



第一章作业

2.1

apt-get 安装软件的整体步骤：

- (1).从list文件中检索可用的源,从其中找到这个软件包
- (2).生成软件依赖树，将需要的依赖包提前列出来，在安装所需软件之前进行安装
- (3).从源镜像站点下载软件包
- (4).安装依赖包及软件并配置

Ubuntu 如何管理软件依赖关系和软件版本：

- (1).Ubuntu使用APT包管理工具，Ubuntu上用与Debian一样的Deb软件管理工具，apt-get就是Ubuntu的Deb软件管理工具，即APT包管理工具。这个软件会从Ubuntu的软件源库里调用安装所需要安装的包，而且可以自动分析和解决依赖关系，并且将所依赖的软件都以此类推的管理安装妥当。
- (2).在Ubuntu下需要安装指定版本软件的时候，我们只用在安装时，在其包名后加上我们需哦需要的版本号，例如我们想安装5.4版本的gcc：`sudo apt-get install gcc-5.4`

2.2

软件源：

Deb软件管理的应用程序安装库存放着软件包，apt命令从软件源中下载软件，在/etc/apt/soure.list中可以为apt配置软件源。Ubuntu中软件源细分两种：

Ubuntu官方软件源和PPA软件源。

Ubuntu 官方软件源中包含了 Ubuntu 系统中所用到的绝大部分的软件，它对应的源列表是 /ect/apt/soure.list。在这个文件中，记录了 Ubuntu 官方源的地址。清华源、阿里源等镜像地址，它和 Ubuntu 官方的镜像是相同的，替换主要是为了加快 apt 安装和更新软件源的速度。

PPA 源出现的背景是因为系统自带的源是很有限的，我们肯定需要一些其他的软件包然而如果是直接下载deb格式的文件的的话，又不能获取到更新和维护，所以这就用到了十分重要的 PPA 源了。

如何更换系统自带的软件源：

第一种通过ubuntu自带软件software&update替换

第二种通过更换源文件替换

由于我使用WSL2，则直接采用第二种方法

备份源文件，打开terminal，以下命令在terminal输入

```
1| cd /etc/apt/
```

```
2| sudo cp sources.list sources.list.bak
```

3| 替换清华源，打开链接选择对应Ubuntu版本：

<https://mirrors.tuna.tsinghua.edu.cn/help/ubuntu/>

选择自己的ubuntu版本，将框内的内容复制；

4| 在ubuntu的termina内执行下列命令：

```
sudo gedit /etc/apt/sources.list
```

会自动打开文件，删除所有内容，将上面复制的清华源粘贴到文件中，点击保存

更新清华源，执行下面命令

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

安装来自第三方软件源中的软件：

以Ubuntu18.04系统为例，这也是推荐的版本以及我在使用的版本，安装搜狗拼音时直接下载deb安装包，用dpkg进行安装，使用 `sudo apt install -f` 进行修复即可。

2.3

除了 apt-get 以外，还有什么方式在系统中安装所需软件：

(1).下载deb格式文件进行离线安装；使用dpkg命令进行安装和卸载deb安装包。

(2).apt-get方式在线安装；主要用于在线从互联网的软件仓库中 搜索、安装、升级、卸载 软件。

(3).make方式安装；对于某些软件仓库没有的库或者安装包，或者下载过慢的这种情况，我们可以选择直接下载其源码，然后进行源码安装。

除了 Ubuntu 以外，其他发行版使用什么软件管理工具？请至少各列举两种：

CentOS：RPM YUM

Fedora：DNF

DNF 是新一代的 rpm 软件包管理器。他首先出现在 Fedora 18 这个发行版中。而最近，它取代了 yum，正式成为 Fedora 22 的包管理器。

2.4

环境变量 PATH 是什么，有什么用途：

系统通过PATH这个环境变量来获得可执行文件所在的位置，然后运行一下对应的可执行文件。

LD_LIBRARY_PATH 是什么：

程序已经成功编译并且链接成功后，使用LD_LIBRARY_PATH来搜索目录，该变量中只有动态库有意义。

指令 ldconfig 有什么用途：

ldconfig是一个动态链接库管理命令，为了让动态链接库为系统所共享，还需运行动态链接库的管理命令ldconfig。ldconfig的用途，主要是在默认搜寻目录（/lib和/usr/lib）以及动态库配置文件/etc/ld.so.conf内所列的目录下,搜索出可共享的动态链接库（lib.so），进而创建出动态装入程序（ld.so）所需的连接和缓存文件.缓存文件默认为 /etc/ld.so.cache，此文件保存已排好序的动态链接库名字列表。

