE *fp = fopen(pFilePath, "w+"); // 以读写方式打开文件
        if (fp == NULL)
            return false;

        // 遍历Serialized结构体的vector并将每个对象的类型和内容写入文件
        for (int i = 0; i < v.size(); i++) {
            fwrite(&(v[i].nType), sizeof(int), 1, fp);

            // 根据对象类型进行相应的序列化
            if (0 == v[i].nType) {
                A *p = (A *) (v[i].pObj);
                p->Serialize(fp);
            } else if (1 == v[i].nType) {
                B *p = (B *) (v[i].pObj);
                p->Serialize(fp);
            }
        }

        fclose(fp);
        return true;
    }
}

```

```

// 从文件反序列化为Serialized结构体的vector
bool Deserialize(const char *pFilePath, vector<Serialized> &v) {
    FILE *fp = fopen(pFilePath, "r+"); // 以读写方式打开文件
    for (;;) {
        int nType;
        int r = fread(&nType, sizeof(int), 1, fp);
        if (-1 == r || 0 == r)
            break;

        // 根据对象类型创建对应的对象并进行反序列化
        if (0 == nType) {
            A *p;
            p = new A();
            p->Deserialize(fp);

            Serialized s;
            s.nType = nType;
            s.pObj = p;
            v.push_back(s);
        } else if (1 == nType) {
            B *p;
            p = new B();
            p->Deserialize(fp);

```

```

s.Deserialize("data", v);

// 遍历Serialized结构体的vector并根据对象类型调用相应的打印函数
for (int i = 0; i < v.size(); i++) {
    if (v[i].nType == 0) {
        A *p = (A *) (v[i].pObj);
        p->f();
    } else if (v[i].nType == 1) {
        B *p = (B *) (v[i].pObj);
        p->f();
    }
}
}

```

- 版本 4：按照面向对象的方法，解决多个类的多个对象序列化到文件的问题

化到文件的问题

```
hc202222081026@ubuntu:~/Desktop/实验二/code/v4$ c++ test4.
cpp
hc202222081026@ubuntu:~/Desktop/实验二/code/v4$ ./a.out
in f(): 2
in f(): 3 4
in f(): 4 5
in f(): 5
hc202222081026@ubuntu:~/Desktop/实验二/code/v4$
```

```
// 定义序列化接口
class ILSerializable {
public:
    // 序列化到文件
    virtual bool Serialize(FILE* fp) = 0;
    // 从文件反序列化
    virtual ILSerializable* Deserialize(FILE* fp) = 0;
    // 获取对象类型
    virtual bool GetType(int& type) = 0;

public:
    ILSerializable() {}
    virtual ~ILSerializable() {}
};

// 类A实现序列化接口
class A : public ILSerializable {
public:
```

```

public:
    // 获取对象类型
    virtual bool GetType(int& type) override {
        type = 0;
        return true;
    }

    // 将A对象的内容写入文件
    virtual bool Serialize(FILE* fp) override {
        if (fp == NULL)
            return false;
        fwrite(&i, sizeof(int), 1, fp);
        return true;
    }

    // 从文件中读取内容反序列化为A对象
    virtual ILSerializable* Deserialize(FILE* fp) override {
        A* p = new A();
        fread(&(p->i), sizeof(int), 1, fp);
        return p;
    }
}

```

```

    // 将B对象的内容写入文件
    virtual bool Serialize(FILE* fp) override {
        if (fp == NULL)
            return false;
        fwrite(&i, sizeof(int), 1, fp);
        fwrite(&j, sizeof(int), 1, fp);
        return true;
    }

    // 从文件中读取内容反序列化为B对象
    virtual ILSerializable* Deserialize(FILE* fp) override {
        B* p = new B();
        fread(&(p->i), sizeof(int), 1, fp);
        fread(&(p->j), sizeof(int), 1, fp);
        return p;
    }
}

```

```
// 序列化器类
class CLSerializer {
public:
    // 将序列化对象的vector序列化到文件
    bool Serialize(const char* pFilePath, std::vector<ILSerializable*>& v) {
        FILE* fp = fopen(pFilePath, "w+");
        if (fp == NULL)
            return false;
        for (int i = 0; i < v.size(); i++) {
            int type;
            v[i]->GetType(type);
            fwrite(&type, sizeof(int), 1, fp);
            v[i]->Serialize(fp);
        }
        fclose(fp);
        return true;
    }

    // 从文件反序列化为序列化对象的vector
    bool Deserialize(const char* pFilePath, std::vector<ILSerializable*>& v) {
        FILE* fp = fopen(pFilePath, "r+");
```

- 版本 5：序列化的目的地不仅可以是文件，还可以是其他，即可配置性

```
hc202222081026@ubuntu:~/Desktop/实验二/code/v5$ c++ test5.
cpp
hc202222081026@ubuntu:~/Desktop/实验二/code/v5$ ./a.out
in f(): 2
in f(): 3 4
in f(): 4 5
in f(): 5
hc202222081026@ubuntu:~/Desktop/实验二/code/v5$
```



a.out



config.txt



data1



test5.cpp


```
public:
    // 获取对象类型
    virtual bool GetType(int& type) override {
        type = 1;
        return true;
    }

    // 将B对象的内容写入文件
    virtual bool Serialize(FILE* fp) override {
        if (fp == NULL)
            return false;
        fwrite(&i, sizeof(int), 1, fp);
        fwrite(&j, sizeof(int), 1, fp);
        return true;
    }

    // 从文件中读取内容反序列化为B对象
    virtual ILSerializable* Deserialize(FILE* fp) override {
        B* p = new B();
        fread(&(p->i), sizeof(int), 1, fp);
        fread(&(p->j), sizeof(int), 1, fp);
        return p;
    }
}
```

```
public:
    // 将序列化对象的vector序列化到文件
    bool Serialize(const char* pFilePath, std::vector<ILSerializable*>& v) {
        FILE* fp = fopen(pFilePath, "w+");
        if (fp == NULL)
            return false;
        for (int i = 0; i < v.size(); i++) {
            int type;
            v[i]->GetType(type);
            fwrite(&type, sizeof(int), 1, fp);
            v[i]->Serialize(fp);
        }
        fclose(fp);
        return true;
    }

    // 从文件反序列化为序列化对象的vector
    bool Deserialize(const char* pFilePath, std::vector<ILSerializable*>& v) {
        FILE* fp = fopen(pFilePath, "r+");
        if (fp == NULL)
            return false;
        for (;;) {
            int nType = -1;
            int r = fread(&nType, sizeof(int), 1, fp);
```

```

// 创建A、B对象并存放在ILSerializable指针向量中
A a1(2);
B b1(3);
B b2(4);
A a2(5);

std::vector<ILSerializable*> v;

v.push_back(&a1);
v.push_back(&b1);
v.push_back(&b2);
v.push_back(&a2);

CLSerializer s;
// 序列化ILSerializable指针向量到文件
s.Serialize(buf, v);
}

{
    CLSerializer s;
    A a;
    B b;
    s.Register(&a);
    s.Register(&b);

    std::vector<ILSerializable*> v;
    // 从文件反序列化ILSerializable指针向量
    s.Deserialize(buf, v);
}

```

```

s.Register(&b);

std::vector<ILSerializable*> v;
// 从文件反序列化ILSerializable指针向量
s.Deserialize(buf, v);

for (int i = 0; i < v.size(); i++) {
    A* p = dynamic_cast<A*>(v[i]);
    if (p != NULL)
        p->f(); // 如果是A对象，则调用f()方法

    B* q = dynamic_cast<B*>(v[i]);
    if (q != NULL)
        q->f(); // 如果是B对象，则调用f()方法
}

return 0;

```

六、实验步骤:

对每个版本的程序进行分析，给出流程图、类图、时序图等分析。

附上源代码，并给出占代码行数 20%以上的注释。

七、总结及心得体会:

对象序列化和反序列化：通过提供类 A 和类 B，学会了如何将这些类的对象序列化为文件，以及如何从文件中反序列化回对象。

ILSerializable 抽象类：通过创建 ILSerializable 抽象类，定义了一个通用的序列化接口，使得不同类可以实现相同的接口以进行统一的处理。

CLSerializer 类：实现了一个序列化器类，能够处理派生自 ILSerializable 的对象。通过注册不同的 ILSerializable 对象，它能够在运行时识别和处理不同类型的对象。

文件操作和错误处理：您使用了文件操作函数（例如 fopen、fwrite、fread）来读写数据。

抽象设计的重要性：使用 ILSerializable 抽象类的设计允许了更通用的实现，使得新的类可以轻松集成序列化功能。

动态类型识别：在 CLSerializer 中使用动态类型识别文件操作

谨慎处理：对于文件操作，在关键的步骤进行了错误检查。

八、对本实验过程及方法、手段的改进建议：

报告评分：

指导教师签字：

电子科技大学

实验报告

学生姓名： 侯晨 学 号： 202222081026 指导教师： 丁 旭 阳

实验地点： 电子科技大学

实验时间： 2023/11/22

一、实验室名称：

Linux 环境高级编程实验室

二、实验项目名称：

基本 TCP 套接口通信库封装

三、实验学时：

8 学时

四、实验目的：

理解基本 TCP 套接口通信机制, 学习五种编程范式, 封装通信库。

五、实验内容：

对基本 TCP 套接口通信机制进行封装。要求使用以下五种编程范式，封装通信库；并使用五种封装后通信库，实现 echo 服务器和客户端。

- 传统 C 的结构化程序设计思想


```

#include <sys/socket.h>
#include <netinet/in.h>
#include <memory.h>
#include <arpa/inet.h>
#include <unistd.h>

#include <iostream>

// 定义函数指针类型，用于传递客户端处理函数
typedef void (*TCPClient)(int nConnectedSocket);

// TCP 客户端主函数
int RunTCPClient(TCPClient ClientFunction, int nServerPort, const char *strServerIP)
{
    // 创建套接字
    int nClientSocket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (-1 == nClientSocket)
    {
        std::cout << "socket error" << std::endl;
        return -1;
    }

    // 设置服务器地址结构体
    sockaddr_in ServerAddress;
    memset(&ServerAddress, 0, sizeof(sockaddr_in));
    ServerAddress.sin_family = AF_INET;

    // 将字符串形式的 IP 地址转换为网络字节序的二进制形式
    if (::inet_pton(AF_INET, strServerIP, &ServerAddress.sin_addr) != 1)
    {
        std::cout << "inet_pton error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }

    // 设置服务器端口号，并将端口号转换为网络字节序
    ServerAddress.sin_port = htons(nServerPort);

    // 连接到服务器
    if (::connect(nClientSocket, (sockaddr *)&ServerAddress, sizeof(ServerAddress)) == -1)
    {
        std::cout << "connect error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }
}

```

```

    }

    // 调用传入的客户端处理函数
    ClientFunction(nClientSocket);

