基于VSCode和CMake实现C/C++开发<Linux篇>

本次课程适合的对象：

- Linux零基础，想了解最常用的**高频Linux命令**的小伙伴
- 只会在Windows开发C/C++，需要转**Linux**开发的小伙伴
- 想深入了解并掌握**GCC**编译器编译语法和规则的小伙伴
- 想深入了解并掌握**GDB**调试器命令行调试的小伙伴
- 想学习使用**CMake**构建**C/C++**工程的小伙伴
- 想学习Linux下使用**VSCode**进行**C/C++**开发的小伙伴

总而言之，本课程将**从零开始，教会你如何在Linux开发C/C++**，带领你一起打开新世界的大门~

> 系统环境：**Ubuntu18.04 LTS** 虚拟机
>
> 开发语言：**C++**
>
> 开发IDE ： **VSCode**

课程宗旨：*无干货，不视频。分享让生活更美好。*

基于VSCode和CMake实现C/C++开发<Linux篇>
- 第一讲：
  - 1.1 目录结构
  - 1.2 指令与选项
  - 1.3 重要指令讲解 +【实战】命令行演练
  - 1.4 文件编辑
- 第二讲：开发环境搭建
  - 2.1 编译器，调试器安装
  - 2.2 CMake安装
- 第三讲：GCC编译器
  - 3.1 编译过程
  - 3.2 g++重要编译参数
  - 3.3 【实战】g++命令行编译
    - 3.3.1 直接编译
    - 3.3.2 生成库文件并编译
    - 3.3.3 运行可执行文件
- 第四讲：GDB调试器
  - 4.1 常用调试命令参数
  - 4.2 【实战】命令行调试
- 第五讲：IDE - VSCode
  - 5.1 界面介绍
  - 5.2 插件安装
  - 5.3 快捷键
  - 5.4【实战】2个小项目
    - 5.4.1 高频使用技巧

5.4.2 代码编写
5.4.3 编译并运行
第六讲：CMake
6.1 Cross-platform development
6.2 语法特性介绍
6.3 重要指令和CMake常用变量
6.3.1 重要指令
6.3.2 CMake常用变量
6.4 CMake编译工程
6.4.1 编译流程
6.4.2 两种构建方式
6.5 【实战】CMake代码实践
6.5.1 最小CMake工程
6.5.2 多目录工程 - 直接编译
6.5.3 多目录工程 - 生成库编译
第七讲：【实战】使用VSCode进行完整项目开发
案例：士兵突击
需求:
开发:
7.1 合理设置项目目录
7.2 编写项目源文件
7.3 编写CMakeLists.txt构建项目编译规则
7.4 使用外部构建，手动编译CMake项目
7.5 配置VSCode的json文件并调试项目

第一讲：

什么是linux系统?

- Linux是**开源**的操作系统

多用户多任务：

- 单用户：一个用户，在登录计算机（操作系统），只能允许**同时**登录一个用户；
- 单任务：一个任务，允许用户**同时**进行的操作任务数量；
- 多用户：多个用户，在登录计算机（操作系统），允许**同时**登录多个用户进行操作；
- 多任务：多个任务，允许用户**同时**进行多个操作任务；

Windows属于：单用户、多任务。

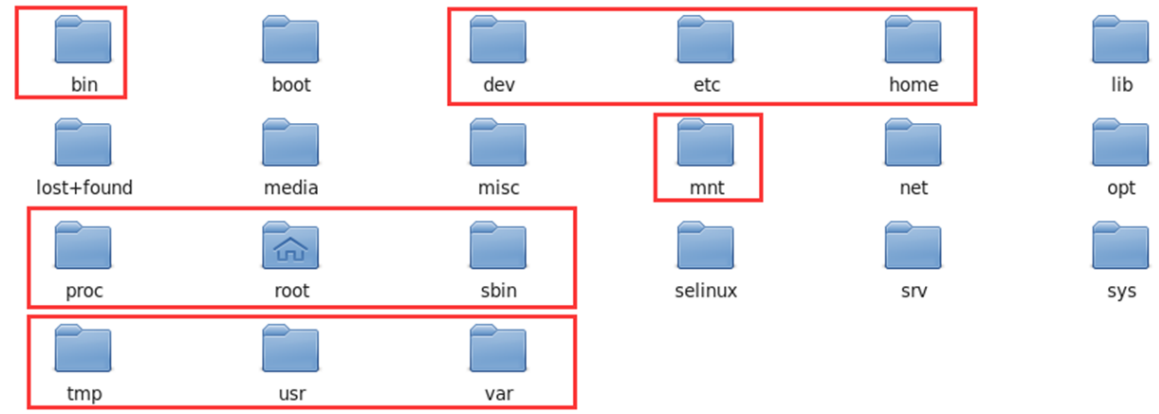
Linux属于：多用户、多任务。

Linux一切皆文件

对于文件的操作的都有哪些种类?

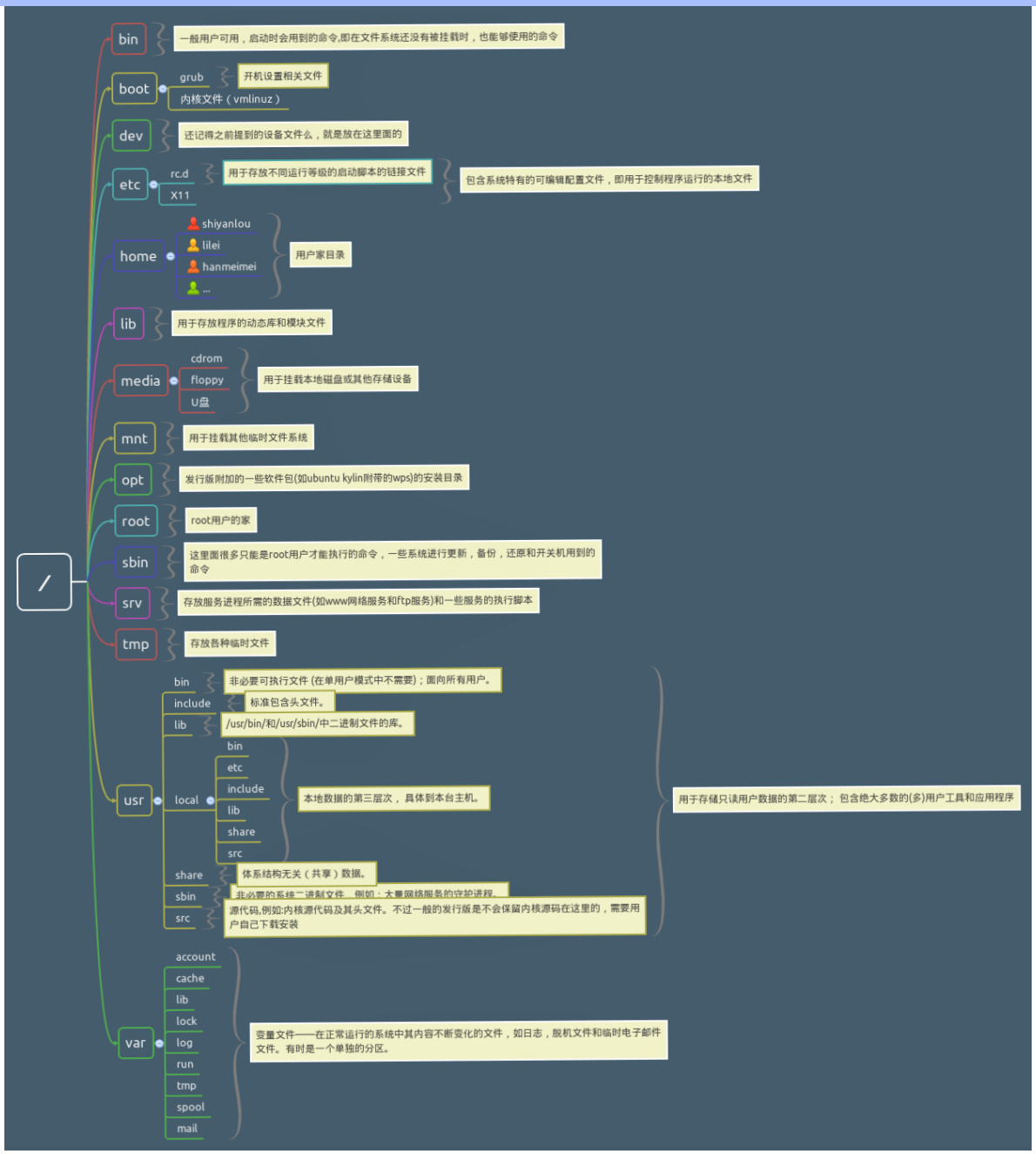
- **创建文件、编辑文件、保存文件、关闭文件、重命名文件、删除文件、恢复文件。**