2.5

Linux文件权限：

r (read)：可读取此档案的实际内容，如读取文本文件的文字内容等；

w (write)：可以编辑、新增或者是修改该档案的内容(但不包含删除该档案)；

x (eXecute)：该档案具有可以被系统执行的权限。

例如：有一个文件的权限数据为『`rwxr-xr--`』，那么前3个字符表示文件的拥有者可以对文件进行读、写、执行三个操作，中间3个字符表示该文件的所属组可以对文件进行读和执行操作，最后3个字符表示其他群组的用户只能对该文件进行读操作。

修改文件权限：

- (1).改变所属群组, `chgrp` 操作指令：`"chgrp 群组名称 档案或目录"`。
- (2).改变档案拥有者, `chown` 操作指令：`"chown [-R] 账号名称 档案或目录"`，此外还可以直接修改群组名称 操作指令：`"chown [-R] 账号名称:组名 档案或目录"`。
- (3).改变权限, `chmod`，此权限可分为数字和字符两种。

数字指令：

档案的权限字符为：『`-rwxrwxrwx`』，这九个权限是三个三个一组的！其中，我们可以使用数字来代表各个权限，各权限的分数对照表如下：

`r:4`

`w:2`

`x:1`

每种身份(owner/group/others)各自的三个权限(r/w/x)分数是需要累加的，例如当权限为：『`-rwxrwx---`』分数则是：

`owner = rwx = 4+2+1 = 7`

`group = rwx = 4+2+1 = 7`

`others= --- = 0+0+0 = 0`

所以等一下我们设定权限的变更时，该档案的权限数字就是770啦！变更权限的指令 `chmod` 的语法是这样的：

`chmod [-R] xyz 档案或目录`

选项与参数：

`xyz`：就是刚刚提到的数字类型的权限属性，为 `rwx` 属性数值的相加。

字符指令：

基本上九个权限分别是(1)user (2)group (3)others 三种身份。可以借由 `u`, `g`, `o` 来代表三种身份的权限！此外，`a` 则代表 `all` 亦即全部的身份！`+`(加入) `-`(除去) `=`(设定)！那么读写的权限就可以写成 `r`, `w`, `x`！举个栗子：

假如我们要『设定』一个档案的权限成为『`-rwxr-xr-x`』时，基本上就是：

user (u)：具有可读、可写、可执行的权限；

group 与 others (g/o)：具有可读与执行的权限。

操作指令为："chmod u=rwx,go=rx .bashrc"。

如果我不知道原先的文件属性，而我只想要增加.bashrc 这个档案的每个人均可写入的权限，那举我就可以使用："chmod a+w .bashrc"。

而如果是要将权限去掉而开更励其他已存在的权限呢？例如要拿掉全部人的可执行权限，则："chmod a-x .bashrc"。

tips：-R：进行递归(recursive)的持续变更，亦即连同次目录下的所有档案都会变更，

只有root或者文件拥有者才可以改变文件属性，群组名称必须是已经存在的。

2.6

Linux 用户和用户组是什么概念？用户组的权限是什么意思？有哪些常见的用户组？

由于Linux是个多人多任务的系统，因此可能常常会有多人同时使用这部主机来进行工作的情况发生，为了考虑每个人的隐私权以及每个人喜好的工作环境，以及每个人想拥有共享或者独享的文件权限，因此诞生了Linux用户的概念。

当存在团队开发时，群组可以保证团队内的用户都可以访问共享信息，而群组外的成员（引申概念：其他人）无法访问，且群组内的每个成员都可以拥有独属于自己的空间，即私人权限。其中一个用户可以加入多个群组。

