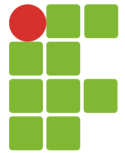
# Introdução à ferramenta GNU Make

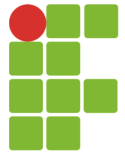
## Programação de Computadores 1 - Ciência da Computação


Prof. Daniel Saad Nogueira Nunes

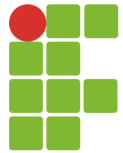
IFB – Instituto Federal de Brasília,
Campus Taguatinga



# Sumário

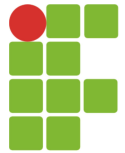


# Sumário



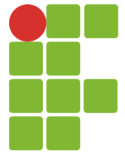
# Introdução

- A ferramenta `make`, presente no ecossistema GNU, é extremamente relevante para ser utilizada em projetos.
- Ela é capaz de determinar quais as partes do sistema que precisam ser compiladas e realiza a compilação propriamente dita.
- Além de compilar códigos para construção do sistema, o `make` pode emitir diversos outros comandos (*e.g.* limpar arquivos objetos), como um *script*.
- Ela pode ser integrada a IDEs, permitindo instruir a IDE o que deve ser feito na construção do sistema.
- É essencial a utilização do `make` para disponibilização de códigos a terceiros.

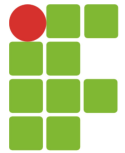


# Introdução

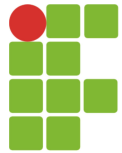
- Para aprender a utilizar a ferramenta, iremos adotar uma abordagem baseada em exemplo.
- Utilizaremos os códigos da nossa biblioteca de operações em vetores.



# Sumário

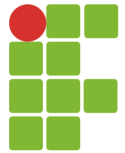


# Sumário

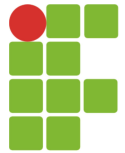


# Makefile

- A ferramenta `make`, quando invocada, executará as **regras** presentes no arquivo chamado `Makefile`.
- Ela recompilará todos os arquivos modificados presentes nas regras para construção do artefato computacional.
- Poupa tempo ao não realizar um retrabalho para os arquivos já compilados na sua última versão.

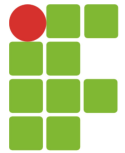


# Sumário



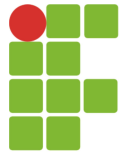
# Sintaxe Básica

- A sintaxe básica de uma regra consiste de:
  - Nome da regra ou objetivo;
  - Pré-requisitos;
  - Receita.
- O nome da regra é separado pelos pré-requisitos pelo símbolo de dois pontos, enquanto os pré-requisitos são separados entre si através dos espaços.
- A receita pode ter várias linhas, instruindo comandos à ferramenta `make`. Estas linhas devem iniciar com uma tabulação (TAB).



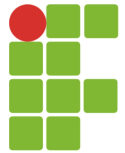
# Sintaxe Básica

```
1 target: prerequisite_1 prerequisite_2 ... prerequisite_n
2     recipe
```



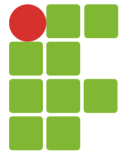
# Sintaxe Básica

- Adaptando o arquivo `Makefile` ao nosso código, temos a seguinte configuração.



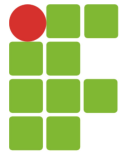
# Sintaxe Básica

```
1  main: main.o leitura.o escrita.o operacao.o
2      cc -Wall main.o leitura.o escrita.o operacao.o -o main
3
4  main.o: main.c
5      cc -c -Wall main.c
6
7  leitura.o: leitura.c leitura.h
8      cc -c -Wall leitura.c
9
10 escrita.o: escrita.c escrita.h
11     cc -c -Wall escrita.c
12
13 operacao.o: operacao.c operacao.h
14     cc -c -Wall operacao.c
15
16 clean:
17     rm main main.o leitura.o escrita.o operacao.o
```



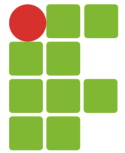
# Sintaxe Básica

- Para gerar o executável `main`, só é necessário usar o comando `make`.
- No caso, o objetivo `main` depende de todos os arquivos objetos, os quais dependem de seus respectivos arquivos `.c` e `.h`.
- Ele só será construído após a compilação dos arquivos fonte.
- Cada arquivo fonte só é compilado se os arquivos objetos estão desatualizados.
- O `make` cuida da resolução de dependências automaticamente.



