

Programmazione C

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

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
 - Assegnazioni
 - 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!");  
    }  
}
```

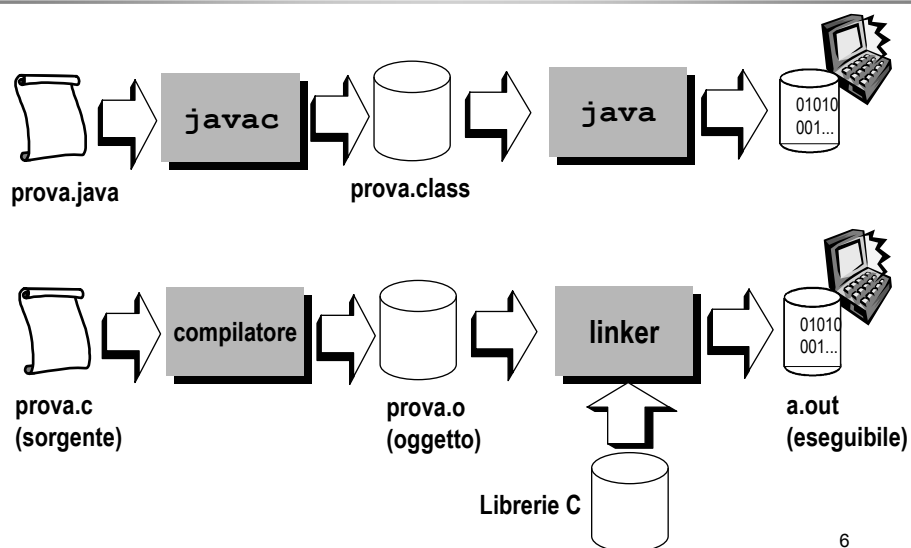
Hw.java

```
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 -g prova.c**
 - Genera a.out con info di debugging
- **gcc -o prova prova.c**
 - Genera un eseguibile con il nome **prova**
- **gcc -c prova.c**
 - Genera il file **prova.o**
- **gcc -o prova -g -lm**
 - Genera un eseguibile con il nome **prova**, 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 dal **main** o dalle singole funzioni, con il relativo tipo

12

Il preprocessore C

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 '#'

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*

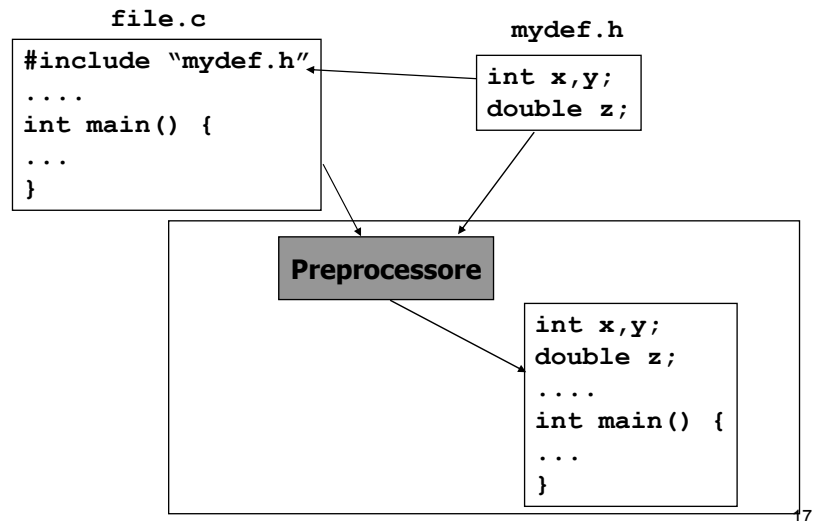
15

Direttive del preprocessore

- **#define**
 - Definizione di un valore costante
 - Ogni riferimento alla costante viene espanso dal preprocessore al valore corrispondente
 - Esempi
 - `#define FALSE 0`
 - `#define SEPARATOR "-----"`

16

La direttiva #include



I dati

Definizione di dati

- Tutti i dati devono essere definiti prima di essere usati
- Definizione di un dato
 - 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
 - `int` interi (complemento a 2)
 - `float` reali (floating point singola precisione)
 - `double` reali (floating point doppia precisione)
- La dimensione precisa di questi tipi dipende dall'architettura (non definita dal linguaggio)
 - `|char|` = 8 bit sempre

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**
 - 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 dati

- Esempi:
`int x;`
`char ch;`
`long int x1,x2,x3;`
`double pi;`
`short int stipendio;`
`long y,z;`
- Usiamo nomi significativi!
 - Es: `int x0a11;`
`int valore;`