用户分为普通用户、管理员用户（root用户）和系统用户。普通用户在系统上的任务是进行普通的工作，root用户对系统具有绝对的控制权，但操作不当会对系统造成损毁。所以在进行简单任务是进行使用普通用户。

用户组是用户的容器，通过组，我们可以更加方便的归类、管理用户。用户能从用户组继承权限，一般分为普通用户组，系统用户组，私有用户组。当创建一个新用户时，若没有指定他所属于的组，系统就建立一个与该用户同名的私有组。当然此时该私有组中只包含这个用户自己。标准组可以容纳多个用户,若使用标准组,在创建一个新的用户时就应该指定他所属于的组。

2.7

常见的 Linux 下 C++ 编译器有哪几种？在你的机器上，默认用的是哪一种？它能够支持 C++ 的 哪个标准？：

JetBrains CLion、 Visual Studio Code、 Clang、 GCC、 Visual Studio 2022 with WSL2

默认安装CLion，它能够支持C++98 ~ C++23

3.1

SLAM应用的方向：

机器人、无人驾驶、无人机、AR/VR、医疗

3.2

SLAM 中定位与建图的关系：

定位是为了精确地确定当前设备在某个环境中的姿态和位置；建图将周围环境的观测部分整合到一个单一的模型中。最初定位和建图是两个相互独立的关系，后来发现这两个步骤是相互依赖的。建图的准确性依赖于定位精度，而定位的实现又离不开精确的建图。

定位的同时需要建图的理由：

SLAM强调在未知环境下进行整个过程，如果在未知环境下进行定位，首先需要能够识别并理解周围的环境。再利用环境中的外部信息作为定位的基准，所以需要对所处的环境进行建图。

3.3

SLAM 发展史：

1986年在旧金山举办的IEEE机器人自动化会议中是最早的关于概率SLAM问题被提及。经过几年的深耕，Smith等人发表了具有里程碑意义的论文。在最早提出SLAM的一系列论文中，当时的人们称它为“空间状态不确定性的估计”。而SLAM这一概念最早由Hugh Durrant-Whyte 和 John J.Leonard提出。SLAM主要用于解决移动机器人在未知环境中运行时定位导航与地图构建的问题。从SLAM的提出到现在已有三十多年。

我们可以将它划分成哪几个阶段：

- (1).基于距离传感器的传统SLAM技术：SLAM技术最早是基于概率统计的扩展卡尔曼滤波的方法（EKF-SLAM），后来经过改进学者们提出了Fast SLAM方法，该方法大大减小计算量，提升运算速度。
- (2).基于视觉传感器的SLAM技术：2010年后，越来越多的SLAM系统采用视觉传感器作为主要的输入信号，提出了基于特征法的SALM、基于直接法的SLAM
- (3).SLAM技术未来研究热点将是多传感器融合SLAM、语义SLAM、SLAM于深度学习的结合

3.4

从什么时候开始 SLAM 区分为前端和后端？

SLAM系统结构出现前后端应该是在出现基于视觉的SLAM方法之后，大概是在2010年后。

为什么我们要把 SLAM 区分为前端和后端？

SLAM系统的体系结构包括两个主要部分：前端（或者叫视觉里程计）与后端。前端将传感器数据抽象成适用于估计的模型，估算相邻图像间相机的运动，以及局部地图的样子；而后端在这些经由前端处理的抽象数据上执行推理。后端接受不同时刻视觉里程计测量的相机位姿，以及回环检测的信息，对它们进行优化，得到全局一致的轨迹和地图。前后端所实现的功能不同，目的不同。

3.5

列举三篇在 SLAM 领域的经典文献：

- [1] Davison A J, Reid I D, Molton N D, et al. MonoSLAM: Real-time single camera SLAM[J]. IEEE transactions on pattern analysis and machine intelligence, 2007, 29(6): 1052-1067.
- [2] Mourikis A, Roumeliotis S. A multi-state constraint Kalman filter for vision-aided inertial navigation[C] //Proceedings of IEEE International Conference on Robotics and Automation. Los Alamitos: IEEE Computer Society Press, 2007: 3565-3572
- [3] Mur-Artal R, Montiel J M M, Tardos J D. ORB-SLAM: a versatile and accurate monocular SLAM system[J]. IEEE transactions on robotics, 2015, 31(5): 1147-1163.

