3 C++ Compilation (Makefiles)

我们可以在 working directory 中输入指令来用 g++ compiler 手动制作 `.exe` 文件.

```
g++ -Wall -Werror -pedantic -g --std=c++17 -Wno-sign-compare -Wno-comment main.cpp -o
main.exe
```

但是如果程序的文件关系比较多，先后编译顺序手写就很麻烦. 因而我们把指令写进一个 `Makefile` 文件来 automate 这个过程.

`make` 指令会读取为 `Makefile` 文件，根据里面的指令来制作 `.exe` 文件.

280并不要求手搓 Makefile, 只需要能看懂 Makefile 并修改其中的一些命令依赖就可以了. 以后再学习 Makefile 文件的编写以及通过 Cmake 工具生成 Makefile 文件.

3.1 MakeFile 语法规则

我们知道，C/C++ Source Code 首先会生成 obj 文件 (`.o`), 然后 `.o` 文件会被编译成 `.exe` 文件.

```
target... : prerequisite...
  recipe1...
  (command1...
  command2...)
```

1. **target (目标文件)**: 本条规则要生成的文件, 比如 `main.exe`. 如果 target 文件的更新时间晚于 prerequisite 的更新时间，则说明 prerequisite 没有改动，target 不需要重新编译; 否则会进行重新编译并更新target.
2. **prerequisite (依赖文件)**: used as input to create the target, 比如 `main.cpp`. 即目标文件由哪些文件生成.
3. **recipe (配方)**：由 prerequisite 生成 target 的 commands, 比如 `g++ main.cpp -o main.exe`. **注意每条命令要换行，且每条命令之前必须有一个tab保持缩进，这是语法要求 (有一些编辑工具默认tab为4个空格，会造成Makefile语法错误)**.

> 具体例子：
>
> ```
> main.exe: main.cpp
>   g++ main.cpp -o main.exe
> ```

3.2 Variable 变量

一个 variable 可以帮助减少 code duplication 从而起到简化作用. 比如要生成多个 `.exe` 文件，于是我们给编译命令 (`g++` 编译器和它的flag) 用 variables (`CXX` 和 `CXXFLAGS`) 来指代.

```
CXX ?= g++
CXXFLAGS ?= -Wall -Werror -pedantic -g --std=c++17 -Wno-sign-compare -Wno-comment

main.exe: main.cpp
  $(CXX) $(CXXFLAGS) main.cpp -o main.exe

unit_tests.exe: unit_tests.cpp unit.cpp

  $(CXX) $(CXXFLAGS) unit_tests.cpp unit.cpp -o unit_tests.exe
```

这些变量名都是标准的. 至于格式和其他东西这里先不学了，参见 GNU make 官方文档 https://www.gnu.org/software/make/manual/make.html#Makefile-Names 以及中译 https://file.elecfans.com/web1/M00/7D/E7/o4YBAFwQthSADYCWAAT9Q1w_4U0711.pdf

变量‘CFLAGS’的值为‘-g’，则您可将‘-g’选项传递给每个编译器。所有的隐含规则编译C程序时都使用‘$CC’获得编译器的名称，并且都在传递给编译器的参数中都包含‘$(CFLAGS)’。
隐含规则使用的变量可分为两类：一类是程序名变量（象cc），另一类是包含程序运行参数的变量（象CFLAGS）。(‘程序名’可能也包含一些命令参数，但是它必须以一个实际可以执行的程序名开始。) 如果一个变量值中包含多个参数，它们之间用空格隔开。
这里是内建规则中程序名变量列表：

AR
: 档案管理程序；缺省为：‘ar'.

AS
: 汇编编译程序；缺省为：‘as'.

CC
: C语言编译程序；缺省为：‘cc'.

CXX
: C++编译程序；缺省为：‘g++'.

CO
: 从RCS文件中解压缩抽取文件程序；缺省为：‘co'.

CPP
: 带有标准输出的C语言预处理程序；缺省为：‘$(CC) -E'.

