Ferramentas de desenvolvimento

make, controle de versões

programação em ambiente Linux

Taisy Silva Weber

Ferramentas...

- Editores
- Auxílio a compilação:
  - comando make e arquivo makefile
  - sintaxe, regras, dependência, macros, regras embutidas (built-in rules), múltiplos alvos.
- Controle de versões:
  - necessidades, definição, RCS, SCCS e CVS.

Matthew99, cap. 8

Ferramentas de desenvolvimento

- ✓ muitas ferramentas disponíveis
  - idênticas ou derivadas de ambientes UNIX
- ✓ geralmente não são amigáveis
  - linha de comandos
- ✓ muito populares
  - `make` - controle de compilação de código fonte
  - `gdb` - depurador

Editores

usar o editor que estiver acostumado

- ✓ mais comuns
  - vi
  - Emacs (Editor MACroS)

necessários quando interface gráfica não está disponível

- ✓ vi
  - pequeno
  - apenas editor
  - permite macros
  - toque simples de teclas habilitam comandos sofisticados

não muito bem documentado

Emacs

- ✓ mais que um editor
  - sistema completo
    - permite ler e escrever e-mails, jogar, executar programas da shell
    - possui linguagem de programação própria: elisp
  - documentação abundante
- ✓ modo vi (chamado viper)
  - permite executar comandos vi dentro do Emacs

make

auxílio a compilação

- descreve como compilar programas
  - pode ser usado também para produzir arquivos de saída a partir de múltiplos arquivos de entrada
    - evtl. também para processar documentos (por exemplo com troff ou TeX)
- makefile
  - arquivo que define como a aplicação é construída
    - geralmente reside no mesmo diretório que os demais arquivos do projeto
  - podem existir vários e diferentes makefiles a qualquer momento

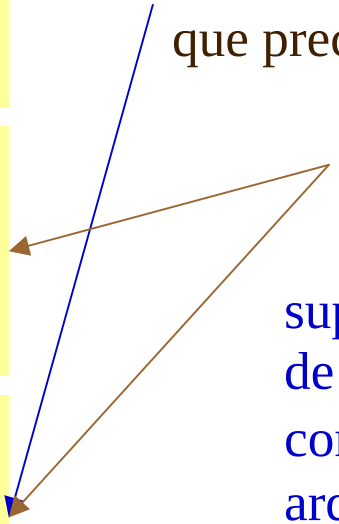
Motivação

problemas com múltiplos arquivos fonte

```
/* main.c */
#include "a.h"
...
```

```
/* 2.c */
#include "a.h"
#include "b.h"
...
```

```
/* 3.c */
#include "b.h"
#include "c.h"
…
```



trocas em `c.h` invalidam `3.c` que precisa ser recompilado

trocas em `b.h` invalidam `2.c` e `3.c` que precisam ser recompilados

suponha que o programador esquece de recompilar `2.c` (o que pode ser comum se um grande número de arquivos fonte estiver envolvido)

