## CMake

### CMake 使用方法

<http://blog.csdn.net/u012150179/article/details/17852273>

     CMake的所有的语句都写在一个叫:CMakeLists.txt的文件中。当CMakeLists.txt文件确定后,可以用ccmake命令对相关 的变量值进行配置。这个命令必须指向CMakeLists.txt所在的目录。配置完成之后,应用cmake命令生成相应的makefile（在Unix like系统下）或者 project文件（指定用window下的相应编程工具编译时）。

    其基本操作流程为：

$**>**  ccmake directory

$**>**  cmake  directory

$**>**  make

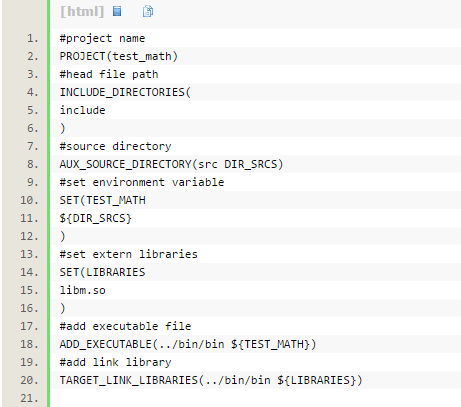
其中directory为CMakeList.txt所在目录；

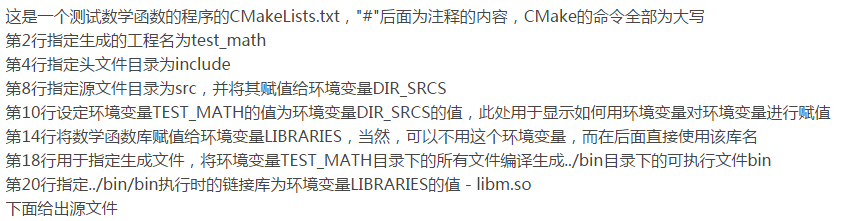
第一条语句用于配置编译选项，如VTK\_DIR目录 ，一般这一步不需要配置，直接执行第二条语句即可，但当出现错误时，这里就需要认为配置了，这一步才真正派上用场；

第二条命令用于根据CMakeLists.txt生成Makefile文件；

第三条命令用于执行Makefile文件，编译程序，生成可执行文件；

CMake的执行就是这么简单，其难点在于如何编写CMakeLists.txt文件，下面结合例子简单介绍CMakeLists.txt的编写，看下面这个CMakeLists.txt





### CMake简介

CMake是一种跨平台编译工具，比make更为高级，使用起来要方便得多。CMake主要是编写CMakeLists.txt文件，然后用cmake命令将CMakeLists.txt文件转化为make所需要的makefile文件，最后用make命令编译源码生成可执行程序或共享库（so(shared object)）。因此CMake的编译基本就两个步骤：

1. cmake

2. make

CMake是一个跨平台的编译自动配置工具，它使用一个名为CMakeLists.txt的文件来描述构建过程，可以产生标准的构建文件。它可以用简单的语句来描述所有平台的安装(编译过程)。它能够输出各种各样的**makefile**或者**project**文件，能测试编译器所支持的C++特性，类似UNIX下的automake。CMake并不直接建构出最终的软件，而是产生标准的建构档(如Unix的Makefile或Windows Visual C++的projects/workspaces)，然后再依一般的建构方式使用。

CMake可以编译源代码、制作程式库、产生适配器(wrapper)、还可以用任意的顺序建构执行档。CMake支持in-place建构(二进档和源代码在同一个目录树中)和out-of-place建构(二进档在别的目录里)，因此可以很容易从同一个源代码目录树中建构出多个二进档。CMake也支持静态与动态程式库的建构。CMake是一个比make更高级的编译配置工具。

         CMake的组态档取名为CMakeLists.txt。组态档是用一种建构软件专用的特殊编程语言写的CMake脚本。文件CMakeLists.txt需要手工编写，也可以通过编写脚本进行半自动的生成。通过编写CMakeLists.txt，可以控制生成的Makefile，从而控制编译过程。

         CMake主要特点：(1)、开放源代码，使用类BSD许可发布；(2)、跨平台，并可生成native编译配置文件，在Linux/Unix平台，生成**makefile**；在苹果平台，可以生成**xcode**；在windows平台，可以生成**msvc**的工程文件；(3)、能够管理大型项目；(4)、简化编译构建过程和编译过程，CMake的工具链非常简单：cmake+make；(5)、高效率；(6)、可扩展，可以为cmake编写特定功能的模块，扩充cmake功能。