4

按照要求组织源代码文件，并书写 CMakeLists.txt，为你的库提供 Findhello.cmake 文件，让其他用户可以通过 find_package 命令找到你的库，并实际测试

源码文件目录结构：

```
root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/hello# tree
.
├── CMakeLists.txt
├── COPYRIGHT
├── README
├── cmake
│   └── Findhello.cmake
├── doc
│   └── hello.txt
├── include
│   └── hello.h
├── log
├── runhello.sh
├── src
│   └── hello.cpp
└── useHello.cpp
```

源码文件

CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.10)

PROJECT(sayHello)

SET(CMAKE_BUILD_TYPE "Release")
INCLUDE_DIRECTORIES(./include)
ADD_LIBRARY(hello SHARED ./src/hello.cpp)

INSTALL(TARGETS hello LIBRARY DESTINATION lib ARCHIVE DESTINATION lib)
INSTALL(FILES ./include/hello.h DESTINATION include)

SET(CMAKE_MODULE_PATH /blueLinux/code/4ofL1/hello/cmake)
MESSAGE(${CMAKE_MODULE_PATH})
```



```

FIND_PACKAGE(hello REQUIRED)
IF(HELLO_FOUND)
    MESSAGE(${CMAKE_MODULE_PATH})
    ADD_EXECUTABLE(sayHello useHello.cpp)
    TARGET_LINK_LIBRARIES(sayHello hello)
ENDIF(HELLO_FOUND)

```

Findhello.cmake

```

FIND_PATH(HELLO_INCLUDE_DIR hello.h /usr/local/include/)
FIND_LIBRARY(HELLO_LIBRARY NAMES hello PATH /usr/local/lib)

IF(HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
    SET(HELLO_FOUND TRUE)
ENDIF(HELLO_INCLUDE_DIR AND HELLO_LIBRARY)

IF(HELLO_FOUND)
    IF(NOT HELLO_FIND_QUIETLY)
        MESSAGE(STATUS "Found Hello: ${HELLO_LIBRARY}")
    ENDIF(NOT HELLO_FIND_QUIETLY)
ELSE(HELLO_FOUND)
    IF(HELLO_FIND_REQUIRED)
        MESSAGE(FATAL_ERROR "Could not find hello library")
    ENDIF(HELLO_FIND_REQUIRED)
ENDIF(HELLO_FOUND)

```

运行顺序

```
cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..
```

```

root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/hello/build# cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
/blueLinux/code/slam-learning/ch1/code/task4/cmake
-- Found Hello: /usr/local/lib/libhello.so
/blueLinux/code/slam-learning/ch1/code/task4/cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /blueLinux/code/4ofL1/hello/build

```

make

```

root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/hello/build# make
[ 25%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[ 50%] Linking CXX shared library libhello.so
[ 50%] Built target hello
[ 75%] Building CXX object CMakeFiles/sayHello.dir/useHello.cpp.o
[100%] Linking CXX executable sayHello
[100%] Built target sayHello

```

make install

```

root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/hello/build# make install
Consolidate compiler generated dependencies of target hello
[ 50%] Built target hello
Consolidate compiler generated dependencies of target sayHello
[100%] Built target sayHello
Install the project...
-- Install configuration: "Release"
-- Installing: /usr/local/lib/libhello.so
-- Installing: /usr/local/include/hello.h

```

运行结果

```
./sayhello
```

```
root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/hello/build# ./sayHello  
Hello SLAM
```

Tips： /usr/local/lib默认没有在Linux的搜索路径中。需要把动态链接库的安装路径（例如 /usr/local/lib ）放到环境变量 LD_LIBRARY_PATH 里。

5.1

gflags, glog, gtest 的安装：

安装过程基本相同，具体过程如下

安装Gflags运行指令

```
git clone https://github.com/gflags/gflags.git  
cd gflags  
mkdir build && cd build  
cmake -DCMAKE_INSTALL_PREFIX=/usr/local -DBUILD_SHARED_LIBS=ON -DGFLAGS_NAMESPACE=gflags  
../  
make -j8  
sudo make install
```

安装Glog运行指令

