ESCREVENDO UM MAKEFILE PARA MULTI-PLATAFORMAS

1 Introdução

Um *Makefile* é um arquivo que contém um conjunto de regras a ser executado pelo software *Make*. O *Make*, por sua vez, é um utilitário feito para automatizar o processo de compilação de programas e bibliotecas. Vale ressaltar que uma característica interessante do uso dessa ferramenta é que, quando recompilando um projeto que teve só algumas partes modificadas, é possível identificar essas partes para que só elas e suas dependências sejam recompiladas.

Desta forma, o objetivo desse documento é ensinar os alunos a escreverem um arquivo *Makefile* que facilite o processo de compilação dos trabalhos a serem desenvolvidos na disciplina Introdução ao Desenvolvimento de jogos, tanto em sistemas operacionais Windows quanto em sistemas operacionais baseados em Unix (como Linux, Mac OS, etc.).

Além disso, ressalta-se que a implementação de um *Makefile* multi-plataforma bem como a padronização dos diretórios dos trabalhos (mais detalhes a seguir) são de caráter **obrigatório** para a disciplina.

2 Pré-requisitos

Dado que os projetos da disciplina terão como dependência as bibliotecas do SDL2, para executar o *Make* é necessário seguir os tutoriais de instalação disponíveis na plataforma Aprender (tanto para o Linux quanto para o Windows). Ressalta-se que no tutorial de instalação para o Windows o passo dado como opcional na seção "TDM-GCC" passa a ser obrigatório.

Em especial, para o Mac OS será necessário baixar os arquivos .framework da biblioteca SDL2 (SDL2, SDL2_image, SDL2_mixer e SDL2_ttf) e colocá-los no caminho: `Machintosh HD/Library/Framworks`.

Em adição, tenha certeza de que o software *Make* está instalado no seu sistema operacional. No caso do Windows, a instalação do TDM-GCC feita no tutorial de instalação já inclui o *Make*, que deve ser executado no cmd (comando: `mingw32-make`). No caso de algumas distribuições do Linux, o *Make* já vem no pacote essencial de desenvolvimento (comando: `make`), mas recomenda-se conferir.

E, para testar o executável gerado pelo *Make*, tenha certeza que o executável tem acesso as bibliotecas do SDL2. No caso do Linux, isso já foi automaticamente feito ao seguir o tutorial de instalação. No caso do Windows, os arquivos *.dll* devem estar ou no mesmo diretório do executável ou no *Working Directory*.

Por fim, observa-se que é **obrigatório** seguir a hierarquia de diretórios especificada a seguir, tanto para o *Makefile* funcionar, quanto para fins de padronização das correções dos trabalhos:

```
<Matrícula>_T<Número do trabalho>
├── include
│   ├── <header>.hpp
│   └── ...
├── src
│   ├── Main.cpp
│   ├── <source>.cpp
│   └── ...
├── assets
│   ├── img
│   │   ├── <image1>.png
│   │   ├── <image2>.jpg
│   │   └── ...
│   ├── font
│   │   ├── <font>.ttf
│   │   └── ...
│   └── audio
│       ├── <audio1>.wav
│       ├── <audio2>.ogg
│       └── ...
└── Makefile
```

Caso deseje colocar os arquivos de source e include da engine numa pasta separada, é possível mas ficará sob sua responsabilidade de adaptar esse makefile para essa estrutura.

3 Variáveis

Uma variável é um nome definido no *Makefile* para representar uma *String*, de modo que o valor da variável é substituído explicitamente onde a variável é declarada. Desta forma, é uma boa prática usar variáveis pois é possível modificar os seus valores sem ter que diretamente mexer nas regras do arquivo.

No *Makefile* a ser desenvolvido, pode-se agrupar as variáveis a serem utilizadas de acordo com o que o seu valor representa:

- Comandos: COMPILER, RMDIR e RM;
- *Flags*: DEP_FLAGS, LIBS, INC_PATHS, FLAGS, DFLAGS e RFLAGS;
- Caminhos: INC_PATH, SRC_PATH, BIN_PATH, DEP_PATH e SDL_PATHS;
- Listas de arquivos: CPP_FILES, INC_FILES, FILE_NAMES, DEP_FILES e OBJ_FILES;
- Arquivo: EXEC.