FC
: Fortran 以及 Ratfor 语言的编译和预处理程序；缺省为：‘f77'.

GET
: 从SCCS文件中解压缩抽取文件程序；缺省为：‘get'.

LEX
: 将 Lex 语言转变为 C 或 Ratfor程序的程序；缺省为：‘lex'.

PC
: Pascal 程序编译程序；缺省为：‘pc'.

YACC
: 将 Yacc语言转变为 C程序的程序；缺省为：‘yacc'.

YACCR
: 将 Yacc语言转变为 Ratfor程序的程序；缺省为：‘yacc -r'.

MAKEINFO
: 将Texinfo 源文件转换为信息文件的程序；缺省为：‘makeinfo'.

TEX
: 从TeX源产生TeX DVI文件的程序；缺省为：‘tex'.

TEXI2DVI
: 从Texinfo源产生TeX DVI 文件的程序；缺省为：‘texi2dvi'.

这里是值为上述程序附加参数的变量列表。在没有注明的情况下，所有变量的值为空值。

ARFLAGS
: 用于档案管理程序的标志，缺省为：‘rv'.

ASFLAGS
: 用于汇编编译器的额外标志 (当具体调用‘.s'或‘.S'文件时)。

CFLAGS
: 用于C编译器的额外标志。

CXXFLAGS
: 用于C++编译器的额外标志。

COFLAGS
: 用于RCS co程序的额外标志。

CPPFLAGS
: 用于C预处理以及使用它的程序的额外标志 (C和 Fortran 编译器)。

FFLAGS
: 用于Fortran编译器的额外标志。

GFLAGS
: 用于SCCS get程序的额外标志。

LDFLAGS
: 用于调用linker（‘ld’）的编译器的额外标志。

LFLAGS
: 用于Lex的额外标志。

PFLAGS
: 用于Pascal编译器的额外标志。

RFLAGS
: 用于处理Ratfor程序的Fortran编译器的额外标志。

YFLAGS
: 用于Yacc的额外标志。Yacc。

3.3 Example

3.3.1 Source files

(1) `main.cpp`:

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

int main() {
  cout << "Hello World!\n";
}
```

(2) `main_test.in`:

```
this program doesn't read stdin,
so this text doesn't matter
```

(3) `main_test.out.correct`:

```
Hello World!
```

3.3.2 command line compilation

我们可以写 command 手动用 g++ 编译这段.

```
g++ -Wall -Werror -pedantic -g --std=c++17 -Wno-sign-compare -Wno-comment main.cpp -o
main.exe
./main.exe
# Hello World!
rm main.exe

./main.exe < main_test.in > main_test.out
diff main_test.out main_test.out.correct
# no output means the file matches
```

3.3.3 `Makefile` compilation

而我们也可以创建一个 Makefile 文件来编译.

(1) 代码结构:

```
tree
.
├── Makefile
└── main.cpp
```

(2) `Makefile`:

```
CXX ?= g++
CXXFLAGS ?= -Wall -Werror -pedantic -g --std=c++17 -Wno-sign-compare -Wno-comment

# Run regression test
test: main.exe
  ./main.exe < main_test.in > main_test.out
  diff main_test.out main_test.out.correct
  echo PASS

# Compile the main executable
main.exe: main.cpp
```

```
  $(CXX) $(CXXFLAGS) main.cpp -o main.exe

# Remove automatically generated files
clean :
  rm -rvf *.exe *~ *.out *.dSYM *.stackdump
```

(3) command:

```
make main.exe #(1) make exe using makefile
./main.exe  #(2) run exe
make test #(3) run a test
make clean  #(4) clean the generated files
```

```
[qiulin] /mnt/c/Users/19680/Desktop/make-example/ $ make main.exe
g++ -Wall -Werror -pedantic -g --std=c++17 -Wno-sign-compare -Wno-comment main.cpp -o main.exe
[qiulin] /mnt/c/Users/19680/Desktop/make-example/ $ ./main.exe
Hello World!
```