### CMake语法

<http://blog.csdn.net/fengbingchun/article/details/46685413>

  CMakeLists.txt的语法比较简单，由命令、注释和空格组成，其中命令是不区分大小写的，参数和变量是大小写相关的，但，推荐全部使用大写指令。符号”#”后面的内容被认为是注释。命令由命令名称、小括号和参数组成，参数之间使用空格或分号进行间隔。变量使用${xxx}引用。

         常用命令说明：

1.      aux\_source\_directory(<dir><variable>) :该命令会把参数<dir>中所有的源文件(不包括头文件)名称赋值给参数<variable>；

2.      find\_path(<VAR> name1[path1 path2 …]):该命令在参数path\*指示的目录中查找文件name1并将查找到的路径保存在变量VAR中(其中使用”[…]”包含的项表示可忽略项，使用”…|…”分割的项表示只能选择其中一项)；

3.      find\_library(${var} NAMES name1[name2 …] PATHS path1 [path2 …] PATH\_SUFFIXES suffix1 [uffix2 …]):搜索一个外部的链接库文件,并将结果的全部路径保存到var变量中。要搜索的链接库文件名字可能是name1,name2等；搜索路径为path1, path2等；此外还可以指定路径的后缀词为suffix1,suffix2等；

4.      find\_package(name):在指定的模块目录中搜索一个名为Find<name>.cmake(例如，FindOSG.cmake)的CMake脚本模块文件，执行其中的内容，意图搜索到指定的外部依赖库头文件和库文件位置；

5.      find\_program:搜索一个外部的可执行程序；

6.      project(name):指定项目名称name；

7.      include(file):在当前文件中包含另一个CMake脚本文件的内容,用来载入CMakeLists.txt文件，也用于载入预定义的cmake模块；

8.      include\_directories:指定头文件的搜索路径，用来向工程添加多个特定的头文件搜索路径，可以多次调用以设置多个路径，相当于指定gcc的-I参数；

9.      link\_directories:添加非标准的共享库搜索路径，设置外部动态链接库或静态链接库的搜素路径，相当于gcc的-L参数；

10.  link\_libraries:添加链接库；

11.  add\_subdirectory:用于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文件存放的位置；

12.  add\_executable:编译可执行程序，指定编译，好像也可以添加.o文件；

13.  add\_definitions(-DMACRO1-DMACRO2 …):添加编译参数，添加-D预编译宏定义，可以一次添加多个；

14.  add\_dependencies:定义target依赖的其它target，确保在编译本target之前，其它的target已经被构建；

15.  add\_library:可以设置要生成的链接库为SHARED或者STATIC，还可以设置MODULE(插件，可动态调用，但不作为其它工程的依赖)；

16.  add\_custom\_target(name COMMANDcmd1 [COMMAND cmd2 ..]):添加一个名为name的编译目录，并指定一个或多个自定义的命令cmd1,cmd2等；注意ADD\_CUSTOM\_COMMAND与这个命令的区别：前者是针对一个已有的子工程进行自定义编译规则的设置；后者则是建立一个新的自定义的目标工程；

17.  target\_link\_libraries:可以用来为target添加需要链接的共享库，指定工程所用的依赖库，添加链接库，添加动态库或静态库，相当于指定-l参数；

18.  message:打印消息，在控制台或者对话框输出一行或多行调试信息；

19.  set:定义一个用户自定义变量；

20.  set\_target\_properties:用来设置输出的名称，对于动态库，还可以用来指定动态库版本和API版本；

21.  cmake\_minimum\_required:设定依赖的cmake版本；

22.  configure\_file(infile outfile):将文件infile复制到outfile的位置，同时执行其中变量的自动配置和更替;

23.  install:安装目标工程到指定的文件夹,此命令用于定义安装规则，安装的内容可以包括目标二进制、动态库、静态库以及文件、目录、脚本等；

24.  option(${var} “text” value):向用户提供一个可选项，提示信息为text，初始值为value，并将最终的结果传递到var变量中；

