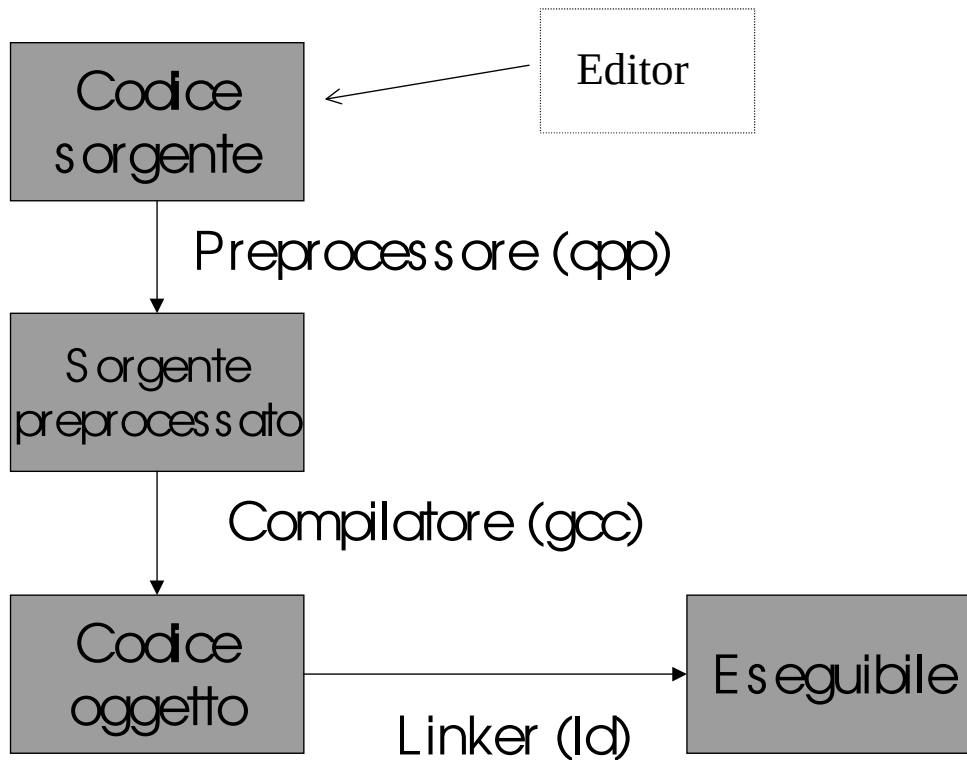


Strumenti di sviluppo in UNIX

Il processo di compilazione



Strumenti di sviluppo in UNIX

Emacs/Xemacs

- Anche altri editor (elvis, jed, nedit, vi, vim, xcoral, xedit,...)

SisOp@linus: xemacs

SisOp@linus: xemacs prova.c

SisOp@linus: xemacs prova.c &

- qualche comando da ricordare:

CxCf: carica un nuovo file in emacs/xemacs

CxCs: salva il file

Ck: cancella dalla posizione del cursore alla fine della linea

Cs <stringa>: cerca la stringa indicata

Cx-2: divide lo schermo orizzontalmente

Strumenti di sviluppo in UNIX

Il preprocessore C (cpp)

- É un tool per trasformare il codice prima della compilazione.
- Le istruzioni del preprocessore sono precedute dal simbolo #.

Definizione di macro ed espansioni

- **macro**: definizione di un identificatore associato ad una stringa
- viene definito con:
`#define NAME expansion`
- tutti i successivi NAME vengono sostituiti con `expansion`
- Per convenzioni i nomi delle macro sono composte da sole lettere maiuscole. Es: `#define MAX 10`
- Le macro possono avere degli argomenti:
`#define MIN(x,y) ((x<y)?(x):(y))`
- L'invocazione: `MIN(a,b)` viene espansa nel codice: `((a<b)?(a):(b))`

Strumenti di sviluppo in UNIX

Inclusione di file

- il preprocessore include un file per mezzo dell'istruzione `#include`
- in genere per un programma/modulo vengono inclusi diversi file header:

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

Strumenti di sviluppo in UNIX

Compilazione condizionale

- Il preprocessore consente di compilare in modo selettivo delle porzioni di programma in base al verificarsi di certe condizioni.
- `#ifdef NAME` (`#ifndef NAME`) inserisce il codice se NAME é stato definito (non definito).