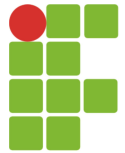
# Sintaxe Básica

- A regra com o objetivo `clean` não possui nenhum pré-requisito.
- Sua função é apenas limpar os arquivos objetos e o executável gerado.
- Para invocá-la é necessário apenas utilizar o comando `make clean`.

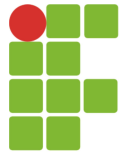


# Sintaxe Básica

- O comando cc refere-se à uma abreviação do compilador gcc, utilizado no Linux.

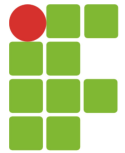


# Sumário

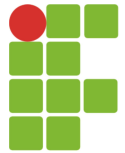


# Processamento

- Ao digitar no terminal o comando `make`, a ferramenta `make` irá ler o `Makefile` disponível e tentar executar as regras presentes.
- A ferramenta começa a execução pela primeira regra no documento. Na presença de pré-requisitos, outras regras são executadas pela ferramenta para completar as dependências.
- Arquivos objetos na sua última versão não necessitam de compilação. Esta gerência é realizada automaticamente pelo `make`.

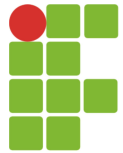


# Sumário



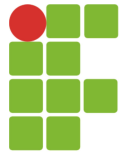
# Variáveis

- O `make` fornece suporte para variáveis.
- A utilização de variáveis permite reduzir a presença de erros relacionados ao ato de copiar e colar, além de deixar o código do `Makefile` mais organizado.
- As atribuições à variáveis são feitas através do operador =, enquanto o acesso ao valor da variável, é feita através do operador `$(nome_da_variavel)`.



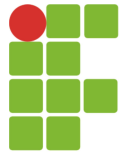
# Sintaxe: Variáveis

```
1  objects =  main.o leitura.o escrita.o operacao.o
2
3  main: $(objects)
4      cc -Wall $(objects) -o main
5
6  main.o: main.c
7      cc -c -Wall main.c
8
9  leitura.o: leitura.c leitura.h
10     cc -c -Wall leitura.c
11
12 escrita.o: escrita.c escrita.h
13     cc -c -Wall escrita.c
14
15 operacao.o: operacao.c operacao.h
16     cc -c -Wall operacao.c
17
18 clean:
19     rm main $(objects)
```



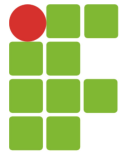
# Variáveis Automáticas

- O `make` fornece uma sintaxe especial para as variáveis automáticas.
- Algumas delas são:
  - `$@`: nome do primeiro objetivo.
  - `$<`: nome do primeiro pré-requisito.
  - `$^`: lista dos pré-requisitos.
- Podemos utilizar esta sintaxe para simplificar o `Makefile`.

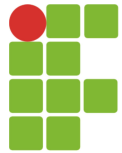


# Sintaxe: Variáveis Automáticas

```
1  OBJ =  main.o leitura.o escrita.o operacao.o
2
3  main: $(OBJ)
4      cc -Wall $^ -o $@
5
6  main.o: main.c
7      cc -c -Wall $<
8
9  leitura.o: leitura.c leitura.h
10     cc -c -Wall $<
11
12 escrita.o: escrita.c escrita.h
13     cc -c -Wall $<
14
15 operacao.o: operacao.c operacao.h
16     cc -c -Wall $<
17
18 clean:
19     rm main $(OBJ)
```

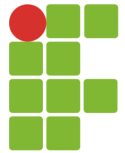


# Sumário



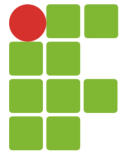
# Dedução Automática

- O `make` consegue deduzir automaticamente que os arquivos objetos dependem do arquivo `.c` do mesmo nome.
- Logo, não é necessário colocar os arquivos `.c` nas dependências e nem a receita, podemos simplificar o `Makefile`.