25.  enable\_testing:用来控制Makefile是否构建test目标，涉及工程所有目录；

26.  exec\_program:用于在指定的目录运行某个程序；

27. execute\_process:执行一个或多个子进程，按指定的先后顺序运行一个或多个命令；

28.  file：文件操作命令；

  内置变量、环境变量：

1.      CMAKE\_C\_COMPILER:指定C编译器；

2.      CMAKE\_CXX\_COMPILER:指定C++编译器；

3.      CMAKE\_C\_FLAGS:指定编译C文件时的编译选项，如-g，也可以通过add\_definitions添加编译选项；

4.      CMAKE\_CXX\_FLAGS:设置C++编译选项；

5.      CMAKE\_BUILD\_TYPE:build类型(Debug,Release,…),CMAKE\_BUILD\_TYPE=Debug;

6.      CMAKE\_COMMAND:也就是CMake可执行文件本身的全路径；

7.      CMAKE\_DEBUG\_POSTFIX：Debug版本生成目标的后缀，通常可以设置为”d”字符；

8.      CMAKE\_GENERATOR:编译器名称，例如”UnixMakefiles”, “Visual Studio 7”等；

9.      CMAKE\_INSTALL\_PREFIX:工程安装目录，所有生成和调用所需的可执行程序，库文件，头文件都会安装到该路径下，Unix/Linux下默认为/usr/local, windows下默认为C:\Program Files;

10.  CMAKE\_MODULE\_PATH:设置搜索CMakeModules模块(.cmake)的额外路径，用来定义自己的cmake模块所在的路径；

11.  CMAKE\_CURRENT\_SOURCE\_DIR:指的是当前处理的CMakeLists.txt所在的路径；

12.  CMAKE\_CURRENT\_BINARY\_DIR:如果是in-source编译，则跟CMAKE\_CURRENT\_SOURCE\_DIR一致；如果是out-of-source，指的是target编译目录；

13.  CMAKE\_CURRENT\_LIST\_FILE:输出调用这个变量的CMakeLists.txt的完整路径；

14.  CMAKE\_CURRENT\_LIST\_LINE:输出这个变量所在的行；

15.  CMAKE\_INCLUDE\_CURRENT\_DIR:自动添加CMAKE\_CURRENT\_BINARY\_DIR和CMAKE\_CURRENT\_SOURCE\_DIR到当前处理的CMakeLists.txt;

16.  CMAKE\_INCLUDE\_DIRECTORIES\_PROJECT\_EFORE:将工程提供的头文件目录始终至于系统头文件目录的前面，当你定义的头文件确定跟系统发生冲突时可以提供一些帮助；

17.  EXECUTABLE\_OUTPUT\_PATH:指定可执行文件的存放路径，最终结果的存放目录；

18.  LIBRARY\_OUTPUT\_PATH:指定库文件存放路径，最终结果的存放目录；

19.  BUILD\_SHARED\_LIBS:指定编译成静态库还是动态库；

20.  PROJECT\_BINARY\_DIR(CMAKE\_BINARY\_DIR):如果是内部构建(in-sourcebuild),指的就是工程顶层目录；如果是外部构建(out-of-source build)，指的是工程编译发生的目录；

21.  PROJECT\_NAME:工程名称，即使用PROJECT命令设置的名称；

22.  PROJECT\_SOURCE\_DIR(CMAKE\_SOURCE\_DIR):工程源代码文件所在的目录，指的是工程顶层目录；

23.  CYGWIN:标识当前系统是否为Cygwin；

24.  MSVC:标识当前系统是否使用MicrosoftVisual C；

25.  UNIX:标识当前系统是否为Unix系列(包括Linux、Cygwin和Apple);

26.  WIN32：标识当前系统是否为Windows及Win64;

  内置变量的使用：

1.      在CMakeLists.txt中指定，使用set;

2.      cmake命令中使用，如cmake-DBUILD\_SHARED\_LIBS=OFF；

CMake调用环境变量的方式：使用$ENV{NAME}指令就可以调用系统的环境变量了。如MESSAGE(STATUS “HOME dir: $ENV {HOME}”)

         设置环境变量的方式是：SET(ENV{变量名} 值)

变量，以${MY\_VAIRABLE}的形式表达，其储存类型为字符串类型，但是可以根据具体命令的要求自动转换成布尔型、整型或者浮点类型。变量可以出现在字符串中，也可以实现”内省”。变量有用户自定义和系统内置两种，用户自定义变量使用SET命令设置；而系统变量由系统自动赋值，例如${PROJECT\_SOURCE\_DIR}。

CMake中的条件语句：IF(expression)… ELSE(expression) … ENDIF(expression) 或者IF(expression1) …ELSEIF(expression2) … ELSE() … ENDIF() ，expression是判断条件，和C/C++类似，CMake的条件也存在”与/或/非”以及”等于/大于/小于”等几种操作符，分别用AND/OR/NOT以及EQUAL/LESS/GREATER来表示。IF控制语句，变量是直接使用变量名引用，而不需要${}.

