

Strumenti per la programmazione in C

Laboratorio Sistemi Operativi

Giuseppe Salvi

Email: giuseppe.salvi@uniparthenope.it

Il linguaggio C

- Studio delle system call (chiamate di sistema) che consentono di sfruttare appieno le potenzialità di Unix quali
 - gestione dei file
 - multitasking
 - interprocess communication (IPC)
- Utilizzeremo il C come linguaggio “ambiente” per sperimentare la programmazione di sistema

Perché il C?

- La scelta del linguaggio di programmazione è un vecchio dibattito in informatica
 - Per lo sviluppo di applicazioni è una questione aperta
- Per la programmazione di sistema ci sono pochi dubbi, il C
 - Il C è molto vicino all'hardware
 - Molti costrutti si traducono facilmente nel codice macchina
 - I tipi di dati disponibili riflettono ciò che supporta direttamente l'hardware
 - L'accesso alla memoria indiretto (via i puntatori) consente ai programmatori di accedere a tutte le parti del sistema
 - Storicamente UNIX scritto in C ed anche il kernel di Linux
 - Molti software di sistema sono scritti in C
 - I driver delle periferiche sono quasi tutti scritti in C
 - Il C è veloce (conseguenza del suo più basso livello di astrazione rispetto ad altri linguaggi)

Strumenti per la programmazione C

- Si assume una conoscenza dei costrutti di base
- Vedremo qualche cenno/ripasso
 - Compilazione e linking
 - Debugging
 - Programmazione modulare
 - Suddivisione di un programma in moduli e header file
 - Dipendenze tra i moduli: make

Compilazione di un programma C in UNIX/Linux

- Ogni versione di Unix ha un compilatore standard per il linguaggio C, generalmente chiamato **cc**
- Nel caso di GNU Linux è presente **gcc** (GNU C compiler) conforme allo standard POSIX (in Linux **cc** è un link simbolico a **gcc**)
- La sintassi esprime in maniera vaga l'ordine degli argomenti nella riga di comando e in effetti non c'è una particolare rigidità

gcc [<opzioni>|<file>]...

- Estensioni tipiche dei nomi dei file
 - .c Sorgente C
 - .o File oggetto
 - .a Libreria di file oggetto

Compilatore standard

- Per compilare un programma sorgente **source.c**

gcc source.c

che genera l'eseguibile **a.out**

- Si può scegliere il nome dell'eseguibile utilizzando l'opzione **-o**

-

gcc source.c -o target

Compilatore standard: Opzioni Comuni

- **-c** genera i file oggetto, senza effettuare il link dell'eseguibile finale
- **-g**
Aggiunge informazioni diagnostiche utili per il debugging attraverso strumenti appositi come **gdb**
- **-o file**
Indica che il file generato dalla compilazione (eseguibile, oggetto o altro) si deve chiamare **file**
- **-l libreria**
Utilizza la libreria il cui nome inizia per **lib**, continua con il nome indicato, e termina con **.a** oppure **.so**
- **-v** genera informazioni dettagliate sulla compilazione

Compilazione

Costruzione di un programma

- Una volta scritto un programma questo deve essere trasformato in un eseguibile
 - La costruzione di un programma consiste in una serie di passi
 - Ad ogni passo, il codice è trasformato in una forma intermedia
 - Preprocessed, codice assembly, codice oggetto e librerie
- E' importante comprendere ciascun passo per diversi motivi
 - Ognuno offre più strategie per risparmiare tempo di programmazione e per massimizzare le risorse del sistema
 - La fase di **preprocessing** fornisce le macro per la sostituzione ripetitiva di stringhe
 - Il passo **assembly** fornisce il programmatore dei nomi per le locazioni di memoria
 - Il passo di **linking** fornisce un modo per il riuso di codice eseguibile esistente in programmi multipli
 - Le forme intermedie possono essere mantenute tra i vari *build* del programma per velocizzare i rebuild successivi

Codice oggetto e linking

- I passi fondamentali nella costruzione di un programma sono la compilazione ed il linking
 - Trasformano il codice sorgente in un file eseguibile
 - Il codice sorgente è il programma C scritto dal programmatore

- **Es:**

```
#include <stdio.h>
int OurSquareRoot(int n){
    if (n==4)
        return(2);
    else{
        printf("Non posso calcolare la radice quadrata di %d\n",n);
        return(-1);
    }
}
```

Codice oggetto e linking

- Se questo codice è memorizzato nel file `sqrt.c`, può essere compilato in un codice oggetto come segue:

```
$ ls
  sqrt.c
$ gcc -c sqrt.c
$ ls
  sqrt.c sqrt.o
```

- Il flag `-c` dice al compilatore di fermarsi dopo la fase di compilazione e di non procedere al linking
- Il codice oggetto (`sqrt.o`) contiene il codice macchina, istruzioni che possono essere eseguite su un processore
 - Un file oggetto, tuttavia, non può essere eseguito direttamente
 - Il codice oggetto deve essere collegato (linked) per poter diventare un programma eseguibile da mandare in esecuzione
 - Il linking è il processo che consiste nel mettere insieme pezzi multipli di codice oggetto e di organizzarli in un eseguibile
 - Il codice oggetto può provenire da file di codice sorgente multipli, ognuno compilato nel proprio codice oggetto

Esempio

```
#include <stdio.h>
int OurSquareRoot(int);
int main(int argc, char *argv[])
{
    int x,s;
    printf("Inserisci un intero: ");
    scanf("%d",&x);
    s=OurSquareRoot(x);
    if(s!=-1)
        printf("La radice quadrata di %d: %d\n",x,s);
}
```

Esempio(1)

- Se il codice è memorizzato nel file main.c, lo possiamo compilare come

```
$ gcc -c sqrt.c
```

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

```
$ ls
```

```
main.c main.o sqrt.c sqrt.o
```

```
$ gcc sqrt.o main.o
```

```
$ ls
```

```
a.out main.c main.o sqrt.c sqrt.o
```

```
$
```