- **Es:**

```
#define FOO
#ifdef FOO
    ... this gets included...
#endif
#ifndef FOO
    ... this does NOT get included...
#endif
```

- **Demo:**

```
cpp -P foo
cpp -P -DINCLUDE_BAR foo
cpp -P -DINCLUDE_BLAH foo
```

Strumenti di sviluppo in UNIX

Compilatore (gcc)

- Può invocare automaticamente tutti gli strumenti necessari alla compilazione: preprocessore, compilatore, assembler e linker
- Trasforma il codice sorgente in codice macchina ma non risolve necessariamente tutti i simboli:

```
gcc <opzioni> <filename>
```

```
SisOp@linus: gcc -c hello.c    # -c compila soltanto ma non
                                # collega le librerie)
```

Opzioni di compilazione per gcc

- Per assicurarsi che il codice sia conforme allo standard ANSI:
 - `-Wall`: mostra tutti i messaggi di warning (avvisi) che gcc può fornire;
 - `-pedantic`: mostra tutti gli errori e i warning richiesti dallo standard ANSI C
- Per ottimizzare:
 - `-O -O1 -O2 -O3`: livelli crescenti di ottimizzazione del compilatore
 - `-O0`: nessuna ottimizzazione
- Per includere le informazioni per il debug: `-g`

Il comando **nm** visualizza i simboli di un file .o

```
gcc -c hello.c # contiene printf("Hello world");  
nm hello.o
```

```
00000000 t gcc2_compiled.
```

```
00000000 T main
```

```
U printf
```

Il file definisce
una funzione globale (T sta
per text cioè codice) di nome
`main`

Il file contiene un riferimento ad un simbolo
`printf` non definito localmente (U sta per
undefined). Il file andrà collegato alla libreria C
che definisce quel simbolo.

Strumenti di sviluppo in UNIX

Linker (ld)

- Risolve i simboli tra programmi e crea gli eseguibili

```
SisOp@linus: gcc -c hello.c
```

```
SisOp@linus: gcc hello.o -o hello # la libreria C (libc) è  
# collegata automaticamente
```

```
SisOp@linus: hello
```

- Esempio: link con altre funzioni

```
SisOp@linus: gcc hello2.c -o hello2 [fallisce]
```

```
SisOp@linus: gcc hello2.c -lm -o hello2 [funziona]
```

È necessario avere la funzione `sqrt` dal file

`/usr/lib/libm.so` (collegamento dinamico - default) oppure `/usr/lib/libm.a` (collegamento statico - opzione `-B` al linker)

- Alcune importanti librerie si trovano in:

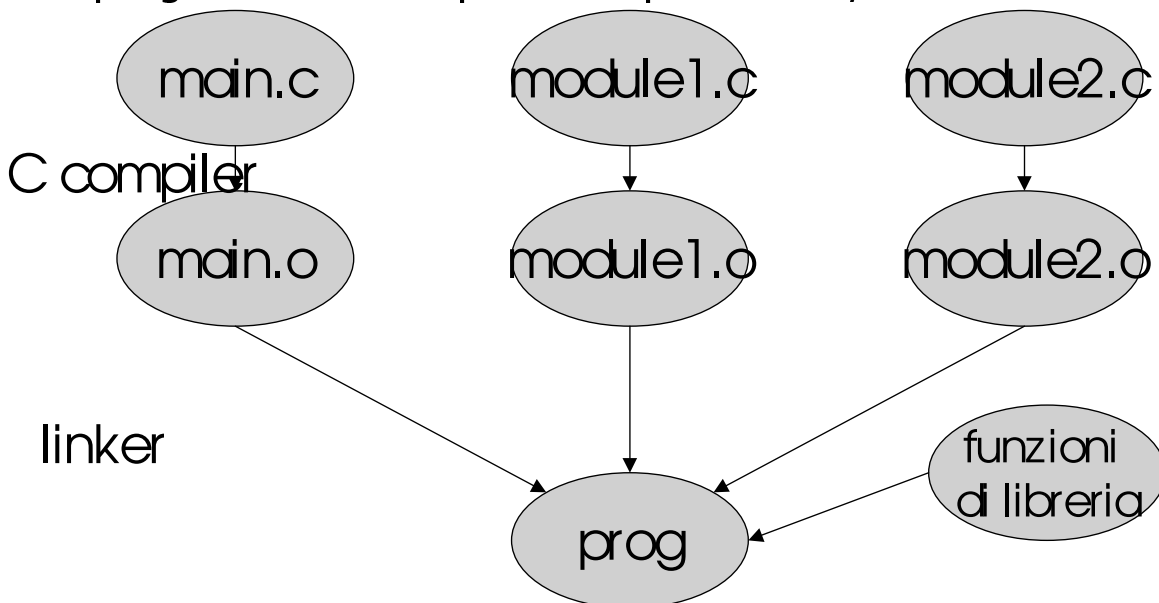
`/usr/lib`

`/lib`

`/usr/X11R6/lib`

Strumenti di sviluppo in UNIX

Se il programma è composto da più moduli/file



```
gcc -c main.c  
gcc -c module1.c  
gcc -c module2.c  
gcc -o prog main.o module1.o module2.o
```

Strumenti di sviluppo in UNIX

Se il programma è composto da più moduli/file:

- vanno inseriti i prototipi per tutte le funzioni prima della loro chiamata:

- Esempio:

```
#include <stdio.h>
/* This is a prototype. Use them */
int blah(int foo);

void main(void) {
    printf("blah(5) = %d\n", blah(5));
}
int blah(int foo) {
    return(2*foo);
}
```

- L'uso dei prototipi definisce una interfaccia che aiuta a prevenire errori

Strumenti di sviluppo in UNIX

Simboli esterni e simboli globali

- Inserire la parola chiave `extern` crea un simbolo ma non alloca spazio per la variabile/funzione: definizione di un riferimento esterno
- In un solo file la variabile/funzione non deve essere definita come `extern`: definizione del simbolo globale
- Esempio:

```
gcc -c main.c # contiene extern int numero ; numero += 1;
gcc -c blah.c # contiene int numero=0 ;
gcc main.o -o example [non funziona]
gcc main.o blah.o -o example [funziona]
```

Strumenti di sviluppo in UNIX

File header

- hanno estensione .h
- sono usati per specificare l'interfaccia di un modulo/file
- Il file header puo` essere incluso in due modi:
 - tramite la notazione "nome_del_file"
 - tramite la notazione <nome_del_file>
- L'opzione di compilazione -I dir indica il percorso di ricerca per i file .h;
- L'header dovrebbe contenere:
 - prototipi delle funzioni condivise
 - dichiarazione extern per le variabili condivise
 - typedefs
 - macros
 - structs, enums, e altri tipi di dati

Strumenti di sviluppo in UNIX

- Uso dei defines per evitare la ricorsione:

```
#ifndef FOO_H_INC
#define FOO_H_INC
#include "foo.h"
...
#endif /* FOO_H_INC */
```

- Gli header file si possono trovare in:

```
/usr/include
/usr/include/X11 (header per X Window System)
```

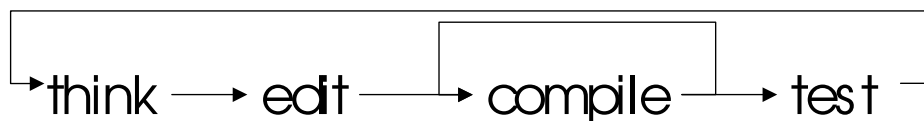
File core

- Contengono lo stato del programma nel momento che ha catturato un segnale, compreso stack, heap, registers,.
- Permette di fare delle analisi post-mortem
- La creazione dei file core puo` essere attivata/disattivata
- Per shells C (csh e tcsh)

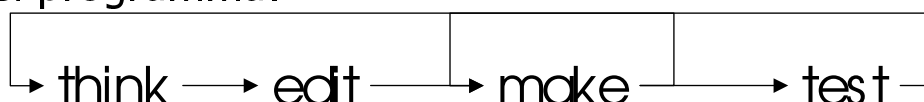
```
limit coredumpsize unlimited  
limit coredumpsize 0  
limit coredumpsize 50k
```