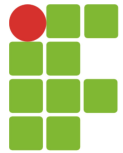


# Sintaxe: Dedução Automática

```
1  OBJ =  main.o leitura.o escrita.o operacao.o
2
3  main: $(OBJ)
4      cc -Wall $(OBJ) -o main
5
6  main.o:
7
8  leitura.o: leitura.h
9
10 escrita.o: escrita.h
11
12 operacao.o: operacao.h
13
14
15 clean:
16     rm main $(OBJ)
```

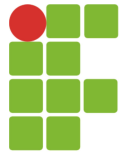


# Sumário



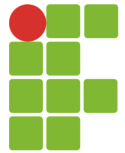
# Limpeza

- Nós já vimos como instruir o make para realizar a limpeza dos arquivos objetos e executável.
- Contudo, podemos ter problemas caso haja um arquivo chamado `clean`, isso pode confundir o `make`.
- Além disso, podemos evitar erros associados ao `rm`, como tentar apagar arquivos que não existem.

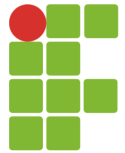


# Sintaxe: Dedução Automática

```
1  OBJ =  main.o leitura.o escrita.o operacao.o
2
3  main: $(OBJ)
4      cc -Wall $(OBJ) -o main
5
6  main.o:
7
8  leitura.o: leitura.h
9
10 escrita.o: escrita.h
11
12 operacao.o: operacao.h
13
14 .PHONY: clean
15
16 clean:
17     rm main $(OBJ)
```

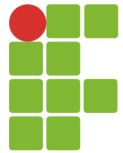


# Sumário



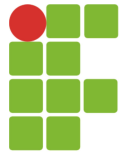
# Wildcards

- As *wildcards* ou coringas permitem que descrevamos uma coleção de arquivos através de uma única expressão.
- Podemos por exemplo selecionar todos os arquivos `.c` ou `.o` com uma única expressão.
- Exemplo: `$(wildcard *.c)` nos fornece a lista dos arquivos fonte.
- Podemos simplificar imensamente o nosso `Makefile` utilizando este mecanismo.



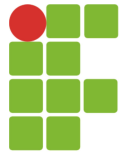
# Sintaxe: Wildcards

```
1  SRC = $(wildcard *.c)
2  OBJ =  $(SRC:.c=.o) # realiza substituição dos nomes .c para os .o
3
4  main: $(OBJ)
5      cc -Wall $(OBJ) -o $@
6
7  main.o:
8
9  leitura.o: leitura.h
10
11 escrita.o: escrita.h
12
13 operacao.o: operacao.h
14
15 .PHONY: clean
16
17 clean:
18     rm main $(OBJ)
```



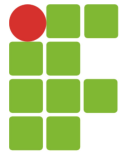
# Sintaxe: Pattern Rules

- Através de regras especiais, podemos simplificar bastante o makefile.
- A regra `%.o: %.c %.h` indica que existe uma regra em que cada arquivo objeto possui o arquivo fonte e o arquivo cabeçalho de mesmo nome como pré-requisitos.

