CMake 以及 VSCode的代码调试学习

1.GCC

1.1编译过程

1. 预处理

```
# -E
g++ -E test.cpp -o test.i //生成了.i文件
```

2. 编译

```
# -S
g++ -S test.i -o test.s
```

3. 汇编

```
#-c
g++ -c test.s -o tset.o
```

4. 链接

```
#直接-o生成可执行文件
g++ test.o -o test
```

上述流程可用以下代码代替:

```
g++ test.cpp -o test
```

1.2 g++常用编译参数

1. **-g** 编译带调试信息的可执行文件

```
#-g 可以使g++产生能被调试器使用的调试信息
#下面代码告诉编译器产生带调试信息的可执行文件test
g++ -g test.cpp -o test
```

2.**-O[n]** 优化源代码

```
g++ -O2 test.cpp #一般使用-O2就可以
```

3.**-l**和**-L** 前者指定库文件 | 后者指定库文件路径

```
#-l(小写) 后面直接加库名
#在/lib , /usr/lib, /usr/local/lib 里的库-l可以直接链接，如:
g++ -lglog test.cpp

#如果库文件不在上面三个目录里，需要先用-L(大写)指定目录，在跟-l,如:

g++ -L/home/bing/fol1 -lmytest test.cpp
```

4.**-I**指定头文件搜索目录

```
#-I
#/usr/include目录里的头文件，gcc可以自己搜索，如果不在则需要另外指定，同时-I的参数可以为绝对路径，也可以为相对路径。如:
g++ -I/myinclude test.cpp
```

5.**-Wall** 打印文件警告信息

6.**-w** 关闭警告信息

7.**-std=c++11** 设置编译标准

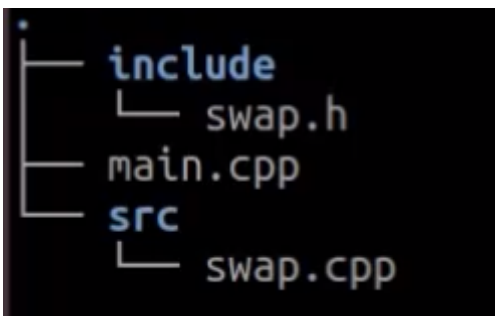
8.**-o** 指定输出文件名

9.**-D** 定义宏

```
// 可以开启关闭宏，如:

#include<stdio.h>

int main(){
    #ifdef DEBUG
      printf("hello");
    #endif
    printf("aaa");

    return 0;
}

// 编译时， 如果写: g++ -DDEBUG main.cpp 则 hello 会被打印
```

1.3 g++命令行实操

对于这样一个结构:



如何使用g++ 将其生成可执行文件呢?

第一种方式我们用比较简单的少参数方式:

```
# 别忘了-I链接头文件目录
g++ main.cpp src/swap.cpp -Iinclude
#以上操作会默认生成一个a.out文件，接下来将其运行:
./a.out
```

第二种方式我们在一的基础上增加一些别的参数:

```
g++ main.cpp src/swap.cpp -Iinclude -Wall -std=c++11 -o b.out
//这样就会生成名为b.out且使用c++11为编译标准的可执行文件
```

接着，我们来使用生成库文件，再将其编译的方式:

第一种，我们生成一个静态库:

```
//此流程为操作过程:
//第一步,进入src目录
cd src
//接着将swap.cpp生成一个二进制.o文件
g++ swap.cpp -c -I../include
//再将生成的swap.o文件生成为静态库libswap.a
ar rs libswap.a swap.o
//回到上级目录
cd ..
// 将库与main.cpp链接，生成可执行文件
g++ main.cpp -Iinclude -Lsrc -lswap -o staticman  //要-I是因为本例中mian.cpp中
引用了swap.h
```

第二种，生成动态库:

```
//生成:
g++ swap.cpp -I../include -fPIC -shared -o libswap.so
//等价于下面这种写法:
g++ swap.cpp -I../include -c -fPIC
g++ -shared -o libswap.so swap.o
```