23

Costanti

- Valori che rappresentano quantità fisse
- Esempi
 - `char`
 - `'f'`
 - `int, short, long`
 - `26`
 - `0x1a` `0X1a`
 - `26L`
 - `26U`
 - `26UL`
 - `float, double`
 - `-212.6`
 - `-2.126e2` `-2.126E2`

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 costanti

- Sintassi
 [const] <*tipo*> <*variabile*> [= <*valore*>] ;
- Esempi
 - `const double pigreco = 3.14159;`
 - `const char separatore = '$';`
 - `const float aliquota = 0.2;`
- 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**>”

 - 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 al `main()`
- *Variabili locali*
 - Definite all'interno del main
 - Più in generale, definite all'interno di un blocco

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

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

29

Le istruzioni

Istruzioni

- Assegnazioni
 - Come in Java
- Operatori
 - Simili a quelli di Java (non tutti)
- Istruzioni di controllo del flusso
 - Come in Java
- Istruzioni di I/O
 - Diverse da Java!

31

L'istruzione `printf()`

- Sintassi
`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

32

L'istruzione `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
```

33

L'istruzione `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);
```

34

I/O formattato avanzato

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

- **printf**

`%[flag][min dim][.precision][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`

35

I/O formattato avanzato

- **scanf**

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

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

36

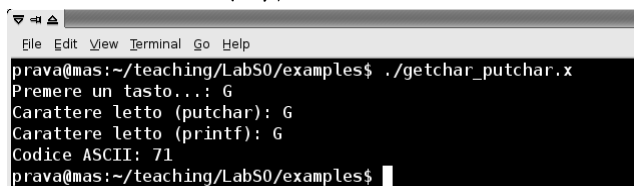
I/O a caratteri

- Acquisizione/stampa di un carattere alla volta
- Istruzioni
 - `getchar()`
 - Legge un carattere da tastiera
 - Il carattere viene fornito come “risultato” di `getchar`
 - (valore intero)
 - In caso di errore il risultato è la costante `EOF` (definita in `stdio.h`)
 - `putchar(<carattere>)`
 - Stampa <carattere> su schermo
 - <carattere>: una dato di tipo `char`

37

I/O a caratteri - Esempio

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char*argv[]){
    char tast;
    printf("Premere un tasto...: ");
    tast = getchar();
    if (tast != EOF) {
        printf("Carattere letto da putchar: ");
        putchar(tast);
        putchar('\n');
        printf("Carattere letto da printf: %c\n", tast);
        printf("Codice ASCII: %d\n", tast);
    } else {
        printf("Letto end of file\n");
    }
    exit(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$
```

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 non è noto
 - Es.: un testo

39

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’)
 - La riga viene fornita come stringa <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

40

I/O a righe

- **NOTA:**

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

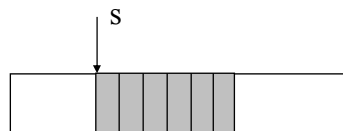
char*

- Significato: stringa = vettore di caratteri (**char**)

- Simbolo `'*'` indica l'indirizzo di partenza della stringa in memoria
- Detto "puntatore"

- Esempio

- `char* s;`



41

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);
    }
}
```

42

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
 - Gestione dei puntatori
 - `puts(s)` è identica a `printf("%s\n",s);`
- Usate meno di frequente degli altre istruzioni di I/O

43

Le funzioni

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

45

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)
{
    ...
}
```

← prototipo

46

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

47

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$
```

48

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 formali posso modificarne il valore*

49

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);}
```

```
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$
```

50

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

51

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>	$\text{atan} (y / x)$
<code>double sinh (double x)</code>	seno iperbolico
<code>double cosh (double x)</code>	coseno iperbolico
<code>double tanh (double x)</code>	tang. iperbolica

52

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

53

Funzioni di utilità

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

54

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 una cifra
<code>int isdigit (char c)</code>	Se c è minuscola
<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

55

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 */
```

56

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>char* strncmp (char* dest, char* src, int n);</code>	cfr. n car. max

57

I file

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 tre stream
 - **stdin**
 - **stdout**
 - **stderr**

59

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

60

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
 - **<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)
 - `"r+"`: lettura/scrittura su file esistente
 - `"w+"`: lettura/scrittura su nuovo file
 - `"a+"`: lettura/scrittura in coda o su nuovo file
 - ritorna il puntatore allo stream in caso di successo, altrimenti ritorna **NULL**