         CMake中的循环语句：FOREACH(vararg1 arg2 …) … ENDFOREACH(var) ,设置一个循环的局部变量var，每次将其赋为arg1, arg2等变量(或者变量数组)中的一个值，并执行循环中的命令段。

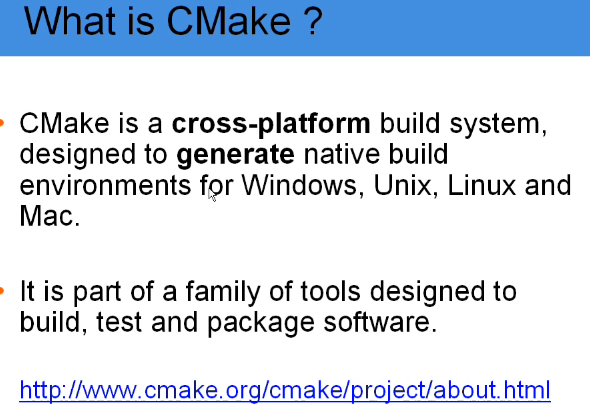
         CMake中的宏函数可以理解为C语言的函数，它改变代码执行跳转的流程并简化了脚本程序的开发：MACRO(funcname [arg1 [arg2 …]]) … ENDMACRO(funcname) ，和函数的编写要求一样，CMake的宏函数必须制定一个函数名funcname，以及零个或多个输入参数arg1,arg2等。需要调用宏函数的时候，只要直接使用funcname(arg1 arg2)的形式就可以了。

         CMake第一次运行的时候，它将产生一个文件叫CMakeCache.txt，该文件可以被看作一个配置文件，它里面的内容就像传递给configure命令的参数集。

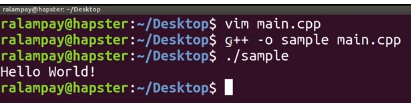
以下举例来说明CMake的详细用法：

总的目录结构为：

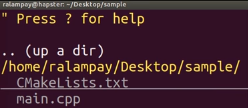
### CMake文件示例

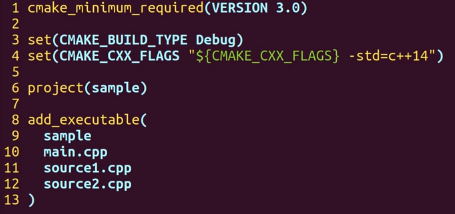


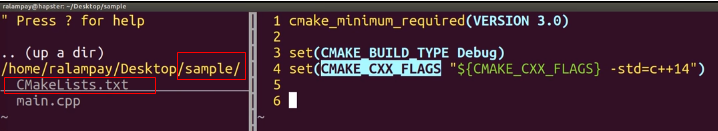
#### Source code



#### CMakeList.txt







### CMake常用指令

<http://blog.csdn.net/z_h_s/article/details/50699905>

### CMake示例

#### 多目录工程的CmakeLists.txt编写（自动添加多目录下的文件）

<http://blog.csdn.net/ktigerhero3/article/details/70313350>

实现类似于vs中工程的CMakeLists.txt的编写。功能为main.cpp调用hello.cpp 的hello()函数，world.cpp的world()函数。使用自动添加多目录下的文件。

##### CMakeLists.txt文件内容

----------------------------------------------------

cmake\_minimum\_required(VERSION 2.8)

project(helloworld)

# Add the source in project root directory

aux\_source\_directory(. DIRSRCS)

# Add header file include directories

include\_directories(./ ./hello ./world)

# Add block directories

add\_subdirectory(hello)

add\_subdirectory(world)

# Target

add\_executable(helloworld ${DIRSRCS})

target\_link\_libraries(helloworld hello world)

----------------------------------------------------

add\_executable:编译可执行程序，指定编译，好像也可以添加.o文件；

 target\_link\_libraries:可以用来为target添加需要链接的共享库，指定工程所用的依赖库，添加链接库，添加动态库或静态库，相当于指定-l参数；

##### 步骤









#### 示例2

<https://www.cnblogs.com/cv-pr/p/6206921.html>

##### 编写CMakeLists.txt

#1.cmake verson，指定cmake版本

cmake\_minimum\_required(VERSION 3.2)