但是，要运行动态库生成的可执行文件则要加上**LD_LIBRARY_PATH=src**再运行

---

2. GDB调试器

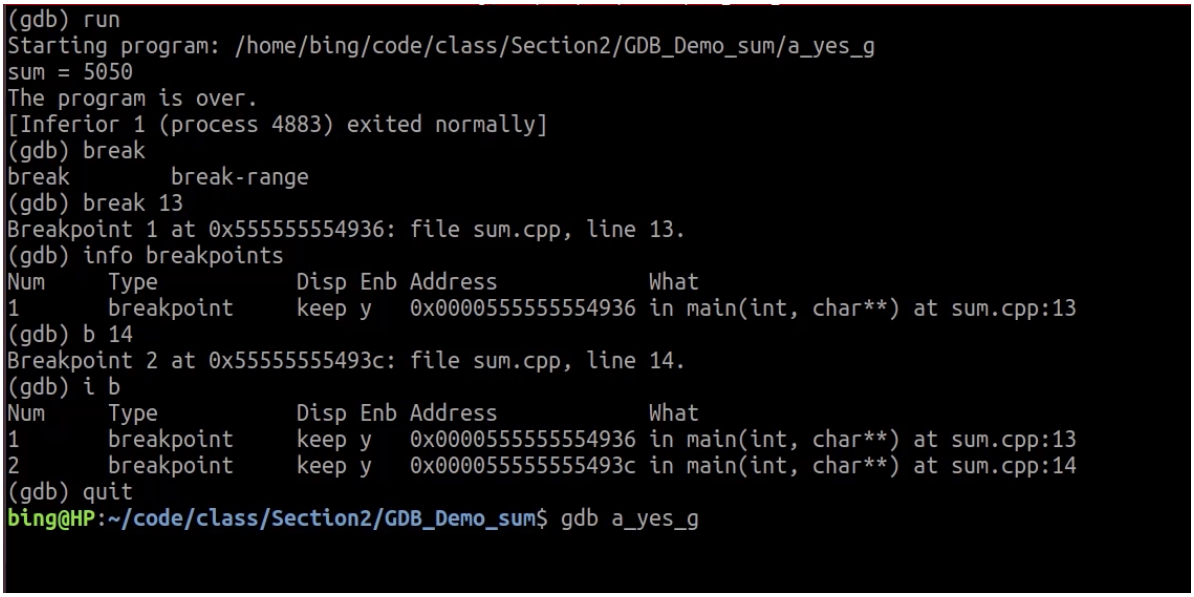
GDB主要功能：

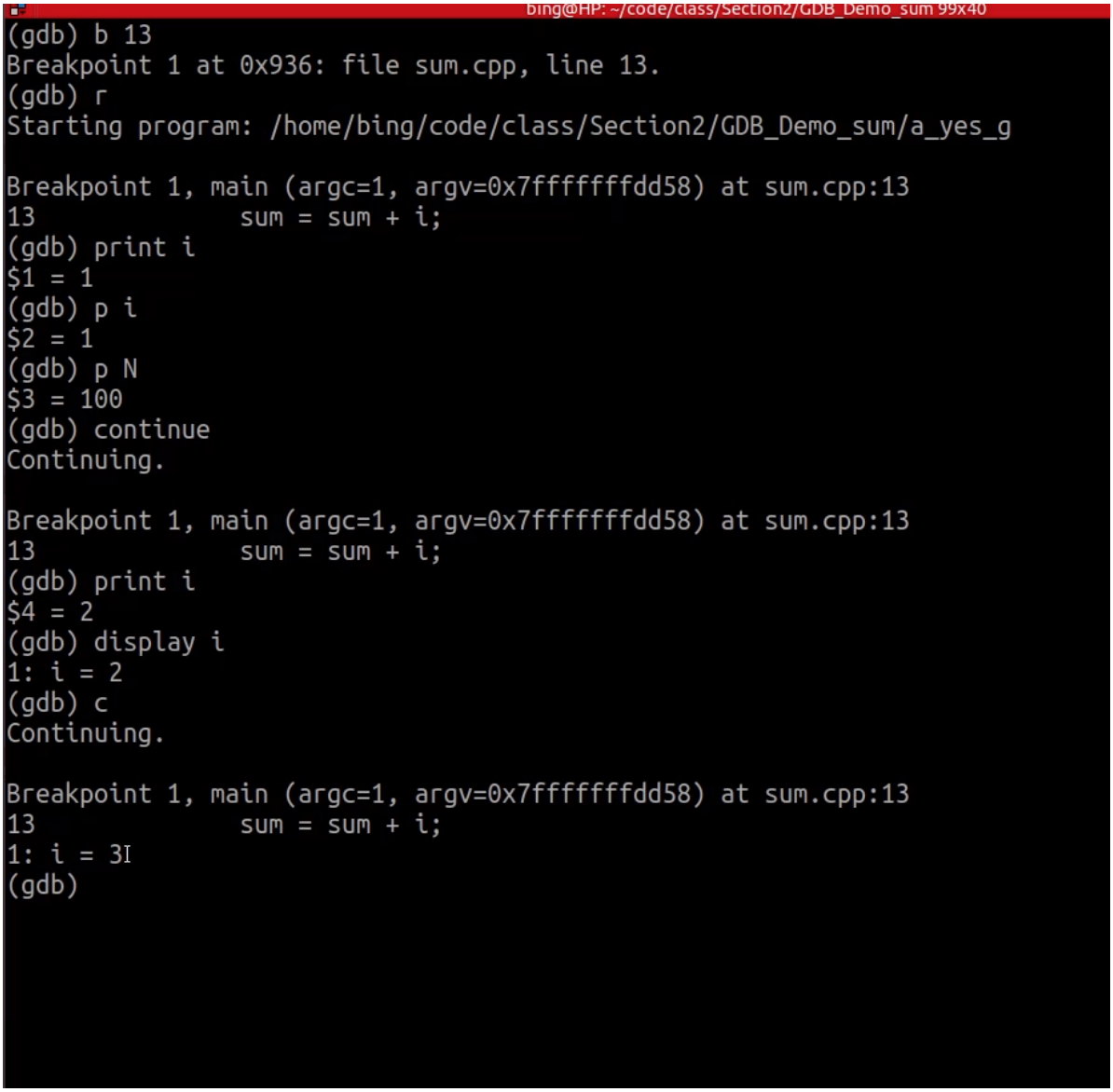
- 设置断点
- 在指定代码行暂停执行
- 单步执行
- 查看变量值
- 动态改变执行环境
- 分析崩溃产生的core文件

gdb常用命令

```
## 格式:  输入 gdb[exefilename] 进入调试，其中exefilename为可执行文件名

run(r):重新开始
start:单步执行，停在第一行
next(n):单步，不会进内部
step(s): 单步，会进内部
finish:完成当前函数
```

执行命令图片(供参考):





2. VSCode 使用

打开工程：进入文件夹后输入**code .**

常用快捷键：

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`

3. CMake

1. 语法特性

- **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**

P.S.:如果在if中使用${}来写变量名是错误的

2. 常用指令

- **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)
```

cmake

- **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main.cpp hello.cpp
set(SRC sayhello.cpp hello.cpp)
```

P.S.:指令中的可选项不一定要都写上

- **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(<option> ...)**

```
# 添加编译参数 -Wall -std=c++11
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)
```

cmake

- **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)
```

- **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])**

```cmake
# 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)**

```cmake
# 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})
```

3. 常用变量

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

```cmake
# 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)

```cmake
# 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

  <projectname>_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。

4. CMake编译工程

CMake的目录结构：

首先，项目主目录要存在一个CMakeLists.txt文件，这是必须的。

接下来，针对不同的文件目录结构，我们有两种编译选择：

1.如果包含源文件的子目录中已经含有CMakeLists.txt , 则我们只需要在主目录的CMakeLists.txt中通过add_subdirectory添加子目录即可。

2.如果包含源文件的子目录中不含CMakeLists.txt文件，则子目录的编译规则体现在主目录的CMakeLists.txt中。

CMake编译流程：

```
* 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
```

P.S.:

我们在构建工程时最好采用外部构建的方式，即先mkdir一个build文件夹，然后在build文件夹中cmake .. ，将产生的一堆中间文件扔到build中，保证其他文件夹的良好可读性。

实战的一些代码：