61

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 (su disco) ad un file logico (stream)
- 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

62

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);
```

63

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)`

64

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: `putchar()` equivale a `putc(stdout)`

65

Lettura a righe

- `char* fgets(char* <s>, int <n>, FILE* <stream>);`
 - Legge una stringa dallo stream fermandosi al più dopo n-1 caratteri
 - L'eventuale `'\n'` NON viene eliminato (diverso da `gets` !)
 - Restituisce il puntatore alla stringa carattere letto o `NULL` in caso di fine file o errore
 - NOTA: `gets()` "equivale" a `fgets(stdin)`

66

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

67

Lettura formattata

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

68

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

69

Le strutture dati aggregate

Tipi aggregati

- 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 metodi!)
- Sintassi (definizione di tipo)

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

I campi sono nel formato
`<tipo> <nome campo>;`

71

struct - Esempio

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

72

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;
```

73

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;
```

74

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 <tipo> <nome nuovo tipo>;
```

- Esempio

```
typedef struct complex {  
    double re;  
    double im;  
} compl; —————> compl z1, z2;
```

75

I puntatori

Puntatori

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

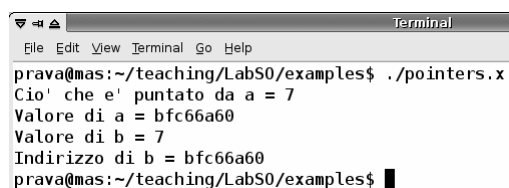
- Esempio:

<code>int *a;</code>	<code>a</code> è un puntatore a una variabile di tipo intero
<code>int b;</code>	<code>b</code> è una variabile di tipo intero
<code>a = 3;</code>	illegale perché <code>a</code> è un puntatore
<code>a = &b;</code>	ad <code>a</code> viene assegnato l'indirizzo di <code>b</code>
<code>*a = 3;</code>	legale perché <code>a</code> punta a <code>b</code> , il valore di <code>b</code> viene modificato (<code>b=3</code>)

77

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);
}
```



A terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Go, Help). The prompt is `prava@mas:~/teaching/LabS0/examples$./pointers.x`. The output is:
`Cio' che e' puntato da a = 7`
`Valore di a = bfc66a60`
`Valore di b = 7`
`Indirizzo di b = bfc66a60`
The prompt returns to `prava@mas:~/teaching/LabS0/examples$`.

78

Stringhe

- Vettori di caratteri terminati da un carattere aggiuntivo '\0' (NULL)

- Memorizzate come i vettori

- Es: `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]

79

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

80

Stringhe, vettori e puntatori

- In generale, è equivalente definire una stringa come
 - `char s[];`
 - `char *s;`
- Lo stesso vale nel caso di funzioni che ricevono una stringa come parametro
 - `int f(char s[])`
 - `int f(char *s)`

81

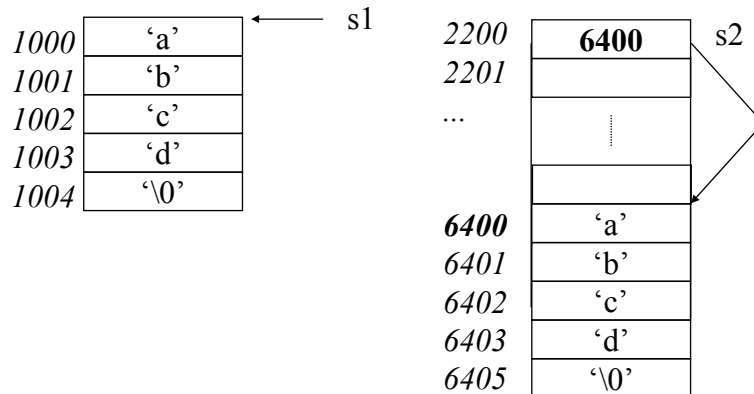
Stringhe, vettori e puntatori

- Differenza sostanziale nella definizione di una stringa come vettore o come puntatore:
`char s1[] = "abcd";`
`char *s2 = "abcd";`
 - `s1` è un vettore
 - I caratteri che lo compongono possono cambiare
 - Si riferisce sempre alla stessa area di memoria
 - `s2` è un puntatore
 - Si riferisce ("punta") ad una stringa costante ("abcd")
 - Si può far puntare altrove (es. scrivendo `s2 = ...`) ma ...
 - ... la modifica del contenuto di `s2` ha risultato NON DEFINITO
- Allocate in "zone" di memoria diverse!!!

82

Stringhe, vettori e puntatori

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



83

Stringhe, v