    // 关闭套接字
    ::close(nClientSocket);

    return 0;
}

// 自定义的客户端处理函数
void MyClient(int nConnectedSocket)
{
    // 读取服务器发送的数据
    char buf[13];
    ::read(nConnectedSocket, buf, 13);

    // 打印接收到的数据
    std::cout << "Received data from the server: " << buf << std::endl;
}

// 主函数
int main()
{
    int nListenSocket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (-1 == nListenSocket) {
        // 如果创建套接字失败, 打印错误信息, 返回-1
        std::cerr << "socket error" << std::endl;
        return -1;
    }

    // 初始化服务器地址结构体
    sockaddr_in ServerAddress;
    memset(&ServerAddress, 0, sizeof(sockaddr_in));
    ServerAddress.sin_family = AF_INET;
    ServerAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    ServerAddress.sin_port = htons(5000);

    // 绑定套接字和地址
    if (::bind(nListenSocket, (sockaddr*)&ServerAddress, sizeof(ServerAddress)) == -1) {
        // 如果绑定失败, 打印错误信息, 关闭套接字, 返回-1
        std::cerr << "bind error" << std::endl;
        ::close(nListenSocket);
    }
}

```

```

        return -1;
    }

// 监听连接，允许 10 个连接请求排队等待处理
if (::listen(nListenSocket, 10) == -1) {
    // 如果监听失败，打印错误信息，关闭套接字，返回-1
    std::cerr << "listen error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

while (true) {
    sockaddr_in ClientAddress;
    socklen_t LengthOfClientAddress = sizeof(sockaddr_in);

    // 接受连接请求
    int nConnectedSocket = ::accept(nListenSocket, (sockaddr *)&ClientAddress,
    &LengthOfClientAddress);

    if (nConnectedSocket == -1) {
        // 如果接受连接失败，打印错误信息，退出循环
        std::cerr << "accept error" << std::endl;
        break; // 退出循环，或者添加适当的错误处理
    }

    // 从客户端接收数据
    char buffer[1024];
    ssize_t bytesRead = ::read(nConnectedSocket, buffer, sizeof(buffer));

    if (bytesRead > 0) {
        // 如果成功接收数据，添加字符串结束符
        buffer[bytesRead] = '\0';
        // 打印从客户端接收到的数据
        std::cout << "Received from client: " << buffer << std::endl;

        // 发送响应到客户端
        const char *response = "Hello from server";
        ::write(nConnectedSocket, response, strlen(response));
    } else {
        // 如果读取失败，打印错误信息
        std::cerr << "Error reading from client" << std::endl;
    }

    // 关闭连接

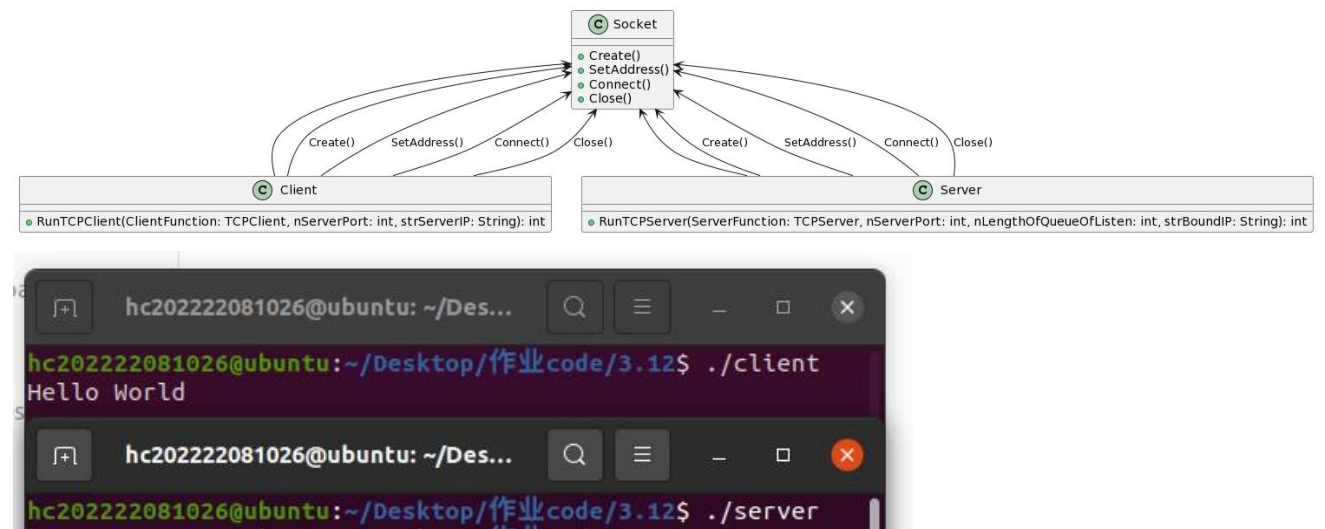
```

```

        ::close(nConnectedSocket);
    }

    // 关闭监听套接字
    ::close(nListenSocket);
}

```



● 面向对象的程序设计思想

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <memory.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

// TCP 服务器基类
class CTCPServer
{
public:
    // 构造函数，用于初始化服务器参数
    CTCPServer(int nServerPort, int nLengthOfQueueOfListen = 100, const char *strBoundIP =
    NULL)
    {
        m_nServerPort = nServerPort;
        m_nLengthOfQueueOfListen = nLengthOfQueueOfListen;

        // 处理绑定 IP
        if (NULL == strBoundIP)
        {
            m_strBoundIP = NULL;

```

```

    }
    else
    {
        int length = strlen(strBoundIP);
        m_strBoundIP = new char[length + 1];
        memcpy(m_strBoundIP, strBoundIP, length + 1);
    }
}

```

// 虚析构函数，释放动态分配的内存

```
virtual ~CTCPServer()
```

```

{
    if (m_strBoundIP != NULL)
    {
        delete[] m_strBoundIP;
    }
}

```

public:

// 启动服务器的主方法

```
int Run()
```

```

{
    // 创建监听套接字
    int nListenSocket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (-1 == nListenSocket)
    {
        std::cout << "socket error" << std::endl;
        return -1;
    }

    // 配置服务器地址
    sockaddr_in ServerAddress;
    memset(&ServerAddress, 0, sizeof(sockaddr_in));
    ServerAddress.sin_family = AF_INET;

    // 处理绑定 IP
    if (NULL == m_strBoundIP)
    {
        ServerAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    }
    else
    {
        if (::inet_pton(AF_INET, m_strBoundIP, &ServerAddress.sin_addr) != 1)
        {

```

```

        std::cout << "inet_pton error" << std::endl;
        ::close(nListenSocket);
        return -1;
    }
}

// 配置服务器端口
ServerAddress.sin_port = htons(m_nServerPort);

// 绑定套接字和地址
if (::bind(nListenSocket, (sockaddr *)&ServerAddress, sizeof(sockaddr_in)) == -1)
{
    std::cout << "bind error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 开始监听连接请求
if (::listen(nListenSocket, m_nLengthOfQueueOfListen) == -1)
{
    std::cout << "listen error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 接受连接请求
sockaddr_in ClientAddress;
socklen_t LengthOfClientAddress = sizeof(sockaddr_in);
int nConnectedSocket = ::accept(nListenSocket, (sockaddr *)&ClientAddress,
&LengthOfClientAddress);
if (-1 == nConnectedSocket)
{
    std::cout << "accept error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 调用虚函数，执行服务器特定的功能
ServerFunction(nConnectedSocket, nListenSocket);

// 关闭连接
::close(nConnectedSocket);
::close(nListenSocket);

```



```

        return 0;
    }

private:
    // 虚函数，用于在派生类中实现具体的服务器功能
    virtual void ServerFunction(int nConnectedSocket, int nListenSocket)
    {
    }

private:
    int m_nServerPort; // 服务器端口
    char *m_strBoundIP; // 绑定的 IP 地址
    int m_nLengthOfQueueOfListen; // 监听队列长度
};

// 派生类，实现具体的服务器功能
class CMyTCPServer : public CTCPSTerver
{
public:
    // 构造函数，调用基类构造函数初始化服务器参数
    CMyTCPServer(int nServerPort, int nLengthOfQueueOfListen = 100, const char
*strBoundIP = NULL) : CTCPSTerver(nServerPort, nLengthOfQueueOfListen, strBoundIP)
    {
    }

    // 虚析构函数
    virtual ~CMyTCPServer()
    {
    }

private:
    // 实现基类中的虚函数，发送 "Hello World" 到客户端
    virtual void ServerFunction(int nConnectedSocket, int nListenSocket)
    {
        ::write(nConnectedSocket, "Hello World\n", 13);
    }
};

// 主函数
int main()
{
    // 创建并运行派生类对象
    CMyTCPServer myserver(4000);
    myserver.Run();
}

```

```
        return 0;
    }
}
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <memory.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <iostream>
```

```
// TCP 客户端基类
```

```
class CTCPClient
```

```
{
```

```
public:
```

```
    // 构造函数，用于初始化客户端参数
```

```
    CTCPClient(int nServerPort, const char *strServerIP)
```

```
    {
```

```
        // 初始化服务器端口
```

```
        m_nServerPort = nServerPort;
```

```
        // 处理服务器 IP
```

```
        int nlength = strlen(strServerIP);
```

```
        m_strServerIP = new char[nlength + 1];
```

```
        strcpy(m_strServerIP, strServerIP);
```

```
    }
```

```
    // 析构函数，释放动态分配的内存
```

```
    virtual ~CTCPClient()
```

```
    {
```

```
        delete[] m_strServerIP;
```

```
    }
```

```
public:
```

```
    // 主方法，运行客户端
```

```
    int Run()
```

```
    {
```

```
        // 创建客户端套接字
```

```
        int nClientSocket = ::socket(AF_INET, SOCK_STREAM, 0);
```

```
        if (-1 == nClientSocket)
```

```
        {
```

```
            // 如果创建套接字失败，打印错误信息，返回-1
```

```
            std::cout << "socket error" << std::endl;
```

```

        return -1;
    }

    // 配置服务器地址
    sockaddr_in ServerAddress;
    memset(&ServerAddress, 0, sizeof(sockaddr_in));
    ServerAddress.sin_family = AF_INET;

    // 处理服务器 IP
    if (::inet_pton(AF_INET, m_strServerIP, &ServerAddress.sin_addr) != 1)
    {
        // 如果转换 IP 地址失败，打印错误信息，关闭套接字，返回-1
        std::cout << "inet_pton error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }

    // 配置服务器端口
    ServerAddress.sin_port = htons(m_nServerPort);

    // 连接到服务器
    if (::connect(nClientSocket, (sockaddr *)&ServerAddress, sizeof(ServerAddress)) ==
-1)
    {
        // 如果连接失败，打印错误信息，关闭套接字，返回-1
        std::cout << "connect error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }

    // 调用虚函数，执行客户端特定的功能
    ClientFunction(nClientSocket);

    // 关闭连接
    ::close(nClientSocket);

    return 0;
}

// 虚函数，用于在派生类中实现具体的客户端功能
virtual void ClientFunction(int nConnectedSocket)
{
    // 默认的客户端功能为空，派生类可重写此函数以实现自定义的客户端行为
}