#2.project name，指定项目的名称，一般和项目的文件夹名称对应

PROJECT(test\_sqrt)

#3.head file path，头文件目录

INCLUDE\_DIRECTORIES(

include

)

#4.source directory，源文件目录

AUX\_SOURCE\_DIRECTORY(src DIR\_SRCS)

#5.set environment variable，设置环境变量，编译用到的源文件全部都要放到这里，否则编译能够通过，但是执行的时候会出现各种问题，比如"symbol lookup error xxxxx , undefined symbol"

SET(TEST\_MATH

${DIR\_SRCS}

)

#6.add executable file，添加要编译的可执行文件

ADD\_EXECUTABLE(${PROJECT\_NAME} ${TEST\_MATH})

#7.add link library，添加可执行文件所需要的库，比如我们用到了libm.so（命名规则：lib+name+.so），就添加该库的名称

TARGET\_LINK\_LIBRARIES(${PROJECT\_NAME} m)

##### 编译和运行

mkdir build

cd build

cmake ..

make

├── build

│ ├── CMakeCache.txt

│ ├── CMakeFiles

│ │ ├── 3.2.2

│ │ │ ├── CMakeCCompiler.cmake

│ │ │ ├── CMakeCXXCompiler.cmake

│ │ │ ├── CMakeDetermineCompilerABI\_C.bin

│ │ │ ├── CMakeDetermineCompilerABI\_CXX.bin

│ │ │ ├── CMakeSystem.cmake

│ │ │ ├── CompilerIdC

│ │ │ │ ├── a.out

│ │ │ │ └── CMakeCCompilerId.c

│ │ │ └── CompilerIdCXX

│ │ │ ├── a.out

│ │ │ └── CMakeCXXCompilerId.cpp

│ │ ├── cmake.check\_cache

│ │ ├── CMakeDirectoryInformation.cmake

│ │ ├── CMakeOutput.log

│ │ ├── CMakeTmp

│ │ ├── feature\_tests.bin

│ │ ├── feature\_tests.c

│ │ ├── feature\_tests.cxx

│ │ ├── Makefile2

│ │ ├── Makefile.cmake

│ │ ├── progress.marks

│ │ ├── TargetDirectories.txt

│ │ └── test\_sqrt.dir

│ │ ├── build.make

│ │ ├── C.includecache

│ │ ├── cmake\_clean.cmake

│ │ ├── DependInfo.cmake

│ │ ├── depend.internal

│ │ ├── depend.make

│ │ ├── flags.make

│ │ ├── link.txt

│ │ ├── progress.make

│ │ └── src

│ │ ├── b.c.o

│ │ └── main.c.o

│ ├── cmake\_install.cmake

│ ├── Makefile

│ └── test\_sqrt

├── CMakeLists.txt

├── include

│ └── b.h

└── src

├── b.c

└── main.c

##### 结果

注意在build的目录下生成了一个可执行的文件test\_sqrt,运行获取结果如下：

命令：

./test\_sqrt

结果：

input a:49.000000

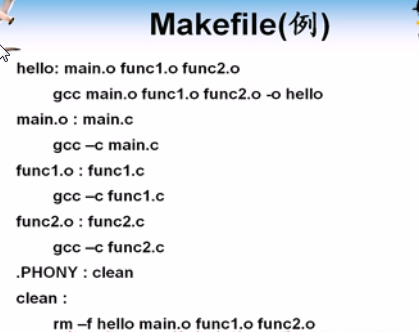
sqrt result:7.000000

## Make

### Makefile语法【make命令】

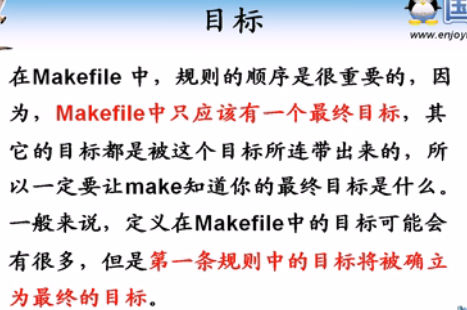
<http://v.youku.com/v_show/id_XMjAwNzAwMDUy.html?spm=a2h0k.8191407.0.0&from=s1.8-1-1.2>