```
#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/Lab50/examples$ ./array_and_pointer.x
Stringa s1 = abcd
Stringa s2 = efgh
Modifico s1...
... valore modificato = bcde
Modifico s2...
Segmentation fault
prava@mas:~/teaching/Lab50/examples$
```

84

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 '`->`'

85

Puntatori e `struct` - Esempio

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

86

Parametri del main

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

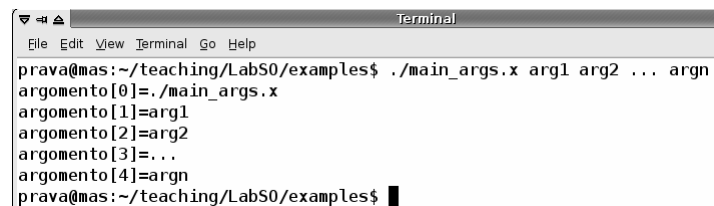
Sintassi

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

87

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]);  
}
```



```
prava@mas:~/teaching/LabS0/examples$ ./main_args.x arg1 arg2 ... argn  
argomento[0]=./main_args.x  
argomento[1]=arg1  
argomento[2]=arg2  
argomento[3]=...  
argomento[4]=argn  
prava@mas:~/teaching/LabS0/examples$
```

88

Memoria dinamica

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** ()
- Due usi
 - Allocazione dinamica di vettori
 - Allocazione dinamica di strutture

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

91

Strutture e malloc

- Bisogna sapere quanto spazio (byte) occupa la struttura

- Funzione **sizeof(<variabile>)**

- Ritorna il numero di byte occupati da <variabile>

- Esempio

```
typedef struct {  
    float x,y;  
} Point;  
  
Point* segment;  
segment = (Point*) malloc(2*sizeof(Point));  
/* vettore di due variabili point */
```

92

Liberare la memoria

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

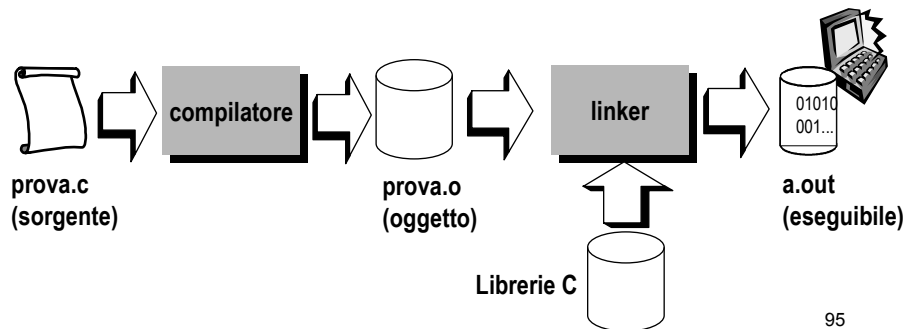
```
segment = (Point*)
    malloc(2*sizeof(Point));
...
free(segment);
```

93

Le librerie

Le librerie

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



95

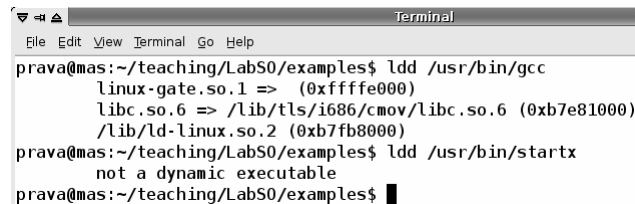
Le librerie

- 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

96

Le librerie

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



```
prava@mas:~/teaching/Lab50/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/Lab50/examples$ ldd /usr/bin/startx
not a dynamic executable
prava@mas:~/teaching/Lab50/examples$
```

97

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 loader. Può essere usato anche per creare librerie dinamiche (.so)

98

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>.so`

99

Compilazione

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

101

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>

102

II Makefile

Esempio

STRINGA = TUTTI E TRE

one:

 @echo UNO

two:

 @echo DUE

three:

 @echo TRE

all: one two three

 @echo \$(STRINGA)

```
prava@mas:~/teaching/Lab50/examples$ make -f makefile_echo
UNO
prava@mas:~/teaching/Lab50/examples$ make -f makefile_echo two
DUE
prava@mas:~/teaching/Lab50/examples$ make -f makefile_echo all
UNO
DUE
TRE
TUTTI E TRE
prava@mas:~/teaching/Lab50/examples$
```

103

II 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

104

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
```

105

Il debugging

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