```

```

private:
    int m_nServerPort; // 服务器端口
    char *m_strServerIP; // 服务器 IP 地址
};

// 派生类，实现具体的客户端功能
class CMyTCPClient : public CTCPClient
{
public:
    // 构造函数，调用基类构造函数初始化客户端参数
    CMyTCPClient(int nServerPort, const char *strServerIP) : CTCPClient(nServerPort,
strServerIP)
    {
    }

    // 虚析构函数
    virtual ~CMyTCPClient()
    {
    }

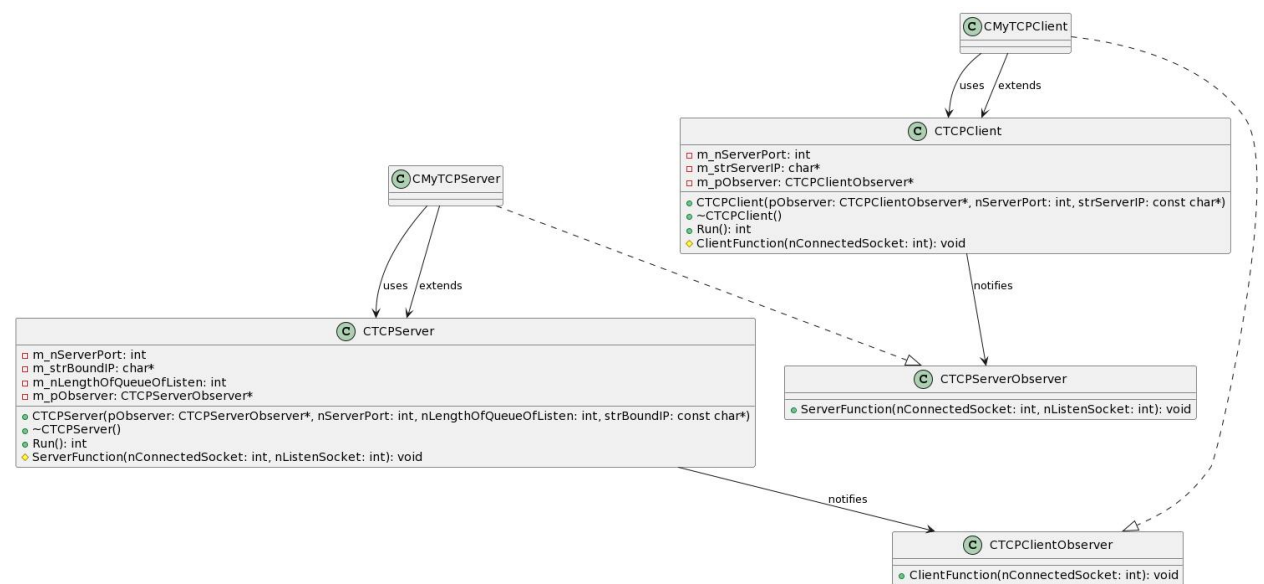
private:
    // 实现基类中的虚函数，从服务器端接收数据并打印
    virtual void ClientFunction(int nConnectedSocket)
    {
        // 读取服务器端发送的数据
        char buf[13];
        ::read(nConnectedSocket, buf, 13);

        // 打印接收到的数据
        std::cout << buf << std::endl;
    }
};

// 主函数
int main()
{
    // 创建并运行派生类对象
    CMyTCPClient client(4000, "127.0.0.1");
    client.Run();

    return 0;
}

```



```

hc20222081026@ubuntu: ~/Des...
hc20222081026@ubuntu: ~/Des...
hc20222081026@ubuntu: ~/Desktop/作业code/3.14$ ./server
  
```

```

hc20222081026@ubuntu: ~/Desktop/作业code/3.14
hc20222081026@ubuntu: ~/Desktop/作业code/3.14$ ./client
Hello World
hc20222081026@ubuntu: ~/Desktop/作业code/3.14$
  
```

● 基于接口的程序设计思想

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <memory.h>
#include <string.h>
#include <unistd.h>
#include <iostream>
  
```

```

// 服务器观察者基类
class CTCPClientObserver
{
public:
    CTCPClientObserver()
    {
        // 服务器观察者基类的构造函数，为空实现
    }
}
  
```

```

virtual ~CTCPServerObserver()
{
    // 服务器观察者基类的虚析构函数，为空实现
}

// 纯虚函数，用于在派生类中实现具体的服务器功能
virtual void ServerFunction(int nConnectedSocket, int nListenSocket) = 0;
};

// TCP 服务器基类
class CTCPServer
{
public:
    // 构造函数，用于初始化服务器参数
    CTCPServer(CTCPServerObserver *pObserver, int nServerPort, int
nLengthOfQueueOfListen = 100, const char *strBoundIP = NULL)
    {
        m_pObserver = pObserver;
        m_nServerPort = nServerPort;
        m_nLengthOfQueueOfListen = nLengthOfQueueOfListen;

        // 处理绑定 IP
        if (NULL == strBoundIP)
        {
            m_strBoundIP = NULL;
        }
        else
        {
            int length = strlen(strBoundIP);
            m_strBoundIP = new char[length + 1];
            memcpy(m_strBoundIP, strBoundIP, length + 1);
        }
    }

    virtual ~CTCPServer()
    {
        if (m_strBoundIP != NULL)
        {
            delete[] m_strBoundIP;
        }
    }

public:
    // 主方法，运行服务器

```

```

int Run()
{
    // 创建监听套接字
    int nListenSocket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (-1 == nListenSocket)
    {
        // 如果创建套接字失败，打印错误信息，返回-1
        std::cout << "socket error" << std::endl;
        return -1;
    }

    // 配置服务器地址
    sockaddr_in ServerAddress;
    memset(&ServerAddress, 0, sizeof(sockaddr_in));
    ServerAddress.sin_family = AF_INET;

    // 处理绑定 IP
    if (NULL == m_strBoundIP)
    {
        ServerAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    }
    else
    {
        if (::inet_pton(AF_INET, m_strBoundIP, &ServerAddress.sin_addr) != 1)
        {
            // 如果转换 IP 地址失败，打印错误信息，关闭套接字，返回-1
            std::cout << "inet_pton error" << std::endl;
            ::close(nListenSocket);
            return -1;
        }
    }

    // 配置服务器端口
    ServerAddress.sin_port = htons(m_nServerPort);

    // 绑定套接字和地址
    if (::bind(nListenSocket, (sockaddr *)&ServerAddress, sizeof(sockaddr_in)) == -1)
    {
        // 如果绑定失败，打印错误信息，关闭套接字，返回-1
        std::cout << "bind error" << std::endl;
        ::close(nListenSocket);
        return -1;
    }
}

```



```

// 开始监听连接请求
if (::listen(nListenSocket, m_nLengthOfQueueOfListen) == -1)
{
    // 如果监听失败，打印错误信息，关闭套接字，返回-1
    std::cout << "listen error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 接受连接请求
sockaddr_in ClientAddress;
socklen_t LengthOfClientAddress = sizeof(sockaddr_in);
int nConnectedSocket = ::accept(nListenSocket, (sockaddr *)&ClientAddress,
&LengthOfClientAddress);
if (-1 == nConnectedSocket)
{
    // 如果接受连接失败，打印错误信息，关闭套接字，返回-1
    std::cout << "accept error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 调用观察者的虚函数，执行服务器特定的功能
if (m_pObserver != NULL)
{
    m_pObserver->ServerFunction(nConnectedSocket, nListenSocket);
}

// 关闭连接
::close(nConnectedSocket);
::close(nListenSocket);

return 0;
}

private:
    int m_nServerPort;           // 服务器端口
    char *m_strBoundIP;          // 绑定的 IP 地址
    int m_nLengthOfQueueOfListen; // 监听队列长度
    CTCPServerObserver *m_pObserver; // 服务器观察者指针
};

// 派生类，实现具体的服务器功能
class CMYTCPServer : public CTCPServerObserver

```

```

{
public:
    // 构造函数，为空实现
    CMyTCPServer()
    {
    }

    // 虚析构函数，为空实现
    virtual ~CMyTCPServer()
    {
    }

private:
    // 实现基类中的虚函数，向连接的客户端发送 "Hello World" 消息
    virtual void ServerFunction(int nConnectedSocket, int nListenSocket)
    {
        ::write(nConnectedSocket, "Hello World\n", 13);
    }
};

// 主函数
int main()
{
    // 创建服务器观察者对象
    CMyTCPServer myserver;

    // 创建并运行服务器对象，将观察者对象传递给服务器
    CTCPServer tcpserver(&myserver, 4000);
    tcpserver.Run();

    return 0;
}

```

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <memory.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

```

```

// 客户端观察者基类
class CTCPClietObserver

```

```

{
public:
    CTCPClientObserver()
    {
        // 客户端观察者基类的构造函数，为空实现
    }

    virtual ~CTCPClientObserver()
    {
        // 客户端观察者基类的虚析构函数，为空实现
    }

    // 纯虚函数，用于在派生类中实现具体的客户端功能
    virtual void ClientFunction(int nConnectedSocket) = 0;
};

// TCP 客户端基类
class CTCPClient
{
public:
    // 构造函数，用于初始化客户端参数
    CTCPClient(CTCPClientObserver *pObserver, int nServerPort, const char *strServerIP)
    {
        m_pObserver = pObserver;
        m_nServerPort = nServerPort;

        // 处理服务器 IP
        int nlength = strlen(strServerIP);
        m_strServerIP = new char[nlength + 1];
        strcpy(m_strServerIP, strServerIP);
    }

    virtual ~CTCPClient()
    {
        delete[] m_strServerIP;
    }

public:
    // 主方法，运行客户端
    int Run()
    {
        // 创建客户端套接字
        int nClientSocket = ::socket(AF_INET, SOCK_STREAM, 0);
        if (-1 == nClientSocket)

