

Programmazione C

Davide Quaglia
a.a. 2010/2011

1

Sommario

- C vs. Java
- Tipi di dati
- Preprocessore C
- Istruzioni
 - Strutture di controllo (condizioni e cicli)
 - I/O
- Sottoprogrammi
- Utilizzo di File
- Strutture dati aggregate
- Altre caratteristiche

2

C vs. Java

- Java: linguaggio ad oggetti
- C: linguaggio procedurale
 - No classi, no oggetti, no costruttori, ...
 - Separazione tra dati e codice
- Diverso flusso di esecuzione
- Sintassi simile
 - Stesse istruzioni
 - Assegnamenti
 - Controllo del flusso
 - Diverse istruzioni di I/O

3

C vs. Java

- Differenza fondamentale
 - A oggetti vs. procedurale!

JAVA

```
public class xyz
{
  <dichiarazione di attributi >
  <dichiarazione di metodi>
}
```

C

```
<dichiarazione di attributi>
<dichiarazione di procedure>
```

4

C vs. Java

- Esempio

```
public class Hw
{
    public static void main(String args[])
    {
        int i;
        System.out.println("Hello World!");
    }
}
```

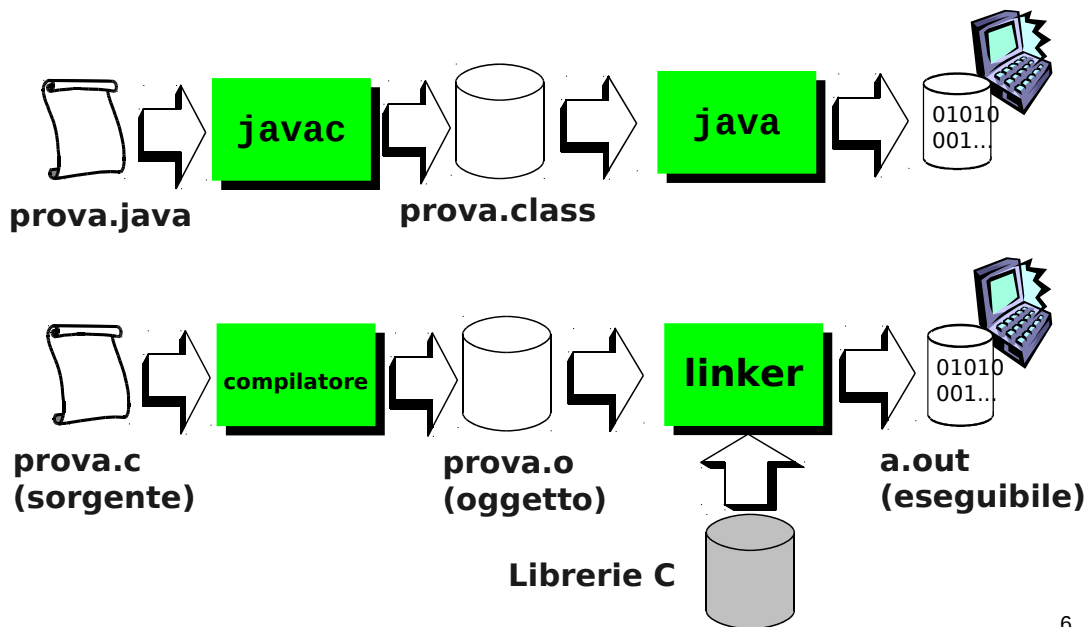
Hw.java

```
void main(int argc, char *argv[])
{
    int i;
    printf("Hello World!");
}
```

Hw.c

5

C vs. Java: flusso di esecuzione



6

C vs. Java: flusso di esecuzione

- Apparente similitudine
- In realtà molto diverso
 - Esecuzione Java = compilatore + interprete (JVM)
 - Esecuzione C = compilatore + linker
 - Linker svolge due funzioni essenziali
 - Collegamento a librerie di codice precedentemente scritto
 - Binding degli indirizzi simbolici in indirizzi rilocabili
 - In ambiente UNIX/Linux, compilazione + link realizzati da un singolo programma (compilatore C): **gcc**

7

Compilatore C

- **gcc**
 - Distribuzione GNU
 - Non è un comando UNIX!
- Uso di base
 - **gcc** <nome sorgente C> (genera direttamente file a.out)
 - Opzioni (combinabili)
 - **gcc -g**: genera le info per il *debugging*
 - **gcc -o** <file>: genera un eseguibile con il nome <file>
 - **gcc -c**: forza la generazione del file .o
 - **gcc -I** <directory>: specifica la directory in cui si trova il sorgente (default directory corrente)
 - **gcc -l**<nome>: specifica il link con la libreria lib<nome>.a

8

Compilatore C

- Esempi:
 - **gcc prova.c**
 - Genera un eseguibile con il nome `a.out`
 - **gcc -o prova.x prova.c**
 - Genera un eseguibile con il nome `prova.x`
 - **gcc -o prova.x -g prova.c**
 - Genera `prova.x` con info di debugging
 - **gcc -c prova.c**
 - Genera il file `prova.o` cioè non effettua il linking
 - **gcc -o prova.g -g -lm prova.c**
Genera un eseguibile con il nome `prova.g`, info di debugging e usando la libreria `libm.a`

9

Struttura di un programma C

- Versione minima

```
Parte dichiarativa globale  
main()  
{  
    Parte dichiarativa locale  
    Parte esecutiva (istruzioni)  
}
```

10

Struttura di un programma C

- Versione più generale

```
Parte dichiarativa globale
main()
{
    Parte dichiarativa locale
    Parte esecutiva (istruzioni)
}
funzione1 ()
{
    Parte dichiarativa locale
    Parte esecutiva (istruzioni)
}
...
funzioneN ()
{
    Parte dichiarativa locale
    Parte esecutiva (istruzioni)
}
```

11

Struttura di un programma C

- Parte dichiarativa globale
 - Elenco dei dati usati in tutto il programma e delle loro caratteristiche (*tipo*)
 - numerici, non numerici
- Parte dichiarativa locale
 - Elenco dei dati usati dalle singole funzioni con il relativo tipo
 - Il **main** è considerato una funzione come le altre anche se ha un nome speciale

12

Preprocessore C

13

Il preprocessore

- La prima fase della compilazione (trasparente all'utente) consiste nell'invocazione del *preprocessore*
- Un programma C contiene specifiche direttive per il preprocessore
 - Inclusioni di file di definizioni (*header file*)
 - Definizioni di costanti
 - Altre direttive
- Individuate dal simbolo '#'

14

Direttive del preprocessore

- **#include**
 - Inclusione di un file di inclusione (tipicamente con estensione `.h`)
 - Esempi
 - `#include <stdio.h>` <- *dalle directory di sistema*
 - `#include "myheader.h"` <- *dalla directory corrente*