`make` assegura que todos os arquivos afetados pelas mudanças serão recompilados

a.h, b.h e c.h são header files

Sintaxe de makefiles

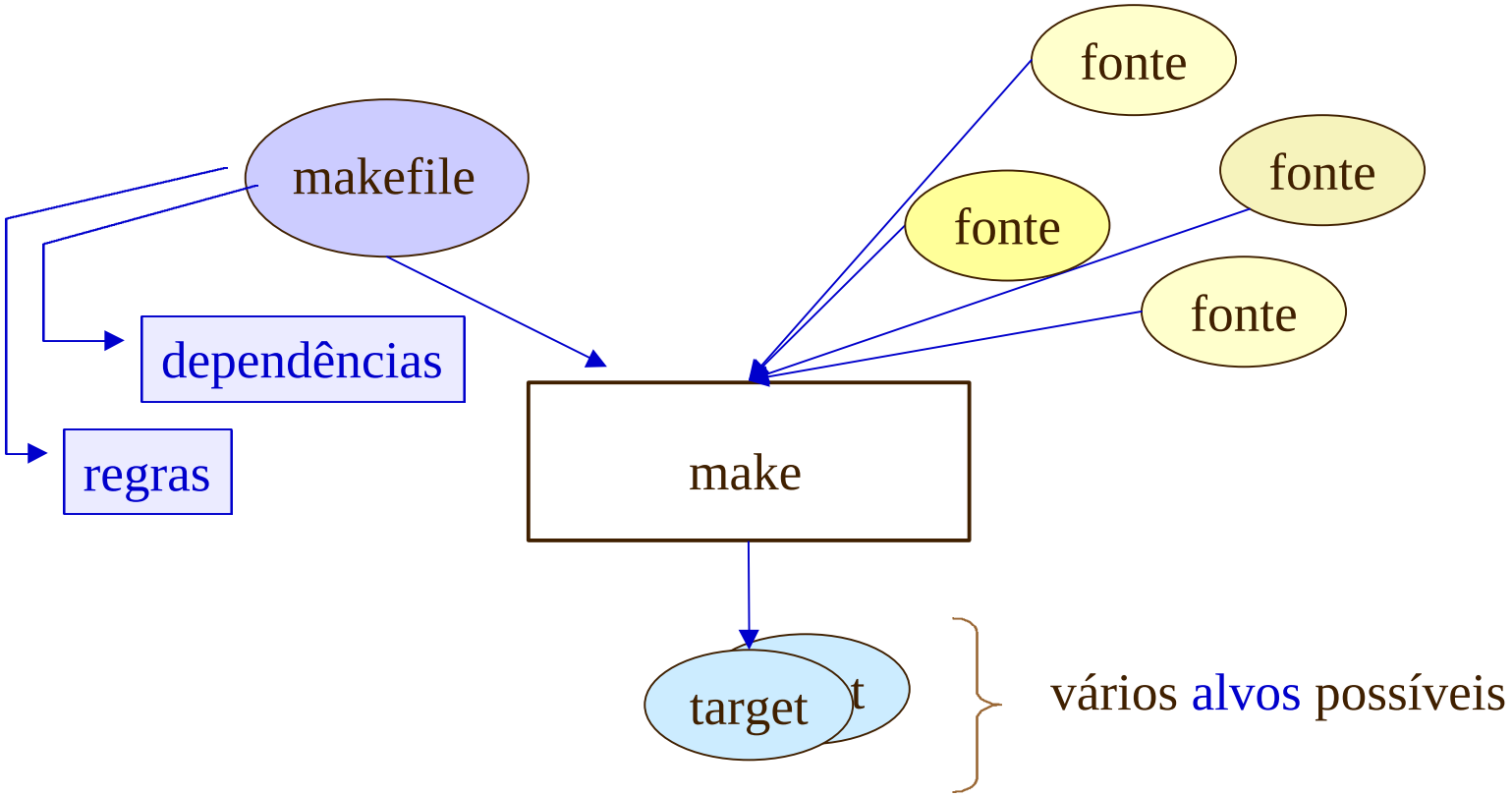
- ✓ makefile

lido pelo comando `make`

`make` compara datas e horas dos arquivos fonte para decidir quais regras aplicar para criar cada alvo

  - conjunto de **dependências** e **regras**
  - dependência
    - alvo (*target*): arquivo a ser construído
      - geralmente um arquivo **executável**
    - conjunto de arquivos fonte do qual ele é dependente
  - regras
      - descrevem como criar o alvo a partir dos arquivos fontes

