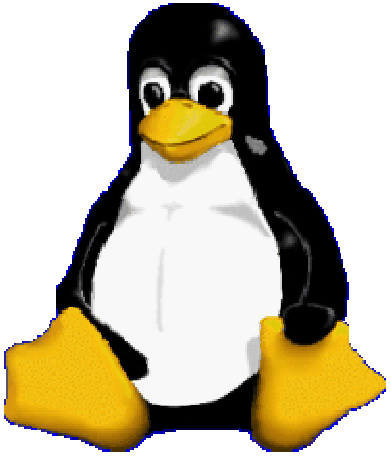
Università degli Studi di Verona

Dipartimento di Biotecnologie

Laurea in Biotecnologie

Corso di Informatica 2014/2015



Introduzione al Linguaggio C

Gennaio 2015 - Sergio Marin Vargas

Struttura di un programma C

Versione minima

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

- Parte dichiarativa globale
  - Elenco dei dati e del loro tipo (numerico, stringa, booleano) usati in tutto il programma.
- Parte dichiarativa locale
  - Elenco dei dati con il relativo tipo usati "solo all'interno" dalla **`main`** o "solo all'interno" dalle **singole funzioni**.

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)
}
```

Il preprocessore C

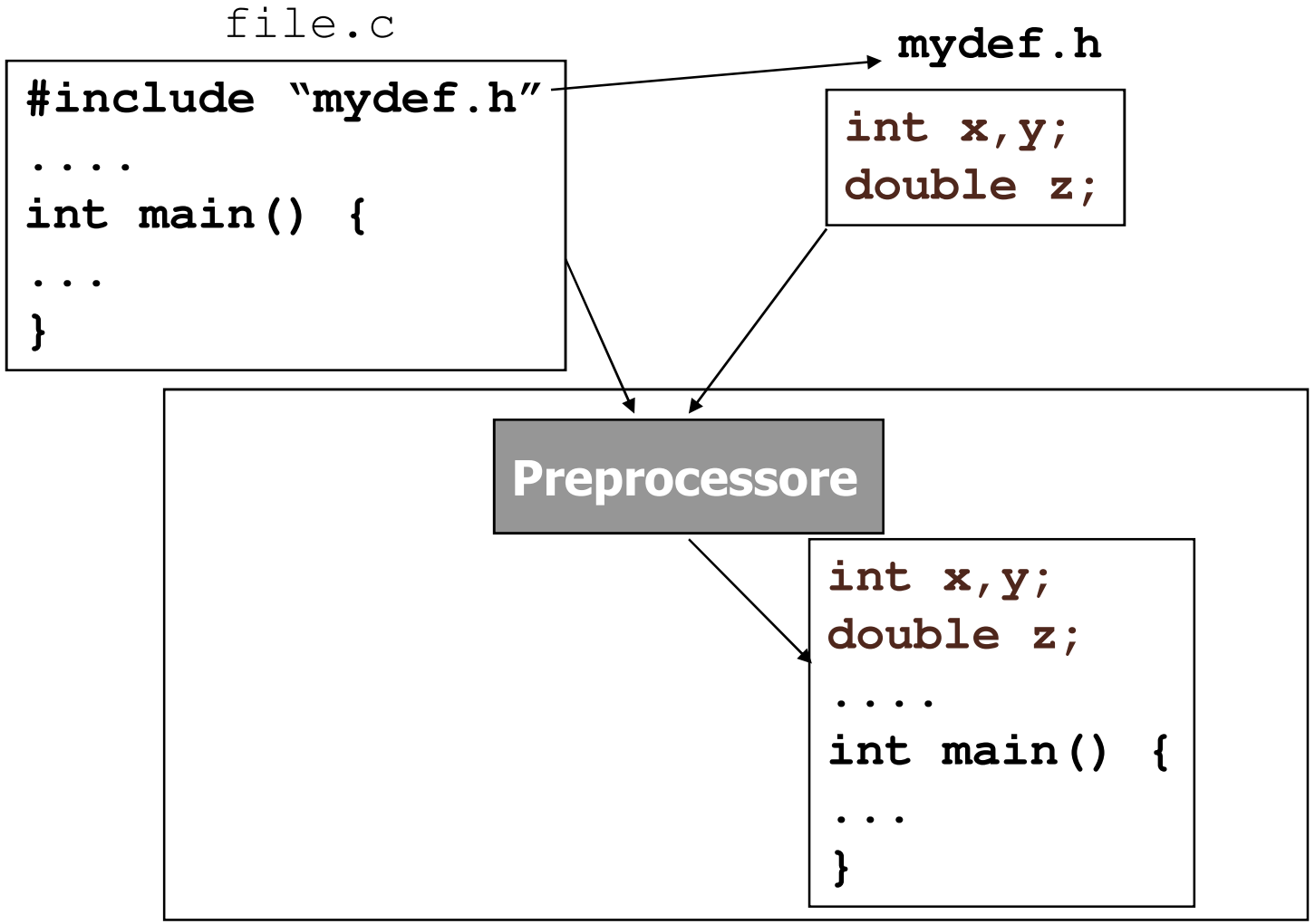
- La prima fase della compilazione (trasparente all'utente) consiste nell'invocazione del *preprocessore*
- Un programma C contiene specifiche direttive per il preprocessore
  - Inclusioni di file di definizioni (*header* file)
  - Definizioni di costanti
  - Altre direttive
- Individuate dal simbolo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

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

La direttiva #include

- Esempio:



I Dati

Ogni variabile è caratterizzata da

- Un nome (Esempio “contatore”, “i”, ecc)
- Un valore
- Un tipo di dato