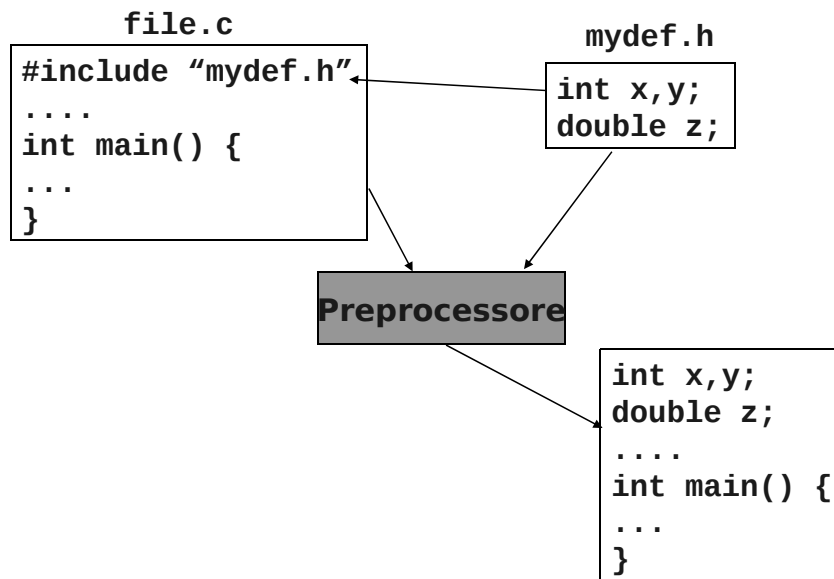
15

Direttive del preprocessore

- **#define**
 - Definizione di un valore costante
 - Ogni riferimento alla costante viene espanso dal preprocessore al valore corrispondente
 - E' buona consuetudine mettere il nome tutto maiuscolo
 - Esempi
 - `#define FALSE 0`
 - `#define SEPARATOR "-----"`

16

La direttiva #include



17

Le variabili

18

Definizione di variabili

- Tutti le variabili devono essere definite prima di essere usate
- Definizione di una variabile
 - riserva spazio in memoria
 - assegna un nome
- Richiede l'indicazione di
 - tipo
 - modalità di accesso (variabili/costanti)
 - nome (identificatore)

19

Tipi base (primitivi)

- Sono quelli forniti direttamente dal C
- Identificati da parole chiave!
 - **char** caratteri ASCII ('a', 'b', '1')
 - **int** interi (complemento a 2)
 - **float** reali (floating point singola precisione)
 - **double** reali (floating point doppia precisione)
- In generale la dimensione precisa di questi tipi dipende dall'architettura (non è definita dal linguaggio C)
 - Eccezione **char** sempre su un byte (8 bit)

20

Modificatori dei tipi base

- Sono previsti dei modificatori, identificati da parole chiave da premettere ai tipi base
- **signed/unsigned**
 - Applicabili ai tipi **char** e **int**
 - **signed**: valore numerico con segno
 - **unsigned**: valore numerico senza segno
- **short/long**
 - 16 (short) / 32 (long)
 - Applicabili al tipo **int**
 - Utilizzabili anche senza specificare **int**

21

Definizione di variabili

- Sintassi
<tipo> <variabile>;
- Sintassi alternativa (definizioni multiple)
<tipo> <lista di variabili>;
 - *<variabile>*: l'identificatore che rappresenta il nome della variabile
 - *<lista di variabili>*: lista di identificatori separati da ','

22

Definizione di variabili

- Esempi:

```
int x;  
char ch;  
long int x1,x2,x3;  
double pi;  
short int stipendio;  
long y,z;
```

- Usiamo nomi significativi!
 - Es: `int x0a11; // NO!`
`int valore; // OK.`

23

Valori costanti

- Valori che rappresentano quantità fisse
- Esempi
 - `char`
 - `'a'`
 - `97`
 - `int, short, long`
 - `26 /* notazione decimale */`
 - `0x1a, 0X1A /* notazione esadecimale */`
 - `032 /* notazione ottale (base 8) */`
 - `26L /* costante forzata a long */`
 - `26U /* costante forzata a unsigned */`
 - `26UL /* costante forzata a unsigned long */`
 - `float, double`
 - `-212.6`
 - `-2.126e2 -2.126E2`
 - `-2.126e-2`

24

Costanti speciali

- Caratteri ASCII non stampabili e/o “speciali”
- Ottenibili tramite “*sequenze di escape*”
 \`<codice ASCII ottale su tre cifre>`
- Esempi
 - ‘\007’
 - ‘\013’
- Caratteri “predefiniti”
 - ‘\b’ **backspace**
 - ‘\f’ **form feed (pagina nuova)**
 - ‘\n’ **line feed (riga nuova)**
 - ‘\t’ **tab (tabulazione)**

25

Definizione di variabili costanti

- Sintassi
[`const`] *<tipo>* *<variabile>* [= *<valore>*] ;
- Esempi
 - `const double pigreco = 3.14159;`
 - `const char separatore = '$';`
 - `const float aliquota = 0.2;`
- Preferibile rispetto all'uso di `#define`
- Convenzione
 - Identificatori delle costanti tipicamente in MAIUSCOLO
 - `const double PIGRECO = 3.14159`

26

Stringhe

- Definizione
 - sequenza di caratteri terminata dal carattere **NULL** ('\0')
- Non è un tipo di base del C
- Costanti stringa
 - “**<sequenza di caratteri>**”
 - Il carattere NULL è già inserito dal compilatore
 - Esempi
 - “Ciao!”
 - “abcdefg\n”

27

Visibilità delle variabili

- Ogni variabile è definita all'interno di un preciso *ambiente di visibilità* (scope)
- *Variabili globali*
 - Definite all'esterno del **main()** e di tutte le altre funzioni
- *Variabili locali*
 - Definite all'interno di un blocco { ... }
 - Es. di blocco
 - funzione main
 - altre funzioni
 - blocchi { ... } all'interno di costrutti **if** e cicli

28

Visibilità delle variabili - Esempio

- **n, x**
visibili in tutto il file
- **a, b, c, y**
visibili in tutto il
main
- **d, z**
visibili nel blocco if

```
int n;  
double x;  
main() {  
    int a,b,c;  
    double y;  
    if (a > b)  
    {  
        int d;  
        double z;  
        .....  
    }  
    .....  
}
```

29

Le istruzioni

30

Istruzioni

- Assegnamenti
 - Come in Java
- Operatori
 - Simili a quelli di Java (non tutti)
- Istruzioni di controllo del flusso
 - Come in Java