Esempio(2)

- Abbiamo creato due file oggetto, main.o e sqrt.o
 - Successivamente, abbiamo creato l'eseguibile, a.out, collegando insieme i due file relativi al codice oggetto
- L'eseguibile è il file che mandiamo in esecuzione

```
$ ./a.out
```

```
Inserisci un intero: 4
```

```
La radice quadrata di 4: 2
```

```
$
```

- Per capire la differenza tra file eseguibile e oggetto

```
$ gcc sqrt.o
```

```
... undefined reference to 'main'
```

```
$
```

- Il compilatore non è in grado di creare l'eseguibile poiché non ha trovato la funzione main() nel file oggetto dato
- Un eseguibile può contenere un qualsiasi numero di funzioni ed esattamente una sola funzione main()
 - In questo modo quando il programma è eseguito sa dove deve cominciare l'esecuzione

Compilazione e linking

- Quando si compila, a meno che non specificato diversamente, gcc procede anche al linking in modo automatico, rimuovendo qualsiasi codice oggetto intermedio
 - Ciò semplifica il lavoro del programmatore che non ha necessità di eseguire separatamente tutti i passi di compilazione e linking
- Tuttavia, quando si compilano file multipli, è conveniente istruire il compilatore in modo che mantenga il codice oggetto
 - Solo i file i cui sorgenti sono modificati devono essere ricompilati
 - Tutto il resto deve solo essere linkato
 - Ciò consente di risparmiare tempo quando il programma è costituito da un numero elevato di file sorgenti

Linking

- Il linking serve innanzitutto per mettere assieme i file oggetto in un eseguibile
 - Serve anche per mettere insieme codice oggetto da file di libreria
 - I file di libreria contengono codice oggetto per funzioni usate in modo frequente
 - In questo modo, il codice sorgente può essere compilato solo una volta e memorizzato in un posto pronto per il link
 - Es.: la libreria più spesso collegata è libc.a, il file di libreria principale per la libreria standard del C
 - Contiene il codice oggetto per le funzioni standard come printf(), strcmp() ecc.
 - Poiché tali funzioni sono usate molto spesso, il loro codice oggetto è tenuto in modo permanente in un file di libreria quindi non necessitano di essere ricomilate ogni volta

Linking ad un file di libreria

```
$ gcc main.o sqrt.o -lc
```

```
$
```

- Il flag `-lc` dice a gcc di linkare un file di libreria chiamato `libc.a`
- Il linking avviene in modo del tutto simile a quanto avviene per i file oggetto
 - **N.B.:** gcc fa il link a `libc.a` di default per cui in generale non è necessario esplicitare `-lc`
 - **-lX** cerca nelle directory standard **`/lib`**, **`/usr/lib`** e **`/usr/local/lib`** un file di libreria chiamato **`libX.a`**
 - **-Lpath** estende la lista di path standard suddetta
- Normalmente, i file oggetto sono copiati nell'eseguibile
 - Se ciò avviene anche per il codice oggetto contenuto nelle librerie il risultato è di avere del codice oggetto ridondante in tutti i file eseguibili che usano la libreria di funzioni

Linking dinamico

- Una libreria può essere collegata in modo dinamico
 - Il codice oggetto non è copiato nell'eseguibile
 - Durante l'esecuzione del programma, se è necessario il codice oggetto da un file di libreria, il codice è caricato in memoria direttamente dal file di libreria

```
$ gcc main.c sqrt.c
```

```
$ ls -al a.out
```

```
-rwxr-xr-x 1 student student 7442 a.out
```

```
$ gcc -static main.c sqrt.c
```

```
$ ls -al a.out
```

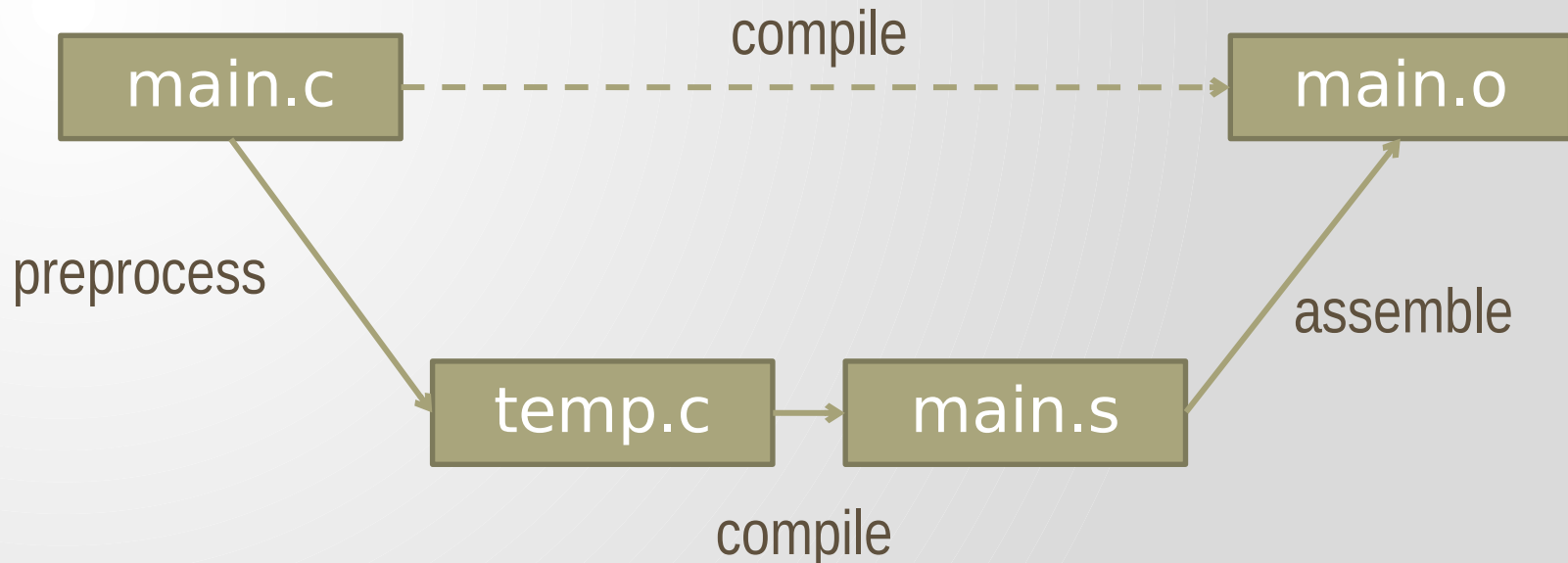
```
-rwxr-xr-x 1 student student 476153 a.out
```

```
$
```