1.1 目录结构



目录结构：

- **Bin：**全称binary，含义是二进制。该目录中存储的都是一些二进制文件，文件都是可以被运行的。
- **Dev：该目录中主要存放的是外接设备，例如盘、其他的光盘等。在其中的外接设备是不能直接被使用的，需要挂载（类似window下的分配盘符）。**
- **Etc：**该目录主要存储一些配置文件。
- **Home：**表示"家"，表示**除了root用户以外**其他用户的家目录，类似于windows下的User/用户目录。
- **Proc：**全称process，表示进程，该目录中存储的是Linux运行时候的进程。
- **Root：**该目录是root用户自己的家目录。
- **Sbin：**全称super binary，该目录也是存储一些可以被执行的二进制文件，但是必须得有super权限的用户才能执行。
- **Tmp：**表示"临时"的，当系统运行时候产生的临时文件会在这个目录存着。
- **Usr：**存放的是用户自己安装的软件。类似于windows下的program files。
- **Var：**存放的程序/系统的日志文件的目录。
- **Mnt：**当外接设备需要挂载的时候，就需要挂载到mnt目录下。



1.2 指令与选项

- **指令含义：**
- **Linux的指令**是指在Linux**终端（命令行）**中输入的内容。
- **指令格式：**
  - 完整指令的**标准格式**：**命令**（空格）[**选项**]（空格）[**操作对象**]
  - 选项和操作对象都可以没有，也可以是多个

```
# 指令示例：以下两条指令等价
ls -l -a -h /home ./
ls -lah /home ./
```

1.3 重要指令讲解 +【实战】命令行演练

- **pwd - Print current working directory**
  - **作用：**打印当前终端所在的目录
  - **用法：** **pwd**

```
# 打印当前目录
pwd
```

- **ls - List directory contents**
  - **作用**：列出当前工作目录下的所有文件/文件夹的名称
  - **用法1**：**ls**

    含义：列出当前工作目录下的文件/文件夹的名称

```
ls
```

  - **用法2**：**ls [路径]**

    含义：列出指定**路径**下的所有文件/文件夹的名称

    - 绝对路径：相对**根目录**的路径；
    - 相对路径：相对**当前目录**的路径；

```
# ls 相对路径
ls ./   #【表示当前目录下】
ls ../  #【上一级目录下】
# ls 绝对路径
ls /home
```

  - **用法3**：**ls [选项] [路径]**

    含义：在列出指定路径下的文件/文件夹的名称，并以指定的格式进行显示。

```
# ls 选项 路径
ls -lah /home
# 选项解释：
    -l：表示list，表示以详细列表的形式进行展示
    -a：表示显示所有的文件/文件夹（包含了隐藏文件/文件夹）
    -h：表示以可读性较高的形式显示
# ls -l 中 “-”表示改行对应的文档类型为文件，“d”表示文档类型为文件夹。
# 在Linux中隐藏文档一般都是以“.”开头
```

- **cd - change directory**
  - **作用**：切换当前的工作目录
  - **用法1**：**cd ； cd ~**

```
# 以下两条命令等价，示直接进入当前用户的家目录下【很常用】
cd
cd ~
```

  - **用法2**：**cd [相对路径]**

```
# 进入到上级目录下
cd ..
# 进入到上级目录中的local目录下
cd ../local
```

  - **用法3**：**cd [绝对路径]**

```
# 进入到/usr/local目录下
cd /usr/local
```

- **mkdir - make directories**
  - **作用：**创建目录
  - **用法1：mkdir 路径**

```
# 在当前路径下创建出目录“myfolder”
mkdir myfolder
```

  - **用法2：mkdir -p 路径**

    含义：一次性创建多层不存在的目录

```
# 创建 ~/a/b/c 目录
mkdir -p ~/a/b/c
```

  - **用法3：mkdir 路径1 [路径2] [路径3]**

    含义：一次性创建多个目录

```
# 在当前目录分别创建a、b、c三个文件夹
mkdir a b c
```

- **touch - change file timestamps**
  - **作用：**创建新文件

```
--------------------------------------------------------------------------------
------------------------------
#                                          【为什么创建新文件是touch】
# 1. touch的作用本来不是创建文件，而是将指定文件的修改时间设置为当前时间。就是假装“碰”（touch）了一下这个文件，假装文件被“修改”了，于是文件的修改时间就是被设置为当前时间。
# 2. 这带来了一个副作用，就是当touch一个不存在的文件的时候，它会创建这个文件。然后，由于touch已经可以完成创建文件的功能了，就不再需要一个单独的create了。
--------------------------------------------------------------------------------
------------------------------
```

  - **用法1：touch [路径]**

```
# 在当前目录下创建linux.txt文件
touch linux.txt

# 在上级目录下创建linux文件
touch ../linux

# 在/home/bing/目录下创建myfile文件
touch /home/bing/myfile
```

  - **用法2：touch [路径1] [路径2]**

```
# 在当前目录下创建file file.txt两个文件
touch file file.txt
```

- **rm - remove files or directories**
  - **作用：**删除文件/目录
  - **用法1：rm [选项] 需要移除的文件路径**

```
# 删除当前路径下的myfile文件
rm myfile
# 删除/usr路径下的myfile文件
rm /usr/myfile
```

  - **用法2：rm [选项] 需要移除的目录**

```
# 删除当前路径下的abc文件
rm -rf myfolder
# 删除/usr路径下的abc文件
rm -rf /usr/myfolder
```

- **cp - copy files and directories**
  - **作用：**复制文件/文件夹到指定的位置
  - **用法1：cp [被复制的文件路径] [文件被复制到的路径]**

```
# cp命令来复制一个文件
cp /home/bing/myfile ./
```

  - **用法2：cp -r [被复制的文件夹路径] [文件夹被复制到的路径]****

    含义：-r 表示递归复制，复制文件夹的时候需要加 -r

```
# 复制/home/bing/myfolder文件夹到根目录/下
cp -r /home/bing/myfolder /
```

  - **用法3：mkdir [路径1] [路径2] [路径3]**

    含义：一次性创建多个目录
- **mv - move (rename) files**
  - **作用：**移动文件到新的位置，或者重命名文件
  - **用法：mv [需要移动的文件路径] [需要保存的位置路径]**

```
# 移动当前目录下myfile文件到根目录/下
mv myfile /myfile

# 移动当前目录下myfolder文件夹到根目录/下
mv myfolder /myfolder

# 移动当前目录下myfile到根目录/下，并重命名为myfile007
mv myfile myfile007
```

- **man - an interface to the system reference manuals**
  - **作用：**包含了Linux中全部命令手册
  - **用法：man [命令]**
  - **含义：**查看命令使用手册，按 q 退出

```
# 查看ls命令的手册
man ls
# 查看cd命令的手册
man cd
# 查看man命令的手册
man man
```

- **reboot - reboot the machine**
  - **作用：**重启linux系统
  - **用法：reboot**

```
# 立即重启
reboot
```

- **shutdown - power-off the machine**
  - **作用：**关机
  - **用法：shut -h [时间]**

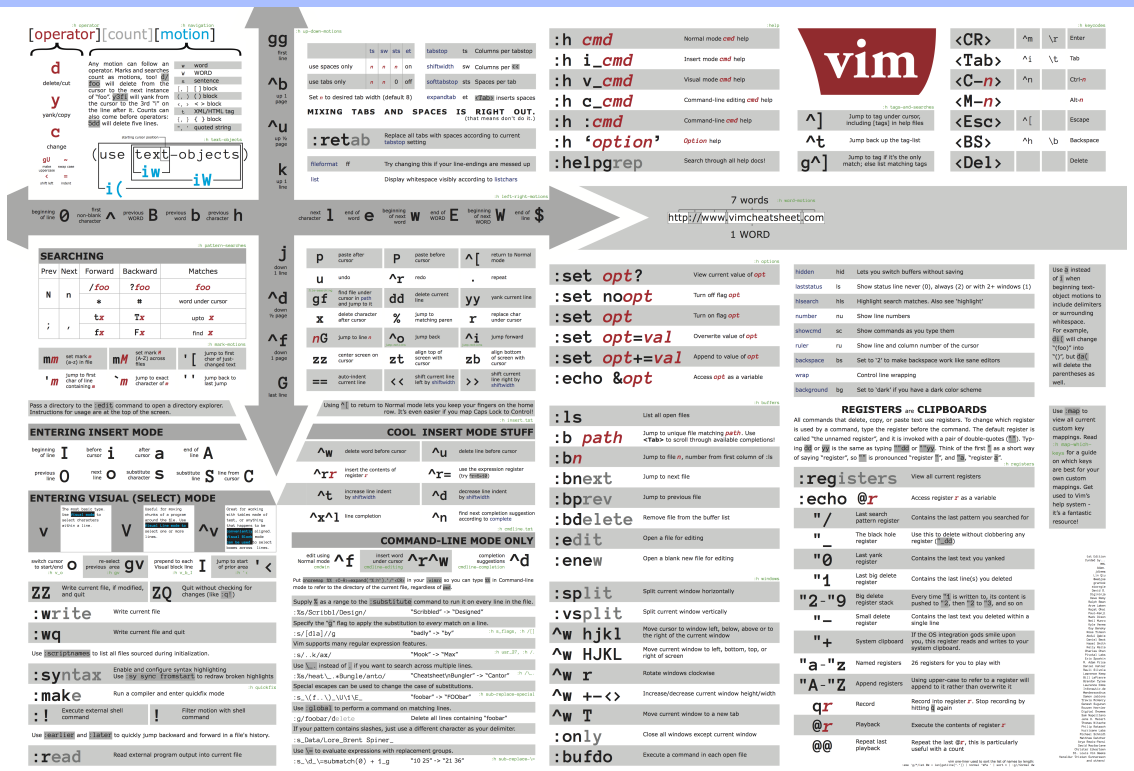
```
# 立即关机
shutdown -h now
```

1.4 文件编辑

- **Vim [file]**
  - 所有的 Linux系统都会内建 Vi/Vim编辑器，其他的编辑器则不一定会存在
  - Vim是所有Unix及Linux系统下标准的编辑器
  - **Vim 具有程序开发的能力，也可以用来对文件进行简单的编辑**