  - Numeri naturali, interi o reali (1, -2, 0.34)
  - Caratteri alfanumerici (stringhe) (A, B, ..)
  - Dati logici o booleani (Vero, Falso)

Criteri di classificazione dei dati

- Visibilità da parte dell'utente
  - visibile (dati di input o output)
  - trasparente (dati in variabili temporanei di supporto)
- Variabilità nel tempo
  - costanti
  - variabili (acquisizione dall'esterno o assegnazione)
- Struttura
  - elementari (interi, alfanumerici, booleani, …)
  - strutturati (array, matrici, …)

Tipi di dato

- Tipi predefiniti
  - Numerici
  - Booleani
  - Alfanumerici
  - Stringa
  - Data
- Variabili strutturate
  - Array (vettori)
  - Matrici (array multidimensionali)
  - Record (strutture complesse definite dall'utente)

Operazioni e tipi di dato

La stessa operazione su "tipi di dati diversi" può portare a risultati diversi

Esempio: "**+**"

Numerico x = 10 → x+1 = 11

Stringa x = "ciao" → x+"a tutti" = "ciao a tutti"

Stringa numerica x = "10" → x+1 = "101"

Data x = 18.11.2010 → x+1= 19.11.2010

Variabili strutturate

Array (vettori):

- Nome
- Tipo
- Può contenere n elementi
- v[i], i=1,…,n (i=0,…,n-1)

Matrici:

- Nome
- Tipo
- Può contenere nxm elementi
- m[i,j], i=1,…,n j=1,…,m

Variabili strutturate

Strutture definite dall'utente:

- Nome
- Campi (componenti)
- Può comprendere componenti di diverso tipo

Esempio:

```
struct prodotto {
char[30] nome;
float costo;
};
prodotto p;
p.costo = 10;
```

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)

Tipi di dati base (primitivi)

- Sono quelli forniti direttamente dal C
- Identificati da le seguenti parole chiavi:
  - `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)
  - 1 `char` = 8 bit = 1 byte (**sempre**)

Modificatori dei tipi di dati base

- Sono previsti dei modificatori, identificati da parole chiave da premettere ai tipi di dati 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`**

Definizione di variabili

- Sintassi:

  <tipo di dato> <variabile>;
- Sintassi alternativa (definizioni multiple):

  <tipo di dato> <lista di variabili>;

- <*variabile*>: è l'identificatore che rappresenta il nome della variabile.
- <*lista di variabili*>: lista di identificatori separati da **virgola** (“**,**”).

Esempio definizione di variabili

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

Definizione di costanti

- Sintassi:

  const <***tipo***> <***variabile***> = <***valore***> ;

- Esempi:
  - `const double PIGRECO = 3.14159;`
  - `const char SEPARATORE = '$';`
  - `const float ALIQUOTA = 0.22;`

- Per convenzione si usa come identificatori delle constanti una variabile in MAIUSCOLO.

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**
  - **'\n' line feed**
  - **'\t' tab**
  - **'\0' null**

Stringhe

- Una stringa è una sequenza di caratteri (array di char) terminata dal carattere **NULL** ('\0')
- Non è un tipo di base del C
- Definizione di una stringa:

  char nome[lunghezza] = "<*sequenza di caratteri*>"
- Esempio:
  - char ciao[6] = "Ciao!\0"
  - char stringa[9] = "abcdefg\n\0";

Visibilità delle variabili

- Ogni variabile è definita all'interno di un preciso *ambiente di visibilità* (*scope*)
- ***Variabili globali***
  - Definite **all'esterno del main()**
- ***Variabili locali***
  - Definite **all'interno del main()** o **all'interno di una funzione**
  - O più in generale, definite all'interno di un blocco

Esempio visibilità delle variabili

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

- **n**,**x**: visibili in tutto il programma
- **a**,**b**,**c**,**y**: visibili in tutto il main (anche nel blocco blu)
- **d**,**z**: visibili solo nel blocco blu

Le istruzioni

- Istruzioni di input/output
- Istruzioni aritmetico–logiche
- Istruzioni di controllo

L'istruzione printf()

- Istruzione di output, che "visualizza" una stringa con dei valori formatti, con la seguente sintassi:

  printf(<*stringa formato*>,<*arg1*>,...,<*argn*>);

  - **<*stringa formato*>:** stringa che determina le direttive del formato di stampa di ognuno dei vari argomenti.
    - Può contenere:
      - Caratteri (stampati come appaiono)
      - Direttive di formato nella forma %<carattere>
        - **%d** **intero**
        - **%u** **intero senza segno (unsigned)**
        - **%s** **stringa**
        - **%c** **carattere**
        - **%x** **esadecimale**
        - **%o** **ottale**
        - **%f** **float**
        - **%g** **double**
  - **<*arg1*>,...,<*argn*>:** gli argomenti ovvero I valori (o espressioni) che si vogliono visualizzare associate alle direttive della "stringa formato" nello stesso ordine in cui appaiono.

Esempi di printf()

```
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
```

L'istruzione scanf()

- Istruzione di input, che "legge" una stringa con dei valori formattati, con la seguente sintassi:

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

  - **<*stringa formato*>**: come per **printf**
  - **<*arg1*>,...,<*argn*>**: le variabili a cui si vogliono assegnare i 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);
```

In input bisogna digitare "un numero intero", "uno spazio" e "un numero decimale" e poi fare invio.

Input/Ouput (I/O) a caratteri

- Acquisizione/Visualizzazione di un carattere alla volta
- Istruzioni:
  - **getchar()**
    - Legge un carattere da tastiera
    - Il carattere viene fornito come "risultato della funzione" getchar, è corrisponde a un valore intero equivalente al codice ASCII del carattere letto dalla tastiera
    - In caso di errore il risultato è la costante **EOF** (definita in **stdio.h)**
  - **putchar(<*carattere*>)**
    - Visualizza il carattere <*carattere*> sullo schermo
    - <*carattere*>: una variabile o costante con tipo di dato **char**

Esempio I/O a caratteri

```
#include <stdio.h>
main()
{
  int tasto;
  printf("Premi un tasto...\n");
  tasto = getchar();
  if (tasto != EOF) /* se non è errore ? */
  {
     printf("Hai premuto %c\n", tasto);
     printf("Codice ASCII = %d\n", tasto);
  }
}
```