- Nel secondo caso la libreria è linkata in modo statico
 - Tutto il codice oggetto è copiato nell'eseguibile
 - Notevole spreco di spazio, ma
 - Maggiore velocità di esecuzione
 - Se una libreria è rimossa l'eseguibile continua a funzionare normalmente

Compilazione più nel dettaglio

- Il processo consiste di tre passi
 - Preprocessing
 - Compiling
 - Assembling



Compile: preprocessing

- Fornisce il meccanismo per il supporto alla sostituzione di testo
 - Chiamato anche macro o sostituzione di macro

```
// Sorgente: pre1.c
#include <stdio.h>
#define PI 3.14 //direttiva per il preprocessore
#define SQRT2 1.7 //direttiva per il preprocessore

int main(int argc, char *argv[])
{
    printf("PI = %lf\n",PI);
    printf("PI = %lf e la radice quadrata di 2 = %lf\n",PI,SQRT2);
}
```

Compile: preprocessing (2)

```
$ gcc -E pre1.c
```

```
.  
.br/>int main(int argc, char *argv[])  
{  
    printf("PI = %lf\n",3.14);  
    printf("PI = %lf e la radice quadrata di 2 = %lf\n",3.14,1.7);  
}
```

- L'opzione `-E` dice a gcc di fermarsi dopo la fase di preprocessing
 - Possiamo salvare il risultato nel file `temp.c` usando l'opzione `-o` e riprendere la compilazione da questo punto invocando gcc su `temp.c`

```
$ gcc temp.c
```

```
$ ./a.out
```

```
PI = 3.140000
```

```
PI = 3.140000 e la radice quadrata di 2 = 1.700000
```

Compile: preprocessing (3)

- Un altro uso della fase di preprocessing è copiare codice sorgente usato frequentemente
- Supponiamo che il seguente codice sia in un file **globals.h**

```
/* variabili globali */
```

```
int x;
```

- Possiamo usarlo nei file sorgenti come segue

```
/* file pre2.c */
```

```
#include "globals.h"
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    x=2;
```

```
    printf("x=%d\n", x);
```

```
}
```

Compile: preprocessing (4)

```
$ gcc -E pre2.c
.
.
/* variabili globali */
int x;
int main(int argc, char *argv[])
{
    x=2;
    printf("x=%d\n", x);
}
```

- Il contenuto del file `globals.h` è copiato, riga per riga, al posto della riga `#include`
 - Ci sono due posti in cui diciamo al preprocessore di ricercare i file da includere
 - `#include <stdio.h>` // cerca nel path di sistema
 - `#include "globals.h"` // cerca nella dir corrente altrimenti uguale a `<>`

Compile: compilazione

- Conversione del codice sorgente in codice assembly

```
/* file pre3.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int x,y,s;
    y=1;
    s=0;

    for (x=0; x<10; x++)
    {
        y=y<<1;
        s=s+y;
    }
    printf("s=%d\n", s);
}
```


Compile: compilazione (2)

- Compiliamo e fermiamoci subito dopo la traduzione in codice assembly (opzione -S di gcc)

```
$ gcc -S pre3.c
.file "pre.c"
.section .rodata
.LC0:
.string "s=%d\n"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
andl $-16, %esp
subl $32, %esp
movl $1, 24(%esp)
movl $0, 20(%esp)
movl $0, 28(%esp)
jmp .L2
[...]
```

- Il passo finale, assembling, consiste nella traduzione del codice assembly in eseguibile

Debug

Debugger

- Il debugger è un programma. Per illustrarne il funzionamento, consideriamo il seguente codice

```
/* sum.c */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i, sum;
    sum=0;
    for (i=0; i<10; i++)
        if (i<5)
            sum=sum+i;
        else
            sum=sum+((i-3)/2+(i/3));
    printf("sum=%d\n", sum);
}
```

Debugger

- **gdb** è il debugger standard di GNU / Linux
 - il file specificato come argomento è caricato nel debugger
 - vari comandi permettono
 - esecuzione passo-passo (**step**)
 - vedere il valore di una variabile (**print**) ed il tipo (**whatis**)
 - analizzare il sorgente (**list**)
 - stabilire dei breakpoint (**break**)
 - visualizzare espressioni (**display**)
 - cercare funzioni
 - occorre compilare il programma con l'opzione **-g** di **gcc** (inserisce informazioni aggiuntive per il debugger)

Avvio di gdb

- Si compila e poi si invoca il debugger sull'eseguibile

```
$ gcc -g sum.c
```

```
$ gdb a.out
```

```
Reading symbols from /home/giusal/a.out...done.
```

```
(gdb) run
```

```
Starting program: /home/giusal/a.out
```

```
sum=29
```

```
Program exited with code 07.
```

```
(gdb) quit
```