make & makefile



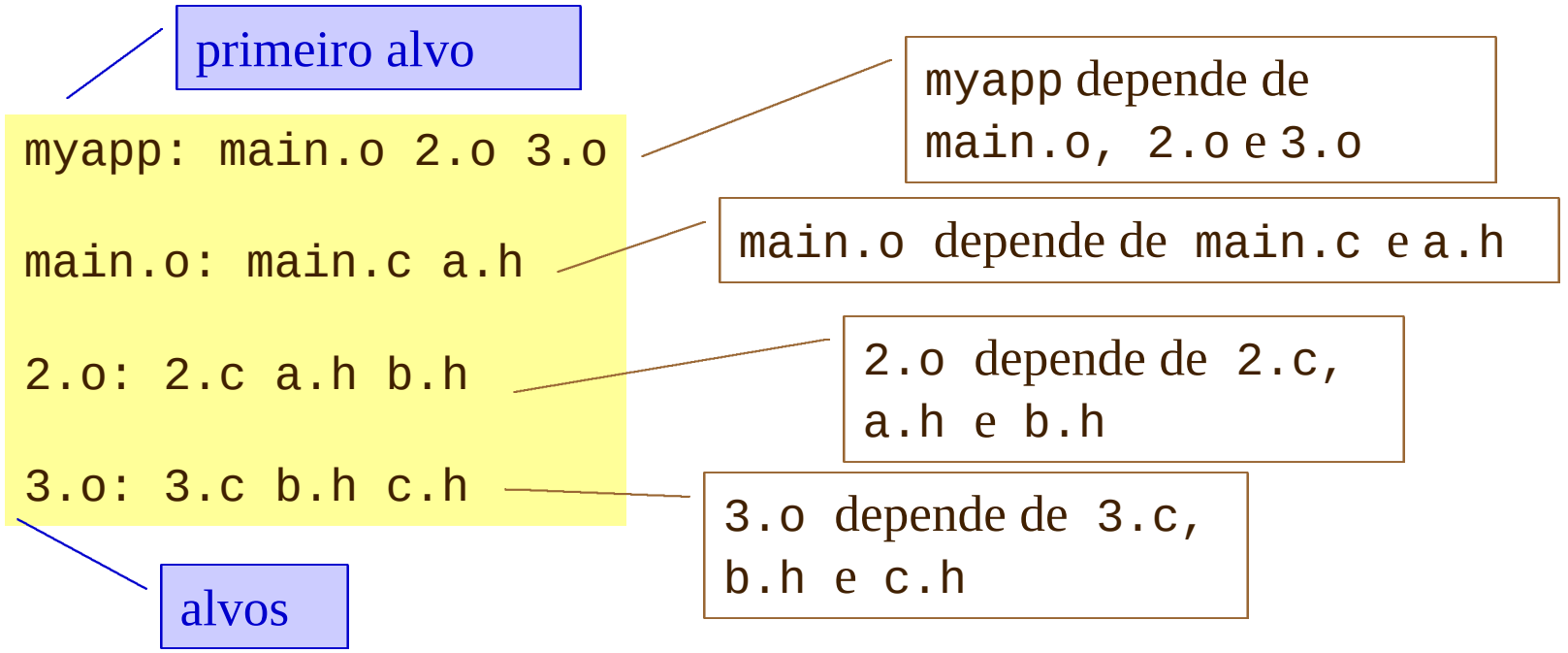
Opções

- `-k`
  - continuar mesmo se erros forem encontrados
- `-n`
  - imprimir o que deveria estar fazendo sem fazê-lo
- `-f <filename>`
  - determina arquivo a ser usado como `makefile`
  - se nada for especificado
    - então `make` procura um arquivo chamado `makefile` no diretório atual
    - se não procura por `Makefile`

Parâmetro

- ✓ nome do arquivo alvo
  - passado como parâmetro
  - se não for passado parâmetro
    - `make` tentará construir (fazer) o primeiro alvo listado no `makefile`
    - programadores geralmente usam `all` como o primeiro alvo no `makefile`
      - `all` permite definir mais do que um arquivo como alvo para `make` construir

Dependências



o exemplo estabelece uma hierarquia de dependências

mostra como os arquivos dependem um do outro

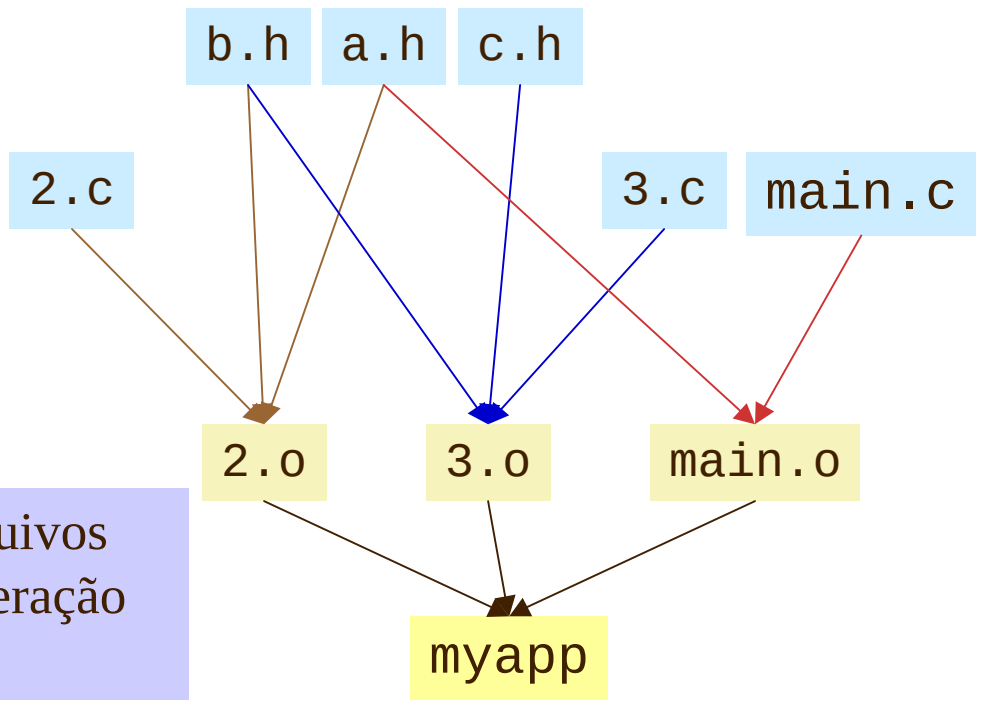
`myapp` será o arquivo que `make` vai construir