Il programma **make**

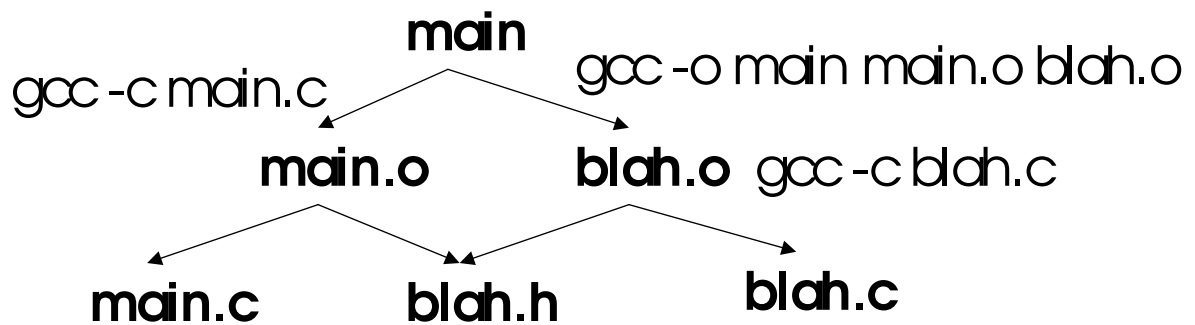
- Ciclo di sviluppo di un programma



- Problemi potenziali:
 - si modifica un file ma ci si dimentica di ricompilarlo
 - si modifica un'interfaccia (.h) ma ci si dimentica di compilare tutti i file che dipendono da essa
 - si compilano anche file non modificati
 - make rende automatica la compilazione e la costruzione del programma:



Make utilizza un grafo delle dipendenze



- ogni nodo rappresenta un file
- di fianco ad ogni nodo vi è il comando che produce ("make") quel file

Per produrre il nodo X

- make tutte le dipendenze di X (quelle modificate piu` recentemente di X)
- aggiorna X con il comando associato
- Es: se **blah.h** o **main.c** sono piu` recenti di **main.o** ricrea **main.o** con **gcc -c main.c**

Makefile

- I file *makefile* o *Makefile* specificano il grafo delle dipendenze

targets: dependents

commands

- I comandi devono iniziare con un carattere di tabulazione (nell'esempio rappresentato da *[Tab]*):

```
main: main.o blah.o
```

```
[Tab]gcc -o main main.o blah.o
```

```
main.o: main.c blah.h
```

```
[Tab]gcc -c main.c
```

```
strset.o: blah.c blah.h
```

```
[Tab]gcc -c blah.c
```

- Per invocare make: `make targets`

```
make blah.o
```

```
make main
```

- Senza argomenti, make crea il primo target nel *makefile*: `SisOp@linus: make`

```
gcc -c main.c
```

```
gcc -c blah.c
```

```
gcc -o main main.o blah.o
```

```
SisOp@linus: touch blah.c
```

```
SisOp@linus: make blah.o
```

```
gcc -c blah.c
```

```
SisOp@linus: make
```

```
gcc -o main main.o blah.o
```

Built-in

- Make contiene dipendenze e comandi già pronti per l'uso

– si suppone che un file “.o” dipenda da un file “.c”

```
main: main.o blah.o
gcc -o main main.o blah.o
main.o blah.o: blah.h
```

Macro

- Make ha la possibilità di definire delle macro. Le macro comunicano con i comandi built-in e semplificano i *makefile*

```
CC = gcc
CFLAGS = -g
LDFLAGS = -g
OBJS=main.o blah.o
a.out: $(OBJS)
$(CC) $(LDFLAGS) $(OBJS)
$(OBJS): blah.h
```

- Target “dummy” per sequenze di comandi comuni:

```
install: a.out
        cp a.out main
        strip main
clean:
        -rm *.o core
clobber: clean
        rm -f a.out main
```

make clean rimuove i file “.o” e i core

- Utilizzare i target dummy per tutti i programmi di “manutenzione”:

```
clean    install    print
release  submit     test
```

Macro dinamiche

- Make contiene un insieme di macro che si modificano dinamicamente in funzione del target:

\$@ nome del target corrente
\$? lista delle dipendenze piu` recenti del target
\$< nome dei dependency file
\$* nome base del target corrente (privato dei suffissi)

Esempio:

```
foo.o: foo.c
        $(CC) -c $(CLAGS) $< -o $@
```

Opzioni di make

<code>make -n</code>	Mostra i comandi che dovrebbero essere eseguiti, senza eseguirli realmente. É utile per verificare la corretta espansione della macro
<code>make -d</code>	Mostra quali sono i criteri grazie a cui make determina se un target è out-of-date
<code>make -k</code>	Continua quanto più possibile dopo che si é verificato un errore
<code>make -f <filename></code>	Make usa <file> invece di makefile o Makefile