- Durante la creazione dell'eseguibile, il compilatore memorizza informazioni aggiuntive sul programma, la **tabella dei simboli**
 - Include una lista di nomi di variabili usate dal programma, nel nostro caso **i** e **sum**
 - Il programma è anche compilato senza ottimizzazione

gdb: breakpoint

- E' utile eseguire il debug impostando dei **breakpoint**
 - Si dice a gdb di eseguire il programma fino al punto indicato e di mettere in pausa il programma

```
(gdb) break 12
```

```
Breakpoint 1 at 0x80483f1: file sum.c, line 12.
```

```
(gdb) run
```

```
Starting program: /home/giusal/a.out
```

```
Breakpoint 1, main (argc=1, argv=0xbffff4a4) at sum.c:12
```

```
12 sum=sum+((i-3)/2+(i/3));
```

```
(gdb) display i
```

```
1: i = 5
```

```
(gdb) where
```

```
#0 main (argc=1, argv=0xbffff4a4) at sum.c:12
```

```
(gdb)
```

gdb: breakpoint (2)

- Quando il programma è in pausa ci sono tre modi per riavviarlo
- **step**: esegue la prossima riga di codice e rimette in pausa
- **next**: uguale a step ma se la riga successiva è una chiamata a funzione, gdb esegue tutte le istruzioni nella funzione e rimette in pausa dopo il ritorno della stessa
- **continue**: continua l'esecuzione fino al successivo breakpoint
- Per rimuovere i breakpoint si usa il comando **clear**

```
(gdb) step
8 for (i=0; i<10; i++)
(gdb) next
9 if (i<5)
(gdb) next
```

```
Breakpoint 1, main (argc=1, argv=0xbffff4a4) at sum.c:12
12 sum=sum+((i-3)/2+(i/3));
(gdb) next
8 for (i=0; i<10; i++)
(gdb) next
9 if (i<5)
(gdb) next
```

```
Breakpoint 1, main (argc=1, argv=0xbffff4a4) at sum.c:12
12 sum=sum+((i-3)/2+(i/3));
(gdb)
```

gdb: visualizzare le variabili

- Esistono due modi
 - **print**: singola richiesta per esaminare il valore
 - visualizza la variabile solo una volta fino a che non gli è nuovamente richiesto
 - **display**: richiesta continua
 - visualizza il valore della variabile ogni volta che il programma è messo in pausa

```
(gdb) break 12
```

```
Breakpoint 1 at 0x80483f1: file sum.c, line 12.
```

```
(gdb) run
```

```
Starting program: /home/giusal/a.out
```

```
Breakpoint 1, main (argc=1, argv=0xbffff4a4) at sum.c:12
```

```
12 sum=sum+((i-3)/2+(i/3));
```

```
(gdb) display i
```

```
1: i = 5
```

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 1, main (argc=1, argv=0xbffff4a4) at sum.c:12
```

```
12 sum=sum+((i-3)/2+(i/3));
```

```
1: i = 6
```

```
(gdb)
```


Esempio 1: Crash dei programmi

```
/* crash1.c */
#include <stdio.h>
int main()
{
    int x,y;
    y=54389;
    for (x=10;x>=0; x--)
        y=y/x;
    printf("%d\n",y);
}
$ gdb crash1
(gdb) run
Starting program: /home/giusal/crash1
```

```
Program received signal SIGFPE, Arithmetic exception.
0x080483e8 in main () at crash1.c:9
9   y=y/x;
(gdb) display x
1: x = 0
(gdb)
```

Esempio 2: Crash dei programmi

```
/* crash2.c */
#include <stdio.h>
int main()
{
    int x,y,z[3];
    y=1200;
    for (x=10;x>0; x--)
        z[y]=y/x;
    printf("%d\n",z[0]);
}
$ ./crash2
Errore di segmentazione
$ gdb crash2
Reading symbols from /home/giusal/crash2...done.
(gdb) run
Starting program: /home/giusal/crash2

Program received signal SIGSEGV, Segmentation fault.
0x080483f0 in main () at crash1.c:7
7   z[y]=y/x;
(gdb)
```

Esempio 2: Crash dei programmi (2)

- Un *segmentation fault* è un accesso illegale alla memoria
 - Il programma ha cercato di accedere ad una locazione di memoria che non appartiene al programma
 - Es.: un array ha una dimensione specifica. Se si tenta di accedere alle celle al di fuori delle dimensioni specificate si causa un accesso errato alla memoria
 - Ritornando all'esempio, possiamo chiedere di stampare il valore di y

```
(gdb) display y
```

```
1: y = 54389
```

```
(gdb)
```

Esempio 3: ciclo infinito

- Quando un programma è in esecuzione per lungo tempo senza visualizzare nulla di nuovo probabilmente è bloccato in un ciclo infinito
 - Il codice che si esegue nel ciclo non causa mai la condizione di uscita del ciclo

```
/* infloop.c */
#include <stdio.h>
int main()
{
    int x,y=10;
    for (x=0; x<10; x++){
        y=y+x;
        if (y>10)
            x--;
    }
}
```

Esempio 3: ciclo infinito (2)

```
$ gdb infloop
(gdb) run
Starting program: /home/giusal/infloop
^C
Program received signal SIGINT, Interrupt.
main () at infloop.c:7
7   for (x=0; x<10; x++)
(gdb) display x
1: x = 0
(gdb) next
9       y=y+x;
1: x = 0
(gdb) next
10      if (y>10)
1: x = 0
```

Esempio 3: ciclo infinito (3)

```
(gdb) next
```

```
11 x--;
```

```
1: x = 0
```

```
(gdb)
```

```
7 for (x=0; x<10; x++)
```

```
1: x = -1
```

```
(gdb)
```

```
9 y=y+x;
```

```
1: x = 0
```

```
(gdb)
```

- Dopo aver guardato un'iterazione completa del ciclo, ci accorgiamo che la variabile contatore x ha lo stesso valore (zero) all'inizio di ogni iterazione
 - Non raggiunge mai il valore 10, quindi non termina mai il ciclo
 - Ispezioniamo il codice nel ciclo che coinvolge x e risolviamo il problema

Organizzazione del codice

Programma in un singolo modulo

- Inverte la stringa data come argomento ...