31

Le funzioni di input / output

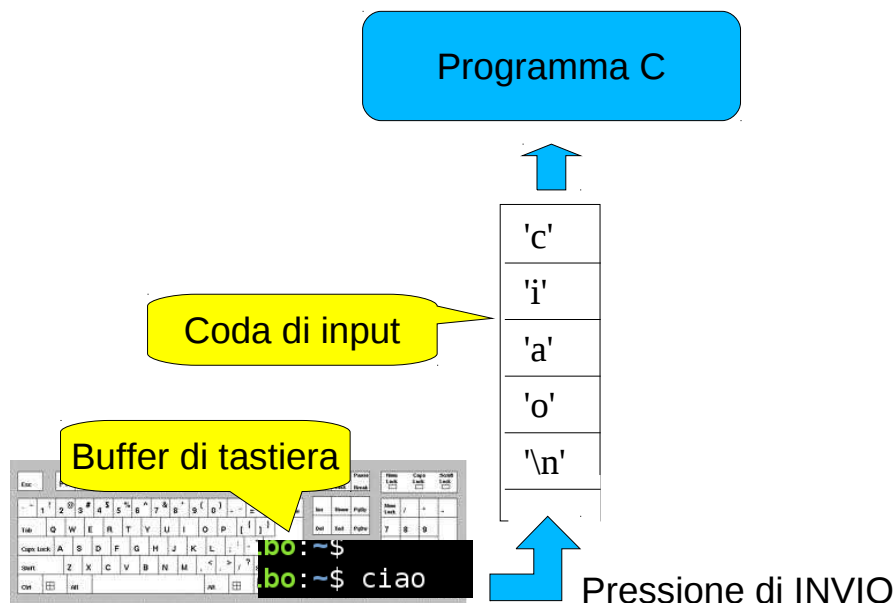
32

Dati in ingresso da tastiera

- I caratteri digitati da tastiera rimangono in un **buffer di tastiera** finché non viene premuto INVIO
 - Possono essere cancellati con BACKSPACE e riscritti
- Alla pressione di INVIO vengono inseriti nella **coda di input**; un carattere in fila a quello successivo compreso il carattere NEWLINE
- Le funzioni di lettura scanf(), getchar(), gets() attingono i caratteri da tale coda consumandoli
 - Se la coda è vuota le funzioni di lettura bloccano il programma in attesa che vengano premuti dei tasti + INVIO
 - I caratteri non letti rimangono in coda in attesa di essere consumati dalle successive chiamate a funzioni di lettura
- La coda di input può essere chiusa con la pressione di CTRL-d
 - nessun nuovo carattere arriverà dal buffer di tastiera per il programma C
 - i caratteri ancora in coda possono essere consumati dalle funzioni di lettura dopodiché queste NON si bloccano in attesa di nuovi caratteri ma restituiscono un msg di errore

33

Dati in ingresso da tastiera



34

La funzione printf()

- `printf(<stringa formato>, <arg1>, . . . , <argn>);`
 - **<stringa formato>**
stringa che determina il formato di stampa di ognuno dei vari argomenti
 - Può contenere
 - Caratteri (stampati come appaiono)
 - Direttive di formato nella forma `%<carattere>`
 - `%d` intero
 - `%u` unsigned
 - `%s` stringa
 - `%c` carattere
 - `%x` esadecimale
 - `%o` ottale
 - `%f` float
 - `%g` double
 - Se il formato di stampa non corrisponde al tipo dell'argomento il risultato potrebbe essere errato

35

La funzione printf()

- **<arg1>, . . . , <argn>**
quantità (espressioni) che si vogliono stampare
 - **Associate alle direttive di formato nello stesso ordine**
- Esempi

```
int x = 2;  
float z = 0.5;  
char c = 'a';
```

```
printf("%d %f %c\n", x, z, c);
```

output

```
2 0.5 a
```

```
printf("%f***%c***%d\n", z, c, x);
```

output

```
0.5***a***2
```

```
printf("%f***%d***%d\n", z, c, x);
```

output

```
0.5***97***2
```

36

La funzione scanf()

- Sintassi

```
scanf(<stringa formato>, <arg1>, . . . , <argn>);
```

- <stringa formato>: come per printf
- <arg1>, . . . , <argn>: le variabili a cui si vogliono assegnare valori
 - **IMPORTANTE: i nomi delle variabili vanno precedute dall'operatore & che indica l'indirizzo della variabile**

- Esempio:

```
int x;  
float z;  
scanf("%d %f", &x, &z);
```

37

La funzione scanf()

- La funzione legge un carattere alla volta dalla coda di input finché tali caratteri corrispondono alla stringa di formato
 - Alla prima violazione la scanf() esce e i caratteri non letti rimangono nella coda di input (verranno letti dalle funzioni di input successive)
- Nella stringa di formato i caratteri *space*, *tab*, *newline* sono equivalenti
 - Definiamolo il generico carattere "spazio"
- La presenza di un carattere "spazio" nella stringa di formato equivale a riconoscere la presenza di "uno o più caratteri spazio"

38

I/O formattato avanzato

- Le direttive della stringa formato di **printf** e **scanf** sono in realtà più complesse

- **printf**

%[flag][min dim][.precisione][dimensione]<carattere>

- **[flag]**: più usati
 - - giustificazione della stampa a sinistra
 - + premette sempre il segno
- **[min dim]**: dimensione minima di stampa in caratteri
- **[precisione]**: numero di cifre frazionarie per numeri reali
- **[dimensione]**: uno tra
 - **h** argomento è short
 - **l** argomento è long
 - **L** argomento è long double

39

I/O formattato avanzato

- **scanf**

%[*][max im][dimensione]<carattere>

- **[*]**: non fa effettuare l'assegnazione (ad es., per "saltare" un dato in input)
- **[max im]**: dimensione massima in caratteri del campo
- **[dimensione]**: uno tra
 - **h** argomento è short
 - **l** argomento è long
 - **L** argomento è long double

40

I/O a caratteri

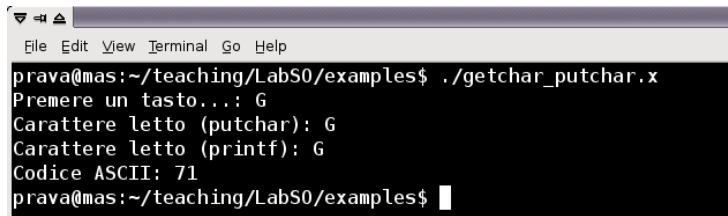
- Acquisizione/stampa di un carattere alla volta
- Istruzioni
 - **getchar()**
 - Legge un carattere dalla coda di input
 - In caso di terminazione della coda (con CTRL-d) il risultato è la costante **EOF** (definita in **stdio.h**)
 - ATT: **getchar** restituisce un valore int e non char in quanto deve restituire o il codice ASCII (numero tra 0 e 255) oppure EOF (non è un carattere ASCII)
 - **putchar(<carattere>)**
 - Stampa <carattere> su schermo
 - <carattere> è un dato di tipo **char**