Vim具有“编辑器之神”的称号，学会Vim便可在Linux的世界里**畅行无阻**，**尤其是在终端中。**

👇Vim操作终极图片👇



- **gedit [file]**
  - Linux 下的一个纯文本编辑器
  - 可以根据不同的语言高亮显现关键字和标识符。
- **nano [file]**
  - nano 是一个小巧的文本编辑器
  - 它比vi/vim要简单得多，比较适合Linux初学者使用。
  - 某些Linux发行版的默认编辑器就是nano。

第二讲：开发环境搭建

2.1 编译器，调试器安装

- **安装GCC，GDB**

```
sudo apt update
# 通过以下命令安装编译器和调试器
sudo apt install build-essential gdb
```

- **安装成功确认**

```
# 以下命令确认每个软件是否安装成功
# 如果成功，则显示版本号
gcc --version
g++ --version
gdb --version
```

2.2 CMake安装

- **安装cmake**

```
# 通过以下命令安装编译器和调试器
sudo apt install cmake
```

- **安装成功确认**

```
# 确认是否安装成功
# 如果成功，则显示版本号
cmake --version
```

第三讲：GCC编译器

前言:

1. GCC 编译器支持编译 Go、Objective-C，Objective-C ++，Fortran，Ada，D 和 BRIG（HSAIL）等程序;
2. Linux 开发C/C++ 一定要熟悉 GCC
3. **VSCode是通过调用GCC编译器来实现C/C++的编译工作的;**

实际使用中:

- 使用 gcc 指令编译 C 代码
- 使用 g++指令编译 C++ 代码

3.1 编译过程

1. **预处理-Pre-Processing　　//.i文件**

```
# -E 选项指示编译器仅对输入文件进行预处理
g++  -E  test.cpp  -o  test.i    //.i文件
```

2. **编译-Compiling　　// .s文件**

```
# -S 编译选项告诉 g++ 在为 C++ 代码产生了汇编语言文件后停止编译
#  g++ 产生的汇编语言文件的缺省扩展名是 .s
g++  -S  test.i  -o  test.s
```

3. **汇编-Assembling　　// .o文件**

```
# -c 选项告诉 g++ 仅把源代码编译为机器语言的目标代码
# 缺省时 g++ 建立的目标代码文件有一个 .o 的扩展名。
g++  -c  test.s  -o  test.o
```

4. **链接-Linking　　// bin文件**

```
# -o 编译选项来为将产生的可执行文件用指定的文件名
g++  test.o  -o  test
```

3.2 g++重要编译参数

1. **-g**　编译带调试信息的可执行文件

```
# -g 选项告诉 GCC 产生能被 GNU 调试器GDB使用的调试信息，以调试程序。

# 产生带调试信息的可执行文件test
g++ -g test.cpp
```

2. **-O[n]** 优化源代码

```
## 所谓优化，例如省略掉代码中从未使用过的变量、直接将常量表达式用结果值代替等等，这些操作会缩减目标文件所包含的代码量，提高最终生成的可执行文件的运行效率。

# -O 选项告诉 g++ 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行的更快。 -O2 选项告诉 g++ 产生尽可能小和尽可能快的代码。 如-O2，-O3，-On（n 常为0-3）
# -O 同时减小代码的长度和执行时间，其效果等价于-O1
# -O0 表示不做优化
# -O1 为默认优化
# -O2 除了完成-O1的优化之外，还进行一些额外的调整工作，如指令调整等。
# -O3 则包括循环展开和其他一些与处理特性相关的优化工作。
# 选项将使编译的速度比使用 -O 时慢， 但通常产生的代码执行速度会更快。

# 使用 -O2优化源代码，并输出可执行文件
g++ -O2 test.cpp
```

3. **-l 和 -L** 指定库文件 | 指定库文件路径

```
# -l参数(小写)就是用来指定程序要链接的库，-l参数紧接着就是库名
# 在/lib和/usr/lib和/usr/local/lib里的库直接用-l参数就能链接

# 链接glog库
g++ -lglog test.cpp

# 如果库文件没放在上面三个目录里，需要使用-L参数(大写)指定库文件所在目录
# -L参数跟着的是库文件所在的目录名

# 链接mytest库，libmytest.so在/home/bing/mytestlibfolder目录下
g++ -L/home/bing/mytestlibfolder -lmytest test.cpp
```

4. **-I** 指定头文件搜索目录

```
# -I
# /usr/include目录一般是不用指定的，gcc知道去那里找，但 是如果头文件不在/usr/icnclude里我们就要用-I参数指定了，比如头文件放在/myinclude目录里，那编译命令行就要加上-I/myinclude 参数了，如果不加你会得到一个”xxxx.h: No such file or directory”的错误。-I参数可以用相对路径，比如头文件在当前 目录，可以用-I.来指定。上面我们提到的-cflags参数就是用来生成-I参数的。

g++ -I/myinclude test.cpp
```

5. **-Wall** 打印警告信息

```
# 打印出gcc提供的警告信息
g++ -Wall test.cpp
```

6. **-w** 关闭警告信息

```
# 关闭所有警告信息
g++ -w test.cpp
```

7. **-std=c++11** 设置编译标准

```
# 使用 c++11 标准编译 test.cpp
g++ -std=c++11 test.cpp
```

8. **-o** 指定输出文件名

```
# 指定即将产生的文件名

# 指定输出可执行文件名为test
g++ test.cpp -o test
```

9. **-D** 定义宏

```
# 在使用gcc/g++编译的时候定义宏

# 常用场景：
# -DDEBUG 定义DEBUG宏，可能文件中有DEBUG宏部分的相关信息，用个DDEBUG来选择开启或关闭DEBUG
```

示例代码：

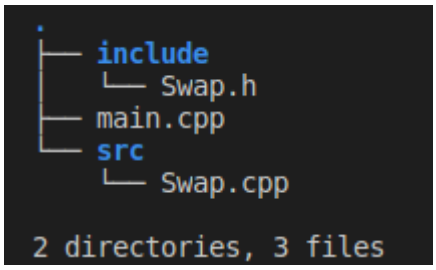
```
// -Dname 定义宏name,默认定义内容为字符串“1”

#include <stdio.h>

int main()
{
    #ifdef DEBUG
        printf("DEBUG LOG\n");
    #endif
        printf("in\n");
}

// 1. 在编译的时候，使用gcc -DDEBUG main.cpp
// 2. 第七行代码可以被执行
```

注：使用 `man gcc` 命令可以查看gcc英文使用手册，见下图

```
GCC(1)                                                                  GNU
                                                                    GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-Wpedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] [@file] infile...

       Only the most useful options are listed here; see below for the remainder.  g++ accepts
mostly the same options as gcc.
```

3.3 【实战】g++命令行编译

案例：最初目录结构: 2 directories, 3 files




```
# 最初目录结构
.
├── include
│   └── Swap.h
├── main.cpp
└── src
    └── Swap.cpp

2 directories, 3 files
```

3.3.1 直接编译

最简单的编译，并运行

```
# 将 main.cpp src/Swap.cpp 编译为可执行文件
g++ main.cpp src/Swap.cpp -Iinclude
# 运行a.out
./a.out
```

增加参数编译，并运行

```
# 将 main.cpp src/Swap.cpp 编译为可执行文件 附带一堆参数
g++ main.cpp src/Swap.cpp -Iinclude -std=c++11 -O2 -Wall -o b.out
# 运行 b.out
./b.out
```


3.3.2 生成库文件并编译

链接**静态库**生成可执行文件①：

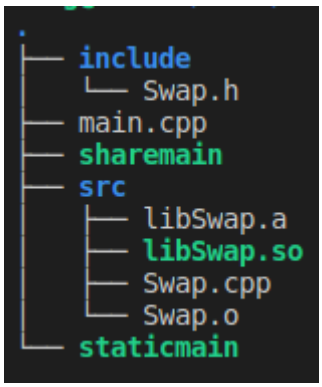
```
## 进入src目录下
$cd src

# 汇编，生成Swap.o文件
g++ Swap.cpp -c -I../include
# 生成静态库libSwap.a
ar rs libSwap.a Swap.o

## 回到上级目录
$cd ..

# 链接，生成可执行文件:staticmain
g++ main.cpp -Iinclude -Lsrc -lSwap -o staticmain
```