```
/* REVERSE.c */
#include <stdio.h>
#include <string.h>

/* Prototipo della funzione */
void reverse ( char *, char * );

int main (int argc, char *argv[]) {
    /* Buffer per la stringa invertita */
    char str [strlen(argv[1])+1];

    /* Inverte la stringa in ingresso */
    reverse (argv[1], str);

    /* Mostra il risultato */
    printf ("reverse (\"%s\") = %s\n", argv[1], str);
    return 0;
}
```


Programma in un singolo modulo (2)

```
void reverse ( char *before, char *after ) {  
    /* before: puntatore alla stringa originaria */  
    /* after: puntatore alla stringa invertita */  
  
    int i, j, len;  
    len = strlen (before);  
    for (j=len-1, i=0; j>=0; j--, i++) /* Ciclo */  
        after[i] = before[j];  
    after[len] = '\0'; /* \0 termina la stringa invertita */  
}
```

Programma in un singolo modulo: problemi

- Realizzare un intero programma come un singolo modulo presenta vari inconvenienti:
 - Ogni (minima) modifica richiede la ricompilazione dell'intero programma
 - tempi di compilazione elevati !!
 - Non è facile riutilizzare funzioni (di utilità generale) definite nel programma (es. **reverse()**)
 - Nota: Per il secondo problema un semplice “cut&paste” delle funzioni è una pessima soluzione per
 - manutenzione: ogni operazione di aggiornamento (es. sostituzione del codice della funzione con uno più efficiente) su ogni copia!
 - efficienza: il cut&paste è lento e ciascuna copia della funzione occupa spazio disco

Suddivisione in più moduli

- Un programma C complesso è normalmente articolato in più file sorgenti
 - compilati indipendentemente
 - collegati in un unico eseguibile
- Risolve i problemi menzionati precedentemente
 - Uno stesso file può essere utilizzato da diversi programmi (funzioni riusabili)
 - La rigenerazione di un eseguibile richiede la ricompilazione dei soli file sorgente modificati ed il linking

Suddivisione in più moduli (2)

- In un programma diviso in più moduli sorgente
 - se un file sorgente utilizza una funzione, non definita nello stesso file, deve contenere la dichiarazione del prototipo della funzione
 - Es: per utilizzare **reverse()** occorre dichiarare
void reverse(char *, char *)
- direttive per il pre-processor / definizioni di tipo devono essere presenti in ogni file che le utilizza
 - Es. (non legate all'esempio)
 - **#define MAX_REVERSABLE 255**
 - **typedef char MyString[MAX_REVERSABLE]**

Suddivisione in più moduli (3)

- Per rendere un modulo **modulo.c** facilmente riusabile si predispose un header file (file di intestazione) **modulo.h** contenente:
 - direttive, definizioni di tipo
 - prototipi delle funzioni definite
- Il file modulo **modulo.c** include il proprio header

```
#include "modulo.h"
```

- Ogni file che utilizza il modulo ne include l'header

```
#include "modulo.h"
```

Suddivisione in più moduli (4)

- Ogni file sorgente può essere compilato separatamente

gcc -c modulo.c

- La compilazione produce il file oggetto **modulo.o** che contiene
 - codice macchina
 - tabella dei simboli
- La tabella dei simboli permette di ricombinare (tramite il compilatore **gcc** o il loader **ld**) il codice macchina con quello di altri moduli oggetto per ottenere file eseguibili

Suddivisione in più moduli: esempio

```
/* reverse.h */
```

```
/* Prototipo della funzione reverse */  
void reverse (char *, char *);
```

```
/* reverse.c */
```

```
#include <stdio.h>  
#include <string.h>  
#include "reverse.h"
```

```
/**  
void reverse ( char *before, char *after ) {  
    /* before: puntatore alla stringa originaria */  
    /* after: puntatore alla stringa invertita */  
    int i, j, len;  
    len = strlen (before);  
    for (j=len-1, i=0; j>=0; j--, i++) /* Ciclo */  
        after[i] = before[j];  
    after[len] = '\0'; /* \0 termina la stringa invertita */  
}
```

Suddivisione in più moduli: esempio (2)

```
/* usaRev.c */
#include <stdio.h>
#include <string.h>
#include "reverse.h" /* contiene il prototipo di reverse */

/*****/
int main (int argc, char *argv[]) {
/* Buffer per la stringa invertita */
char str[strlen(argv[1])+1];
/* Inverte la stringa in ingresso */
reverse (argv[1], str);
/* Mostra il risultato */
printf ("reverse (\"%s\") = %s\n", argv[1], str);
return 0;
}
```


Compilazione separata

- Con l'opzione **-c** ...

```
$ gcc -c reverse.c
```

```
$ gcc -c usaRev.c
```

si generano i file oggetto **reverse.o** e **usaRev.o**

```
$ ls -l
```

```
-rw-r--r-- 1 giusal giusal 517 Apr 27 11:07 usaRev.c  
-rw-r--r-- 1 giusal giusal 1056 Apr 27 11:16 usaRev.o  
-rw-r--r-- 1 giusal giusal 503 Apr 27 11:14 reverse.c  
-rw-r--r-- 1 giusal giusal 99 Apr 27 11:13 reverse.h  
-rw-r--r-- 1 giusal giusal 884 Apr 27 11:14 reverse.o
```

- Alternativamente, con un solo comando

```
$ gcc -c reverse.c usaRev.c
```

Compilazione separata (2)

- Infine il linking dei file oggetto, risolve i riferimenti incrociati e crea un file eseguibile **usaRev**

```
$ gcc reverse.o usaRev.o -o usaRev
$ ls -l usaRev
-rwxr-xr-x 1 giusal giusal 4325 Apr 27 usaRev
$ ./usaRev gatto
reverse ("gatto") = ottag
```

- Se volessimo cambiare **reverse.c** dovremmo ricompilare solo questo file e poi rifare il linking

Esempio riutilizzo di reverse()

- Verifica se una stringa è palindroma

```
/* palindroma.h */
int palindroma (char *);

/* palindroma.c */
#include <string.h>
#include "reverse.h"
#include "palindroma.h"
/*****/
int palindroma (char *str) {
    /* Buffer per la stringa invertita */
    char revstr [strlen(str)+1];

    /* Inverte la stringa in ingresso */
    reverse (str, revstr);
    /* vero se le stringhe sono uguali */
    return !strcmp(str, revstr);
}
```

Esempio riutilizzo di reverse() (2)

- Il main...