I/O a righe

- Acquisizione/visualizzazione di una riga alla volta
  - Riga = serie di caratteri terminata da '\n' (a capo)
- Istruzioni:
  - **gets**(<*variabile stringa*>)
    - **Legge una riga da tastiera (fino al '\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*>)
    - **Visualizza la stringa <*stringa*> sullo schermo**
    - **Aggiunge sempre '\n' alla fine della stringa**

Le istruzioni

- Istruzioni di input/output
- Istruzioni aritmetico–logiche
- Istruzioni di controllo

Operatori su interi (int)

=	Assegnamento
+	Somma
++	Incremento + 1
-	Sottrazione
--	Decremento - 1
*	Moltiplicazione
/	Divisione con troncamento della parte frazionaria
%	Resto della divisione intera
==	Relazione di uguaglianza
!=	Relazione di diversità
<	Minore di…
>	Maggiore di…
<=	Minore o uguale
>=	Maggiore o uguale

Operatori su decimali (float)

=	Assegnamento
+	Somma
-	Sottrazione
*	Moltiplicazione
/	Divisione a risultato reale
==	Relazione di uguaglianza
!=	Relazione di diversità
<	Minore di…
>	Maggiore di…
<=	Minore o uguale
>=	Maggiore o uguale

Operatori logici

! **NOT** (Negazione di una espressione logica)

if ! (a > b)
→ L'espressione sarà vera se **non è vero** che "a > b"

&& **AND** (Unione con "and" di due espressioni logiche)

if ((a > b) && (b > c))
→ L'espressione sarà vera se "a > b" **e** "b > c"

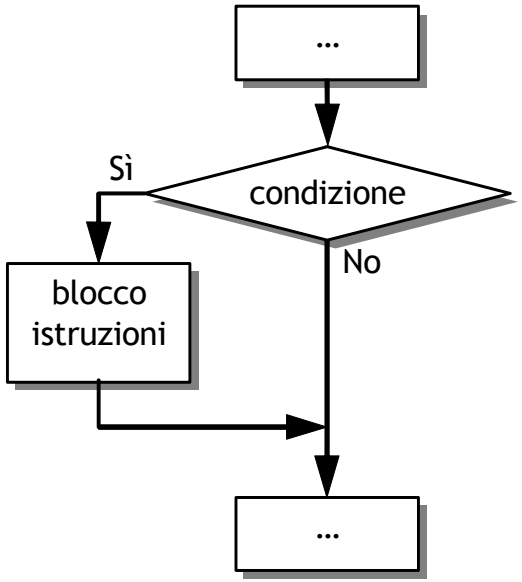
|| **OR** (Unione con "or" di due espressioni logiche)

if ((a > b) || (b > c))
→ L'espressione sarà vera se "a > b" **o** "b > c"

Le istruzioni

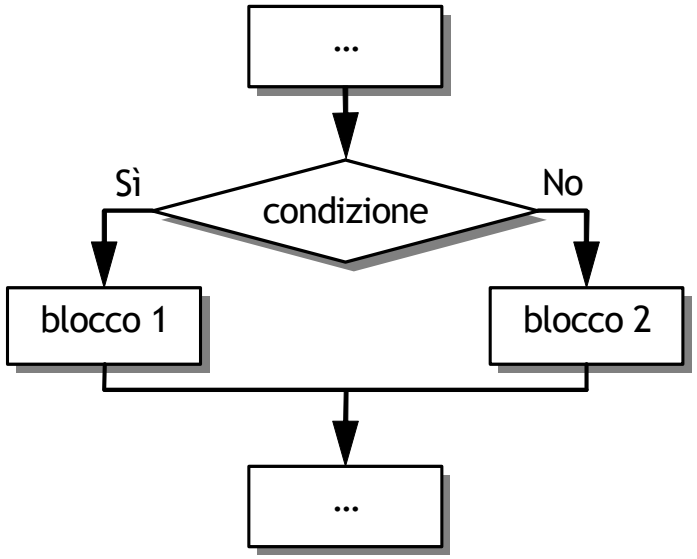
- Istruzioni di input/output
- Istruzioni aritmetico–logiche
- Istruzioni di controllo