#### 实例



#### 规则【目标、依赖、命令】

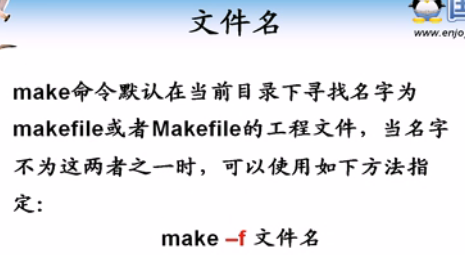




#### 伪目标



#### 命令【make [-f filename]】

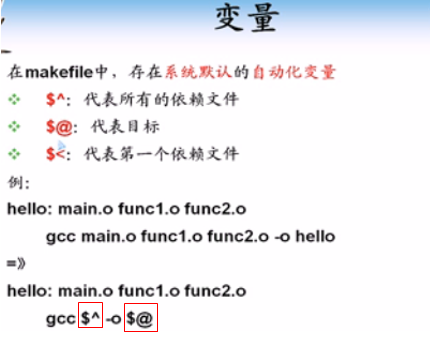


#### 变量

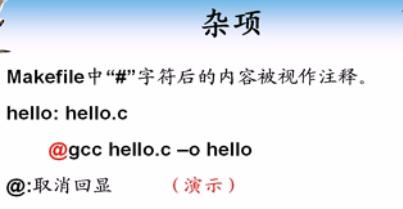
##### 自定义变量



##### 系统变量



#### 杂项【注释#、取消回显@、缩进是TAB而不是空格】



#### 示例OSS C SDK DEMO

# OSS C SDK DEMO FOR Ubuntu

INCLUDES = -I.

SRCS = demo.c sts/general.c sts/jsmn.c sts/request.c sts/time.c sts/token.c sts/util.c

OBJS = $(SRCS:.c=.o)

CC = gcc

CFLAGS = -O -g

LIBS = -lpthread -lcurl

EXE = demo

$(EXE):$(OBJS)

$(CC) $^ -o $@ $(LIBS)

%.o: %.c

$(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

clean:

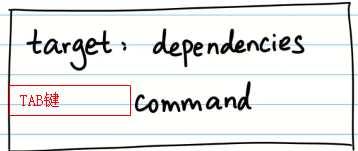
rm -f $(EXE) $(OBJS)