```

```

{
    // 如果创建套接字失败，打印错误信息，返回-1
    std::cout << "socket error" << std::endl;
    return -1;
}

// 配置服务器地址
sockaddr_in ServerAddress;
memset(&ServerAddress, 0, sizeof(sockaddr_in));
ServerAddress.sin_family = AF_INET;

// 处理服务器 IP
if (::inet_pton(AF_INET, m_strServerIP, &ServerAddress.sin_addr) != 1)
{
    // 如果转换 IP 地址失败，打印错误信息，关闭套接字，返回-1
    std::cout << "inet_pton error" << std::endl;
    ::close(nClientSocket);
    return -1;
}

// 配置服务器端口
ServerAddress.sin_port = htons(m_nServerPort);

// 连接到服务器
if (::connect(nClientSocket, (sockaddr *)&ServerAddress, sizeof(ServerAddress)) ==
-1)
{
    // 如果连接失败，打印错误信息，关闭套接字，返回-1
    std::cout << "connect error" << std::endl;
    ::close(nClientSocket);
    return -1;
}

// 调用观察者的虚函数，执行客户端特定的功能
if (m_pObserver != NULL)
{
    m_pObserver->ClientFunction(nClientSocket);
}

// 关闭连接
::close(nClientSocket);

return 0;
}

```

```
private:
    int m_nServerPort;           // 服务器端口
    char *m_strServerIP;        // 服务器 IP 地址
    CTCPClientObserver *m_pObserver; // 客户端观察者指针
};
```

// 派生类，实现具体的客户端功能

```
class CMyTCPClient : public CTCPClientObserver
```

```
{
```

```
public:
```

```
    // 构造函数，为空实现
```

```
    CMyTCPClient()
```

```
{
```

```
}
```

```
    // 虚析构函数，为空实现
```

```
    virtual ~CMyTCPClient()
```

```
{
```

```
}
```

```
private:
```

```
    // 实现基类中的虚函数，从服务器端接收数据并打印
```

```
    virtual void ClientFunction(int nConnectedSocket)
```

```
{
```

```
    // 读取服务器端发送的数据
```

```
    char buf[13];
```

```
    ::read(nConnectedSocket, buf, 13);
```

```
    // 打印接收到的数据
```

```
    std::cout << buf << std::endl;
```

```
}
```

```
};
```

// 主函数

```
int main()
```

```
{
```

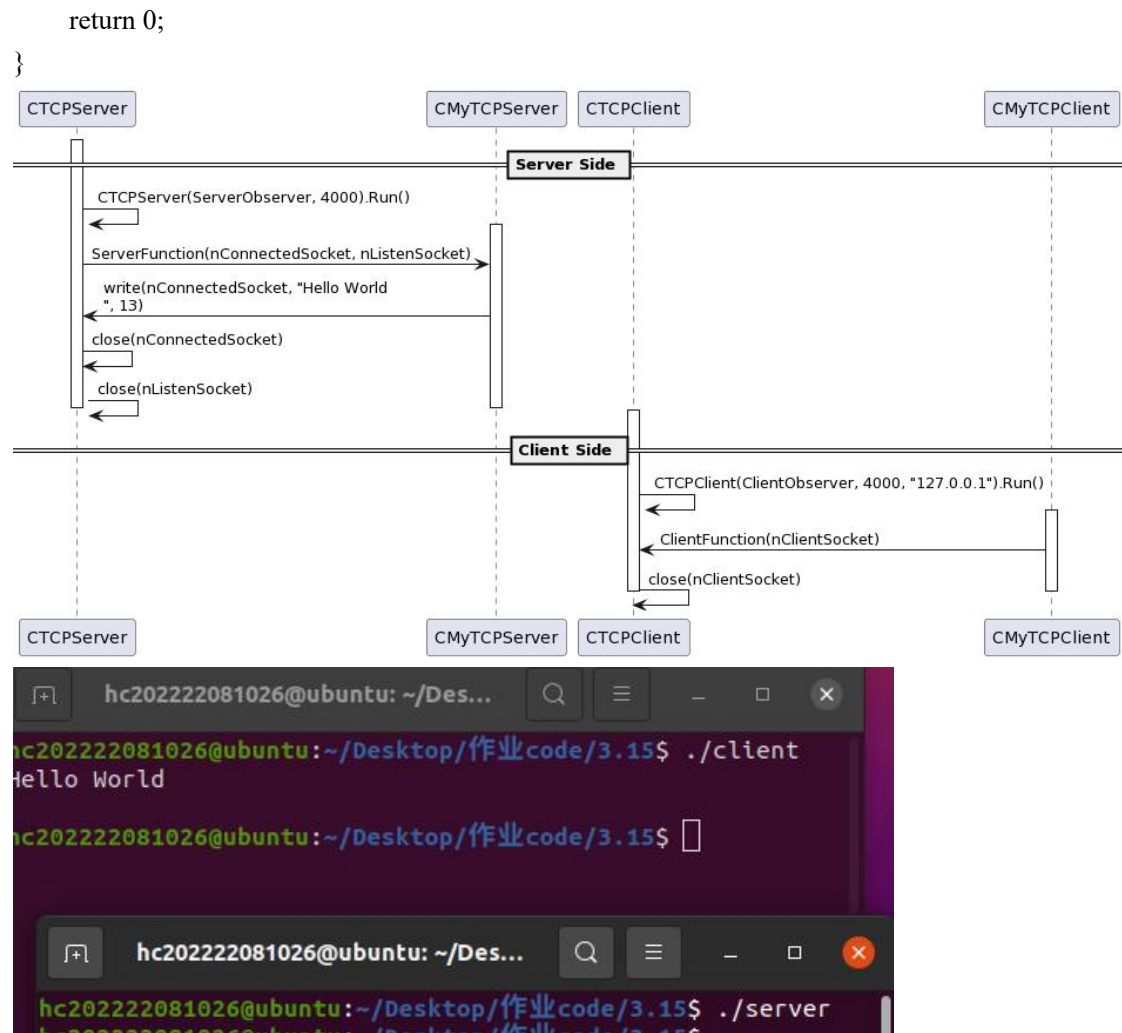
```
    // 创建客户端观察者对象
```

```
    CMyTCPClient client;
```

```
    // 创建并运行客户端对象，将观察者对象传递给客户端
```

```
    CTCPClient tcpclient(&client, 4000, "127.0.0.1");
```

```
    tcpclient.Run();
```



● 静态的面向对象的程序设计思想

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <memory.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

```

// 泛型 TCP 服务器模板类

```
template<typename T>
```

```
class CTCPServer
```

```
{
```

```
public:
```

```
// 构造函数，初始化服务器参数
```

```
CTCPServer(int nServerPort, int nLengthOfQueueOfListen = 100, const char *strBoundIP =
NULL)
```

```
{
```

```

        m_nServerPort = nServerPort;
        m_nLengthOfQueueOfListen = nLengthOfQueueOfListen;

        // 处理绑定 IP
        if (NULL == strBoundIP)
        {
            m_strBoundIP = NULL;
        }
        else
        {
            int length = strlen(strBoundIP);
            m_strBoundIP = new char[length + 1];
            memcpy(m_strBoundIP, strBoundIP, length + 1);
        }
    }

    // 虚析构函数，释放资源
    virtual ~CTCPServer()
    {
        if (m_strBoundIP != NULL)
        {
            delete [] m_strBoundIP;
        }
    }

public:
    // 主方法，运行服务器
    int Run()
    {
        // 创建监听套接字
        int nListenSocket = ::socket(AF_INET, SOCK_STREAM, 0);
        if (-1 == nListenSocket)
        {
            std::cout << "socket error" << std::endl;
            return -1;
        }

        // 配置服务器地址
        sockaddr_in ServerAddress;
        memset(&ServerAddress, 0, sizeof(sockaddr_in));
        ServerAddress.sin_family = AF_INET;

        // 处理绑定 IP
        if (NULL == m_strBoundIP)

```



```

{
    ServerAddress.sin_addr.s_addr = htonl(INADDR_ANY);
}
else
{
    if (::inet_pton(AF_INET, m_strBoundIP, &ServerAddress.sin_addr) != 1)
    {
        std::cout << "inet_pton error" << std::endl;
        ::close(nListenSocket);
        return -1;
    }
}

// 配置服务器端口
ServerAddress.sin_port = htons(m_nServerPort);

// 绑定监听套接字
if (::bind(nListenSocket, (sockaddr *)&ServerAddress, sizeof(sockaddr_in)) == -1)
{
    std::cout << "bind error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 开始监听
if (::listen(nListenSocket, m_nLengthOfQueueOfListen) == -1)
{
    std::cout << "listen error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 接受连接请求
sockaddr_in ClientAddress;
socklen_t LengthOfClientAddress = sizeof(sockaddr_in);
int nConnectedSocket = ::accept(nListenSocket, (sockaddr *)&ClientAddress,
&LengthOfClientAddress);
if (-1 == nConnectedSocket)
{
    std::cout << "accept error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

```

```

        // 获取当前对象的指针，并调用模板类的虚函数
        T *pT = static_cast<T *>(this);
        pT->ServerFunction(nConnectedSocket, nListenSocket);

        // 关闭连接
        ::close(nConnectedSocket);
        ::close(nListenSocket);

        return 0;
    }

private:
    void ServerFunction(int nConnectedSocket, int nListenSocket)
    {
        // 默认服务器函数为空实现，派生类可重写该函数以定义具体的服务器行为
    }

private:
    int m_nServerPort;        // 服务器端口
    char* m_strBoundIP;       // 绑定的 IP 地址
    int m_nLengthOfQueueOfListen; // 监听队列长度
};

// 派生类，实现具体的服务器行为
class CMyTCPServer : public CTCPSTerver<CMyTCPServer>
{
public:
    // 构造函数，调用基类构造函数初始化参数
    CMyTCPServer(int nServerPort, int nLengthOfQueueOfListen = 100, const char
*strBoundIP = NULL)
        : CTCPSTerver<CMyTCPServer>(nServerPort, nLengthOfQueueOfListen, strBoundIP)
    {
    }

    // 虚析构函数，为空实现
    virtual ~CMyTCPServer()
    {
    }

    // 重写基类的虚函数，实现具体的服务器行为
    void ServerFunction(int nConnectedSocket, int nListenSocket)
    {
        // 向连接的客户端发送消息
        ::write(nConnectedSocket, "Hello World\n", 13);
    }
}

```

```

    }
};

// 主函数
int main()
{
    // 创建并运行服务器对象
    CMyTCPServer myserver(4000);
    myserver.Run();

    return 0;
}

#include <sys/socket.h>
#include <netinet/in.h>
#include <memory.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

// 泛型 TCP 客户端模板类
template<typename T>
class CTCPClient
{
public:
    // 构造函数，初始化服务器端口和 IP 地址
    CTCPClient(int nServerPort, const char *strServerIP)
    {
        m_nServerPort = nServerPort;

        // 处理服务器 IP 地址
        int nlength = strlen(strServerIP);
        m_strServerIP = new char [nlength + 1];
        strcpy(m_strServerIP, strServerIP);
    }

    // 虚析构函数，释放资源
    virtual ~CTCPClient()
    {
        delete [] m_strServerIP;
    }

public:

```

```

// 主方法，运行客户端
int Run()
{
    // 创建客户端套接字
    int nClientSocket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (-1 == nClientSocket)
    {
        std::cout << "socket error" << std::endl;
        return -1;
    }