If (selezione semplice)



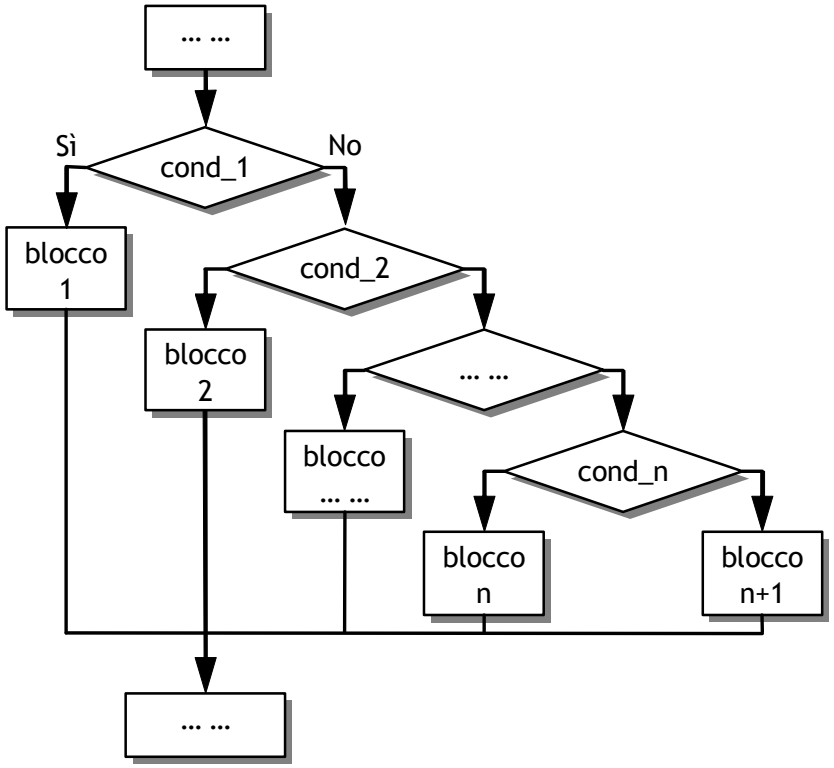

```
main()
{ ...
  /* selezione semplice */
  if (condizione) {
    /* blocco istruzioni
       eseguito solo se
       la condizione è vera */
    ...
  }
  ...
}
```

If-else (selezione a due vie)



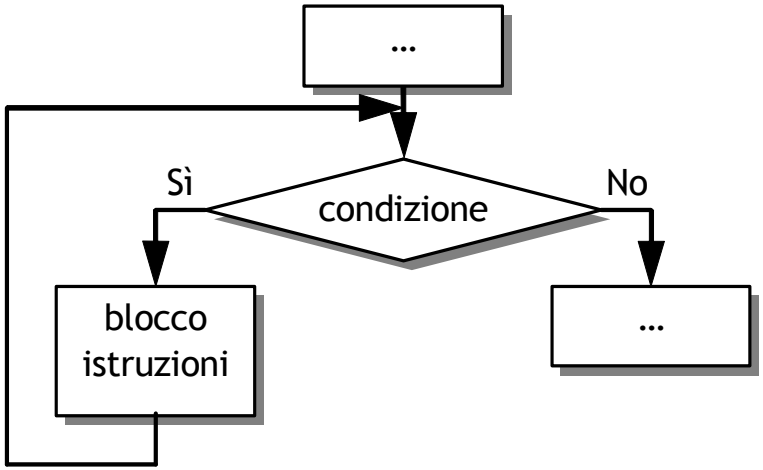

```
main()
{  ...
    /* selezione a due vie */
   if (condizione) {
      ... ... /* blocco 1 */
   }
   else {
      ... ... /* blocco 2 */
   }
   ...
}
```