41

I/O a caratteri - Esempio

```
#include <stdio.h>

int main(int argc, char*argv[]){
    int tasto;
    printf("Premere un tasto...: ");
    tasto = getchar();
    if (tasto != EOF) {
        printf("Carattere letto da getchar: ");
        putchar(tasto);
        putchar('\n');
        printf("Carattere letto da getchar: %c\n", tasto);
        printf("Codice ASCII: %d\n", tasto);
    } else {
        printf("Letto end of file\n"); //è stato premuto CTRL-d
    }
    return 0;
}
```



```
prava@mas:~/teaching/LabS0/examples$ ./getchar_putchar.x
Premere un tasto...: G
Carattere letto (putchar): G
Carattere letto (printf): G
Codice ASCII: 71
prava@mas:~/teaching/LabS0/examples$
```

42

scanf/printf -- getchar/putchar

- **scanf** e **printf** sono “costruite” a partire da **getchar/putchar**
- **scanf/printf** utili quando è noto il formato (tipo) del dato che viene letto
 - Es.: serie di dati tabulati con formato fisso
- **getchar/putchar** utili quando il formato del dato da leggere non è noto
 - Es.: un testo

43

I/O a righe

- Acquisizione/stampa di una riga alla volta
 - Riga = serie di caratteri terminata da ‘\n’
- Istruzioni:
 - **gets(<variabile stringa>)**
 - Legge una riga da tastiera (fino a ‘\n’ o EOF dovuto a CTRL-D)
 - La riga viene fornita come stringa in <variabile stringa> senza il carattere ‘\n’
 - In caso di errore il risultato è la costante NULL (definita in `stdio.h`)
 - **puts(<stringa>)**
 - Stampa <stringa> su schermo
 - Aggiunge sempre ‘\n’ in coda alla stringa

44

I/O a righe

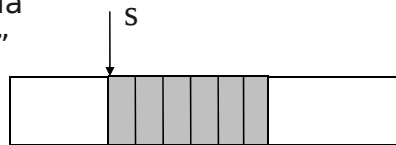
- NOTA:
 - L'argomento di **gets/puts** è di un tipo non primitivo, definito come segue:

char*

- Significato: stringa = vettore di caratteri (**char**)
 - Simbolo '*' indica l'indirizzo di partenza della stringa in memoria
 - Detto "puntatore"

- Esempio

- `char* s;`



45

I/O a righe - Esempio

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    char s[10]; /* non char* s; */
    char *res;
    printf("Scrivi qualcosa\n");
    res = gets(s);
    if (res != NULL) /* errore ? */
    {
        puts("Hai inserito: ");
        puts(s);
    }
}
```

46

I/O a righe

- NOTE
 - **puts/gets** sono “costruite” a partire da **getchar/putchar**
 - Uso di **gets** richiede l’allocazione dello spazio di memoria per la riga letta in input
 - ATTENZIONE: se la riga in input contiene più caratteri di quelli allocati il programma termina con errore!
 - **puts(s)** è identica a **printf(“%s\n”, s);**
- Usate meno di frequente degli altre istruzioni di I/O

47

Le funzioni

48

Funzioni

- Un programma C consiste di una o più funzioni
 - Almeno `main()`
- Funzionamento identico ai metodi
- Definizione delle funzioni
 - Prima della definizione di `main()`
 - Dopo della definizione di `main()` → necessario premettere in testa al file il *prototipo* della funzione
 - Nome
 - Argomenti
 - I prototipi si possono raccogliere in header file da includere con la direttiva `#include`

49

Funzioni e prototipi: esempio

```
double f(int x)
{
    ...
}
int main ()
{
    ...
    z = f(y);
    ...
}
```

```
double f(int);
int main ()
{
    ...
    z = f(y);
    ...
}
double f(int x)
{
    ...
}
```

dichiarazione

definizione

50

Passaggio dei parametri

- In C, il passaggio dei parametri avviene *per valore*
 - Significato: Il valore dei parametri attuali viene copiato in variabili locali della funzione
- Implicazione
 - I parametri attuali non vengono MAI modificati dalle istruzioni della funzione

51

Passaggio dei parametri - Esempio

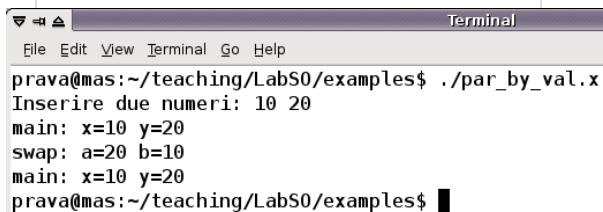
```
#include<stdio.h>
#include<stdlib.h>

void swap(int a,int b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
    printf("swap: a=%d
           b=%d\n", a, b);}

```

```
int main(int argc, char*
         argv[]) {
    int x,y;
    printf("Inserire due
           numeri: ");
    scanf("%d %d",&x,&y);
    printf("main: x=%d
           y=%d\n", x, y);
    swap(x,y);
    /* x e y NON VENGONO
       MODIFICATI */
    printf("main: x=%d
           y=%d\n", x, y);}

```



```
prava@mas:~/teaching/Lab50/examples$ ./par_by_val.x
Inserire due numeri: 10 20
main: x=10 y=20
swap: a=20 b=10
main: x=10 y=20
prava@mas:~/teaching/Lab50/examples$
```

52

Passaggio dei parametri

- Per modificare i parametri
 - Passaggio *per indirizzo (by reference)*
 - parametri attuali = indirizzi di variabili
 - Ottenibile con l'operatore '&' da premettere al nome della variabile
 - parametri formali = *puntatori* al tipo corrispondente dei parametri attuali
- Concetto
 - *Passando gli indirizzi dei parametri attuali posso modificarne il valore*

53

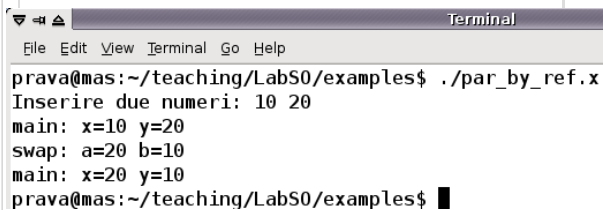
Passaggio dei parametri - Esempio

```
#include<stdio.h>
#include<stdlib.h>
void swap(int *a,int *b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
    printf("swap: a=%d
           b=%d\n", *a, *b);}

```