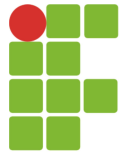


# Sintaxe: Wildcards

```
1  SRC = $(wildcard *.c)
2  OBJ =  $(SRC:.c=.o) # realiza substituição dos nomes .c para os .o
3
4  main: $(OBJ)
5      cc -Wall $(OBJ) -o $@
6
7  %.o: %.c %.h
8      cc -Wall -c $<
9
10 .PHONY: clean
11
12 clean:
13     rm main $(OBJ)
```

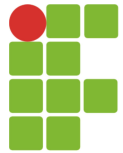


# Sumário



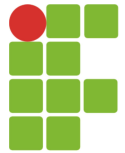
# Valores Padrão

- Por padrão, o `make` utiliza algumas variáveis com valores padrão, as quais podem ser modificadas para se adequarem ao projeto.
- Algumas delas são:
- `CC`: indica o compilador, o valor padrão é `cc`.
- `CFLAGS`: indica as flags utilizadas na compilação. O valor padrão é vazio.
- `LDFLAGS`: indica as flags utilizadas na ligação. O valor padrão é vazio.

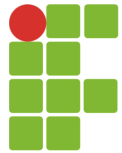


# Sintaxe: Wildcards

```
1  CFLAGS = -Wall
2  CC = gcc
3  # LDFLAGS = -lm (poderia ser necessária em outro projeto que utilizasse a math.h)
4
5  SRC = $(wildcard *.c)
6  OBJ =  $(SRC:.c=.o) # realiza substituição dos nomes .c para os .o
7
8
9  main: $(OBJ)
10     $(CC) $(OBJ) -o $@ $(LDFLAGS)
11
12 %.o: %.c %.h
13     $(CC) $(CFLAGS) -c $<
14
15 .PHONY: clean
16
17 clean:
18     rm main $(OBJ)
```

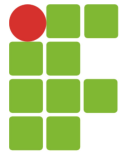


# Sumário



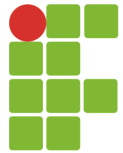
# Exemplo

- Podemos fazer um `Makefile` mais complexo, que insere os arquivos objetos e o executável em pastas específicas.



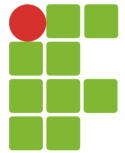
# Exemplo

```
1  CFLAGS = -Wall
2  CC = gcc
3  # LDFLAGS = -lm (poderia ser necessária em outro projeto que utilizasse a math.h)
4
5  OBJDIR = objects
6  BINDIR = bin
7  SRC = $(wildcard *.c)
8  OBJ =  $(patsubst %.c,$(OBJDIR)/%.o,$(SRC))
9
10 all: binfolder objfolder bin/main
11
12 bin/main: $(OBJ)
13     @ echo "Compilando os arquivos objetos no executável"
14     $(CC) $(CFLAGS) $(OBJ)  -o $@ $(LDFLAGS)
15
16 binfolder:
17     @ echo "Criando pasta dos binários"
```

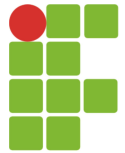


# Exemplo

```
18      mkdir -p $(BINDIR)
19
20  objfolder:
21      @ echo "Criando pasta dos objetos"
22      mkdir -p $(OBJDIR)
23
24  objects/main.o: main.c
25      $(CC) $(CFLAGS) -c $< -o $@
26
27  objects/%.o: %.c %.h
28      $(CC) $(CFLAGS) -c $< -o $@
29
30
31  .PHONY: clean
32
33  clean:
34      rm bin/* objects/*
35      rmdir bin objects
```

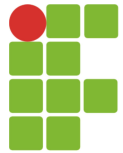


# Sumário



# Make

- A ferramenta `make` é essencial para construir artefatos computacionais em projetos mais complexos.
- Facilita bastante a distribuição do código e a compilação.
- Existe um trabalho pequeno para elaborá-lo, mas que é recompensado pela automatização das tarefas.
- Além da compilação, é possível criar regras para executar *benchmarks* ou testes unitários, as possibilidades são ilimitadas.



# Alternativas ao Make

- Além do `make`, existem diversos outros sistemas de construção de artefatos computacionais, dos quais podemos citar:
  - `CMake`;
  - `Ninja`;
  - `Bazel`;
  - `SCons`;
  - *etc...*