链接**动态库**生成可执行文件②：

```
## 进入src目录下
$cd src

# 生成动态库libSwap.so
g++ Swap.cpp -I../include -fPIC -shared -o libSwap.so
## 上面命令等价于以下两条命令
# gcc Swap.cpp -I../include -c -fPIC
# gcc -shared -o libSwap.so Swap.o

## 回到上级目录
$cd ..

# 链接，生成可执行文件:sharemain
g++ main.cpp -Iinclude -Lsrc -lSwap -o sharemain
```

编译完成后的目录结构

最终目录结构：2 directories, 8 files




```
# 最终目录结构
.
├── include
|   └── Swap.h
```

```
├── main.cpp
├── sharemain
├── src
│   ├── libSwap.a
│   ├── libSwap.so
│   ├── Swap.cpp
│   └── Swap.o
└── staticmain

2 directories, 8 files
```

3.3.3 运行可执行文件

运行可执行文件①

```
# 运行可执行文件
./staticmain
```

运行可执行文件②

```
# 运行可执行文件
LD_LIBRARY_PATH=src ./sharemain
```

第四讲：GDB调试器

前言：

- **GDB(GNU Debugger)**是一个用来**调试C/C++程序**的功能强大的**调试器**，是Linux系统开发C/C++最常用的调试器
- 程序员可以**使用GDB来跟踪程序中的错误**，从而减少程序员的工作量。
- Linux 开发C/C++ 一定要熟悉 GDB
- **VSCode是通过调用GDB调试器来实现C/C++的调试工作的；**

> Windows 系统中，常见的集成开发环境（IDE），如 VS、VC等，它们内部已经嵌套了相应的调试器

GDB主要功能：

- 设置**断点**(断点可以是条件表达式)
- 使程序在指定的代码行上暂停执行，便于观察
- **单步**执行程序，便于调试
- 查看程序中变量值的变化
- 动态改变程序的执行环境
- 分析崩溃程序产生的core文件

4.1 常用调试命令参数

调试开始：执行**gdb [exefilename]**，进入gdb调试程序，其中exefilename为要调试的可执行文件名


```
## 以下命令后括号内为命令的简化使用，比如run（r），直接输入命令 r 就代表命令run
```

```
$(gdb)help(h)    # 查看命令帮助，具体命令查询在gdb中输入help + 命令

$(gdb)run(r)     # 重新开始运行文件（run-text：加载文本文件，run-bin：加载二进制文件）

$(gdb)start      # 单步执行，运行程序，停在第一行执行语句

$(gdb)list(l)    # 查看原代码（list-n,从第n行开始查看代码。list+ 函数名：查看具体函数）

$(gdb)set        # 设置变量的值

$(gdb)next(n)    # 单步调试（逐过程，函数直接执行）

$(gdb)step(s)    # 单步调试（逐语句：跳入自定义函数内部执行）

$(gdb)backtrace(bt) # 查看函数的调用的栈帧和层级关系

$(gdb)frame(f)   # 切换函数的栈帧

$(gdb)info(i)    # 查看函数内部局部变量的数值

$(gdb)finish     # 结束当前函数，返回到函数调用点

$(gdb)continue(c)    # 继续运行

$(gdb)print(p)   # 打印值及地址

$(gdb)quit(q)    # 退出gdb
```

```
$(gdb)break+num(b)           # 在第num行设置断点

$(gdb)info breakpoints       # 查看当前设置的所有断点

$(gdb)delete breakpoints num(d) # 删除第num个断点

$(gdb)display                # 追踪查看具体变量值

$(gdb)undisplay              # 取消追踪观察变量

$(gdb)watch                  # 被设置观察点的变量发生修改时，打印显示

$(gdb)i watch                # 显示观察点

$(gdb)enable breakpoints     # 启用断点

$(gdb)disable breakpoints    # 禁用断点

$(gdb)x                      # 查看内存x/20xw 显示20个单元，16进制，4字节每单元

$(gdb)run argv[1] argv[2]    # 调试时命令行传参

$(gdb)set follow-fork-mode child#Makefile项目管理：选择跟踪父子进程（fork()）
```

> **Tips:**
>
> 1. 编译程序时需要加上-g，之后才能用gdb进行调试：gcc -g main.c -o main
> 2. 回车键：重复上一命令

4.2 【实战】命令行调试

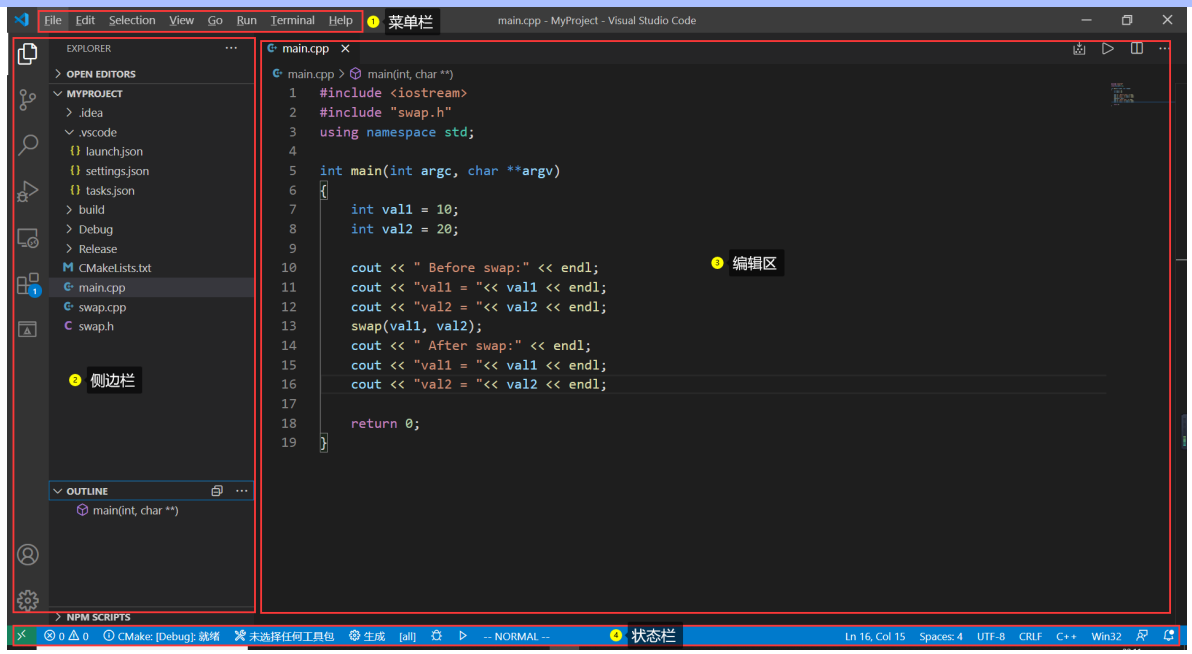
给出一段简单代码，准备调试。

```
#include <iostream>
using namespace std;

int main(int argc,char **argv)
{
    int N = 100;
    int sum = 0;
    int i = 1;

// calculate sum from 1 to 100
    while (i <= N)
    {
        sum = sum + i;
        i = i + 1;
    }

    cout << "sum = " << sum << endl;
    cout << "The program is over."   << endl;

    return 0;
}
```