Nota-se que o valor de algumas variáveis depende do sistema operacional no qual o *Make* está sendo utilizado. Para detectar automaticamente qual o sistema operacional, pode-se usar comandos condicionais para checar o valor de uma variável. No caso, é utilizada a variável OS para detectar se o SO é Windows e a variável UNAME_S para detectar se o SO é Mac. Ademais, assume-se que o SO é Linux.

Por fim, dado que uma variável está declarada no ambiente, pode-se referenciar/chamar ela em qualquer parte do *Makefile*. A forma padrão de fazer isso é: `$(<nome da variável>)`. Mais detalhes a respeito de variáveis poderão ser vistos na seção Regras e na seção Extras.

```
1  #------------------------------------------------------------------------
2  # Assume-se uma distribuição Linux como sistema operacional padrão
3  #------------------------------------------------------------------------
4
5  # O compilador
6  COMPILER = g++
7  # Comando para remover pastas
8  RMDIR = rm -rdf
9  # Comando para remover arquivos
10 RM = rm -f
11
12 # "Flags" para geração automática de dependências
13 DEP_FLAGS = -M -MT $@ -MT $(BIN_PATH)/$(*F).o -MP -MF $@
14 # Bibliotecas a serem linkadas
15 LIBS = -lSDL2 -lSDL2_image -lSDL2_mixer -lSDL2_ttf -lm
16 # Caminho dos includes
17 INC_PATHS = -I$(INC_PATH) $(addprefix -I,$(SDL_INC_PATH))
18
19 # Diretivas de compilacao
20 FLAGS = -std=c++11 -Wall -pedantic -Wextra -Wno-unused-parameter -Werror=init-self
21 # Diretivas extras para debug
22 DFLAGS = -ggdb -O0 -DDEBUG
23 # Diretivas extras para release
24 RFLAGS = -O3 -mtune=native
25
26 INC_PATH = include
27 SRC_PATH = src
28 BIN_PATH = bin
29 DEP_PATH = dep
30
31 # Uma lista de arquivos por extensão:
32 CPP_FILES = $(wildcard $(SRC_PATH)/*.cpp)
33 INC_FILES = $(wildcard $(INC_PATH)/*.h)
34 FILE_NAMES = $(sort $(notdir $(CPP_FILES:.cpp=)) $(notdir $(INC_FILES:.h=)))
35 DEP_FILES = $(addprefix $(DEP_PATH)/,$(addsuffix .d,$(FILE_NAMES)))
36 OBJ_FILES = $(addprefix $(BIN_PATH)/,$(notdir $(CPP_FILES:.cpp=.o)))
37
38 # Nome do executável
39 EXEC = JOGO
40
```

Figura 1: Definição geral de variáveis (assumindo que o SO é Linux)

É possível perceber que existem algumas variáveis que são usadas antes de serem declaradas mas o make ainda funciona. Isso acontece porque as variáveis podem ter seus valores avaliados em dois momentos diferentes. Quando se usa um igual simples ("="), o valor da variável só é avaliado quando a mesma é utilizada. Quando se usa dois pontos e igual (":="), ela é avaliada imediatamente. Por isso SDL_INC_PATH pode ser usado (linha 17) antes de ser declarado (linha 56) e LIBS não dá loop infinito (linha 60). Se uma variável não for encontrada, ela é substituida por espaço vazio.

```
40
41 #------------------------------------------------------------------
42 # Caso o sistema seja windows
43 #------------------------------------------------------------------
44 ifeq ($(OS),Windows_NT)
45
46 # Comando para remover um diretório recursivamente
47 RMDIR = rd /s /q
48
49 # Comando para deletar um único arquivo
50 RM = del /q
51
52 # Possíveis Path da SDL. Caso seja possível ter mais de um local, adicione com espaço entre eles
53 # Por ex.: SDL_PATHs = C:/SDL2 D:/Tools/SDL2 C:/dev-tools/SDL2
54 SDL_PATHS = C:/SDL2/x86_64-w64-mingw32 C:/Tools/msys64/mingw64
55
56 SDL_INC_PATH += $(addsuffix /include,$(SDL_PATHS))
57 LINK_PATH = $(addprefix -L,$(addsuffix /lib,$(SDL_PATHS)))
58 FLAGS += -mwindows
59 DFLAGS += -mconsole
60 LIBS := -lmingw32 -lSDL2main $(LIBS)
61
62 # Nome do executável
63 EXEC := $(EXEC).exe
64
65 else
66
67 UNAME_S := $(shell uname -s)
68
69 #------------------------------------------------------------------
70 # Caso o sistema seja Mac
71 #------------------------------------------------------------------
72
73 ifeq ($(UNAME_S), Darwin)
74
75 LIBS = -lm -framework SDL2 -framework SDL2_image -framework SDL2_mixer -framework SDL2_ttf
76
77 endif
78 endif
79
```