```
git clone https://ghproxy.com/https://github.com/google/glog  
sudo apt-get install autoconf automake libtool  
cd glog  
mkdir build && cd build  
cmake ..  
make  
sudo make install
```

Tips： 要求cmake最低版本3.16，若低于此版本需要更新后安装

安装Gtest运行指令

```
git clone https://github.com/google/googletest  
cd googletest  
mkdir build && cd build
```

```
cmake ..  
make  
sudo make install
```

测试Gtest

测试代码：

```
#include<gtest/gtest.h>  
  
int add(int a,int b){  
    return a+b;  
}  
  
TEST(testCase,test0){  
    EXPECT_EQ(add(2,3),5);  
}  
  
int main(int argc,char **argv){  
    testing::InitGoogleTest(&argc,argv);  
    return RUN_ALL_TESTS();  
}
```

终端运行

```
g++ test.c -lgtest -lpthread -v -o test  
./test
```

输出结果：

```
root@DESKTOP-78JR5UA:/blueLinux/code/testgtest# ./test  
[=====] Running 1 test from 1 test suite.  
[-----] Global test environment set-up.  
[-----] 1 test from testCase  
[ RUN      ] testCase.test0  
[      OK  ] testCase.test0 (0 ms)  
[-----] 1 test from testCase (0 ms total)  
  
[-----] Global test environment tear-down  
[=====] 1 test from 1 test suite ran. (0 ms total)  
[ PASSED  ] 1 test.
```

5.2

用glog 的打印方式替代 std::cout 的输出

code

```
#include<iostream>
#include<logging.h>
int main( int argc, char** argv ) {
    google::InitGoogleLogging("sayHello");
    google::SetLogDestination(google::GLOG_INFO, "./log/");
    LOG(INFO) << "HELLO SLAM!";
}
```

运行结果

```
root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/usehello/build/bin/log# cat 20220904-155628.1077
Log file created at: 2022/09/04 15:56:28
Running on machine: DESKTOP-78JR5UA
Running duration (h:mm:ss): 0:00:00
Log line format: [IWEF]yyyymmdd hh:mm:ss.uuuuuu threadid file:line] msg
I20220904 15:56:28.678534 1077 useHello.cpp:8] HELLO SLAM!
```

5.3

在 useHello.c 中增加一个 gflags 以指明打印的次数 print_times，默认为 1。当用户传递该参数时，即打印多少遍 Hello SLAM。

code

```
#include <iostream>
#include <gflags.h>

DEFINE_int32(print_times, 1, "1");

int main( int argc, char** argv ) {
    gflags::ParseCommandLineFlags(&argc, &argv, true);
    for(int i = 0; i < FLAGS_print_times; i++)
```

```
std::cout << "HELLO SLAM" << std::endl;  
}
```

运行结果

```
root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/usehello/build/bin# ./hello -print_times=3  
HELLO SLAM  
HELLO SLAM  
HELLO SLAM
```

5.4

书写一个 gtest 单元测试程序来测试你的工程能够正常运行。修改你的 CMakeLists.txt 来增加这个单元测试。

testHello.cpp

```
#include <gtest/gtest.h>  
#include <glog/logging.h>  
#include "hello.h"  
  
TEST(sayHelloTest, sayHelloNoThrow)  
{  
    EXPECT_NO_THROW(sayHello());  
}  
  
int main(int argc, char *argv[])  
{  
    google::InitGoogleLogging(argv[0]);  
    google::SetStderrLogging(google::GLOG_INFO);  
    testing::InitGoogleTest(&argc, argv);  
    return RUN_ALL_TESTS();  
}
```

CMakeLists.txt修改此两行即可

```
ADD_EXECUTABLE(testHello testHello.cpp)  
TARGET_LINK_LIBRARIES(testHello hello glog gflags gtest gmock pthread)
```

运行结果

```
root@DESKTOP-78JR5UA:/blueLinux/code/4ofL1/hello/build# ./testHello
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from sayHelloTest
[ RUN      ] sayHelloTest.sayHelloNoThrow
Hello SLAM
[      OK   ] sayHelloTest.sayHelloNoThrow (0 ms)
[-----] 1 test from sayHelloTest (0 ms total)