第五讲：IDE - VSCode

5.1 界面介绍

1. 侧边栏
2. 菜单栏
3. 编辑区
4. 状态栏

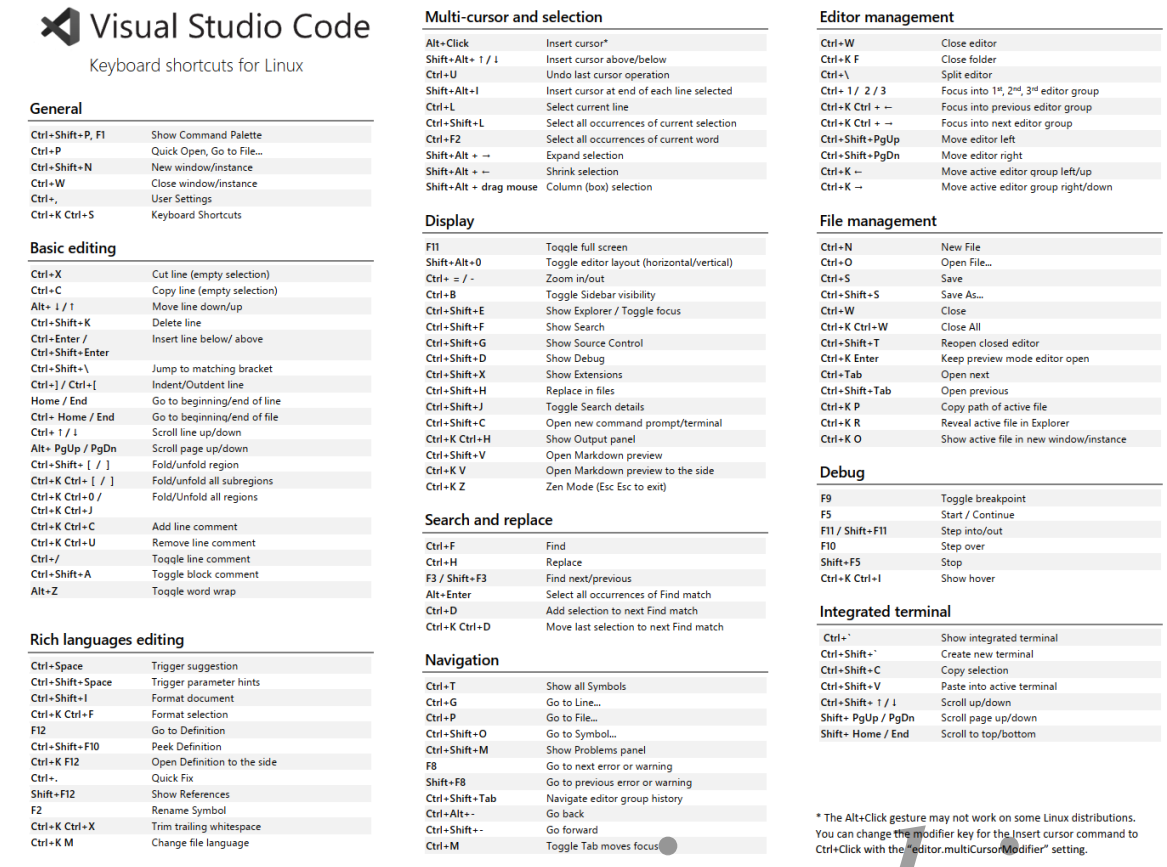


5.2 插件安装

以下三款插件是在Linux下开发C/C++的三款必备插件~

- C/C++
- CMake
- CMake Tools

5.3 快捷键



Visual Studio Code

Keyboard shortcuts for Linux

General

Ctrl+Shift+P, F1	Show Command Palette
Ctrl+P	Quick Open, Go to File...
Ctrl+Shift+N	New window/instance
Ctrl+W	Close window/instance
Ctrl+,	User Settings
Ctrl+K Ctrl+S	Keyboard Shortcuts

Basic editing

Ctrl+X	Cut line (empty selection)
Ctrl+C	Copy line (empty selection)
Alt+ ↓ / ↑	Move line down/up
Ctrl+Shift+K	Delete line
Ctrl+Enter / Ctrl+Shift+Enter	Insert line below/ above
Ctrl+Shift+\	Jump to matching bracket
Ctrl+] / Ctrl+[	Indent/Outdent line
Home / End	Go to beginning/end of line
Ctrl+ Home / End	Go to beginning/end of file
Ctrl+ ↑ / ↓	Scroll line up/down
Alt+ PgUp / PgDn	Scroll page up/down
Ctrl+Shift+ [/]	Fold/unfold region
Ctrl+K Ctrl+ [/]	Fold/unfold all subregions
Ctrl+K Ctrl+0 / Ctrl+K Ctrl+J	Fold/Unfold all regions
Ctrl+K Ctrl+C	Add line comment
Ctrl+K Ctrl+U	Remove line comment
Ctrl+/	Toggle line comment
Ctrl+Shift+A	Toggle block comment
Alt+Z	Toggle word wrap

Rich languages editing

Ctrl+Space	Trigger suggestion
Ctrl+Shift+Space	Trigger parameter hints
Ctrl+Shift+I	Format document
Ctrl+K Ctrl+F	Format selection
F12	Go to Definition
Ctrl+Shift+F10	Peek Definition
Ctrl+K F12	Open Definition to the side
Ctrl+.	Quick Fix
Shift+F12	Show References
F2	Rename Symbol
Ctrl+K Ctrl+X	Trim trailing whitespace
Ctrl+K M	Change file language

Multi-cursor and selection

Alt+Click	Insert cursor*
Shift+Alt+ ↑ / ↓	Insert cursor above/below
Ctrl+U	Undo last cursor operation
Shift+Alt+I	Insert cursor at end of each line selected
Ctrl+L	Select current line
Ctrl+Shift+L	Select all occurrences of current selection
Ctrl+F2	Select all occurrences of current word
Shift+Alt + →	Expand selection
Shift+Alt + ←	Shrink selection
Shift+Alt + drag mouse	Column (box) selection

Display

F11	Toggle full screen
Shift+Alt+0	Toggle editor layout (horizontal/vertical)
Ctrl+ = / -	Zoom in/out
Ctrl+B	Toggle Sidebar visibility
Ctrl+Shift+E	Show Explorer / Toggle focus
Ctrl+Shift+F	Show Search
Ctrl+Shift+G	Show Source Control
Ctrl+Shift+D	Show Debug
Ctrl+Shift+X	Show Extensions
Ctrl+Shift+H	Replace in files
Ctrl+Shift+J	Toggle Search details
Ctrl+Shift+C	Open new command prompt/terminal
Ctrl+K Ctrl+H	Show Output panel
Ctrl+Shift+V	Open Markdown preview
Ctrl+K V	Open Markdown preview to the side
Ctrl+K Z	Zen Mode (Esc Esc to exit)

Search and replace

Ctrl+F	Find
Ctrl+H	Replace
F3 / Shift+F3	Find next/previous
Alt+Enter	Select all occurrences of Find match
Ctrl+D	Add selection to next Find match
Ctrl+K Ctrl+D	Move last selection to next Find match

Navigation

Ctrl+T	Show all Symbols
Ctrl+G	Go to Line...
Ctrl+P	Go to File...
Ctrl+Shift+O	Go to Symbol...
Ctrl+Shift+M	Show Problems panel
F8	Go to next error or warning
Shift+F8	Go to previous error or warning
Ctrl+Shift+Tab	Navigate editor group history
Ctrl+Alt+-	Go back
Ctrl+Shift+-	Go forward
Ctrl+M	Toggle Tab moves focus

Editor management

Ctrl+W	Close editor
Ctrl+K F	Close folder
Ctrl+\	Split editor
Ctrl+ 1 / 2 / 3	Focus into 1st, 2nd, 3rd editor group
Ctrl+K Ctrl + ←	Focus into previous editor group
Ctrl+K Ctrl + →	Focus into next editor group
Ctrl+Shift+PgUp	Move editor left
Ctrl+Shift+PgDn	Move editor right
Ctrl+K ←	Move active editor group left/up
Ctrl+K →	Move active editor group right/down

File management

Ctrl+N	New File
Ctrl+O	Open File...
Ctrl+S	Save
Ctrl+Shift+S	Save As...
Ctrl+W	Close
Ctrl+K Ctrl+W	Close All
Ctrl+Shift+T	Reopen closed editor
Ctrl+K Enter	Keep preview mode editor open
Ctrl+Tab	Open next
Ctrl+Shift+Tab	Open previous
Ctrl+K P	Copy path of active file
Ctrl+K R	Reveal active file in Explorer
Ctrl+K O	Show active file in new window/instance

Debug

F9	Toggle breakpoint
F5	Start / Continue
F11 / Shift+F11	Step into/out
F10	Step over
Shift+F5	Stop
Ctrl+K Ctrl+I	Show hover

Integrated terminal

Ctrl+`	Show integrated terminal
Ctrl+Shift+`	Create new terminal
Ctrl+Shift+C	Copy selection
Ctrl+Shift+V	Paste into active terminal
Ctrl+Shift+ ↑ / ↓	Scroll up/down
Shift+ PgUp / PgDn	Scroll page up/down
Shift+ Home / End	Scroll to top/bottom

* The Alt+Click gesture may not work on some Linux distributions. You can change the modifier key for the Insert cursor command to Ctrl+Click with the "editor.multiCursorModifier" setting.

高频使用快捷键：

功能	快捷键	功能	快捷键
转到文件 / 其他常用操作	`Ctrl + P`	关闭当前文件	`Ctrl + W`
打开命令面板	`Ctrl + Shift + P`	当前行上移/下移	`Alt + Up/Down`
打开终端	`Ctrl + `	变量统一重命名	`F2`
关闭侧边栏	`Ctrl + B`	转到定义处	`F12`
复制文本	`Ctrl+C`	粘贴文本	`Ctrl+V`
保存文件	`Ctrl+S`	撤销操作	`Ctrl+Z`

在 `Ctrl+P` 窗口下还可以:

- 直接输入文件名，跳转到文件
- `?` 列出当前可执行的动作
- `!` 显示 `Errors` 或 `Warnings`，也可以 `Ctrl+Shift+M`
- `:` 跳转到行数，也可以 `Ctrl+G` 直接进入
- `@` 跳转到 `symbol`（搜索变量或者函数），也可以 `Ctrl+Shift+O` 直接进入
- `@` 根据分类跳转 `symbol`，查找属性或函数，也可以 `Ctrl+Shift+O` 后输入:进入
- `#` 根据名字查找 `symbol`，也可以 `Ctrl+T`