```
int main(int argc, char*
          argv[]) {
    int x,y;
    printf("Inserire due
           numeri: ");
    scanf("%d %d",&x,&y);
    printf("main: x=%d
           y=%d\n", x, y);
    swap(&x,&y);
    /* x e y ORA VENGONO
       MODIFICATI */
    printf("main: x=%d
           y=%d\n", x, y);}

```



```
Terminal
File Edit View Terminal Go Help
prava@mas:~/teaching/LabS0/examples$ ./par_by_ref.x
Inserire due numeri: 10 20
main: x=10 y=20
swap: a=20 b=10
main: x=20 y=10
prava@mas:~/teaching/LabS0/examples$ █
```

54

Funzioni di libreria

- Il C prevede numerose funzioni predefinite per scopi diversi
- Particolarmente utili sono
 - Funzioni matematiche
 - Funzioni di utilità
- Definite in specifiche *librerie*
- Tutte descritte nel *man*

55

Funzioni matematiche

- Utilizzabili con `#include <math.h>`

| <i>funzione</i> | <i>definizione</i> |
|--|-----------------------------|
| <code>double sin (double x)</code> | seno |
| <code>double cos (double x)</code> | coseno |
| <code>double tan (double x)</code> | tangente |
| <code>double asin (double x)</code> | arcoseno |
| <code>double acos (double x)</code> | arcoseno |
| <code>double atan (double x)</code> | arcotangente |
| <code>double atan2 (double y, double x)</code> | <code>atan (y / x)</code> |
| <code>double sinh (double x)</code> | seno iperbolico |
| <code>double cosh (double x)</code> | coseno iperbolico |
| <code>double tanh (double x)</code> | tang. iperbolica |

56

Funzioni matematiche (2)

- Utilizzabili con `#include <math.h>`

| funzione | definizione |
|---|-----------------------|
| <code>double pow (double x, double y)</code> | x^y |
| <code>double sqrt (double x)</code> | radice quadrata |
| <code>double log (double x)</code> | logaritmo naturale |
| <code>double log10 (double x)</code> | logaritmo decimale |
| <code>double exp (double x)</code> | e^x |
| <code>double ceil (double x)</code> | <code>ceil(x)</code> |
| <code>double floor (double x)</code> | <code>floor(x)</code> |
| <code>double fabs (double x)</code> | valore assoluto |
| <code>double fmod (double x, double y)</code> | modulo |

57

Funzioni di utilità

- Varie categorie
 - Classificazione caratteri
`#include <ctype.h>`
 - Funzioni matematiche intere
`#include <stdlib.h>`
 - Stringhe
`#include <string.h>`

58

Funzioni di utilità

- Classificazione caratteri

| <i>funzione</i> | <i>definizione</i> |
|-----------------------------------|-------------------------------|
| <code>int isalnum (char c)</code> | Se c è lettera o cifra |
| <code>int isalpha (char c)</code> | Se c è lettera |
| <code>int isascii(char c)</code> | Se c è lettera o cifra |
| <code>int islower(char c)</code> | Se c è minuscola |
| <code>int isdigit (char c)</code> | Se c è cifra |
| <code>int isupper (char c)</code> | Se c è maiuscola |
| <code>int isspace(char c)</code> | Se c è spazio,tab,\n |
| <code>int iscntrl(char c)</code> | Se c è di controllo |
| <code>int isgraph(char c)</code> | Se c è stampabile, non spazio |
| <code>int isprint(char c)</code> | Se c è stampabile |
| <code>int ispunct(char c)</code> | Se c è di interpunzione |

59

Funzioni di utilità

- Funzioni matematiche intere

| <i>funzione</i> | <i>definizione</i> |
|--|--------------------------------------|
| <code>int abs (int n)</code> | valore assoluto |
| <code>long labs (long n)</code> | valore assoluto |
| <code>div_t div (int numer, int denom)</code> | quoto e resto della divisione intera |
| <code>ldiv_t ldiv(long numer, long denom)</code> | quoto e resto della divisione intera |

Nota: `div_t` e `ldiv_t` sono di un tipo aggregato particolare fatto di due campi (int o long a seconda della funzione usata):

```
quot /* quoziente */  
rem /* resto */
```

60

string.h

- Funzioni per Stringhe

| <i>funzione</i> | <i>definizione</i> |
|---|---------------------------|
| <code>char* strcat (char* s1, char* s2);</code> | concatenazione di s1 e s2 |
| <code>char* strchr (char* s, int c);</code> | trova c dentro s |
| <code>int strcmp (char* s1, char* s2);</code> | confronto |
| <code>char* strcpy (char* s1, char* s2);</code> | copia s2 in s1 |
| <code>int strlen (char* s);</code> | lunghezza di s |
| <code>char* strncat (char* s1, char* s2, int n);</code> | concat. n car. max |
| <code>char* strncpy (char* s1, char* s2, int n);</code> | copia n car. max |
| <code>int strncmp(char* dest, char* src, int n);</code> | cfr. n car. max |

61

l file

62

File sequenziali

- Accesso tramite variabile di tipo *stream* (flusso)
- Definita in **stdio.h**
- Definizione

FILE *<identificatore>;

- Al momento dell'attivazione di un programma vengono automaticamente attivati (aperti) tre stream
 - **stdin**
 - **stdout**
 - **stderr**

63

File sequenziali

- **stdin** automaticamente associato allo standard input (tastiera)
- **stdout** e **stderr** automaticamente associati allo standard output (video)
- **stdin, stdout, stderr** direttamente utilizzabili nelle istruzioni per l'accesso a file

64

Apertura di un file

- Per accedere ad un file è necessario aprirlo:
 - Apertura = connessione di un file fisico (su disco) ad un file logico (stream)
`FILE* fopen(char* <nomefile>, char* <modo>);`
 - <nomefile>: nome del file fisico (eventualmente con percorso assoluto o relativo rispetto alla cartella di lancio del programma)
 - <modo>: il tipo di accesso al file
 - "r": sola lettura
 - "w": sola scrittura (cancella il file se esiste)
 - "a": *append* (aggiunge in coda ad un file o crea file se non esiste)
 - "r+": lettura/scrittura su file esistente
 - "w+": lettura/scrittura (cancella il file se esiste o crea file se non esiste)
 - "a+": lettura/scrittura-in-coda (crea file se non esiste)
 - ritorna il puntatore allo stream in caso di successo, altrimenti ritorna **NULL**

65

Chiusura di un file

- Quando l'utilizzo del file fisico è terminato, è consigliabile chiudere il file
 - Chiusura: cancellazione della connessione di un file fisico
- Funzione
`int fclose(FILE* <stream>);`
 - <stream>: uno stream aperto in precedenza con `fopen()`
 - Valore di ritorno
 - 0 se ritorna correttamente
 - EOF in caso di errore

66

Apertura e chiusura di un file

- Esempio