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

6.1

下载 ORB-SLAM2 源代码截图：

```
终端: 本地 × 本地 (2) × + v
luis@DESKTOP-78JR5UA:/blueLinux/code/ORB-SLAM2$ sudo git clone git@github.com:raulmur/ORB_SLAM2.git
Cloning into 'ORB_SLAM2'...
remote: Enumerating objects: 566, done.
remote: Total 566 (delta 0), reused 0 (delta 0), pack-reused 566
Receiving objects: 100% (566/566), 41.41 MiB | 1.70 MiB/s, done.
Resolving deltas: 100% (182/182), done.
Checking out files: 100% (228/228), done.
luis@DESKTOP-78JR5UA:/blueLinux/code/ORB-SLAM2$
```

6.2

(a) ORB-SLAM2 将编译出什么结果？有几个库文件和可执行文件？

生成一个动态链接库ORB_SLAM2，它就是在/lib文件夹下的libORB_SLAM2.so文件；

生成的可执行文件：

rgbd_tum、stereo_kitti、stereo_euroc、mono_tum、mono_kitti、mono_euroc

(b) ORB-SLAM2 中的 include, src, Examples 三个文件夹中都含有什么内容？

include文件夹中主要是程序的头文件，src为SLAM部分的核心源代码，Examples包含一些根据不同相机的使用样例包括Monocular（单目）、RGB-D、ROS和Stereo。

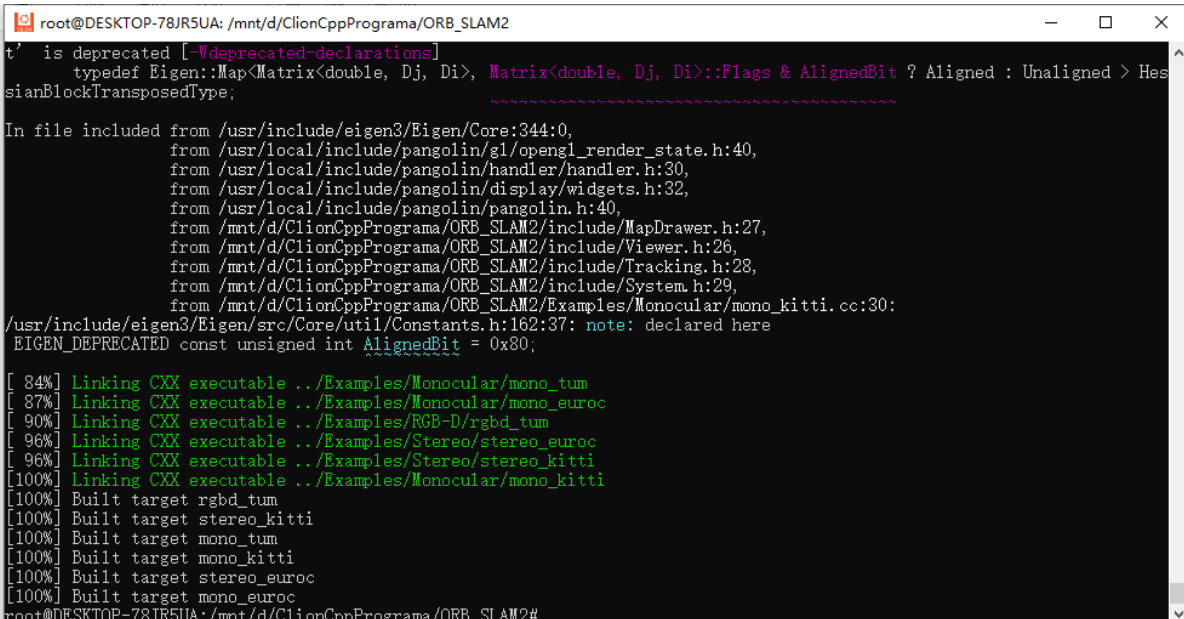
(c) ORB-SLAM2 中的可执行文件链接到了哪些库？它们的名字是什么？

链接到了他自身生成的libORB_SLAM2.so库，还有OpenCV_LIBS、EIGEN3_LIBS、Pangolin_LIBRARIES、libDBow2.so、libg2o.so库文件。