快捷键：编辑器与窗口管理

1. 打开一个新窗口： `Ctrl+Shift+N`
2. 关闭窗口： `Ctrl+Shift+W`
3. 同时打开多个编辑器（查看多个文件）
4. 新建文件 `Ctrl+N`
5. 文件之间切换 `Ctrl+Tab`
6. 切出一个新的编辑器（最多 `3` 个） `Ctrl+\`，也可以按住 `Ctrl` 鼠标点击 `Explorer` 里的文件名
7. 左中右 `3` 个编辑器的快捷键 `Ctrl+1` `Ctrl+2` `Ctrl+3`
8. `3` 个编辑器之间循环切换 `Ctrl+`
9. 编辑器换位置， `Ctrl+k` 然后按 `Left` 或 `Right`

↓ 代码编辑相关的快捷键 ↓

快捷键：格式调整

1. 代码行缩进 `Ctrl+[` 、 `Ctrl+]`
2. `Ctrl+C` 、 `Ctrl+V` 复制或剪切当前行/当前选中内容
3. 代码格式化： `Shift+Alt+F`，或 `Ctrl+Shift+P` 后输入 `format code`
4. 上下移动一行： `Alt+Up` 或 `Alt+Down`
5. 向上向下复制一行： `Shift+Alt+Up` 或 `Shift+Alt+Down`
6. 在当前行下边插入一行 `Ctrl+Enter`
7. 在当前行上方插入一行 `Ctrl+Shift+Enter`

快捷键：光标相关

1. 移动到行首： `Home`
2. 移动到行尾： `End`
3. 移动到文件结尾： `Ctrl+End`
4. 移动到文件开头： `Ctrl+Home`
5. 移动到定义处： `F12`

6. 定义处缩略图：只看一眼而不跳转过去 `Alt+F12`
7. 移动到后半个括号： `Ctrl+Shift+]`
8. 选择从光标到行尾： `Shift+End`
9. 选择从行首到光标处： `Shift+Home`
10. 删除光标右侧的所有字： `Ctrl+Delete`
11. 扩展/缩小选取范围： `Shift+Alt+Left` 和 `Shift+Alt+Right`
12. 多行编辑(列编辑)： `Alt+Shift+鼠标左键`， `Ctrl+Alt+Down/Up`
13. 同时选中所有匹配： `Ctrl+Shift+L`
14. `Ctrl+D` 下一个匹配的也被选中 (在 sublime 中是删除当前行，后面自定义快键键中，设置与 `Ctrl+Shift+K` 互换了)
15. 回退上一个光标操作： `Ctrl+U`

快捷键：重构代码

1. 找到所有的引用： `Shift+F12`
2. 同时修改本文件中所有匹配的： `Ctrl+F12`
3. 重命名：比如要修改一个方法名，可以选中后按 `F2`，输入新的名字，回车，会发现所有的文件都修改了
4. 跳转到下一个 `Error` 或 `Warning`：当有多个错误时可以按 `F8` 逐个跳转
5. 查看 `diff`： 在 `explorer` 里选择文件右键 `Set file to compare`，然后需要对比的文件上右键选择 `Compare with file_name_you_chose`

快捷键：查找替换

1. 查找 `Ctrl+F`
2. 查找替换 `Ctrl+H`
3. 整个文件夹中查找 `Ctrl+Shift+F`

快捷键：显示相关

1. 全屏： `F11`
2. zoomIn/zoomOut： `Ctrl +/-`
3. 侧边栏显/隐： `Ctrl+B`
4. 显示资源管理器 `Ctrl+Shift+E`
5. 显示搜索 `Ctrl+Shift+F`
6. 显示 Git `Ctrl+Shift+G`
7. 显示 Debug `Ctrl+Shift+D`
8. 显示 Output `Ctrl+Shift+U`

5.4【实战】2个小项目

通过手写2个小项目，呈现出基本的C++工程建立的过程，并编译运行这两个小项目

- **项目1：Hello world**
- **项目2：Swap with class**

5.4.1 高频使用技巧

- 左右分屏
- 固定打开的文件
- 格式化代码
- Rename Symbol
- 显示minimap

- 侧边栏查看OUTLINE
- 全屏
- 关闭多个打开的文件

5.4.2 代码编写

5.4.3 编译并运行

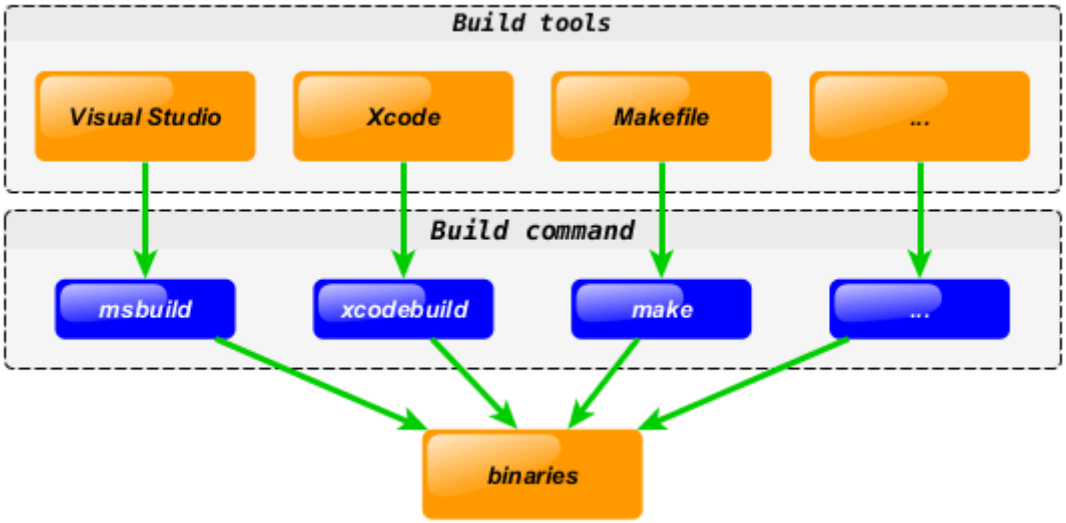
第六讲：CMake

前言：

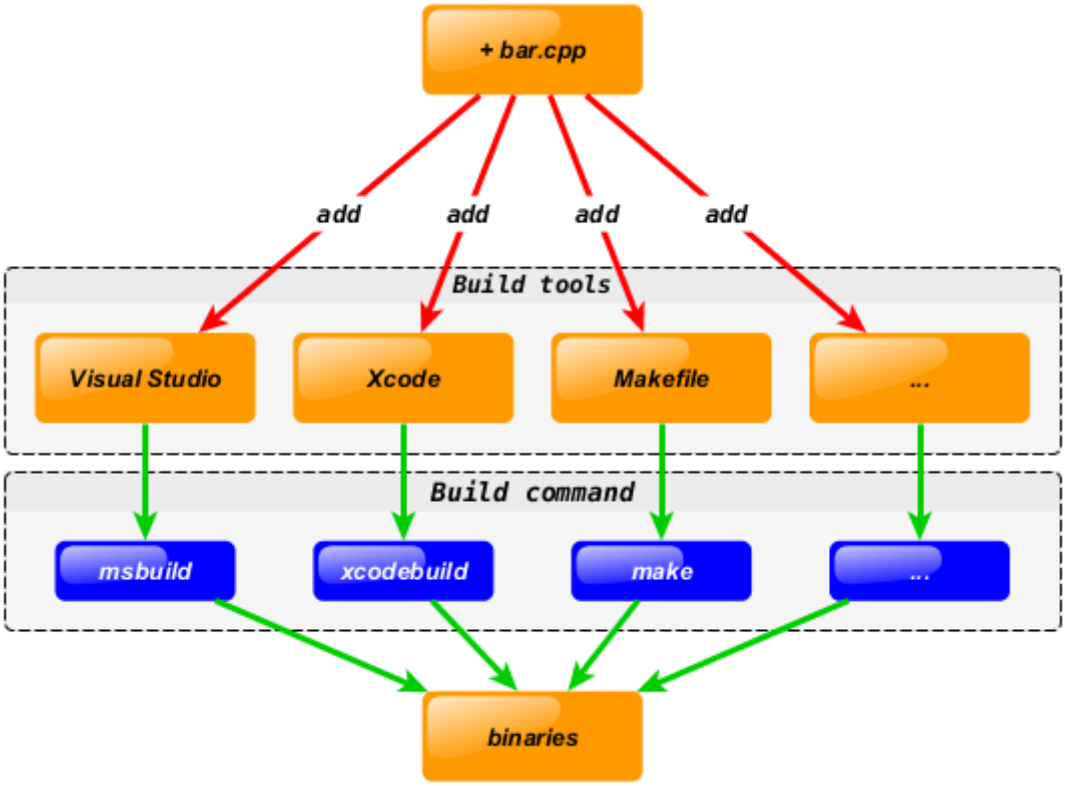
- **CMake**是一个**跨平台**的安装**编译工具**，可以用**简单**的语句来描述**所有平台**的安装(编译过程)。
- CMake可以说已经成为**大部分C++开源项目标配**

6.1 Cross-platform development

Let's assume you have some cross-platform project with C++ code shared along different platforms/IDEs. Say you use `Visual Studio` on Windows, `Xcode` on OSX and `Makefile` for Linux:

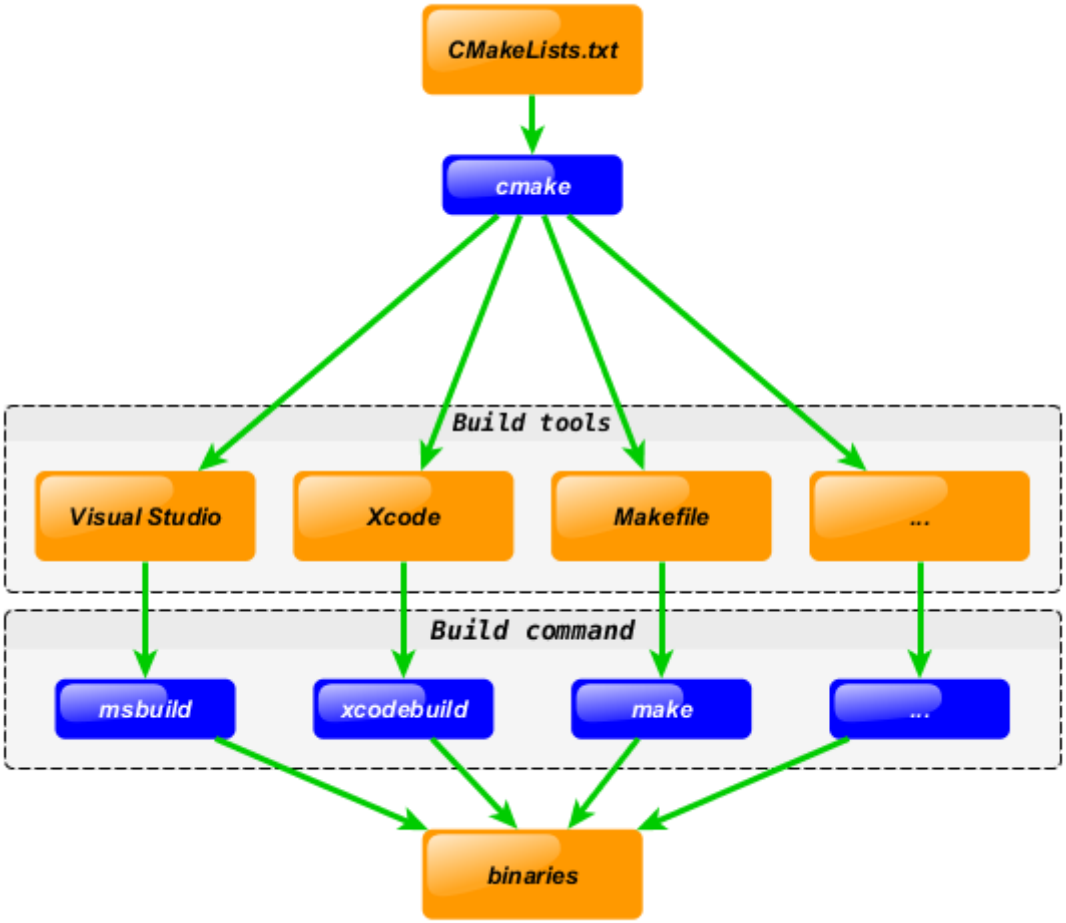


What you will do if you want to add new `bar.cpp` source file? You have to add it to every tool you use:

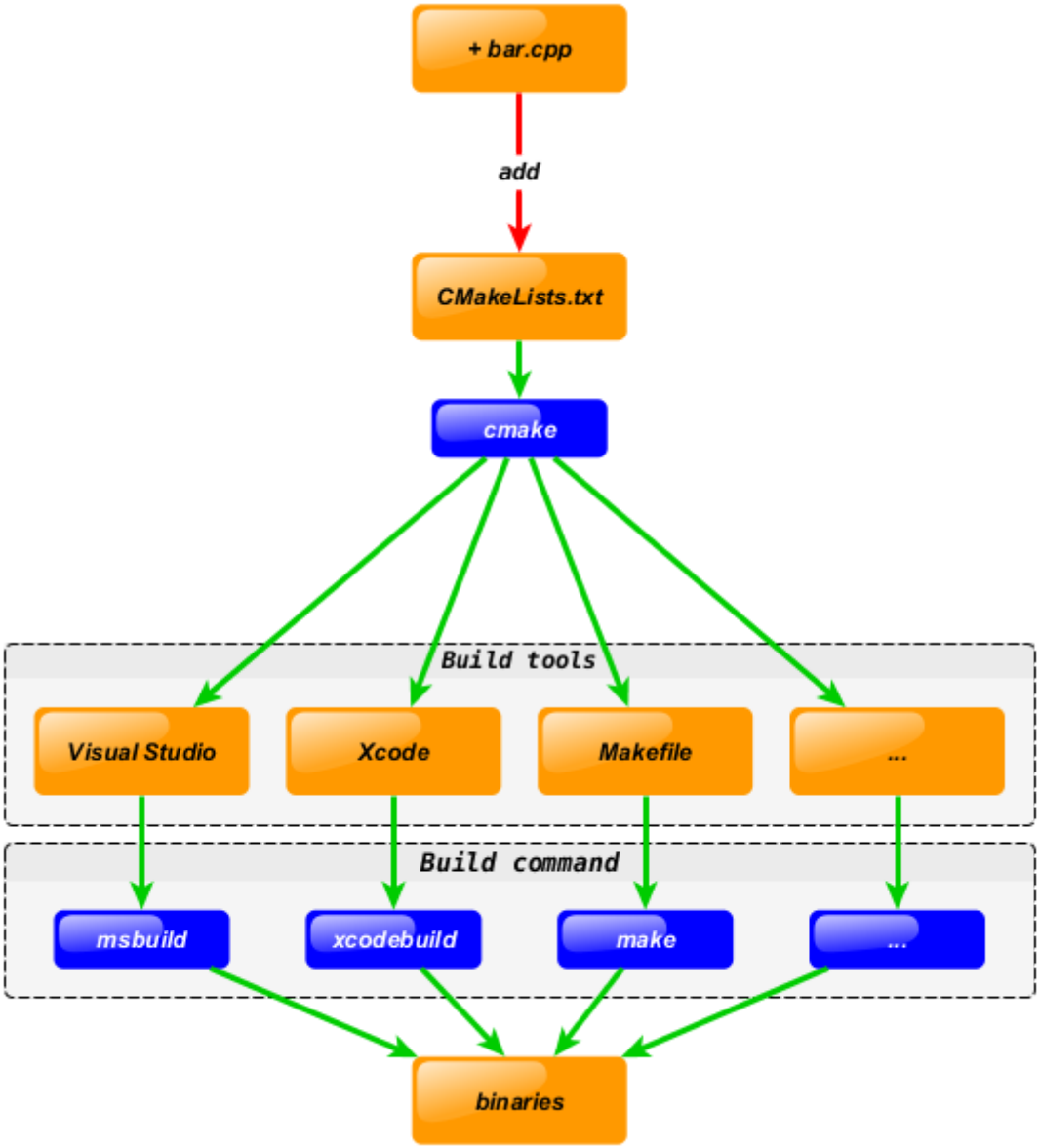


To keep the environment consistent you have to do the similar update several times. And the most important thing is that you have to do it **manually** (arrow marked with a red color on the diagram in this case). Of course such approach is error prone and not flexible.

CMake solve this design flaw by adding extra step to development process. You can describe your project in `CMakeLists.txt` file and use CMake to generate tools you currently interested in using cross-platform CMake code:



Same action - adding new `bar.cpp` file, will be done in **one step** now:



Note that the bottom part of the diagram **was not changed**. I.e. you still can keep using your favorite tools like `Visual Studio/msbuild`, `Xcode/xcodebuild` and `Makefile/make`!