Dependências

`make` vai **construir** `myapp`

```
myapp: main.o 2.o 3.o

main.o: main.c a.h

2.o: 2.c a.h b.h

3.o: 3.c b.h c.h
```

hierarquia de dependências



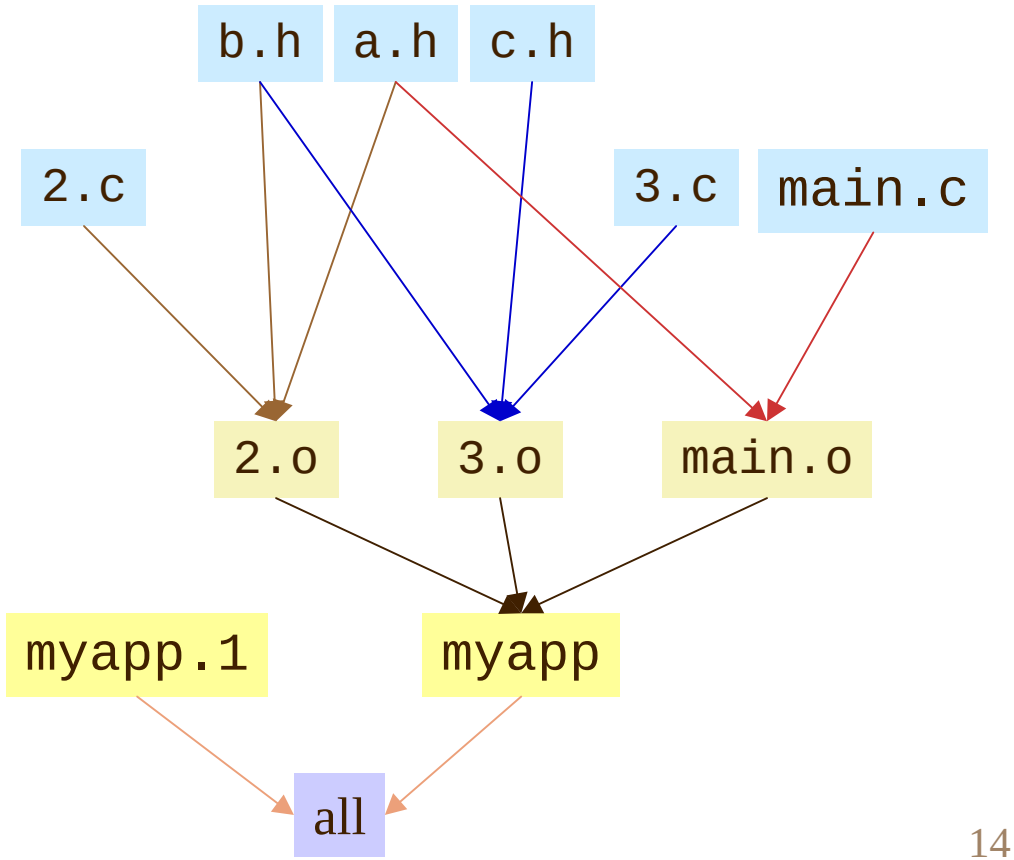
fica fácil verificar quais arquivos são afetados por alguma alteração (em `b.h`, por exemplo)