```
...
FILE *fp; /* variabile di tipo stream */
...
fp = fopen("testo.dat", "r");
/* apro 'testo.dat' in lettura*/
if (fp == NULL)
    printf("Errore nell'apertura\n");
else {
    /* qui posso accedere a 'testo.dat' usando fp */
    fclose(fp);
}
...
```

67

Lettura a caratteri

- `int getc (FILE* <stream>);`
- `int fgetc (FILE* <stream>);`
 - Legge un carattere alla volta dallo stream
 - Restituisce il carattere letto o **EOF** in caso di fine file o errore
- **NOTA:** `getchar()` equivale a `getc(stdin)` e `fgetc(stdin)`

68

Scrittura a caratteri

- `int putc (int c, FILE* <stream>);`
- `int fputc (int c, FILE* <stream>);`
 - Scrive un carattere alla volta sullo stream
 - Restituisce il carattere scritto o **EOF** in caso di errore
- **NOTA:** `putc()` equivale a `putc(stdout)`

69

Lettura a righe

- `char* fgets(char* <s>, int <n>, FILE* <stream>);`
 - n è la lunghezza del vettore
 - Legge una riga dallo stream fermandosi al più dopo $n-1$ caratteri
 - L'eventuale `'\n'` NON viene eliminato (diverso da `gets` !)
 - Restituisce il puntatore alla stringa o `NULL` in caso di fine file o errore

70

Scrittura a righe

- `int fputs(char* <s>, FILE* <stream>);`
 - Scrive la stringa **s** sullo stream senza aggiungere `'\n'` (diverso da `puts` !)
 - Restituisce l'ultimo carattere scritto, oppure **EOF** in caso di errore

71

Lettura formattata

- `int fscanf(FILE* <stream>, char *<formato>, ...);`
 - Come `scanf()`, con un parametro addizionale che rappresenta uno stream
 - Restituisce il numero di campi convertiti, oppure **EOF** in caso di fine file

72

Scrittura formattata

- `int fprintf(FILE* <stream>, char *<formato>, ...);`
 - Come `printf()`, con un parametro aggiuntivo che rappresenta uno stream
 - Restituisce il numero di byte scritti, oppure **EOF** in caso di errore

73

I tipi di dato strutturati

74

Tipi strutturati

- In C è possibile definire dati composti da elementi eterogenei (*record*), aggregandoli in una singola variabile
 - Individuata dalla keyword **struct**
 - Simile alla classe (ma no ha metodi!)
- Sintassi (definizione di tipo)

```
struct <identificatore> {  
campi  
};
```

I campi sono nel formato

```
<tipo> <nome campo>;
```

75

struct - Esempio

```
struct complex {  
  double re;  
  double im;  
};  
  
struct identity {  
  char nome[30];  
  char cognome[30];  
  char codicefiscale[15];  
  int altezza;  
  char stato_civile;  
};
```

76

struct

- Un definizione di struct equivale ad una definizione di tipo
- Successivamente una struttura può essere usata come un tipo per definire variabili
- Esempio

```
struct complex {  
double re;  
double im;  
};  
...  
struct complex num1, num2;
```

77

Accesso ai campi

- Una struttura permette di accedere ai singoli campi tramite l'operatore '.', applicato a variabili del corrispondente tipo struct
<variabile>.<campo>

```
• Esempio  
struct complex {  
double re;  
double im;  
}  
...  
struct complex num1, num2;  
num1.re = 0.33; num1.im = -0.43943;  
num2.re = -0.133; num2.im = -0.49;
```

78

Definizione di struct come tipi

- E' possibile definire un nuovo tipo a partire da una **struct** tramite la direttiva **typedef**
 - Passabili come parametri
 - Indicizzabili in vettori
- Sintassi
typedef **<vecchio tipo>** **<nuovo tipo>;**

- Esempio
typedef **struct complex** {

double re;

double im;

} complex_t;

—————>

complex_t z1, z2;

79

I puntatori

80

Puntatori

- Un puntatore è una variabile che contiene un indirizzo di memoria
- Esempio:

`int *a;` `a` è un puntatore a una variabile di tipo intero

`int b;` `b` è una variabile di tipo intero

ATT: `a` DEVE essere inizializzata

`a = 3;` illegale perché `a` è un puntatore

`a = &b;` ad `a` viene assegnato l'indirizzo di `b`

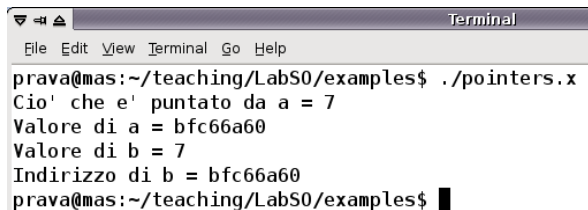
`*a = 3;` legale perché `a` punta a `b`, il valore di `b` viene modificato (`b=3`)

81

Puntatori

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
    int *a;
    int b;
    a = &b;
    *a = 7;
    printf("Cio' che e' puntato da a = %d\n", *a);

    printf("Valore di a = %x\n", a);
    printf("Valore di b = %d\n", b);
    printf("Indirizzo di b = %x\n", &b);
}
```



```
Terminal
File Edit View Terminal Go Help
prava@mas:~/teaching/LabS0/examples$ ./pointers.x
Cio' che e' puntato da a = 7
Valore di a = bfc66a60
Valore di b = 7
Indirizzo di b = bfc66a60
prava@mas:~/teaching/LabS0/examples$ █
```

82

Stringhe

- Vettori di caratteri terminati da un carattere aggiuntivo '\0' (NULL)
- Memorizzate come i vettori

83

Stringhe

- Modi di dichiarazione
 - `char s[5]; // non inizializzata`
 - `char s[] = "ciao!";`

| | | | | | |
|-----|-----|-----|-----|-----|------|
| 'c' | 'i' | 'a' | 'o' | '!' | '\0' |
|-----|-----|-----|-----|-----|------|

s[0] s[1] s[2] s[3] s[4] s[5]

- NOTA: la stringa vuota non è un vettore "vuoto"!
 - Es: `char s[] = "";`

| |
|------|
| '\0' |
|------|

s[0]

84

Stringhe, vettori e puntatori

- Esiste uno stretto legame tra stringhe e puntatori, come per qualsiasi altro vettore
Nome del vettore = indirizzo del primo elemento
- Es: `char nome[20];`
 - `*nome` equivale a `nome[0]`
 - `nome` equivale a `&nome[0]`
- NOTA: Per effettuare qualsiasi operazione su stringhe è necessario utilizzare le funzioni di libreria