6.2 语法特性介绍

- **基本语法格式：指令(参数 1 参数 2...)**
  - 参数使用**括弧**括起
  - 参数之间使用**空格**或**分号**分开
- **指令是大小写无关的，参数和变量是大小写相关的**

```
set(HELLO hello.cpp)
add_executable(hello main.cpp hello.cpp)
ADD_EXECUTABLE(hello main.cpp ${HELLO})
```

- **变量使用${}方式取值，但是在 IF 控制语句中是直接使用变量名**

6.3 重要指令和CMake常用变量

6.3.1 重要指令

- **cmake_minimum_required - 指定CMake的最小版本要求**
  - 语法： **cmake_minimum_required(VERSION versionNumber [FATAL_ERROR])**

```
# CMake最小版本要求为2.8.3
cmake_minimum_required(VERSION 2.8.3)
```

- **project - 定义工程名称，并可指定工程支持的语言**
  - 语法： **project(projectname [CXX] [C] [Java])**

```
# 指定工程名为HELLOWORLD
project(HELLOWORLD)
```

- **set - 显式的定义变量**
  - 语法：**set(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE])**

```
# 定义SRC变量，其值为sayhello.cpp hello.cpp
set(SRC sayhello.cpp hello.cpp)
```

- **include_directories - 向工程添加多个特定的头文件搜索路径** --->相当于指定g++编译器的-I参数
  - 语法： **include_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 ...)**

```
# 将/usr/include/myincludefolder 和 ./include 添加到头文件搜索路径
include_directories(/usr/include/myincludefolder ./include)
```

- **link_directories - 向工程添加多个特定的库文件搜索路径** --->相当于指定g++编译器的-L参数
  - 语法： **link_directories(dir1 dir2 ...)**

```
# 将/usr/lib/mylibfolder 和 ./lib 添加到库文件搜索路径
link_directories(/usr/lib/mylibfolder ./lib)
```

- **add_library - 生成库文件**
  - 语法： **add_library(libname [SHARED|STATIC|MODULE] [EXCLUDE_FROM_ALL] source1 source2 ... sourceN)**

```
# 通过变量 SRC 生成 libhello.so 共享库
add_library(hello SHARED ${SRC})
```

- **add_compile_options** - 添加编译参数
  - 语法：**add_compile_options(**

```
# 添加编译参数 -Wall -std=c++11 -O2
add_compile_options(-Wall -std=c++11 -O2)
```

- **add_executable - 生成可执行文件**
  - 语法：**add_executable(exename source1 source2 ... sourceN)**

```
# 编译main.cpp生成可执行文件main
add_executable(main main.cpp)
```

- **target_link_libraries - 为 target 添加需要链接的共享库** --->相同于指定g++编译器-l参数
  - 语法： **target_link_libraries(target library1<debug | optimized> library2...)**

```
# 将hello动态库文件链接到可执行文件main
target_link_libraries(main hello)
```

- **add_subdirectory - 向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制存放的位置**
  - 语法： **add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])**

```
# 添加src子目录，src中需有一个CMakeLists.txt
add_subdirectory(src)
```

- **aux_source_directory - 发现一个目录下所有的源代码文件并将列表存储在一个变量中，这个指令临时被用来自动构建源文件列表**
  - 语法： **aux_source_directory(dir VARIABLE)**

```
# 定义SRC变量，其值为当前目录下所有的源代码文件
aux_source_directory(. SRC)
# 编译SRC变量所代表的源代码文件，生成main可执行文件
add_executable(main ${SRC})
```

6.3.2 CMake常用变量

- **CMAKE_C_FLAGS** gcc编译选项
- **CMAKE_CXX_FLAGS** g++编译选项

```
# 在CMAKE_CXX_FLAGS编译选项后追加-std=c++11
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
```

- **CMAKE_BUILD_TYPE** 编译类型(Debug, Release)

```
# 设定编译类型为debug，调试时需要选择debug
set(CMAKE_BUILD_TYPE Debug)
# 设定编译类型为release，发布时需要选择release
set(CMAKE_BUILD_TYPE Release)
```

- **CMAKE_BINARY_DIR**

  PROJECT_BINARY_DIR

  _BINARY_DIR

> 1. 这三个变量指代的内容是一致的。
> 2. 如果是 in source build，指的就是工程顶层目录。
> 3. 如果是 out-of-source 编译,指的是工程编译发生的目录。

> 4. PROJECT_BINARY_DIR 跟其他指令稍有区别，不过现在，你可以理解为他们是一致的。

- **CMAKE_SOURCE_DIR**

  PROJECT_SOURCE_DIR
  _SOURCE_DIR

> 1. 这三个变量指代的内容是一致的,不论采用何种编译方式,都是工程顶层目录。
> 2. 也就是在 in source build时,他跟 CMAKE_BINARY_DIR 等变量一致。
> 3. PROJECT_SOURCE_DIR 跟其他指令稍有区别,现在,你可以理解为他们是一致的。

- **CMAKE_C_COMPILER：指定C编译器**
- **CMAKE_CXX_COMPILER：指定C++编译器**
- **EXECUTABLE_OUTPUT_PATH：可执行文件输出的存放路径**
- **LIBRARY_OUTPUT_PATH：库文件输出的存放路径**

6.4 CMake编译工程

CMake目录结构：项目主目录存在一个CMakeLists.txt文件

两种方式设置编译规则：

1. 包含源文件的子文件夹**包含**CMakeLists.txt文件，主目录的CMakeLists.txt通过add_subdirectory添加子目录即可；
2. 包含源文件的子文件夹**未包含**CMakeLists.txt文件，子目录编译规则体现在主目录的CMakeLists.txt中；

6.4.1 编译流程

在 linux 平台下使用 CMake 构建C/C++工程的流程如下:

- 手动编写 CMakeLists.txt。
- 执行命令 `cmake PATH` 生成 Makefile (PATH 是顶层CMakeLists.txt 所在的目录)。
- 执行命令 `make` 进行编译。

```
# important tips
.       # 表示当前目录
./      # 表示当前目录

..      # 表示上级目录
../     # 表示上级目录
```

6.4.2 两种构建方式

- **内部构建(in-source build)**：不推荐使用

  内部构建会在同级目录下产生一大堆中间文件，这些中间文件并不是我们最终所需要的，和工程源文件放在一起会显得杂乱无章。

```
## 内部构建

# 在当前目录下，编译本目录的CMakeLists.txt，生成Makefile和其他文件
cmake .
# 执行make命令，生成target
make
```

- **外部构建(out-of-source build)**：推荐使用

  将编译输出文件与源文件放到不同目录中

```
## 外部构建

# 1. 在当前目录下，创建build文件夹
mkdir build
# 2. 进入到build文件夹
cd build
# 3. 编译上级目录的CMakeLists.txt，生成Makefile和其他文件
cmake ..
# 4. 执行make命令，生成target
make
```

6.5 【实战】CMake代码实践

针对第五章写的两个小项目来写对应的CMakeLists.txt

6.5.1 最小CMake工程

```
# Set the minimum version of CMake that can be used
cmake_minimum_required(VERSION 3.0)

# Set the project name
project (HELLO)

# Add an executable
add_executable(hello_cmake main.cpp)
```

6.5.2 多目录工程 - 直接编译

```
# Set the minimum version of CMake that can be used
cmake_minimum_required(VERSION 3.0)

#project name
project(SWAP)

#head file pat
include_directories( include )

#source directory files to var
add_subdirectory( src DIR_SRCS )

#add executable file
add_executable(swap_02 ${TEST_MATH})

#add link library
```

```
target_link_libraries(${FS_BUILD_BINARY_PREFIX}sqrt ${LIBRARIES})
```

6.5.3 多目录工程 - 生成库编译

```
# Set the minimum version of CMake that can be used
cmake_minimum_required(VERSION 3.0)

#project name
project(SWAP_LIBRARY)

#add compile options
add_compile_options("-Wall -std=c++11")

#set CMAKE_BUILD_TYPE
set( CMAKE_BUILD_TYPE Debug )

# set output binary path
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)

###########################################################
# Create a library
###########################################################

#Generate the static library from the library sources
add_library( swap_library STATIC src/Swap.cpp )

target_include_directories( swap_lib PUBLIC ${PROJECT_SOURCE_DIR}/include )

###########################################################
# Create an executable
###########################################################

# Add an executable with the above sources
add_executable( swap_01 main.cpp )

# link the new swap_01 target with the swap_lib target
target_link_libraries( swap_01 swap_liby )
```

第七讲：【实战】使用VSCode进行完整项目开发

案例：士兵突击

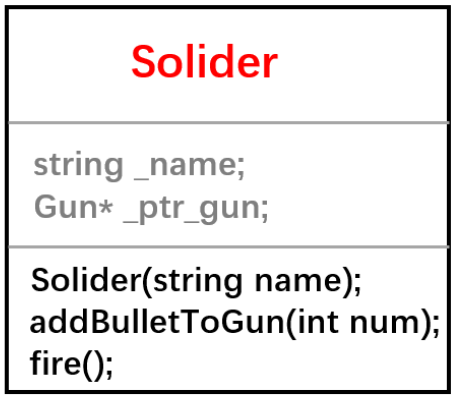
需求：

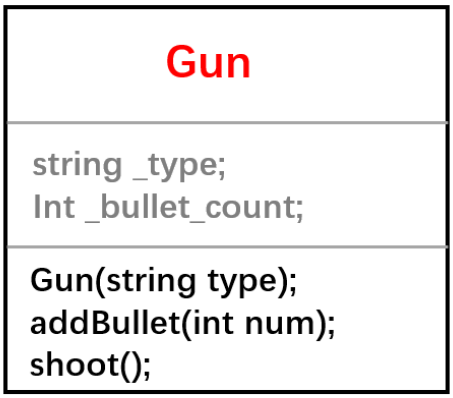
1. **士兵 许三多** 有一把**枪**，叫做 **AK47**
2. **士兵** 可以 **开火**
3. **士兵** 可以 **给枪装填子弹**
4. **枪** 能够 **发射** 子弹
5. **枪** 能够 **装填子弹** —— **增加子弹数量**

开发：

- 开发枪类

- 开发士兵类





7.1 合理设置项目目录

7.2 编写项目源文件

7.3 编写CMakeLists.txt构建项目编译规则

7.4 使用外部构建，手动编译CMake项目

7.5 配置VSCode的json文件并调试项目