Figura 2: Redefinição de variáveis caso o SO seja Windows ou Mac

4 Regras

Para a compilação de um determinado código, existem um conjunto de regras a serem seguidas. Cada regra é descrita como um conjunto de passos a ser executado em ordem, o que é chamado de receita. O *Make* usa receitas escritas no arquivo *Makefile* como instruções para compilação.

A sintáxe de uma receita em um arquivo *Makefile* é a seguinte:

```
<regra>: <dependências>
[TAB]    <comando>
[TAB]    <comando>
            ...
```

Assim, o que é passado após o comando do *Make* é a regra que será seguida. Por padrão, se nada for passado após o comando do *Make* é a regra `all` que será seguida.

A figura abaixo mostra que a regra `all` tem como dependência a existência de um executável de nome contido na variável `EXEC`. Se esse executável não existir, o *Make* procurá se existe uma regra para criação do mesmo, que, nesse caso, é a regra subsequente. A regra de criação do EXEC tem como dependências todos os objetos de compilação do projeto. Se essas dependências forem cumpridas, a regra de criação do EXEC executará o comando que o

```
84
85 # Regra geral
86 all: $(EXEC)
87
88 # Gera o executável
89 $(EXEC): $(OBJ_FILES)
90     $(COMPILER) -o $@ $^ $(LINK_PATH) $(LIBS) $(FLAGS)
91
```

Figura 3: Receita da regra `all` e receita da regra que gera sua dependência

cria, que, nesse caso, usa o compilador especificado e as dependências do projeto para *linkar* os objetos de modo a gerar o executável final. Para isso, esse comando usa duas formas de referenciar variáveis além da padrão:

- `$@` : Referencia o nome da regra.
- `$^` : Referencia as dependências da regra.

Em especial, existem regras chamadas "Regras de inferência". Elas generalizam o processo de compilação de modo que não é necessário especificar o nome da regra explicitamente para cada ocorrência. Essas regras são distinguíveis pelo uso de caracteres especiais, como o "%". Na figura, mostra-se a regra de criação dos objetos e a regra de criação de dependências.

```
91
92 # Gera os arquivos objetos
93 $(BIN_PATH)/%.o: $(DEP_PATH)/%.d | folders
94     $(COMPILER) $(INC_PATHS) $(addprefix $(SRC_PATH)/,$(notdir $(<:.d=.cpp))) -c $(FLAGS) -o $@
95
96 # Gera os arquivos de dependencia
97 $(DEP_PATH)/%.d: $(SRC_PATH)/%.cpp | folders
98     $(COMPILER) $(INC_PATHS) $< $(DEP_FLAGS) $(FLAGS)
99
```

Figura 4: Receita da regra de inferência de criação dos objetos e da de geração de dependências

Explicação do operador pipe ("|") nas dependências da receita caso você seja curioso.

Ademais, é necessário que se especifique no *Makefile* não só como gerar os arquivos, mas que os mesmos sejam incluídos para que suas regras de dependências sejam aplicadas. Para isso, usa-se o comando `-include` (como mostrado na figura 5). Observe a linha que estão no makefile.

```
138 .SECONDEXPANSION:
139 -include $$(DEP_FILES)
```

Figura 5: Comando para incluir as dependências de objetos

Por fim, adiciona-se a regra `clean`, regra padrão em *Makefiles*. Ela possui uma receita simples que tem por objetivo limpar todos os arquivos gerados pelo *Make*.

```
100 clean:
101     -$(RMDIR) $(DEP_PATH)
102     -$(RMDIR) $(BIN_PATH)
103     -$(RM) $(EXEC)
104
```

Figura 6: Receita da regra clean

5 Extras

5.1 Regras especiais padrões do *Make*

A seguir, três regras especiais interessantes são explicadas através de exemplos:

- `.PHONY` : Impede conflitos de arquivos com mesmo nome do declarado por PHONY
- `.PRECIOUS` : Estabelece que dados arquivos são preciosos, ou seja, que devem tentar ser preservados a todo custo (mesmo que o *Make* seja interrompido ou encerrado)
- `.SECONDEXPANSION` (figura 5): Faz com que todas regras abaixo desse comando sejam expandidos uma segunda vez.

```
79
80 ###############################################################################################
81
82 .PRECIOUS: $(DEP_FILES)
83 .PHONY: release debug clean folders help
84
```

Figura 7: Exemplos de algumas regras especiais interessantes para se adicionar ao *Makefile*

Para mais informações a respeito de regras especiais padrões do *Make*, acessar a documentação do GNU que fala a respeito de *Special Built-in Target Names*.

5.2 Outras regras interessantes

Tanto para gerar executáveis mais otimizados, quanto para fins de *depuração*, as receitas de regras mostradas na figura 8 podem ser úteis e interessantes de incluir no *Makefile* da disciplina:

- `release` : Regra que inclui duas diretivas nas compilações com otimização dos objetos:
  - `O3` : Gera o executável com otimização do tipo 3, voltada para eficiência em tempo de execução.
  - `mtune=native` : Caso o processador do computador possua otimizações próprias, executa elas.
- `debug` : Regra que inclui duas diretivas nas compilações dos objetos:
  - `ggdb` : Produz informação de *debug* para ser usada no compilador GDB.
  - `O0` : Garante que nenhuma otimização será feita pelo compilador, garantindo um *debug* mais preciso.

  - `DDEBUG` : Define uma macro DEBUG.

- `folders` : Regra que gera as pastas do projeto.
- `print-%` : Regra interessante para *depuração* do Makefile. Para usá-la deve-se chamar *Make* passando `print-<nome de uma variável do Makefile>`. Ela auxilia no entendimento de como variáveis funcionam e ajuda a depurar construções mais complexas de valores de variáveis.
- `help` : Regra que imprime um menu de ajuda.

```
104
105 release: FLAGS += $(RFLAGS)
106 release: $(EXEC)
107
108 debug: FLAGS += $(DFLAGS)
109 debug: $(EXEC)
110
111 folders:
112 ifeq ($(OS), Windows_NT)
113     @if NOT exist $(DEP_PATH) ( mkdir $(DEP_PATH) )
114     @if NOT exist $(BIN_PATH) ( mkdir $(BIN_PATH) )
115     @if NOT exist $(INC_PATH) ( mkdir $(INC_PATH) )
116     @if NOT exist $(SRC_PATH) ( mkdir $(SRC_PATH) )
117 else
118     @mkdir -p $(DEP_PATH) $(BIN_PATH) $(INC_PATH) $(SRC_PATH)
119 endif
120
121 # Regra pra debug
122 print-% : ; @echo $* = $($*)
123
124 help:
125 ifeq ($(OS), Windows_NT)
126     @echo.
127 endif
128     @echo Available targets:
129     @echo - release:  Builds the release version
130     @echo - debug:    Builds the debug version
131     @echo - clean:    Cleans generated files
132     @echo - folders:  Generates project directories
133     @echo - help:     Shows this help
134 ifeq ($(OS), Windows_NT)
135     @echo.
136 endif
137
```

Figura 8: Exemplos de regras interessantes para se incluir no *Makefile*

5.3 Funcões para nome de arquivos

Na declaração de variáveis do Makefile produzido (vide seção Variáveis), algumas dessas Funções foram utilizadas, sendo elas:

- `wildcard` : Define um padrão e faz uma lista de arquivos (composta de arquivos que obteram casamento de padrão) que se expande no momento que a variável é utilizada.
- `addprefix`: Um prefixo é concatenado ao nome dos arquivos sequentes.
- `notdir`: Retira a parte do nome do arquivo referente ao caminho de diretórios.

5.4 Dicas finais

- Se você está usando o git para versionar seus trabalhos, não se esqueça de incluir no seu .gitignore os arquivos e diretórios criados pelo Makefile (Exemplo: `bin/ dep/ <nome do executável>`). Se não estiver usando, recomenda-se que você comece a usar!
- Se você quer aprender mais a respeito do *Make*, recomenda-se a documentação do GNU.