Vários alvos

primeiro alvo é `all`: `make` vai construir **todos** arquivos da lista

```
all: myapp myapp.1

myapp: main.o 2.o 3.o

main.o: main.c a.h

2.o: 2.c a.h b.h

3.o: 3.c b.h c.h
```

para construir `all`,
todas as dependências
devem ser resolvidas



Regra

- define o comando a ser usado com os arquivos da lista de dependência
  - por exemplo: `gcc -c 2.c`
- cada linha de comando é executada em uma nova shell
- formato:
  - uma linha de regra começa com tab
  - nenhuma linha em `makefile` pode terminar com espaço

atenção: deve ser tab e não espaços

Exemplo de regras

arquivo `Makefile1`

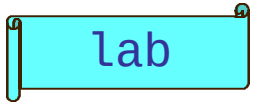
```
myapp: main.o 2.o 3.o
	gcc -o myapp main.o 2.o 3.o

main.o: main.c a.h
	gcc -c main.c

2.o: 2.c a.h b.h
	gcc -c 2.c

3.o: 3.c b.h c.h
	gcc -c 3.c
```

começa com tab

invocação

```
$ make -f Makefile1
```

resultado será uma mensagem de erro pois `main.c` ainda não foi criado (vários outros também não, mas `main.c` é o primeiro encontrado)



Arquivos fonte para Makefile1

```
$ touch a.h
$ touch b.h
$ touch c.h
```

cria *header files* vazios

esses programas não fazem nada

```
/* main.c */
#include "a.h"
#define EXIT_SUCCESS 0
extern void function_two();
extern void function_three();

int main()
{
    function_two();
    function_three();
    exit (EXIT_SUCCESS);
}
```

```
/* 2.c */
#include "a.h"
#include "b.h"

void function_two() {
}
```

```
/* 3.c */
#include "b.h"
#include "c.h"

void function_three() {
}
```

lab

Invocando `make` novamente

```
$ make -f Makefile1
gcc -c main.c
gcc -c 2.c
gcc -c 3.c
gcc -c myapp main.o 2.o 3.o
$
```

`make` mostra os comandos conforme vai executando-os

tentar:

```
$ touch b.h
$ make -f Makefile1

$ rm 2.o
$ make -f Makefile1
```

header foi alterado

objeto foi removido

`make` determina o número mínimo de comandos para execução

`lab`

Comentários em makefile

✓ #

- comentário inicia com # e continua até o fim da linha

não esquecer de documentar abundantemente todos os seus `makefiles` assim como seus programas

não esquecer também que uma linha em `makefile` **não pode terminar com caracter espaço** e que **espaço** e **tab** são interpretados de forma diferentes

Macros em makefile ...

arquivo `Makefile2`

```
all: myapp
```

`all` usado por hábito, mesmo sem múltiplos alvos

```
# Which compiler
CC = gcc
```

macro

```
# Where are include files kept
INCLUDE = .

# Options for development
CFLAGS = -g -Wall -ansi

# Options for release
# CFLAGS = -O -Wall -ansi
```

formato:
`MACRONAME=valor`

também aparece na literatura como definição de variáveis

`lab`

... macros em makefile

```
myapp: main.o 2.o 3.o
	$(CC) -o myapp main.o 2.o 3.o

main.o: main.c a.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c main.c

2.o: 2.c a.h b.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c

3.o: 3.c b.h c.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

acesso ao valor da macro

acesso ao valor: usando `$(MACRONAME)` ou `${MACRONAME}`

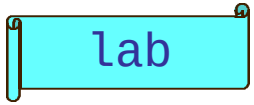
lab

Invocando `make`

```
$ rm *.o myapp
$ make -f Makefile2
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

durante a execução, `make` substitui as macros pelas suas definições

por exemplo; para alterar o compilador C, do gcc para o c89, basta alterar uma linha de `makefile` onde aparece a definição de CC



Caracteres especiais & macros

- ✓ caracteres
  - usados em makefile antes de comandos
    - `-` `make` deve ignorar erros
    - `@` `make` não deve imprimir o comando
- ✓ macros pré-definidas
    - `$@` nome do alvo atual (current target)
    - `$<` nome da dependência atual
    - `$*` nome da dependência atual sem sufixo

existem várias outras macros pré-definidas

Alvos múltiplos

- coletar vários grupos de comandos em um único lugar
- definir mais do que um arquivo alvo

`make` vai tentar construir **apenas um** alvo a cada invocação

  - qual alvo será construído é determinado pelo parâmetro na invocação de `make`
  - no caso de nenhum parâmetro ser passado, `make` constrói o primeiro alvo de `makefile`

Alvos múltiplos: exemplo ...