If-else-if (selezione a più vie)



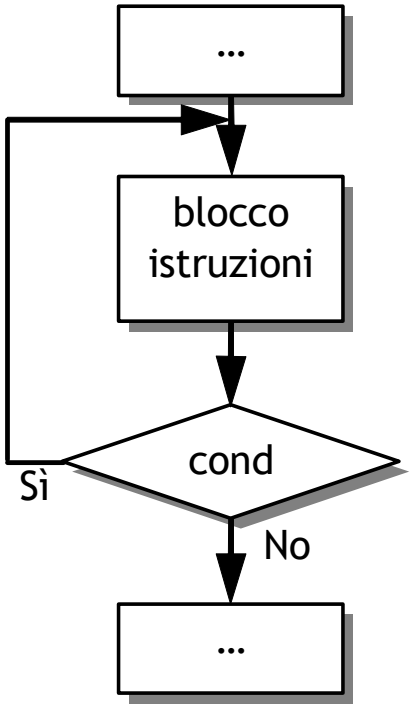

```
main()
{  ... /* sel a più vie */
 if (cond_1) {
    ...  /* blocco_1 */
 }
else if (cond_2) {
    ... /* blocco_2 */
 }
else if (... ...) {
    ... /* blocco_... */
 }
else if (cond_n) {
    ... /* blocco_n */
 }
else {
    ... /* blocco_n+1*/
 }
 ...
}
```

While (ciclo a condizione iniziale)




```
main()
{ … …
/* ciclo a condizione iniziale */
  while (condizione) {
     … … /* blocco istruzioni */
     /* ripetuto finché condizione è vera */
  }
  … /* eseguito quando condizione diviene
falsa */
}
```

While (ciclo a condizione finale)




```
main()
{ …
/* ciclo a condizione finale */
  do {
     … … /* blocco istruzioni */
     /* eseguito una volta e
        ripetuto se la condizione è vera */
  } while (cond)
  … … /* eseguito se la condizione è falsa */
}
```

For (ciclo con contatore)

- Può essere usato per sostituire il ciclo while quando si vuole eseguire il ciclo un numero finito di volte
- Utilizzato quando si devono utilizzare contatori

```
main()
{ … …
/* ciclo a condizione iniziale */
  for (espr1;cond2;espr3) {
     … … /* blocco istruzioni */
     /* ripetuto finché cond2 è vera */
  }
}
```

Tipi strutturati: gli array

- In C non esiste il tipo array (vettore), ma è possibile definirlo utilizzando il costruttore di tipo array
- La sintassi di specifica di un array è

  <tipo> <variabile>[<dimensione>]
- Gli elementi sono ordinati e accessibili tramite un indice di posizione
- Gli elementi sono di tipo omogeneo
- Si può definire **un tipo array** usando l'istruzione **`typedef`**

  `typedef` <tipo> <variabile>[<dimensione>]

`Esempio:`

```
typedef int array_definito[20];
array_definito vett1, vett2;
```

Riempire un array

Algoritmo

```
Dati
   n = 100 intero
   f[ ]  vettore di interi
   i  intero positivo
Risoluzione
   ...
   i ← 1

   finché (i ≤ n) ripeti
      f[i] ← 0
      i ← i + 1
   fine ciclo
   ...
```

Programma in C

```
main() {
   int f[100];
   int i;
   ...
   i = 0;

   while (i ≤ 99) {
      f[i] = 0;
      i = i + 1;
   }
...
}
```

```
main() {
   int f[100];
   int i;
   ...

   for (i=0;i ≤ 99;i++) {
      f[i] = 0;
}
...
}
```

Tipi strutturati: le matrici

- Una matrice è **un array di array**

  Esempio:
  typedef int matrice[20][20];
  matrice mat1,mat2;

  Oppure (array di array):
  typedef int array[20];
  array matrice[20];