7.1

编译ORB-SLAM2的过程和截图：

根据usleep的错误解决方案和谷歌到的教程安装完成



```
root@DESKTOP-78JR5UA: /mnt/d/ClionCppPrograma/ORB_SLAM2
t' is deprecated [-Wdeprecated-declarations]
typedef Eigen::Map<Matrix<double, Dj, Di>, Matrix<double, Dj, Di>::Flags & AlignedBit ? Aligned : Unaligned > Hes
sianBlockTransposedType;
~~~~~
In file included from /usr/include/eigen3/Eigen/Core:344:0,
from /usr/local/include/pangolin/gl/opengl_render_state.h:40,
from /usr/local/include/pangolin/handler/handler.h:30,
from /usr/local/include/pangolin/display/widgets.h:32,
from /usr/local/include/pangolin/pangolin.h:40,
from /mnt/d/ClionCppPrograma/ORB_SLAM2/include/MapDrawer.h:27,
from /mnt/d/ClionCppPrograma/ORB_SLAM2/include/Viewer.h:26,
from /mnt/d/ClionCppPrograma/ORB_SLAM2/include/Tracking.h:28,
from /mnt/d/ClionCppPrograma/ORB_SLAM2/include/System.h:29,
from /mnt/d/ClionCppPrograma/ORB_SLAM2/Examples/Monocular/mono_kitti.cc:30:
/usr/include/eigen3/Eigen/src/Core/util/Constants.h:162:37: note: declared here
EIGEN_DEPRECATED const unsigned int AlignedBit = 0x80;

[ 84%] Linking CXX executable ../Examples/Monocular/mono_tum
[ 87%] Linking CXX executable ../Examples/Monocular/mono_euroc
[ 90%] Linking CXX executable ../Examples/RGB-D/rgbd_tum
[ 96%] Linking CXX executable ../Examples/Stereo/stereo_euroc
[ 96%] Linking CXX executable ../Examples/Stereo/stereo_kitti
[100%] Linking CXX executable ../Examples/Monocular/mono_kitti
[100%] Built target rgbd_tum
[100%] Built target stereo_kitti
[100%] Built target mono_tum
[100%] Built target mono_kitti
[100%] Built target stereo_euroc
[100%] Built target mono_euroc
root@DESKTOP-78JR5UA: /mnt/d/ClionCppPrograma/ORB_SLAM2#
```

7.2

如何将 myslam.cpp或 myvideo.cpp 加入到 ORB-SLAM2 工程中?请给出你的 CMakeLists.txt 修改方案。

进入ORB-SLAM2项目，在Examples文件夹中创建MyVideo文件夹，将提供的代码myvideo.cpp、myvideo.mp4、myslam.cpp、myvideo.yaml放置到MyVideo文件夹下。