85

Puntatori a stringhe

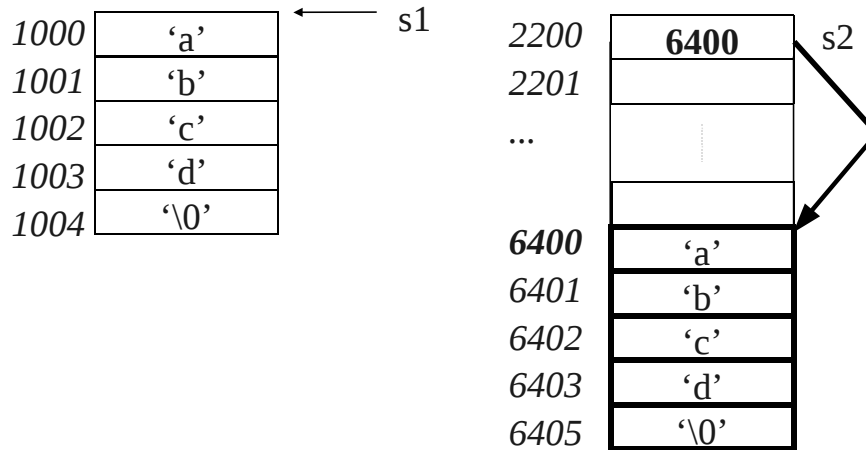
- Dichiarazioni di puntatori a stringa
 - `char s[];`
 - `char *s;`

NOTA: non riservano memoria !!!
- Lo stesso formato si usa nei prototipi di funzioni che ricevono una stringa come parametro (la memoria deve essere riservata dal chiamante)
 - `int f(char s[])`
 - `int f(char *s)`

86

Stringhe, vettori e puntatori

```
char s1[] = "abcd";  
char *s2 = "abcd";
```



87

Stringhe, vettori e puntatori

```
#include<stdio.h>  
#include<stdlib.h>  
  
int main(int argc, char* argv[]){  
    char s1[] = "abcd"; char *s2 = "efgh";  
    int i;  
    printf("Stringa s1 = %s\n", s1);  
    printf("Stringa s2 = %s\n", s2);  
    printf("Modifico s1...\n");  
    for(i=0;i<4;i++) s1[i] = s1[i]+1;  
    printf("... valore modificato = %s\n", s1);  
    printf("Modifico s2...\n");  
    for(i=0;i<4;i++) s2[i] = s2[i]+1;  
    printf("... valore modificato = %s\n", s2);  
}
```

```
Terminal  
File Edit View Terminal Go Help  
prava@mas:~/teaching/LabS0/examples$ ./array_and_pointer.x  
Stringa s1 = abcd  
Stringa s2 = efgh  
Modifico s1...  
... valore modificato = bcde  
Modifico s2...  
Segmentation fault  
prava@mas:~/teaching/LabS0/examples$ █
```

88

Puntatori e struct

- E' tipico in C utilizzare le **struct** tramite variabili di tipo **struct*** (puntatori a struttura)
 - Particolarmente vantaggioso nel caso di **struct** che sono argomenti a funzione
- Accesso ai campi tramite operatore '**->**'

89

Puntatori e struct - Esempio

```
struct complex {  
double re;  
double im;  
}  
  
struct complex *num1;  
struct complex num2;  
/* num1 è puntatore a struct, num2 è struct */  
  
num1 = &num2;  
  
num1->re = 0.33;  
num1->im = -0.43943;  
  
num2.re = -0.133;  
num2.im = -0.49;
```

90

Parametri del main

Come in JAVA è possibile passare dei parametri direttamente dalla linea di comando

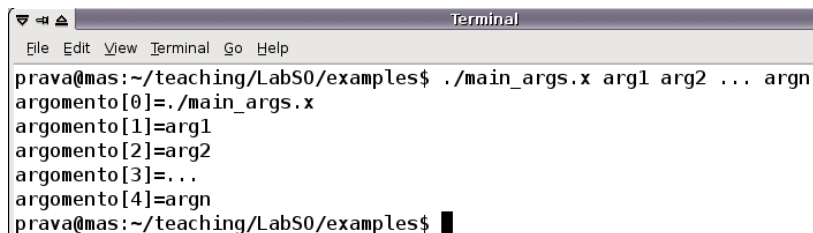
Sintassi

- Numero di parametri: **argc**
- Vettore di parametri: **argv[]**
- **argv[0] = nome del programma**

91

Parametri al main - Esempio

```
void main(int argc, char *argv[]){
    int t;
    for(i = 0; i < argc; i++)
        printf("argomento[%d]=%s\n", i, argv[i]);
}
```



A terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Go, Help). The prompt is "prava@mas:~/teaching/LabS0/examples\$". The user enters the command "./main_args.x arg1 arg2 ... argn". The output shows the program printing the arguments: "argomento[0]=./main_args.x", "argomento[1]=arg1", "argomento[2]=arg2", "argomento[3]=...", and "argomento[4]=argn". The prompt returns to "prava@mas:~/teaching/LabS0/examples\$".

92

Memoria dinamica

93

Memoria dinamica

- E' possibile creare strutture dati allocate nella memoria dinamica del processo (*heap*)
 - Allocazione al tempo di esecuzione
- Funzione **malloc()**
 - **void* malloc(int <numero di byte>)**
 - **Per allocare il tipo desiderato si usa l'operatore di cast ()**
- Richiede `#include <stdlib.h>`
- Due usi
 - Allocazione dinamica di vettori
 - Allocazione dinamica di strutture

94

Vettori e malloc

- Bisogna dichiarare la dimensione del vettore
 - Esempi
 - per allocare un vettore di 10 caratteri

```
char* s;  
s = (char*) malloc(10);
```
 - per allocare un vettore di n caratteri

```
char* s;  
int n = atoi(argv[1]);  
s = (char*) malloc(n);
```

95

Strutture e malloc

- Bisogna sapere quanto spazio (byte) occupa la struttura
- Funzione **sizeof(<tipo>)**
 - Ritorna il numero di byte occupati da <tipo>
- Esempio

```
typedef struct {  
float x,y;  
} Point;  
  
Point* segment;  
segment = (Point*) malloc(2*sizeof(Point));  
/* vettore di due variabili point */  
/* equivalente a  
Point segment[2] */  
if (segment == NULL)  
{ fprintf(stderr, "Out of memory\n");  
return -1;  
}  
...
```

96

Liberare la memoria

- E' buona regola "liberare" sempre la memoria allocata con la **funzione free()**
 - `free(<puntatore allocato con malloc>);`

- Esempio

```
segment = (Point*)malloc(2*sizeof(Point));
if (segment == NULL)
{ fprintf(stderr, "Out of memory\n");
  return -1;
}
...
free(segment);
```