    // 配置服务器地址
    sockaddr_in ServerAddress;
    memset(&ServerAddress, 0, sizeof(sockaddr_in));
    ServerAddress.sin_family = AF_INET;

    // 处理服务器 IP 地址
    if (::inet_pton(AF_INET, m_strServerIP, &ServerAddress.sin_addr) != 1)
    {
        std::cout << "inet_pton error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }

    // 配置服务器端口
    ServerAddress.sin_port = htons(m_nServerPort);

    // 连接到服务器
    if (::connect(nClientSocket, (sockaddr*)&ServerAddress, sizeof(ServerAddress)) == -1)
    {
        std::cout << "connect error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }

    // 获取当前对象的指针，并调用模板类的虚函数
    T *pT = static_cast<T*>(this);
    pT->ClientFunction(nClientSocket);

    // 关闭客户端套接字
    ::close(nClientSocket);

    return 0;
}

```

```

// 虚函数，用于在派生类中定义具体的客户端行为
virtual void ClientFunction(int nConnectedSocket)
{
    // 默认客户端函数为空实现，派生类可重写该函数以定义具体的客户端行为
}

private:
    int m_nServerPort; // 服务器端口
    char *m_strServerIP; // 服务器 IP 地址
};

// 派生类，实现具体的客户端行为
class CMyTCPClient : public CTCPClient<CMyTCPClient>
{
public:
    // 构造函数，调用基类构造函数初始化参数
    CMyTCPClient(int nServerPort, const char *strServerIP) :
    CTCPClient<CMyTCPClient>(nServerPort, strServerIP)
    {
    }

    // 虚析构函数，为空实现
    virtual ~CMyTCPClient()
    {
    }

    // 重写基类的虚函数，实现具体的客户端行为
    void ClientFunction(int nConnectedSocket)
    {
        // 从连接的服务器接收消息
        char buf[13];
        ::read(nConnectedSocket, buf, 13);

        // 打印接收到的消息
        std::cout << buf << std::endl;
    }
};

// 主函数
int main()
{
    // 创建并运行客户端对象
    CMyTCPClient client(4000, "127.0.0.1");
}

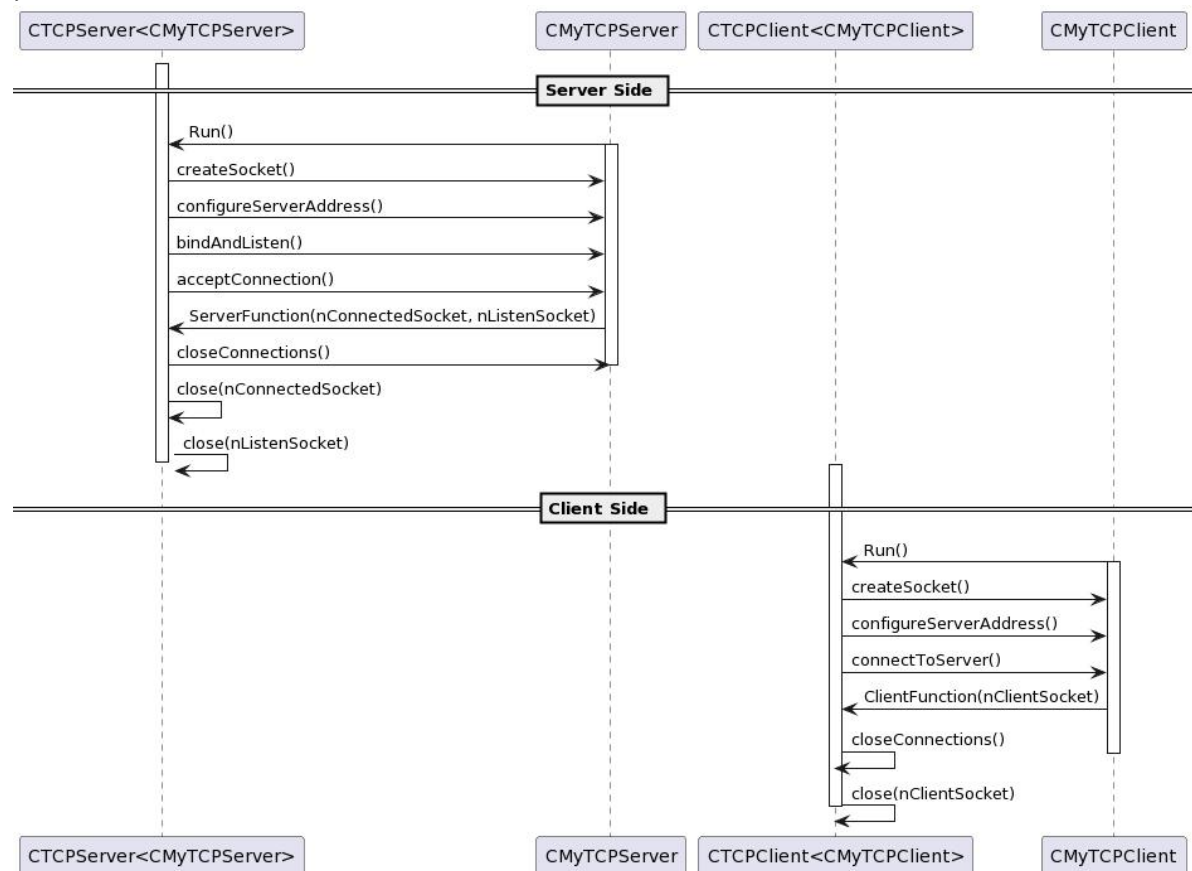
```

```

client.Run();

return 0;
}

```



```

hc202222081026@ubuntu: ~/Des...
hc202222081026@ubuntu:~/Desktop/作业code/3.16$ ./server
hc202222081026@ubuntu:~/Desktop/作业code/3.16$

hc202222081026@ubuntu: ~/Des...
hc202222081026@ubuntu:~/Desktop/作业code/3.16$ ./client
Hello World

```

● 面向方面的程序设计思想

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <memory.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

```

```

// 泛型 TCP 服务器模板类，继承连接处理器
template<typename ConnectionProcessor>
class CTCPServer : public ConnectionProcessor
{
public:
    // 构造函数，初始化服务器端口、监听队列长度和绑定 IP 地址
    CTCPServer(int nServerPort, int nLengthOfQueueOfListen = 100, const char *strBoundIP =
NULL)
    {
        m_nServerPort = nServerPort;
        m_nLengthOfQueueOfListen = nLengthOfQueueOfListen;

        // 处理绑定 IP 地址
        if (NULL == strBoundIP)
        {
            m_strBoundIP = NULL;
        }
        else
        {
            int length = strlen(strBoundIP);
            m_strBoundIP = new char[length + 1];
            memcpy(m_strBoundIP, strBoundIP, length + 1);
        }
    }

    // 虚析构函数，释放资源
    virtual ~CTCPServer()
    {
        if (m_strBoundIP != NULL)
        {
            delete [] m_strBoundIP;
        }
    }

public:
    // 主方法，运行服务器
    int Run()
    {
        // 创建服务器套接字
        int nListenSocket = ::socket(AF_INET, SOCK_STREAM, 0);
        if (-1 == nListenSocket)
        {
            std::cout << "socket error" << std::endl;
            return -1;
        }
    }
};

```



```

}

// 配置服务器地址
sockaddr_in ServerAddress;
memset(&ServerAddress, 0, sizeof(sockaddr_in));
ServerAddress.sin_family = AF_INET;

// 处理绑定 IP 地址
if (NULL == m_strBoundIP)
{
    ServerAddress.sin_addr.s_addr = htonl(INADDR_ANY);
}
else
{
    if (::inet_pton(AF_INET, m_strBoundIP, &ServerAddress.sin_addr) != 1)
    {
        std::cout << "inet_pton error" << std::endl;
        ::close(nListenSocket);
        return -1;
    }
}

// 配置服务器端口
ServerAddress.sin_port = htons(m_nServerPort);

// 绑定服务器套接字
if (::bind(nListenSocket, (sockaddr *)&ServerAddress, sizeof(sockaddr_in)) == -1)
{
    std::cout << "bind error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 监听连接请求
if (::listen(nListenSocket, m_nLengthOfQueueOfListen) == -1)
{
    std::cout << "listen error" << std::endl;
    ::close(nListenSocket);
    return -1;
}

// 接受连接并处理
sockaddr_in ClientAddress;
socklen_t LengthOfClientAddress = sizeof(sockaddr_in);

```

```

        int nConnectedSocket = ::accept(nListenSocket, (sockaddr *)&ClientAddress,
&LengthOfClientAddress);
        if (-1 == nConnectedSocket)
        {
            std::cout << "accept error" << std::endl;
            ::close(nListenSocket);
            return -1;
        }

        // 获取当前对象的指针，并调用模板类的虚函数
        ConnectionProcessor *pProcessor = static_cast<ConnectionProcessor *>(this);
        pProcessor->ServerFunction(nConnectedSocket, nListenSocket);

        // 关闭连接套接字和监听套接字
        ::close(nConnectedSocket);
        ::close(nListenSocket);

        return 0;
    }

```

private:

```

        int m_nServerPort; // 服务器端口
        char* m_strBoundIP; // 绑定 IP 地址
        int m_nLengthOfQueueOfListen; // 监听队列长度
    };

```

// 连接处理器类，实现具体的连接处理行为

class CMyTCPServer

{

public:

// 构造函数，为空实现

CMyTCPServer()

```

{
}

```

// 虚析构函数，为空实现

virtual ~CMyTCPServer()

```

{
}

```

// 具体的连接处理行为，向连接的客户端发送消息

void ServerFunction(int nConnectedSocket, int nListenSocket)

```

{

```

```

    ::write(nConnectedSocket, "Hello World\n", 13);

```

```

    }
};

// 主函数
int main()
{
    // 创建并运行服务器对象，使用 CMyTCPServer 作为连接处理器
    CTCPServer<CMyTCPServer> myserver(4000);
    myserver.Run();

    return 0;
}

```

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <memory.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

// 泛型 TCP 客户端模板类，继承连接处理器
template<typename ConnectionProcessor>
class CTCPCClient : public ConnectionProcessor
{
public:
    // 构造函数，初始化服务器端口和 IP 地址
    CTCPCClient(int nServerPort, const char *strServerIP)
    {
        m_nServerPort = nServerPort;

        int nlength = strlen(strServerIP);
        m_strServerIP = new char[nlength + 1];
        strcpy(m_strServerIP, strServerIP);
    }

    // 虚析构函数，释放资源
    virtual ~CTCPCClient()
    {
        delete[] m_strServerIP;
    }

public:

```

```

// 主方法，运行客户端
int Run()
{
    // 创建客户端套接字
    int nClientSocket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (-1 == nClientSocket)
    {
        std::cout << "socket error" << std::endl;
        return -1;
    }