```
root@DESKTOP-78JR5UA:/mnt/d/ClionCppPrograma# tree ORB_SLAM2/Examples/MyVideo/  
ORB_SLAM2/Examples/MyVideo/  
├── myslam.cpp  
├── myvideo  
├── myvideo.cpp  
├── myvideo.mp4  
└── myvideo.yaml
```

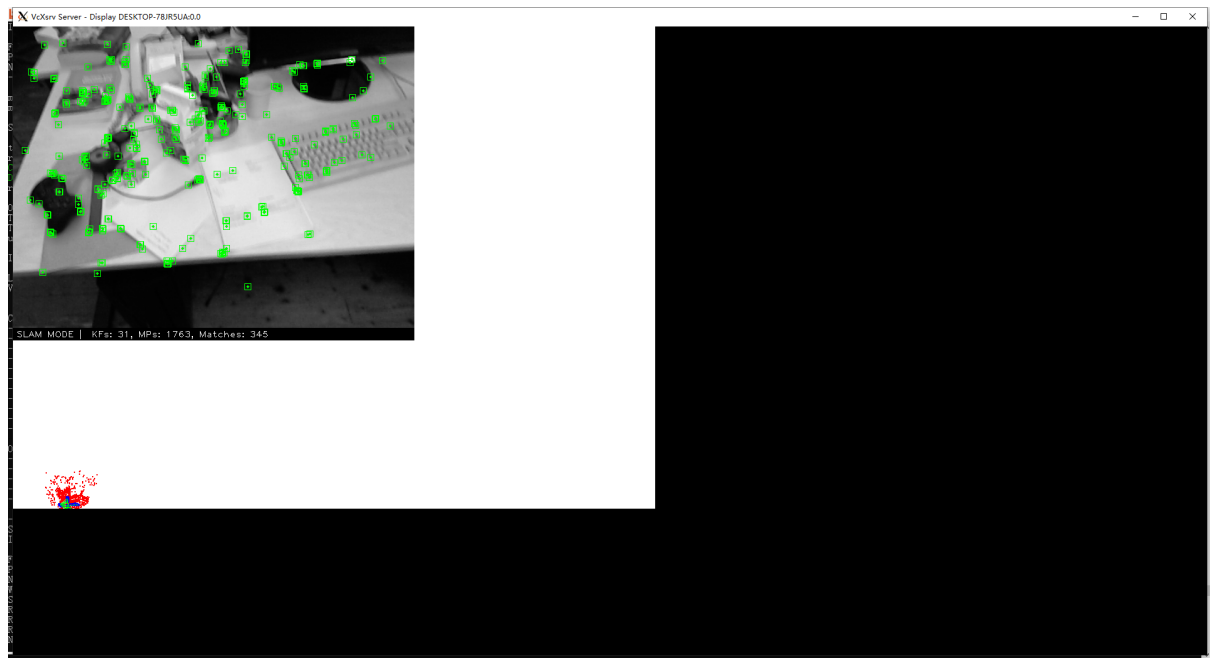
CMakeLists.txt文件，在原文件末尾添加命令：

```
SET(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/Examples/MyVideo)  
  
ADD_EXECUTABLE(myvideo Examples/MyVideo/myvideo.cpp)  
  
TARGET_LINK_LIBRARIES(myvideo ${PROJECT_NAME})
```

7.3

现在，用这个文件让 ORB-SLAM2 运行起来，看看 ORB-SLAM2 的实际效果，请给出运行截图，并谈谈你在运行过程中的体会。

由于我在WSL2上运行的ORB-SLAM2项目，一开始遇到了图形界面的问题，根据谷歌逐一解决后，还是无法正常运行。遂下载了一个数据集试运行，发现可以运行并打印



```

root@DESKTOP-78JRSUA:/mnt/d/CtfonAppPrograma/ORB_SLAM2# ./Examples/Monocular/mono_tum Vocabulary/ORBvoc.txt Examples/Monocular/TUM1.yaml data/rgbd_dataset_freiburg1_xyz

ORB-SLAM2 Copyright (C) 2014-2016 Raul Mur-Artal, University of Zaragoza.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See LICENSE.txt.

Input sensor was set to: Monocular

Loading ORB Vocabulary. This could take a while...
Vocabulary loaded!

Camera Parameters:
- fx: 517.306
- fy: 516.469
- cx: 318.643
- cy: 255.314
- k1: 0.262383
- k2: -0.953104
- k3: 1.16331
- p1: -0.005358
- p2: 0.002628
- fps: 30
- color order: RGB (ignored if grayscale)

ORB Extractor Parameters:
- Number of Features: 1000
- Scale Levels: 8
- Scale Factor: 1.2
- Initial Fast Threshold: 20
- Minimum Fast Threshold: 7

-----
Start processing sequence ...
Images in the sequence: 798

Framebuffer with requested attributes not available. Using available framebuffer. You may see visual artifacts.X11 Error: GLXBadFBConfig
Pangolin X11: Indirect GLX rendering context obtained
New Map created with 89 points
Wrong initialization, resetting...
System Resetting
Resetting Local Mapper... done
Resetting Loop Closing... done
Resetting Database... done
New Map created with 104 points
-----

median tracking time: 0.029159
mean tracking time: 0.0316161

Saving keyframe trajectory to KeyFrameTrajectory.txt ...
trajectory saved!

```

但运行自己的数据集会在黄线处直接退出，于是重新回来查看文件结构，发现我的视频文件未放置在根目录下，导致项目运行中断，最后准备好全部文件运行成功。