```
/* usaPal.c */
#include <stdio.h>
#include "palindroma.h"
/*****/

int main (int argc, char *argv[]) {
    if (palindroma(argv[1]))
        printf ("La stringa \"%s\" e` palindroma.\n", argv[1]);
    else
        printf ("La stringa \"%s\" non e` palindroma.\n", argv[1]);
}
```

Riutilizzo di reverse(): compilazione e linking

```
$ gcc -c palindroma.c
$ gcc -c usaPal.c
$ gcc reverse.o palindroma.o usaPal.o -o usaPal
$ ls -l reverse.o palindroma.o usaPal.o usaPal
-rwxr-xr-x 1 giusal giusal 4596 Apr 27 11:26 usaPal
-rw-r--r-- 1 giusal giusal 1052 Apr 27 11:25 usaPal.o
-rw-r--r-- 1 giusal giusal 892 Apr 27 11:25 palindroma.o
-rw-r--r-- 1 giusal giusal 884 Apr 27 11:14 reverse.o
$ ./usaPal elena
La stringa "elena" non è palindroma
$ ./usaPal anna
La stringa "anna" è palindroma.
```

L'utility make

Gestione delle dipendenze

- La ricompilazione di un programma suddiviso in moduli può essere un'operazione laboriosa: si potrebbe predisporre uno script di shell ...

```
#!/bin/bash
if [ reverse.o -ot reverse.c ] ||
  [ reverse.o -ot reverse.h ]; then
  gcc -c reverse.c
fi

if [ palindroma.o -ot palindroma.c ] ||
  [ palindroma.o -ot palindroma.h ] ||
  [ palindroma.o -ot reverse.h ]; then
  gcc -c palindroma.c
...

```

Progetto

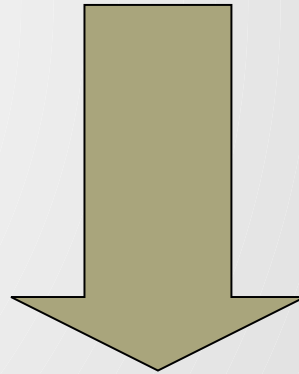
Progetto (2)

- Supponiamo che:
 - **main.o** non esista
 - **prog1.o** e **prog4.o** siano antecedenti all'ultima versione di **prog1.c** e **prog4.c**
 - **prog2.o** e **prog3.o** siano relativi all'ultima versione di **prog2.c** e **prog3.c**

```
$ gcc -c main.c
$ gcc -c prog1.c
$ gcc -c prog4.c
$ gcc -o prog main.o prog1.o ... prog4.o
```

Progetto (3)

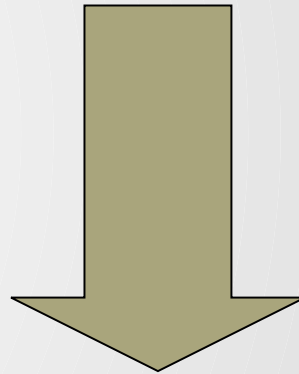
- Se i sorgenti sono modificati frequentemente, allora
 - controllare se qualche sorgente è stato modificato
 - ricompilare gli eventuali file modificati
 - ricostruire l'eseguibile **prog**



operazioni ripetitive!!!

make informalmente

- Unix mette a disposizione una utility apposita
- Il comando make consente la manutenzione e l'aggiornamento dei programmi



- controlla se i file sorgenti sono stati modificati dopo l'ultima modifica
- compila i file sorgenti modificati dopo i corrispondenti file oggetto
- ricostruisce la versione aggiornata di **prog**

make formalmente: gestione dipendenze

make [-f makefile]

- Aggiorna un “progetto” sulla base di regole di dipendenza contenute in un file con un formato speciale detto makefile
 - Per default, make si aspetta che le regole di dipendenza si trovino nel file **Makefile** (anche **makefile**)
 - Con l'opzione **-f** si può specificare un makefile diverso (per convenzione con prefisso “**make**”)

Makefile

- Il makefile contiene un insieme di regole/blocchi del tipo:
targetList: dependencyList
[TAB] commandList
- dove
 - **targetList** contiene una lista di file target
 - **dependencyList** è una lista di file da cui i file target dipendono
 - **commandList** è una lista di zero o più comandi separati da newline (che “ricostruiscono” i target)
 - ogni riga di **commandList** inizia con un **TAB!**
- All'interno di un makefile, si possono inserire commenti preceduti da **#**

Makefile: esempio

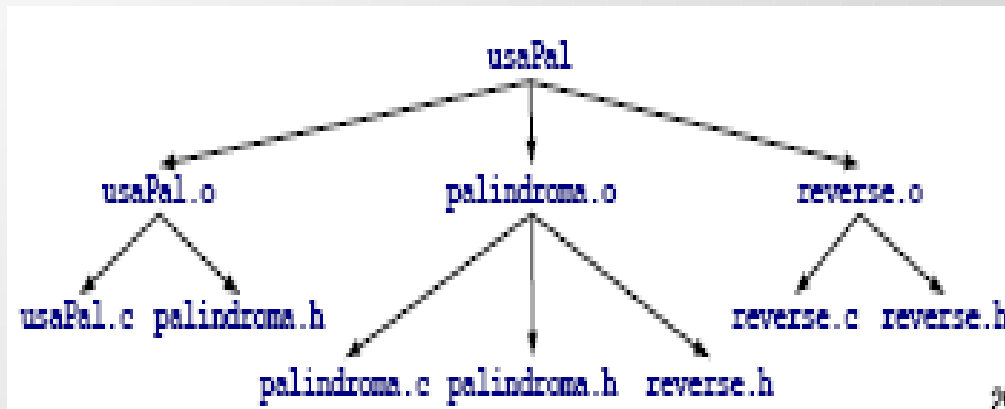
```
# makefile per prog
# versione 1
prog: main.o prog1.o prog2.o prog3.o prog4.o
    gcc main.o prog1.o ... prog4.o -o prog
main.o: main.c
    gcc -c main.c
prog1.o: prog1.c
    gcc -c prog1.c
...
prog4.o: prog4.c
    gcc -c prog4.c
```

Regole del makefile

- Una **linea di dipendenza** definisce la relazione tra un **file dipendente** o target (a sx del carattere:) e uno o più **file di dipendenza**
- Una **linea di comando** o regola, definisce le operazioni che **make** deve effettuare per passare da un **file di dipendenza** al **target**
- Una linea di comando del **makefile** è eseguita se i relativi **file di dipendenza** sono stati modificati più recentemente dei **file dipendenti** associati
- Digitando **make** si ottiene l'esecuzione delle linee di comando relative al primo file dipendente
- Se i **file di dipendenza** sono superati (non esistono), vengono eseguite anche le **linee di comando** necessarie al loro aggiornamento (loro creazione)

Make: funzionamento

- Costruisce un albero delle dipendenze esaminando le regole nell'ordine:
 - ogni file nella **targetList** è radice di un albero i cui figli sono i file nella **dependencyList**
- Visita l'albero in profondità ed esegue la **commandList** associata ad un nodo se:
 - la data di ultima modifica di un figlio è più recente di quella del nodo genitore
 - non esiste il file che etichetta il nodo



Make: regole predefinite

- Non sempre è necessario specificare tutte le dipendenze e le regole
 - **make** è in grado di riconoscere le relazioni esistenti tra alcuni file e di applicarvi regole predefinite
 - **make** riconosce i file con il suffisso **.c** come sorgenti in linguaggio C ed esegue, se necessario, la loro compilazione