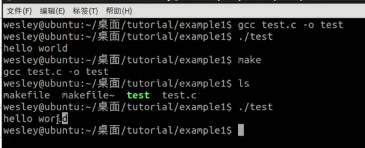
### Makefile示例：文件写法

<https://www.youtube.com/watch?v=E1_uuFWibuM>

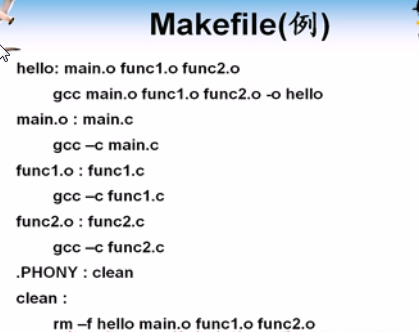
#### Makefile文件名

文件名makefile，现在也支持Makefile

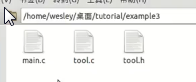


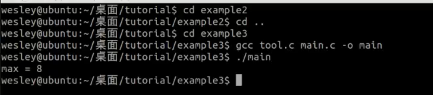


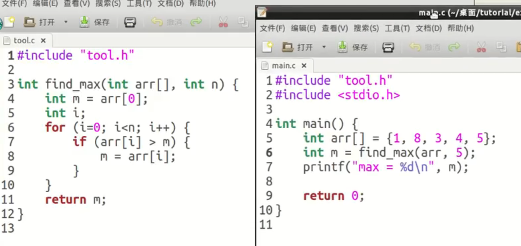


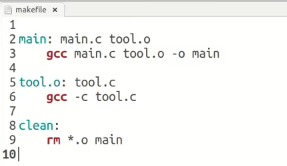


#### 示例









obj=main.o tool.o

main: $obj

gcc $obj -o main

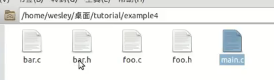
main.o: main.c

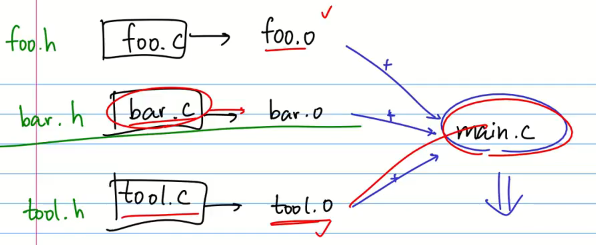
gcc -c main.c -o main.o

tool.o : tool.c

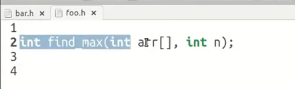
gcc -c tool.c -o tool.c

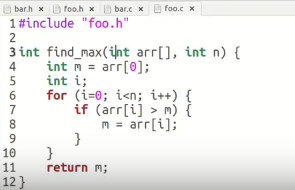
#### 示例4

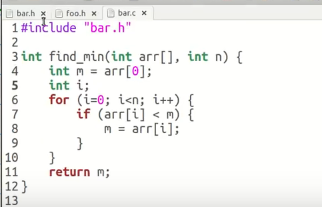