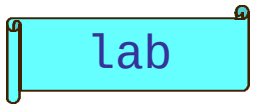
arquivo `Makefile3`

```
all: myapp

# Which compiler
CC = gcc
# Where to install
INSTDIR = /usr/local/bin
# Where are include files kept
INCLUDE = .
# Options for development
CFLAGS = -g -Wall -ansi
# Options for release
# CFLAGS = -O -Wall -ansi
```

essa linha especifica um alvo apenas; para fazer `all` é necessário fazer `myapp`

macro nova

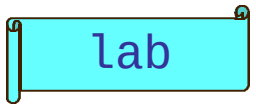
se nenhum parâmetro for passado, `make` irá construir apenas `myapp`



... exemplo ...

```
myapp: main.o 2.o 3.o
	$(CC) -o myapp main.o 2.o 3.o

main.o: main.c a.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c main.c

2.o: 2.c a.h b.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c

3.o: 3.c b.h c.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

esse trecho idêntico a Makefile2



... exemplo

clean e install são alvos

- ignorar erros

@ não imprimir o comando

\ indica que a linha continua

```
clean:
	-rm main.o 2.o 3.o

install: myapp
	@if [ -d $(INSTDIR) ]; \
	then \
		cp myapp $(INSTDIR);\
		chmod a+x $(INSTDIR)/myapp;\
		chmod og-w $(INSTDIR)/myapp;\
		echo "Installed in $(INSTDIR)";\
	else \
		echo "Sorry, $(INSTDIR) does not exist";\
	fi
```

shell script

cada linha de comando é executada em uma nova shell

lab

comentários sobre exemplo

```
clean:
	-rm main.o 2.o 3.o
```

`clean` não possui dependências e será sempre executado se `make` for invocado com o parâmetro `clean`

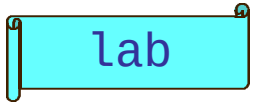
```
install: myapp
```

`install` depende de `myapp` e, para ser construído, `myapp` precisa ser resolvido

as regras para construir `install` formam um script;
como `make` invoca uma shell para cada linha, deve ser forçada a invocação da shell com uma única linha lógica

Invocando make (alvos múltiplos)

```
$ rm *.o myapp
$ make -f Makefile3
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$ make -f Makefile3
make: Nothing to be done for 'all'.
$ rm myapp
$ make -f Makefile3 install
gcc -o myapp main.o 2.o 3.o
Installed in /usr/local/bin
$ make -f Makefile3 clean
rm main.o 2.o 3.o
$
```



Regras embutidas

- simplificam makefiles
  - muitas regras podem ser omitidas
  - por exemplo:
    - `make` sabe como invocar um compilador
    - comando de compilação pode ser omitido
- para descobrir quais as regras embutidas
  - usar a opção `-p`

```
make -p
```

existe grande quantidade de regras embutidas

muitas vezes chamadas de regras de inferência

Exemplo de regra embutida

foo.c

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    exit (EXIT_SUCCESS);
}
```

```
$ make foo
cc foo.c  -o foo
$
```

sem usar makefile

mas invocou `cc` no lugar de `gcc`

```
$ rm foo
$ make CC = gcc CFLAGS = "-Wall -g" foo
gcc -Wall -g  foo.c  -o foo
$
```

usando macros pré definidas CC e CFLAGS

Regras de sufixo

- Unix não diferencia sufixos
- sufixos em `make`
  - `make` usa sufixos de arquivos para determinar qual regra embutida usar
  - exemplo:
    - gerar um arquivo `.o` a partir de um arquivo `.c` usando o compilador
  - novas regras de sufixo podem ser criadas

```
.<old suffix>.<new suffix>:
```

arquivo com novo sufixo criado com mesmo nome

Regras de sufixo: exemplo

regra embutida para manipular bibliotecas

```
.<old suffix>.<new suffix>:
```

`.a` (para **archive**)

```
.c.a:
    $(CC) -C $(CFLAGS) $<
    $(AR) $(ARFLAGS) $@ $*.o
```

`$(AR)` expande para `ar`
`$(ARFLAGS)` expande para `rv`

`$@` nome do alvo atual (current target)
`$<` nome da dependência atual
`$*` nome da dependência atual sem sufixo

para construir uma biblioteca `.a` a partir de um arquivo `.c` `make` deve aplicar duas regras

Manipulação de bibliotecas

- biblioteca
  - contém uma coleção de objetos `.o`
  - arquivos com extensão `.a`
- para referenciar bibliotecas em `make`
  - `lib(file.o)`
  - significa: objeto `file.o` na biblioteca `lib.a`