97

Vettori e matrici

- Dichiarazione nei **prototipi**

`float matrice[]` oppure `float *matrice` // 1 dimensione

`float matrice[][COLS]` oppure `float *matrice[COLS]` // 2 dim

`float matrice[][N][M]` oppure `float *matrice[N][M]` // 3 dim

- Le n-1 dim esplicite servono al compilatore per la trasformazione

`matrice[i][j] ---> matrice[i*num_colonne + j]`

98

Vettori e matrici

□ Allocazione statica

```
float matrice[15] // 1 dimensione
float matrice[10][10] // 2 dim
float matrice[12][16][19] // 3 dim
```

□ Allocazione dinamica

```
float *matrice; // indep. dalle dimensioni
matrice=(float*)malloc(dim1*...*dimN*sizeof(float));
if (matrice == NULL)
{
    fprintf(stderr, "Out of memory\n");
    return -1; //oppure exit(-1);
}
...
free(matrice);
```

99

Vettori e matrici

□ Accesso ad una matrice **bidimensionale** allocata staticamente

```
float matrice[10][10] // 2 dim
...
matrice[i][j] = 3.14;
```

□ Accesso ad una matrice **bidimensionale** allocata dinamicamente

```
float *matrice;
...
matrice[i*num_colonne+j] = 3.14;
```

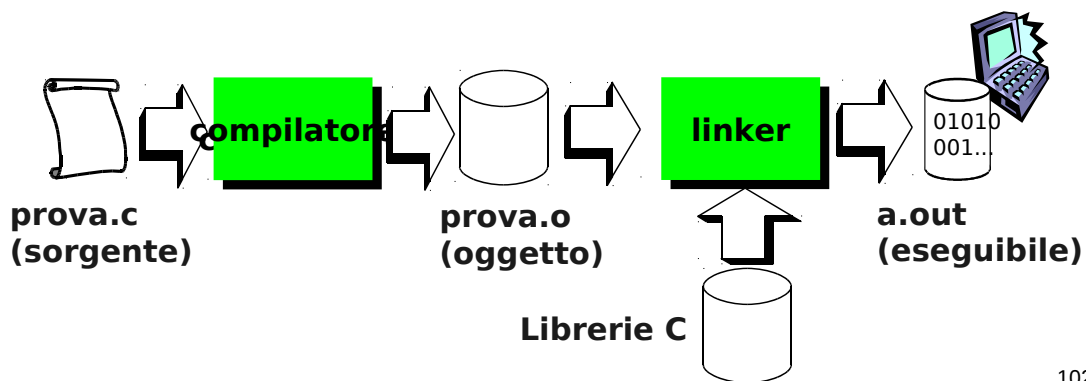
100

Le librerie

101

Le librerie

- Quando un file viene compilato, dopo la fase di “LINK” ho a disposizione l'eseguibile, per il sistema operativo desiderato



102

Le librerie

- L'eseguibile può contenere tutto il codice delle librerie necessario per l'esecuzione (link statico), oppure contenere solo i riferimenti ai file di libreria (link dinamico)
- Se l'eseguibile è "linkato dinamicamente" è necessario che siano state "installate" tutte le librerie richieste dall'eseguibile

103

Le librerie

- Librerie dinamiche: .so
- Librerie statiche: .a
- Comando ldd
 - E' possibile vedere le librerie dinamiche richieste da un eseguibile

```
prava@mas:~/teaching/LabS0/examples$ ldd /usr/bin/gcc
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e81000)
/lib/ld-linux.so.2 (0xb7fb8000)
prava@mas:~/teaching/LabS0/examples$ ldd /usr/bin/startx
not a dynamic executable
prava@mas:~/teaching/LabS0/examples$
```

104

Le librerie

- Comando nm
 - E' possibile vedere il contenuto (simboli) delle librerie dinamiche e statiche
- Comando ar
 - E' possibile creare una libreria statica (.a) unendo un insieme di file .o
- Comando ld
 - E' il linker. Può essere usato anche per creare librerie dinamiche (.so)

105

Le librerie

- Per specificare dove si trovano le librerie esistono due sistemi di configurazione
 - /etc/ld.so.conf
 - LD_LIBRARY_PATH
- gcc -l<nome>
specifica il link con la libreria lib<nome>.a

106

Compilazione di progetti software complessi

107

Make

- È possibile compilare un insieme di file con il comando make
- Il make si basa su un file Makefile che contiene le direttive di compilazione dei vari file
- Il make sfrutta il concetto di dipendenza e i marcatori temporali dei file C per decidere cosa compilare

108

Il Makefile - target e regole

- E' un elenco di target e regole corrispondenti

Target 1: lista dei file da analizzare

<tabulazione> Regola

Target 2: lista dei file da analizzare

<tabulazione> Regola

...

- make
 - Invoca il primo target
- make <nome_target>
 - Invoca il target <nome_target>

109

Il Makefile - Esempio

Esempio

STRINGA = TUTTI E TRE

one:

 @echo UNO

two:

 @echo DUE

three:

 @echo TRE

all: one two three

 @echo \$(STRINGA)

```
prava@mas:~/teaching/LabS0/examples$ make -f makefile_echo
UNO
prava@mas:~/teaching/LabS0/examples$ make -f makefile_echo two
DUE
prava@mas:~/teaching/LabS0/examples$ make -f makefile_echo all
UNO
DUE
TRE
TUTTI E TRE
prava@mas:~/teaching/LabS0/examples$
```

110

Il Makefile - Struttura

Questo è un commento

VARIABILE = valore

#\$ (VAR) espande la variabile VAR in valore

alcune macro predefinite

\$@ : L'eseguibile

\$^ : lista delle dipendenze

\$< : prima dipendenza

\$? : file modificati

% : file corrente

111

Il makefile - Esempio (2)

OBJECTS=main.o myfunc.o

CFLAGS=-g

LIBS=-lm

CC=gcc

PROGRAM_NAME=prova

.SUFFIXES : .o .c

.c.o :

\$(CC) \$(CFLAGS) -c \$<

\$(PROGRAM_NAME) : \$(OBJECTS)

\$(CC) \$(CFLAGS) -o \$(PROGRAM_NAME) \$(OBJECTS) \$(LIBS)

@echo "Compilazione completata!"

clean :

rm -f *.o prova core

112

Debugging

113

Il debugging

- E' possibile eseguire/provare un programma passo/passo per analizzare errori run-time tramite un debugger
- GDB (gnu debug) è il debugger della GNU
 - Esistono numerosi front-end grafici per il GDB, il più famoso è sicuramente il DDD
- Per poter analizzare un eseguibile bisogna compilarlo con
 - gcc -g: genera le info per il debugging
 - gcc -ggdb: genera le info per il debugging GDB

114