```
# makefile di prog
# versione 2
prog: main.o prog1.o ... prog4.o
    gcc main.o ... -o prog
main.o:  main.c
.
.
prog4.o: prog4.c
```

Make: regole predefinite (2)

- Se uno dei file oggetto specificati non è aggiornato
 - **make** ricerca nella directory corrente un file sorgente con lo stesso nome e ne avvia la compilazione

```
# makefile di prog
#versione 3

prog: main.o prog1.o ... prog4.o
    gcc main.o ... -o prog
```

Makefile parametrici

- E' possibile rendere i makefile parametrici impiegando le macro

```
# makefile di prog
# versione 4

OGGETTI = main.o prog1.o ...
prog: $(OGGETTI)
    gcc $(OGGETTI) -o prog
```

- Le macro si usano per definire elenchi di file, opzioni dei compilatori, librerie e comandi

Make: macro

- Una macro è un assegnamento di una stringa ad un nome, che da quel momento la rappresenta

nome = stringa

- Il valore della macro è quindi riferito con ***\$(nome)*** oppure ***\${nome}***

- Le macro si utilizzano spesso per definire path di interesse. Es.

prefix=/usr/local
bin_prefix=\${prefix}/bin

- Possono poi essere personalizzate ...

Esempio: makefile parametrico

```
$ cat makeprova
OGGETTI = qdrig.o pippo.o
OGGETTI1 = qdcol.o pluto.o
MAIN = driver.o
MAIN1 = driver_uno.o
rows: $(OGGETTI) $(MAIN)
      gcc $(OGGETTI) $(MAIN) -o rows
columns: $(OGGETTI1) $(MAIN1)
        gcc $(OGGETTI1) $(MAIN1) -o columns

$ make -f makeprova rows
gcc -c qdrig.c
gcc -c pippo.c
gcc -c driver.c
gcc driver.o qdrig.o pippo.o -o rows

$ make -f makeprova rows
'rows' is up to date.
```

Make: esempio

- Il makefile per il programma **usaPal** (stringhe palindrome) può essere:

```
# Makefile per usaPal, versione 1
```

```
usaPal: usaPal.o palindroma.o reverse.o  
    gcc usaPal.o palindroma.o reverse.o -o usaPal
```

```
usaPal.o: usaPal.c palindroma.h  
    gcc -c usaPal.c
```

```
palindroma.o: palindroma.c palindroma.h reverse.h  
    gcc -c palindroma.c
```

```
reverse.o: reverse.c reverse.h  
    gcc -c reverse.c
```

Regole predefinite: semplificazione

- La presenza delle regole predefinite consente di semplificare il makefile
 - Es. per le stringhe palindrome ...

Makefile per usaPal, versione semplificata

usaPal: usaPal.o palindroma.o reverse.o

gcc usaPal.o palindroma.o reverse.o -o usaPal

usaPal.o: palindroma.h

palindroma.o: palindroma.h reverse.h

reverse.o: reverse.h

- e anche la parte sottolineata può essere rimossa

Makefile tipico

- Il makefile tipico automatizza le fasi legate alla compilazione ed installazione di un programma
- Si distinguono tre **target** comuni
 - **all**: azione da compiere quando non si indica alcun target (tipicamente la compilazione)
 - **install**: installa l'eseguibile dopo la compilazione
 - **clean**: elimina i file oggetto e binari compilati
- Quindi
 - `$ make`
richiama il target **all** e compila il programma
 - `$ make install`
installa gli eseguibili nella destinazione prevista
 - `$ make clean`
pulisce la directory utilizzata per la compilazione

Makefile tipico: esempio

- Makefile per creare / installare **usaRev** e **usaPal**