    // 配置服务器地址
    sockaddr_in ServerAddress;
    memset(&ServerAddress, 0, sizeof(sockaddr_in));
    ServerAddress.sin_family = AF_INET;
    if (::inet_pton(AF_INET, m_strServerIP, &ServerAddress.sin_addr) != 1)
    {
        std::cout << "inet_pton error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }

    // 配置服务器端口
    ServerAddress.sin_port = htons(m_nServerPort);

    // 连接到服务器
    if (::connect(nClientSocket, (sockaddr *)&ServerAddress, sizeof(ServerAddress)) ==
-1)
    {
        std::cout << "connect error" << std::endl;
        ::close(nClientSocket);
        return -1;
    }

    // 获取当前对象的指针，并调用模板类的虚函数
    ConnectionProcessor *pProcessor = static_cast<ConnectionProcessor *>(this);
    pProcessor->ClientFunction(nClientSocket);

    // 关闭客户端套接字
    ::close(nClientSocket);

    return 0;
}

```

```

private:
    int m_nServerPort; // 服务器端口
    char *m_strServerIP; // 服务器 IP 地址
};

// 连接处理器类，实现具体的连接处理行为
class CMyTCPClient
{
public:
    // 构造函数，为空实现
    CMyTCPClient()
    {
    }

    // 虚析构函数，为空实现
    virtual ~CMyTCPClient()
    {
    }

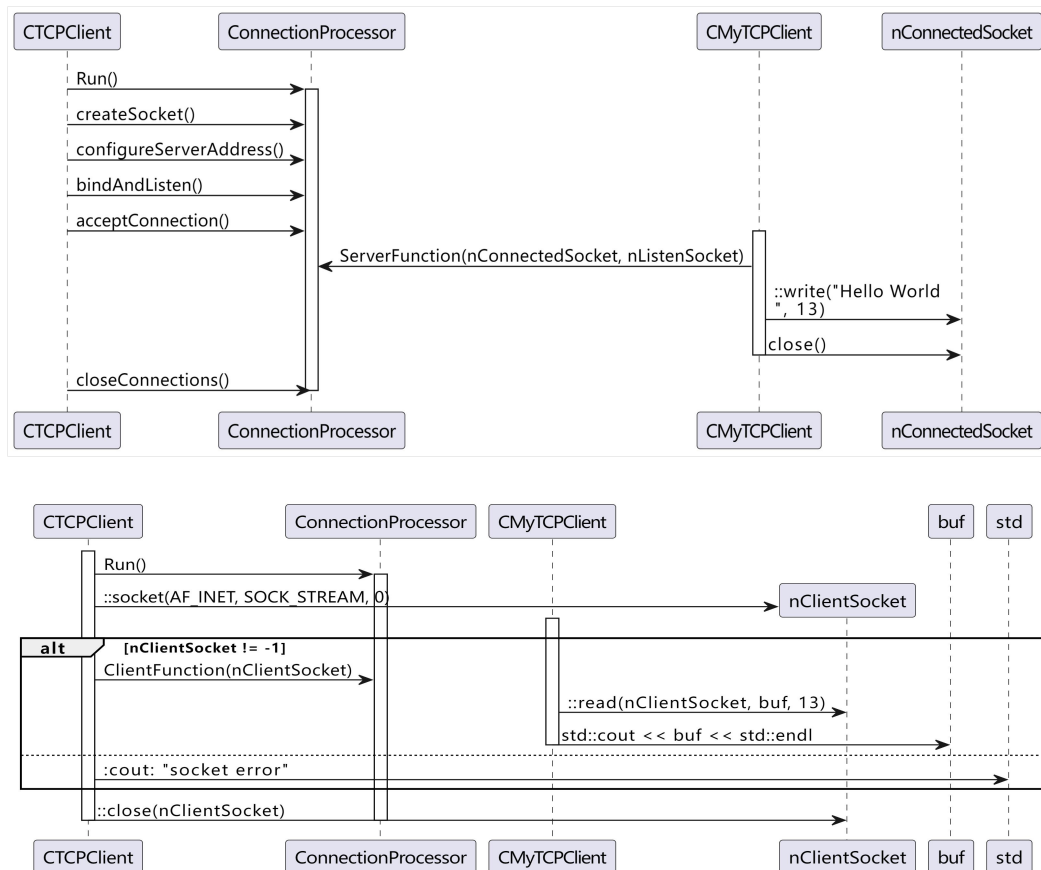
    // 具体的连接处理行为，从连接的服务器读取消息并输出
    void ClientFunction(int nConnectedSocket)
    {
        char buf[13];
        ::read(nConnectedSocket, buf, 13);

        std::cout << buf << std::endl;
    }
};

// 主函数
int main()
{
    // 创建并运行客户端对象，使用 CMyTCPClient 作为连接处理器
    CTCPCClient<CMyTCPClient> client(4000, "127.0.0.1");
    client.Run();

    return 0;
}

```



六、实验步骤:

给出五种通信库的封装源码、echo 服务器、客户端源码。分析各自的流程、时序、类关系等。代码中注释的行数占总行数的 20% 以上。

七、总结及心得体会:

让我深入了解了 TCP 服务器和客户端的基本原理。首先，服务器在一个特定端口监听，等待客户端的连接请求。一旦有客户端连接上来，服务器就和它建立起一个通信管道，可以在这个管道上传输数据。

八、对本实验过程及方法、手段的改进建议：

注释的重要性： 在代码中添加详细的注释，解释每个关键步骤的目的和作用。

错误处理： 考虑在代码中增加更多的错误处理机制，以便更好地处理潜在的错误情况。这将使代码更健壮，同时也有助于排除问题。

模块化： 将功能拆分成更小的模块或函数，以提高代码的可读性和可维护性。

报告评分：

指导教师签字：

电子科技大学

实验报告

学生姓名： 侯晨 学 号： 202222081026 指导教师： 丁 旭 阳

实验地点： 电子科技大学 实验时间： 2023/11/29

一、实验室名称：

Linux 环境高级编程实验室

二、实验项目名称：

插件框架实验

三、实验学时：

4 学时

四、实验目的：

基础知识

实例 1：插件的更新

实例 2：多插件的使用

实例 3：多插件的选择与使用

实例 4：插件设计优化--减少接口

综合练习

五、实验内容：

实验 1:

对文件进行编译，发现执行错误，

```
hc202222081026@ubuntu: ~/Desktop/code4/4.1
hc202222081026@ubuntu:~/Desktop/code4/4.1$ make
g++ -o test.o -c test.cpp -ldl
g++ -fpic -shared -o libtest.so a1.cpp a2.cpp
g++ -o test test.o -ldl
rm *.o
hc202222081026@ubuntu:~/Desktop/code4/4.1$ g++ -o test test.cpp -ldl
hc202222081026@ubuntu:~/Desktop/code4/4.1$ test
hc202222081026@ubuntu:~/Desktop/code4/4.1$ ./test
f1 error
./libtest.so: undefined symbol: f
Segmentation fault (core dumped)
hc202222081026@ubuntu:~/Desktop/code4/4.1$ a
```

寻找错误发现:

```
hc202222081026@ubuntu:~/Desktop/code4/4.1$ nm libtest.so
0000000000004040 b completed.8061
                U __cxa_atexit@@GLIBC_2.2.5
                w __cxa_finalize@@GLIBC_2.2.5
00000000000010c0 t deregister_tm_clones
0000000000001130 t __do_global_ctors_aux
0000000000003df0 d __do_global_ctors_aux_fini_array_entry
0000000000004038 d __dso_handle
0000000000003df8 d _DYNAMIC
00000000000012b4 t _fini
0000000000001170 t frame_dummy
0000000000003dd8 d __frame_dummy_init_array_entry
0000000000002190 r __FRAME_END__
0000000000001215 T g
0000000000004000 d _GLOBAL_OFFSET_TABLE_
00000000000011fc t _GLOBAL__sub_I_a1.cpp
0000000000001298 t _GLOBAL__sub_I_a2.cpp
                w __gmon_start__
000000000000200c r __GNU_EH_FRAME_HDR
0000000000001000 t _init
                w _ITM_deregisterTMCloneTable
                w _ITM_registerTMCloneTable
00000000000010f0 t register_tm_clones
0000000000004040 d __TMC_END__
0000000000001179 T _Z1fv
00000000000011af t _Z41_static_initialization_and_destruction_0ii
```

f 函数实际上在动态库中的名字是: `_Z1fv`

T 表示这是一个全局定义的文本符号，通常是代码段中的函数。

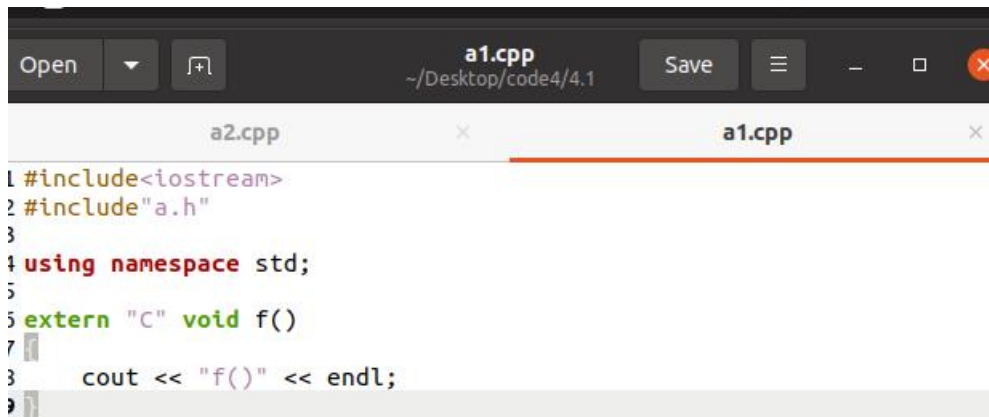
g 是在 `a2.cpp` 文件中声明并定义的函数。

`_Z1fv` 是 f 函数的名称，这是由于 C++ 使用了名称修饰 (name mangling) 机制，将函数名进行了改编，以包含参数类型等信息。在这个例子中，`_Z1fv` 表示 f 函数没有参数 (因为 `_Z1` 表示一个参数的函数，而 f 是函数名，v 表示返回类型为 `void`)。

对于函数 f，其修饰后的名称是 `_Z1fv`，这个修饰是由编译器根据函数的特征生成的。在这里，`_Z` 是一个标记，表示这是一个 C++ 修饰的名称，1 表示函数的参数个数，f 是函数的原始名称。

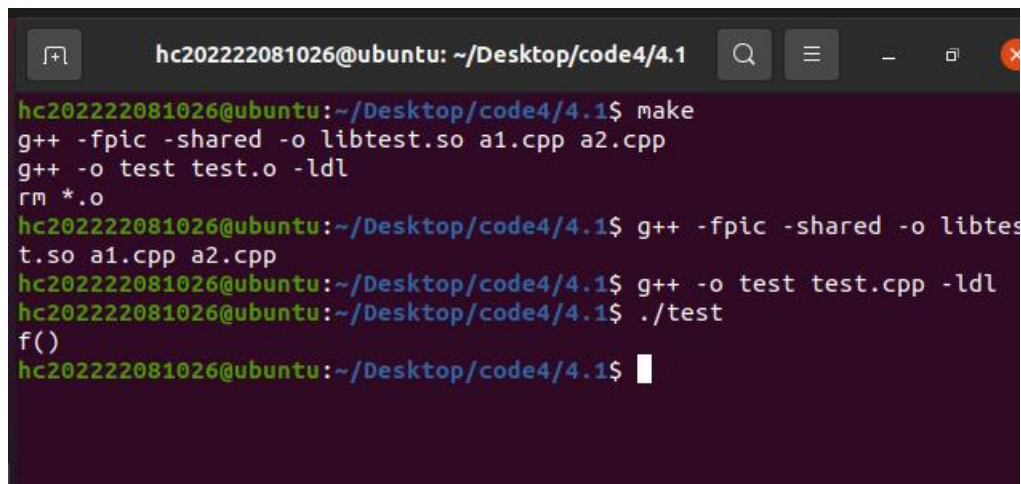
而对于函数 g，由于使用了 `extern "C"` 声明，它的名称没有进行 C++ 的名称修饰，因此在库中保留了原始的名称 g。

如果修改 `a1.cpp`:



```
Open  a1.cpp  ~/Desktop/code4/4.1  Save  -  □  ×
a2.cpp  a1.cpp
1 #include<iostream>
2 #include "a.h"
3
4 using namespace std;
5
6 extern "C" void f()
7 {
8     cout << "f()" << endl;
9 }
```

同时修改 a.h 中的声明，再次运行代码便可成功：



```
hc202222081026@ubuntu: ~/Desktop/code4/4.1
hc202222081026@ubuntu:~/Desktop/code4/4.1$ make
g++ -fpic -shared -o libtest.so a1.cpp a2.cpp
g++ -o test test.o -ldl
rm *.o
hc202222081026@ubuntu:~/Desktop/code4/4.1$ g++ -fpic -shared -o libtest.so a1.cpp a2.cpp
hc202222081026@ubuntu:~/Desktop/code4/4.1$ g++ -o test test.cpp -ldl
hc202222081026@ubuntu:~/Desktop/code4/4.1$ ./test
f()
hc202222081026@ubuntu:~/Desktop/code4/4.1$
```

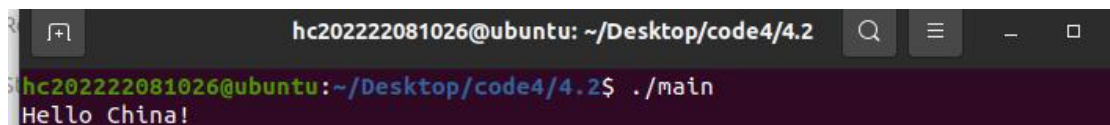
实验 2:

在原本文件中链接动态库，



```
function.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 extern "C" void Print()
6 {
7     cout << "Hello China!" << endl;
8 }
```

可以输出 hello china



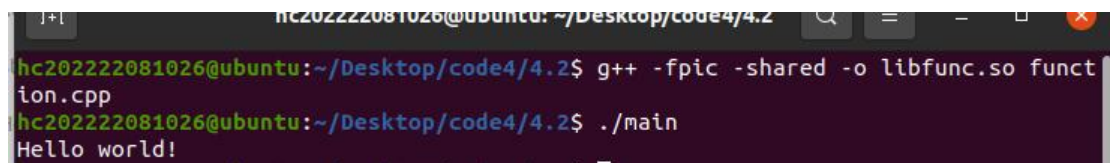
```
hc202222081026@ubuntu: ~/Desktop/code4/4.2
hc202222081026@ubuntu:~/Desktop/code4/4.2$ ./main
Hello China!
```

我们不改变 main，修改 function.cpp



```
function.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 extern "C" void Print()
6 {
7     cout << "Hello world!" << endl;
8 }
```

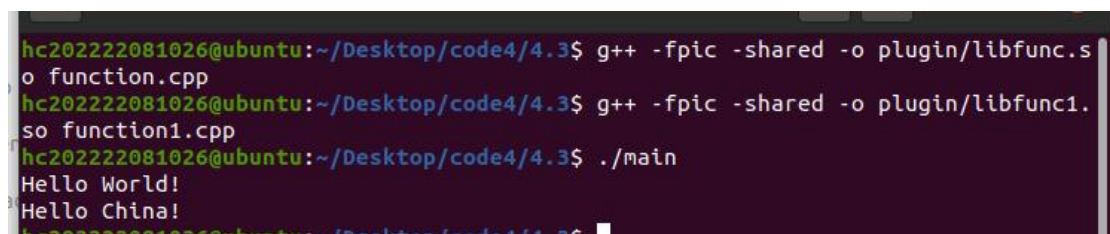
同时加载修改后的动态库，输出 main:



```
hc202222081026@ubuntu: ~/Desktop/code4/4.2
hc202222081026@ubuntu:~/Desktop/code4/4.2$ g++ -fpic -shared -o libfunc.so function.cpp
hc202222081026@ubuntu:~/Desktop/code4/4.2$ ./main
Hello world!
```

成功修改输出！

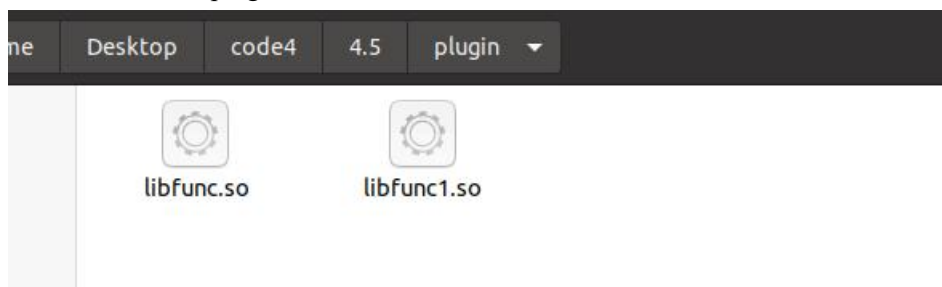
版本二:



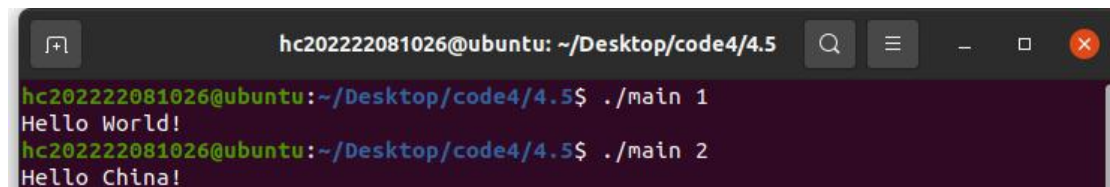
```
hc202222081026@ubuntu:~/Desktop/code4/4.3$ g++ -fpic -shared -o plugin/libfunc.so function.cpp
hc202222081026@ubuntu:~/Desktop/code4/4.3$ g++ -fpic -shared -o plugin/libfunc1.so function1.cpp
hc202222081026@ubuntu:~/Desktop/code4/4.3$ ./main
Hello World!
Hello China!
```

版本三:

将动态库放置于 plugin 中

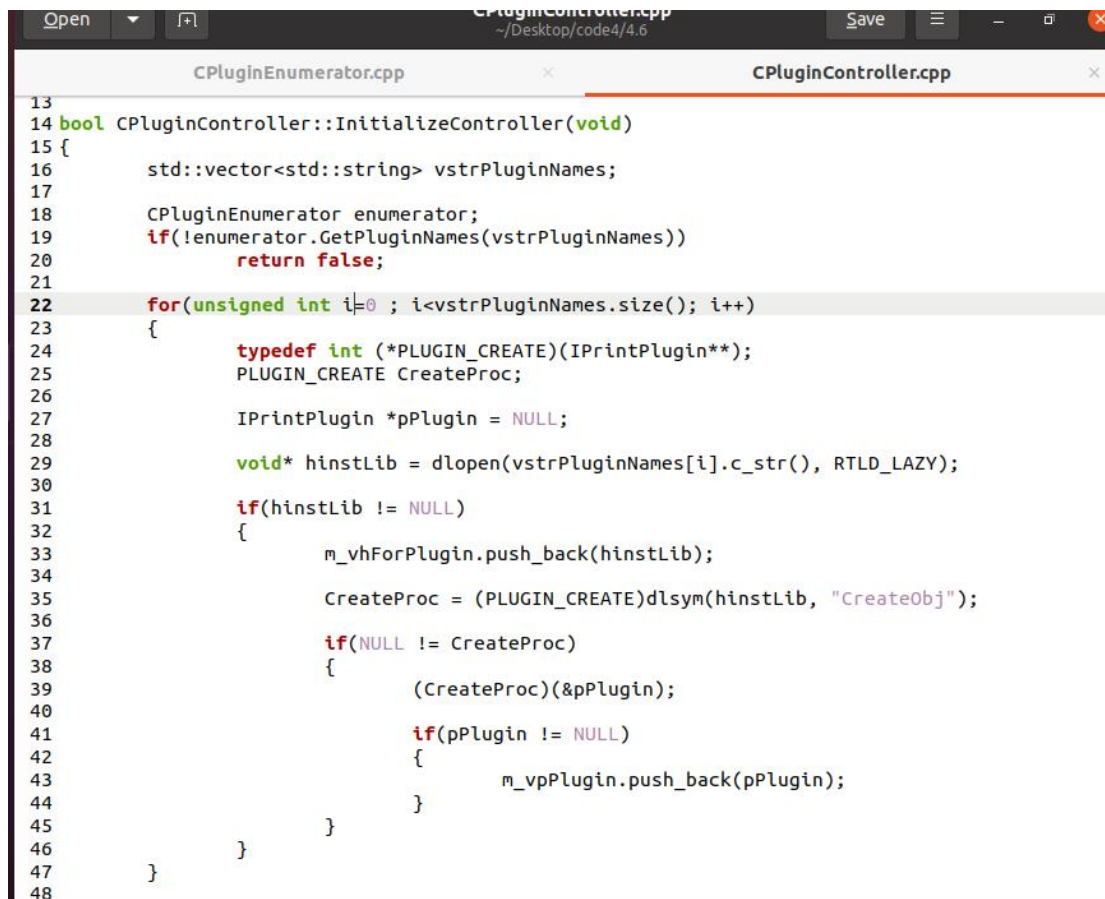


通过设置第二个参数调用不同的任务（执行不同的输出）