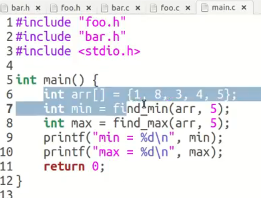


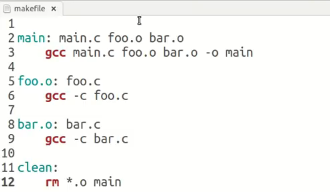
h头文件

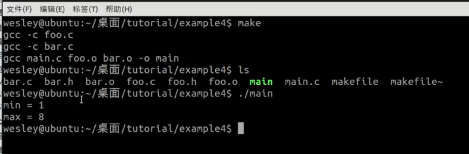




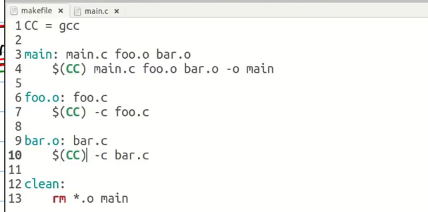


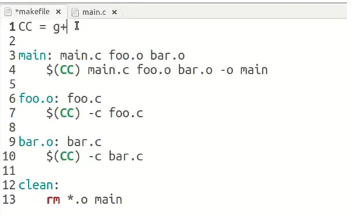


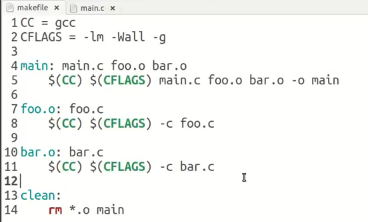




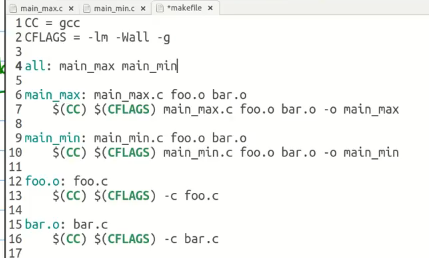
#### 变量







#### 输出2个目标结果

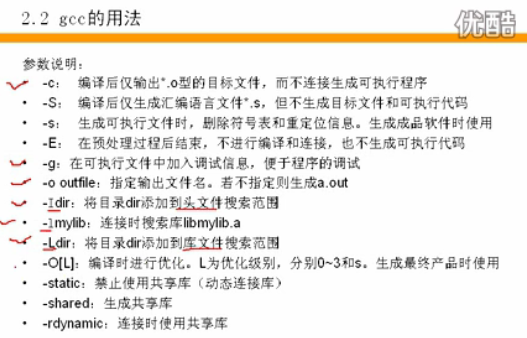


#### g++



## GCC编译链接

### gcc语法



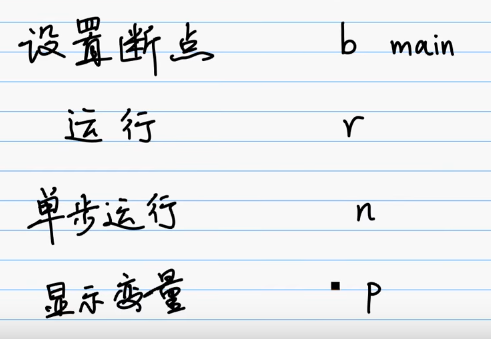
### 编译C程序



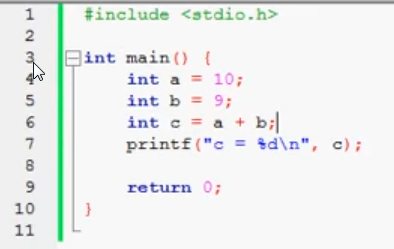
### 编译C++程序



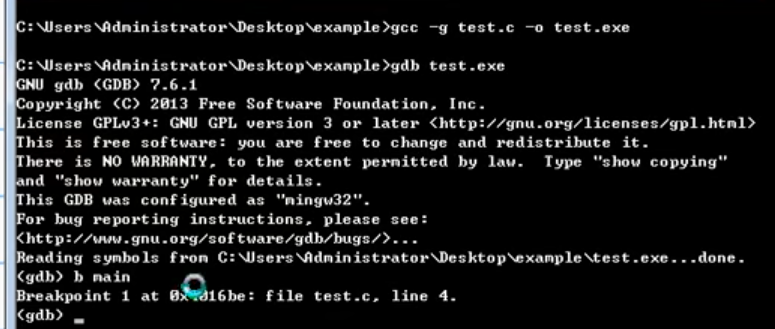
## GDB调试



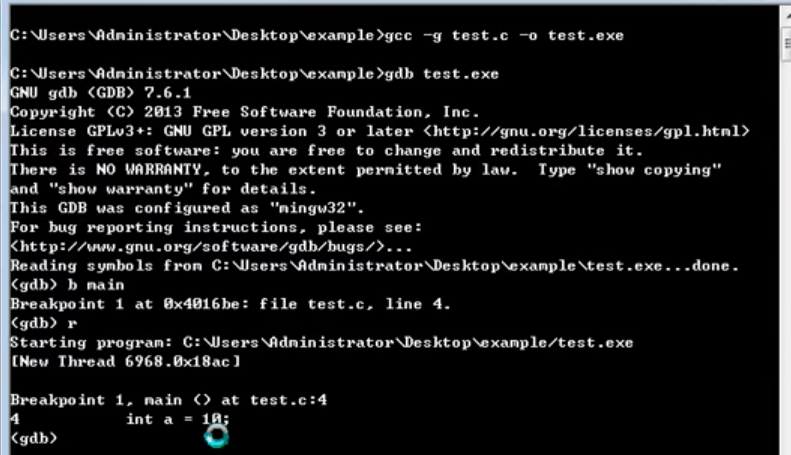
### 源代码

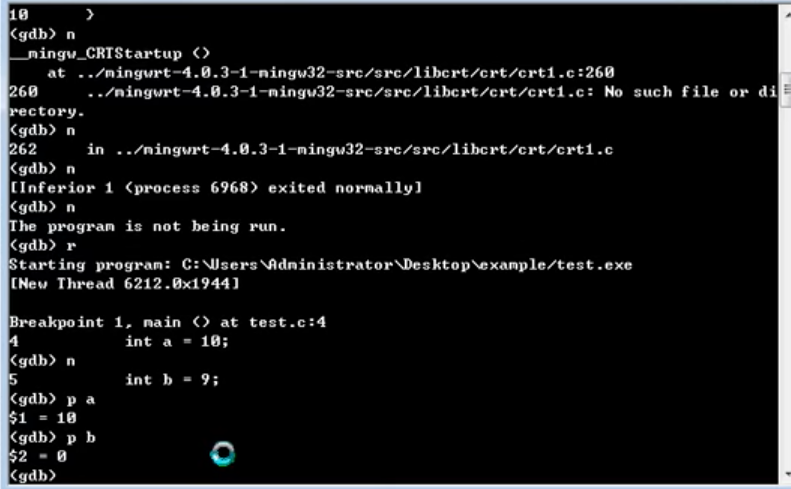


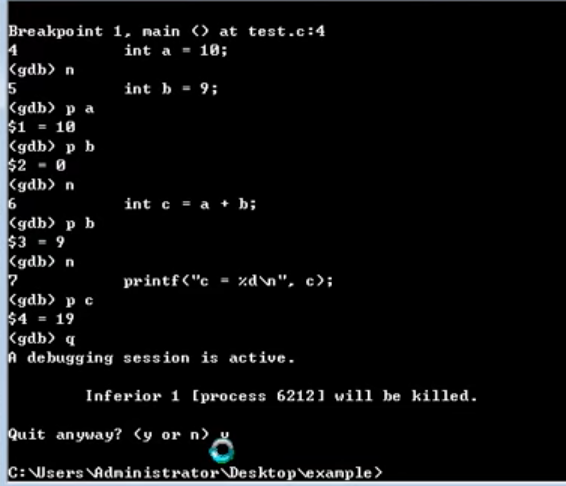
### GDB设置断点 b



### GDB运行命令 r







### 循环调试