```
# directory destinazione
prefix=/usr/local
bindir=${prefix}/bin

all: usaPal usaRev

install:
    cp usaPal usaRev ${bindir}
clean:
    rm *.o usaPal usaRev
usaPal: palindroma.o reverse.o
usaPal.o: palindroma.h
palindroma.o: palindroma.h reverse.h

usaRev: reverse.o
usaRev.o: reverse.h
reverse.o: reverse.h
```

Compressione dei file

Archiviazione dei file

- Il programma di servizio **tar** crea archivi contenenti file e directory
 - È possibile archiviare file, aggiornarli all'interno degli archivi ed aggiungere agli archivi nuovi file quando serve
 - E' stato originariamente progettato per creare archivi su nastro ("tar" sta per "tape archive")
 - In Linux tar viene utilizzato per creare archivi su dispositivi o file
 - si utilizza l'opzione **f** ed il nome del dispositivo o file
 - Quando si crea un archivio tar al nome del file viene assegnata l'estensione .tar
 - Se si specifica il nome di una directory, nell'archivio verranno incluse anche tutte le sue sottodirectory

- Esempio:

```
$ tar opzionif nome-archivio.tar nomi-di-directory-e-file
```

Creazione di un archivio

- Per creare un archivio si usa l'opzione **c**
 - Insieme all'opzione **f** crea un archivio su file o dispositivo (è necessario specificare l'opzione **c** prima di **f**)
- Esempio
 - La directory `mydir` e tutte le sue sottodirectory sono salvate nel file `myarch.tar`

```
$ tar cvf myarch.tar mydir
```

- Successivamente l'utente potrà estrarre la directory dall'archivio utilizzando l'opzione **x**

- L'opzione **xf** estrae i file da un file o un archivio. L'operazione di estrazione genera tutte le subdirectory archiviate

```
$ tar xvf myarch.tar
```

Compressione e decompressione

- **tar** è comunemente utilizzato in unione con un'utility esterna di compressione dati, quali ad esempio **gzip**, **bzip2** o, non più in uso, **compress**
 - **tar** da solo non ha la capacità di comprimere i file
 - La versione GNU di tar supporta le opzioni a riga di comando **z** (gzip), **j** (bzip2), e **Z** (compress), che abilitano la compressione per il file che viene creato in un unico passo

Esempi

- Per archiviare e comprimere in due passi:

```
tar cf nome_tarball.tar file_da_archiviare1  
file_da_archiviare2 ...  
gzip nome_tarball.tar
```
- Per vedere il contenuto di un archivio **.tar.gz**:

```
tar tvf  
nome_tarball.tar.gz
```
- Per estrarre i file dall'archivio
 - Semplice file di tar:

```
tar xf nome_tarball.tar
```
 - File compresso (un passaggio alla volta):

```
gunzip nome_tarball.tar.gz  
tar xf nome_tarball.tar
```
- Per utilizzare **bzip2** invece di **gzip**, è sufficiente sostituire con **bzip2** dove viene utilizzato **gzip** e con **bunzip2** dove viene usato **gunzip** nelle varie righe di comando

Compressione e decompressione

- Utilizzando il flag della compressione previsto per il tar di GNU:
 - Per comprimere:
 - Utilizzando **gzip**: `tar czf nome_tarball.tgz file_da_archiviare1 file_da_archiviare2 ...`
 - Utilizzando **bzip2**: `tar cjf nome_tarball.tbz2 file_da_archiviare1 file_da_archiviare2 ...`
 - Utilizzando **compress**: `tar czf nome_tarball.tar.Z file_da_archiviare1 file_da_archiviare2`
 - Per decomprimere ed estrarre i file dall'archivio:
 - Archivio compresso con **gzip**: `tar xzf nome_tarball.tar.gz`
 - Archivio compresso con **bzip2**: `tar xjf nome_tarball.tar.bz2`
 - Archivio compresso con **compress**: `tar xZf nome_tarball.tar.Z`

Compilare i sorgenti del libro (Stevens)

- Scaricare il software (src.tar.gz) dal sito:
<http://www.apuebook.com/>
- Decomprimere il file in una cartella di lavoro (il contenuto è la cartella apue.3e). Ad esempio:
`/home/username/programmi/apue.3e`
- Modificare il file Make.defines.linux
- Compilare tutti i sorgenti digitando make al prompt della shell
`$ make`

Modificare il contenuto di Make.defines.linux

```
# Common make definitions, customized for each platform
# Definitions required in all program directories to compile and
  link
# C programs using gcc.
WKDIR=/home/sar/apue.3e
CC=gcc
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) -c
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDDIR) $(LDFLAGS)
LDDIR=-L../lib
LDLIBS=../lib/libapue.a $(EXTRALIBS)
CFLAGS=-DLINUX -ansi -I$(WKDIR)/include -Wall -D_GNU_SOURCE
  $(EXTRA)

# Our library that almost every program needs.
LIB=../libapue.a

# Common temp files to delete from each directory.
TEMPFILES=core core.* *.o temp.* *.out typescript*
```

Modificare inserendo la directory di lavoro, ovvero la directory dove avete decompresso i sorgenti. Ad esempio, /home/username/programmi/apue.3e

File header (apue.h) e di libreria (libapue.a)

- Per esercitarsi con i singoli programmi:
 - copiate il file **libapue.a**, che trovate in **.../apue.3e/lib**, in **/usr/lib**
 - Copiate **apue.h** in **/usr/include**
- Il file libapue.a è la libreria creata in fase di compilazione con make, che deve essere linkata ogni qualvolta si compilano i file sorgenti
- L'header apue.h contiene molti degli header necessari per eseguire i programmi descritti nel libro, più un insieme di prototipi di funzione e definizioni di costanti