```
hc202222081026@ubuntu: ~/Desktop/code4/4.5
hc202222081026@ubuntu:~/Desktop/code4/4.5$ ./main 1
Hello World!
hc202222081026@ubuntu:~/Desktop/code4/4.5$ ./main 2
Hello China!
```

版本四:



```
13
14 bool CPluginController::InitializeController(void)
15 {
16     std::vector<std::string> vstrPluginNames;
17
18     CPluginEnumerator enumerator;
19     if(!enumerator.GetPluginNames(vstrPluginNames))
20         return false;
21
22     for(unsigned int i=0 ; i<vstrPluginNames.size(); i++)
23     {
24         typedef int (*PLUGIN_CREATE)(IPrintPlugin**);
25         PLUGIN_CREATE CreateProc;
26
27         IPrintPlugin *pPlugin = NULL;
28
29         void* hinstLib = dlopen(vstrPluginNames[i].c_str(), RTLD_LAZY);
30
31         if(hinstLib != NULL)
32         {
33             m_vhForPlugin.push_back(hinstLib);
34
35             CreateProc = (PLUGIN_CREATE)dlsym(hinstLib, "CreateObj");
36
37             if(NULL != CreateProc)
38             {
39                 (CreateProc>(&pPlugin);
40
41                 if(pPlugin != NULL)
42                 {
43                     m_vpPlugin.push_back(pPlugin);
44                 }
45             }
46         }
47     }
48 }
```

这个函数的作用是初始化插件控制器。它使用 CPluginEnumerator 类获取所有可用插件的文件名，并尝试加载每个插件。对于每个插件，它使用 dlopen 函数加载动态链接库，获取插件工厂函数的地址，然后调用该函数创建插件对象，并将插件对象添加到成员变量中。

```
52 bool CPluginController::ProcessRequest(int FunctionID)
53 {
54     for(unsigned int i = 0; i < m_vpPlugin.size(); i++)
55     {
56         if(m_vpPlugin[i]->GetID() == FunctionID)
57         {
58             m_vpPlugin[i]->Print();
59             break;
60         }
61     }
62     return true;
63 }
64 }
```

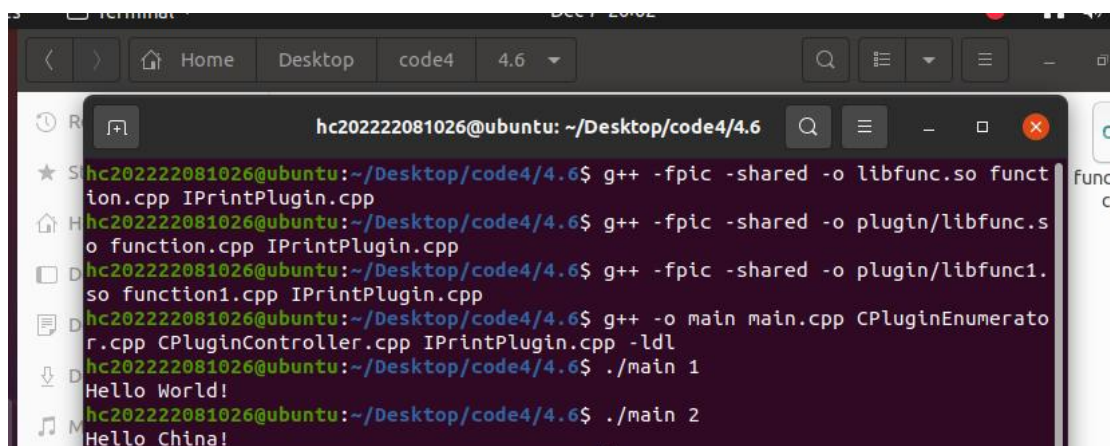
这个函数用于处理请求，通过遍历插件列表，查找匹配请求 ID 的插件，并调用其 Print 方法。


```

65
66 bool CPluginController::ProcessHelp(void)
67 {
68     std::vector<std::string> vstrPluginNames;
69
70     CPluginEnumerator enumerator;
71     if(!enumerator.GetPluginNames(vstrPluginNames))
72         return false;
73
74     for(unsigned int i=0 ; i<vstrPluginNames.size(); i++)
75     {
76         typedef int (*PLUGIN_CREATE)(IPrintPlugin**);
77         PLUGIN_CREATE CreateProc;
78
79         IPrintPlugin *pPlugin = NULL;
80
81         void* hinstLib = dlopen(vstrPluginNames[i].c_str(), RTLD_LAZY);
82
83         if(hinstLib != NULL)
84         {
85             CreateProc = (PLUGIN_CREATE)dlsym(hinstLib, "CreateObj");
86
87             if(NULL != CreateProc)
88             {
89                 (CreateProc>(&pPlugin);
90
91                 if(pPlugin != NULL)
92                 {
93                     pPlugin->Help();
94                 }
95             }
96         }

```

这个函数用于显示帮助信息。它遍历所有插件，加载每个插件并调用其 Help 方法。



```

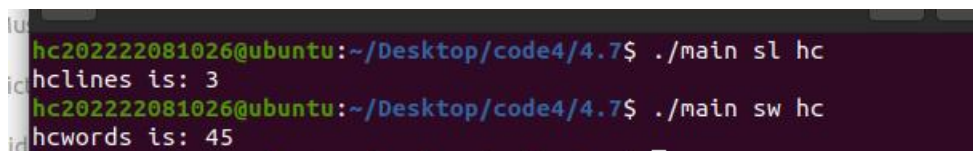
hc202222081026@ubuntu: ~/Desktop/code4/4.6
$ g++ -fpic -shared -o libfunc.so function.cpp IPrintPlugin.cpp
$ g++ -fpic -shared -o plugin/libfunc.so function.cpp IPrintPlugin.cpp
$ g++ -fpic -shared -o plugin/libfunc1.so function1.cpp IPrintPlugin.cpp
$ g++ -o main main.cpp CPluginEnumerator.cpp CPluginController.cpp IPrintPlugin.cpp -ldl
$ ./main 1
Hello World!
$ ./main 2
Hello China!

```

编译 main.cpp、CPluginEnumerator.cpp、CPluginController.cpp 和 IPrintPlugin.cpp 四个源文件，生成可执行文件 main。它通过 -ldl 链接到动态链接库。

综合练习：

装载动态库后进行编译 main.cpp 以后 可以进行运行对应函数



```

hc202222081026@ubuntu: ~/Desktop/code4/4.7$ ./main sl hc
hclines is: 3
hc202222081026@ubuntu: ~/Desktop/code4/4.7$ ./main sw hc
hcwords is: 45

```

代码中，统计字符数的逻辑是通过 read 函数读取文件内容并逐字符增加计数的。在英文文本中，通常每个字符都是一个字节。但是对于中文字符，情况就不同了。

在 UTF-8 编码中，一个中文字符可能由多个字节组成，因此对于包含中文字符的文本，一个字符的字节数可能大于 1。

创建了一个名为 hc 的 txt 文档，放于 main 的同级目录下，内容是：

```
1 black lives matter
2 原神启动
3 一眼丁真
```

如果以后可以添加功能，例如打印文本内容功能（print words,pw），添加代码 function2.cpp

```
virtual void Fun(char *Document) override
{
    int fp;
    char buffer[4096];
    ssize_t bytesRead;

    // open file
    if ((fp = open(Document, O_RDONLY)) == -1)
    {
        cout << "Can not open: " << Document << endl;
        return;
    };

    cout << "Content of " << Document << ":" << endl;

    // read and print the content
    while ((bytesRead = read(fp, buffer, sizeof(buffer))) > 0)
    {
        cout.write(buffer, bytesRead);
    }

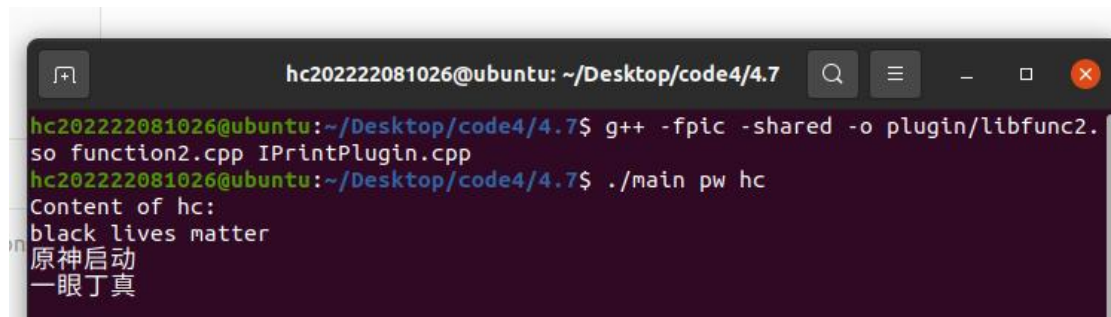
    close(fp);
    cout << endl;
};
```

同时命名 pw 可供调用：



```
1 #include <iostream>
2 #include "IPrintPlugin.h"
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <string.h>
6
7 using namespace std;
8
9 const int FUNC_ID = 4;
10 char FUNC_NAME[] = "pw"; // print words
11
```

然后进行动态库的生成与调用：



```
hc202222081026@ubuntu: ~/Desktop/code4/4.7
hc202222081026@ubuntu:~/Desktop/code4/4.7$ g++ -fpic -shared -o plugin/libfunc2.so function2.cpp IPrintPlugin.cpp
hc202222081026@ubuntu:~/Desktop/code4/4.7$ ./main pw hc
Content of hc:
black lives matter
原神启动
一眼丁真
```

实验成功

六、实验步骤：

PPT 上的 4 个版本程序，以及综合练习

七、总结及心得体会：

动态链接库的使用： 通过示例代码，学到了如何使用动态链接库（DLL）实现插件式架构。动态链接库可以在程序运行时动态加载，为应用程序提供灵活的扩展性。

C++ 类的设计： 了解了如何设计和实现 C++类，包括构造函数、析构函数、虚函数、成员函数等。这些都是面向对象编程的基础概念，用于构建模块化和可维护的代码。

文件操作和 IO： 学到了使用 C++进行文件操作的基本方法，包括打开文件、读取文件内容和关闭文件。这对于处理文本文件、读取插件等方面是很有用的。

插件式架构的设计： 了解了如何设计一个插件式架构，通过动态加载不同的插件实现不同的功能。这样的架构使得程序更加灵活，可以在运行时扩展功能。

命令行参数的处理： 学到了如何在 C++程序中处理命令行参数，

根据不同的参数执行不同的逻辑。这对于实现命令行工具或应用程序非常有用。

八、对本实验过程及方法、手段的改进建议：

建议 **ftp** 网站上文件可以传一个 **zip** 压缩文件，不然对于文件多的实验只能一个一个点击下载很浪费时间。

报告评分：

指导教师签字：