Mais um exemplo

arquivo `Makefile6`

```
# Local Libraries
MYLIB = mylib.a

myapp: main.o $(MYLIB)
	$(CC) -o myapp main.o $(MYLIB)

$(MYLIB): $(MYLIB)(2.o) $(MYLIB)(3.o)
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h

clean:
	-rm main.o 2.o 3.o $(MYLIB)
```

macro

o resto do arquivo corresponde a trechos de Makefile3

`lib(file.o)`

usando regras embutidas para compilação

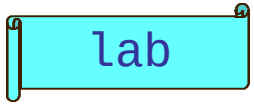
Invocando make (com lib) ...

remove todos os objetos e biblioteca

```
$ rm -f myapp *.o mylib.a
$ make -f Makefile6
gcc -g -Wall -ansi  -c main.c -o main.0
gcc -g -Wall -ansi  -c 2.c -o 2.o
ar rv mylib.a 2.o
ar: creating mylib.a
c - 2.o
gcc -g -Wall -ansi  -c 3.c -o 3.o
ar rv mylib.a 3.o
c - 3.o
gcc -o myapp main.o mylib.a
```

cria lib

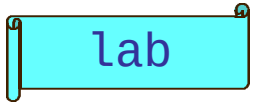
linka `main.o` com a lib



Invocando make (com lib)

```
$ touch c.h
$ make -f Makefile6
gcc -g -Wall -ansi  -c 3.c -o 3.o
ar rv mylib.a 3.o
a - 3.o
gcc -o myapp main.o mylib.a
$
```

`c.h` foi alterado;
`make` deve recompilar `3.c`, atualizar a lib e finalmente relinkar para construir o executável `myapp`



GNU make e gcc

- opção interessante para gcc
  - -MM
    - produz uma lista de dependência
    - lista pode ser usada diretamente em make
    - facilita o uso de make

```
$ gcc -MM main.c 2.c 3.c
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
$
```

lista construída

Controle de código fonte

- ✓ gerenciar mudanças no código fonte
  - em projetos com múltiplos programadores
  - vários programadores alterando o mesmo código
  - RCS
  - ⇨ Revision Control System
    - Free Software Foundation
  - SCCS
    - Source Code Control System
      - introduzido pela AT&T no System V
      - incorporada no padrão X/Open

código fonte do sistema é livre

RCS

- conjunto de comandos para gerenciar arquivos fonte
- mantém um único arquivo e uma lista de alterações
  - lista permite recriar qualquer versão anterior
- permite armazenar comentários associados a cada alteração
- eficiente em termos de espaço
  - armazena apenas as alterações

Principais comandos ...

- rcs
  - inicializa um um arquivo de controle RCS
  - permissões de acesso: apenas leitura
- ci check in
  - verifica e armazena versão atual
- co check out
  - verifica e recria a versão atual
  - altera permissões de acesso para permitir escrita no arquivo

... principais comandos

- rlog
  - mostra um sumário das alterações em um arquivo
  - inicia pela mais recente
- rcsdiff
  - mostra apenas o que mudou entre duas revisões

o uso de RCS será demonstrado a partir de um exemplo

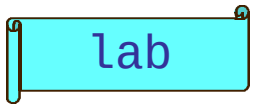
Exemplo RCS

important.c

```
/*
   This is an important file for managing the project.
   It implements the canonical "Hello World" program.
*/

#include <stdlib.h>
#include <stdio.h>

int main()
{
   printf("Hello World\n");
   exit (EXIT_SUCCESS);
}
```

o comando `rcs -i` inicializa o arquivo de controle

e permite entrar com comentários



Exemplo comando `rcs`

```
$ rcs -i important.c
RCS file: important.c,v
enter description, ....
NOTE: This is NOT the log message!
>> esse e' um arquivo de demonstracao
>> .
done
$
```

arquivo read-only com extensão `,v`

experimente `ls -l`

permitidas múltiplas linhas de comentários

entrada de comentários encerra com `.` sozinho numa linha ou com Crtl-D

`lab`

Exemplo comando `ci`

```
$ ci important.c
important.c,v <- important.c
initial revision: 1.1
done
$
```

armazena versão atual no arquivo RCS

o arquivo `important.c` é deletado;
o conteúdo do arquivo original e todas as informações de controle estão todas armazenadas no arquivo de controle `important.c,v`

se o comando `rcs -i` foi esquecido, RCS vai perguntar por uma descrição para o arquivo

