

Leitura 1 - compilação com gcc e make

Igor

Rafael

Hashimoto

20-02-2018

Na última aula (19/02) fizemos atividades usando um compilador via browser. Esta leitura visa familiarizá-lo com as ferramentas de compilação de código em *C* e com a organização de projetos usando *Make*. Recomendamos que executem todos os comandos apresentados no guia para verificar seu funcionamento.

Instalação local das ferramentas dos cursos

Adotaremos o Ubuntu Linux como sistema padrão do mutirão para aprender *C*. Vocês podem realizar todas as atividades em outros sistemas, mas os guias serão feitos especificamente para Ubuntu.

As ferramentas que precisamos para o mutirão estão disponíveis nos pacotes **build-essential** e **gdb**. Também é recomendado instalar os pacotes **valgrind** e **kcachegrind** (também via *apt*) e o *voltron*, uma interface visual para o *gdb*.

```
sudo apt install build-essential gdb valgrind kcachegrind
```

Praticamente qualquer editor de texto ou IDE suporta colorização para *C*, porém recomendamos utilizar alguma ferramenta que também corrija erros de sintaxe. Editores como Visual Studio Code (com plugin para C/C++ e Make) e Eclipse CDT são boas escolhas por serem multi-plataforma e suficientemente completos.

Após o mutirão as atividades de *Computação Embarcada* serão feitas em Windows usando Atmel Studio. Desafios de programação e Sistemas de Hardware e Software continuarão usando Ubuntu Linux.

Compilação (simples) via linha de comando

Usamos o **gcc** para compilar programas em *C*. Para diminuir a quantidade de erros nos programas podemos passar como argumento algumas flags para exibir erros comuns e para fazer somente otimizações que não atrapalhem debug (**-Og**).

```
gcc -Wall -pedantic -std=c99 -Og arquivo.c -o executavel
```

Usando este comando podemos compilar um programa definido em um único arquivo *.c*.

Tarefa: compile o arquivo *printf.c* usado na aula passada no terminal e nomeie o executável *exemplo0*. Rode ele e verifique que tudo continua funcionando.

Organizando o código em vários arquivos

Conforme projetos ficam maiores precisamos dividir o código em vários arquivos. Formalmente, *C* não possui o conceito de módulo ou pacotes, como Java ou Python, porém podemos fazer esta divisão escrevendo nossos códigos em arquivos *.c* e criando arquivos cabeçalho (*headers*) com extensão *.h*.

No arquivo *.h* colocamos as definições de todas as funções que queremos exportar. Ou seja, todas as funções que serão usadas em outros arquivos. Quando definimos uma função não a implementamos, somente declaramos seu tipo de retorno e argumentos. Exemplo:

```
double media_do_vetor(double vetor[], int tam);
```

No arquivo *.c* fazemos a implementação das funções definidas no *header* correspondente, além de outras funções que não serão exportadas.

Usamos a diretiva `#include` para incluir nossos cabeçalhos em outros arquivos *.c* – da mesma maneira que incluímos a biblioteca padrão (`stdio.h`, `stdlib.h`, etc).

Veja o conteúdo dos arquivos da pasta *exemplo1*. Temos um arquivo *main.c* contendo a função principal do programa e arquivos *modulo1.c/h* contendo funções auxiliares.

Pesquisa: O uso das diretivas `#ifndef` e `#define` como visto em *modulo1.h* é chamado de *ifdef guards*. Pesquise para quê servem e como usá-las.

Podemos compilar este programa usando o seguinte comando.

```
gcc -Wall -pedantic -std=c99 -Og main.c modulo1.c -o  
main-exemplo1
```

Apesar de funcionar, compilar desta maneira é ineficiente pois todos os arquivos são recompilados mesmo que só façamos modificações em um deles.

Uma maneira melhor de realizar este processo é dividir a compilação em duas fases: na primeira compilamos individualmente cada arquivo e na segunda juntamos todos os resultados em um executável.

Para compilar um arquivo sem gerar executável usamos a diretiva `-c`.

```
gcc -Wall -pedantic -std=c99 -Og -c main.c      gcc -Wall  
-pedantic -std=c99 -Og -c modulo1.c
```

Estes comandos irão gerar dois arquivos com extensão *.o*. Juntamos os resultados gerados usando o seguinte comando.

```
gcc main.o modulo1.o -o exemplo1-main
```

Desta maneira se mexermos no arquivo *main.c* precisamos recompilá-lo de novo e depois gerar o executável. Como não modificamos *modulo1.c*, o arquivo *modulo1.o* já está atualizado. Em projetos grandes o tempo de compilação salvo é considerável, agilizando significativamente o desenvolvimento.

Projetos usando Make

Ficar lembrando qual arquivo já foi compilado e qual precisa ser atualizado não é viável. O *Make* é um programa usado para gerenciar dependências de compilação deste tipo e é muito usado em C/C++ para gerenciar projetos. Vamos primeiro compilar o projeto que usamos na seção anterior. Basta entrar no diretório *exemplo1* e executar

```
make
```

Um executável *exemplo1-make* deve ter sido criado. Ao executá-lo obtemos o mesmo resultado que com o executável criado manualmente na seção anterior.

Um projeto usando *Make* é definido por um arquivo *Makefile* que define diversas regras (*rules*), cada uma representando um arquivo *target* a ser compilado/gerado por uma sequência de comandos listados dentro de seu escopo. Cada regra tem um conjunto de pré-requisitos, que são *targets* que precisam ser executados antes de sua execução. A sintaxe de um *Makefile* é bem simples:

```
target_name: dependencia1 dependencia2
(tab) comando1
(tab) comando2
(tab) ...
```

Vejamos abaixo o *Makefile* que usamos para compilar o exemplo da seção anterior.

```
CFLAGS = -Wall -pedantic -Og -std=c99
```

```
main.o: main.c modulo1.h
gcc -c $(CFLAGS) main.c
```

```
modulo1.o: modulo1.c modulo1.h
gcc -c $(CFLAGS) modulo1.c
```

```
executavel: main.o modulo1.o
gcc main.o modulo1.o -o exemplo1-make
```

Definimos três regras: *main.o*, *modulo1.o* e *exemplo1-make*. As regras *main.o* e *modulo1.o* dependem, respectivamente, dos arquivos *main.c* e *modulo1.c/h*. A regra *exemplo1-make* depende de *main.o* e *modulo1.o*, que são produtos intermediários. Modificações em um pré-requisito (intermediário ou não) resultam

na reexecução dos comandos da regra correspondente. Vejamos um exemplo: o projeto já foi compilado uma vez e agora estamos corrigindo um bug no arquivo *main.c*. Ao modificarmos este arquivo e rodarmos **make** será executado

1. o target *main.o*, que compila o arquivo *main.c* e depende explicitamente de *main.c*;
2. o target *main*, que depende de *main.o*, que foi modificado por depender de *main.c*

Juntando tudo

A pasta *exemplo2* contém alguns arquivos *.c* e *.h* que gostaríamos de compilar usando **make**. Você tem as seguintes tarefas:

1. Identifique quais arquivos resultam na criação de um executável e quais são apenas definições de funções auxiliares.
2. Compile os programas na mão e corrija os erros, se houverem. Não se esqueça de checar se os programas funcionam corretamente.
3. Crie um Makefile para o projeto.

Dica: o arquivo *util.c* usa a biblioteca *math* de *C*. Se você estiver com dificuldades de encontrar como usar esta biblioteca busque por *gcc undefined reference to sqrt*.