`lab`

Exemplo comando `co`

lock

```
$ co -l important.c
important.c,v -> important.c
revision 1.1 (locked)
done
$
```

o arquivo `important.c` é recriado e pode ser editado

o arquivo `important.c` é recriado com permissão de escrita

bloqueio (lock) garante que apenas um programador estará alterando o arquivo

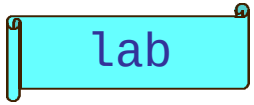
experimente `ls -l`

após usar `co`, alterar o arquivo e usar `ci` novamente; se quiser continuar alterando o arquivo, usar `ci -l` para reter o bloqueio

`lab`

Alterando o arquivo

```
/*
   This is an important file for managing the project.
   It implements the canonical "Hello World" program.
*/
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    printf("This is an extra line added later\n");
    exit (EXIT_SUCCESS);
}
```



Salvando alterações e recuperando versões anteriores

```
$ ci important.c
important.c,v <- important.c
new revision: 1.2; previous revision: 1.1
enter description, ....
>> linha extra de impressao adicionada
>> .
done
$
```

suponha que queremos voltar a versão 1.1

```
$ co -r1.1 important.c
important.c,v -> important.c
revision 1.1
done
$
```

lab

Exemplo `rlog` e `rcsdiff`

Experimente o comando:

```
$ rlog important.c
```

`rlog` mostra um sumário das mudanças no arquivo

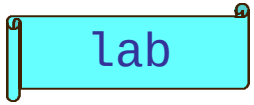
a primeira parte fornece uma descrição do arquivo e as opções que **RCS** está usando

a segunda parte lista as revisões do arquivo, iniciando pela mais recente, e mostrando os comentários digitados

Experimente também o comando:

```
$ rcsdiff -r1.1 -r1.2 important.c
```

`rcsdiff` mostra apenas o que mudou entre duas revisões



Identificando revisões

- macros RCS dentro dos fontes
  - ajudam a identificar revisões
- mais comuns
  - `$RCSfile$`

    expande para o nome do arquivo
  - `$Id$` expande para *string* identificando revisão

consultar *manual pages* para outras macros

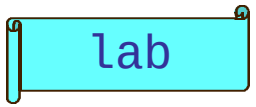
editando novamente o arquivo

```
$ co -l important.c
important.c,v -> important.c
revision 1.2 (locked)
done
$
```

lab

Exemplo macros de identificação

```
/*
   This is an important file for managing the project.
   It implements the canonical "Hello World" program.

   Filename: $RCSfile$
*/
#include <stdlib.h>
#include <stdio.h>

static char *RCSinfo = "$Id$";
int main()
{
    printf("Hello World\n");
    printf("This is an extra line added later\n");
    printf("This file is under RCS control. ");
    printf("Its ID is \n%s\n", RCSinfo);
    exit (EXIT_SUCCESS);
}
```

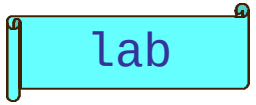


Exemplo macros de identificação

depois de alterado, check in novamente

```
$ ci important.c
important.c,v <- important.c
new revision: 1.3; previous revision: 1.3
enter description, ....
>> adicionadas macros $RCSfile$ e $Id$
>> .
done
$
```

usando comando co e examinando a versão atual do arquivo pode ser verificado que as macros foram expandidas no código fonte



GNU `make` e RCS

- `make` possui uma regra que permite recriar um arquivo `.c` a partir de um arquivo `.c,v`
  - `make` invoca `co` criando o arquivo `.c`
  - após a compilação, `make` remove os arquivos `.o` e `.c` gerados

```
$ rm -f important.c
$ make important
```

para forçar `make` a invocar `co`

lab

SCCS

- muito similar ao RCS
  - comandos possuem nomes diferentes
- especificado em X/Open
  - versões mais recentes do UNIX devem suportar SCCS
- entretanto
  - RCS é mais popular
  - RCS é software livre

não é descrito nos livros texto da disciplina

CVS

- CVS: concurrent versions system
  - usável em ambiente de rede e na Internet
    - não restrito a um diretório local compartilhado como RCS
  - vários programadores podem trabalhar simultaneamente em um arquivo
    - e não apenas um de cada vez como RCS
  - semelhança com RCS
  - muito usado no GNU project para desenvolvimento de software livre pela Internet

brevíssima descrição em Mattew99

Fim...

- ✓ de ferramentas de desenvolvimento
  - vistos make e RCS