- Sintassi per definire una matrice:
  <tipo> <variabile>[<dimRighe>][<dimColonne>]

Tipi strutturati: I record

- In C è possibile definire dati composti da elementi eterogenei (***record***), aggregandoli in una singola variabile individuata dalla keyword **struct**

- Sintassi:

  struct <*identificatore***> {**

  campi

  };

- I **campi** sono nel formato

  <tipo> <nome campo>;

Esempio di record (struct)

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

struct identity {
    char nome[30];
    char cognome[30];
    char codicefiscale[16];
    int altezza;
    char statocivile;
}
struct identity identita;
```

Accesso ai campi di un record

- Una struttura record permette di accedere ai singoli campi tramite l'operatore "**.**", applicato alle variabili del corrispondente tipo struct

  <*variabile***>** **.** **<***campo***>**

- Esempio:

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

Definizione di struct come tipo

- E' possibile definire un nuovo tipo a partire da una **struct** tramite **typedef**

Esempio:

```
typedef struct complex {
   double re;
   double im;
} complesso;
complesso z1,z2;
```

Funzioni

- Un programma C consiste di una o più funzioni
  - Almeno una la "**main()**"
- Definizione delle funzioni:
  - Prima della definizione di **main()**
  - Dopo la definizione di **main()**, in questo caso è necessario premettere in testa al file il *prototipo* della funzione
    - Nome
    - Argomenti

Funzioni e prototipi: esempio

Prima di main():

```
double f(int x)
{
  …
}
int main()
{
  …
}
```

Dopo di main():

```
double f(int);
```

Prototipo della funzione

```
int main ()
{
  …
}
double f(int x)
{
  …
}
```

Funzioni di libreria

- Il C prevede numerose funzioni predefinite per scopi diversi, definite in **specifiche *librerie***

- Varie categorie
  - Funzioni matematiche

    #include <math.h>
  - Funzioni di classificazione caratteri

    #include <ctype.h>
  - Funzioni matematiche intere

    #include <stdlib.h>
  - Funzione di stringhe

    #include <string.h>

Funzioni matematiche I

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

funzione	*definizione*
`double sin (double x)`	
`double cos (double x)`	
`double tan (double x)`	
`double asin (double x)`	
`double acos (double x)`	
`double atan (double x)`	
`double atan2 (double y, double x)`	atan (y / x)
`double sinh (double x)`	
`double cosh (double x)`	
`double tanh (double x)`	

Funzioni matematiche 2

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

funzione	*definizione*
`double pow (double x, double y)`	x^y
`double sqrt (double x)`	radice quadrata
`double log (double x)`	logaritmo naturale
`double log10 (double x)`	logaritmo decimale
`double exp (double x)`	e^x
`double ceil (double x)`	ceiling(x)
`double floor (double x)`	floor(x)
`double fabs (double x)`	valore assoluto
`double fmod (double x, double y)`	modulo

Funzioni di classificazione caratteri

- Utilizzabili con **#include <ctype.h>**

funzione	*definizione*
`int isalnum (char c)`	Se c è lettera o cifra
`int isalpha (char c)`	Se c è lettera
`int isascii(char c)`	Se c è lettera o cifra
`int isdigit (char c)`	Se c è una cifra
`int islower(char c)`	Se c è minuscola
`int isupper (char c)`	Se c è maiuscola
`int isspace(char c)`	Se c è spazio,tab,\n
`int iscntrl(char c)`	Se c è di controllo
`int isgraph(char c)`	Se c è stampabile ma non spazio
`int isprint(char c)`	Se c è stampabile
`int ispunct(char c)`	Se c è punteggiatura

Funzioni matematiche intere

- Utilizzabili con **#include <stdlib.h>**

funzione	*definizione*
`int abs (int n)`	**valore assoluto**
`long labs (long n)`	**valore assoluto**
`div_t div (int numer, int denom)`	quoto e resto della divisione intera
`ldiv_t ldiv (long numer, long denom)`	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):

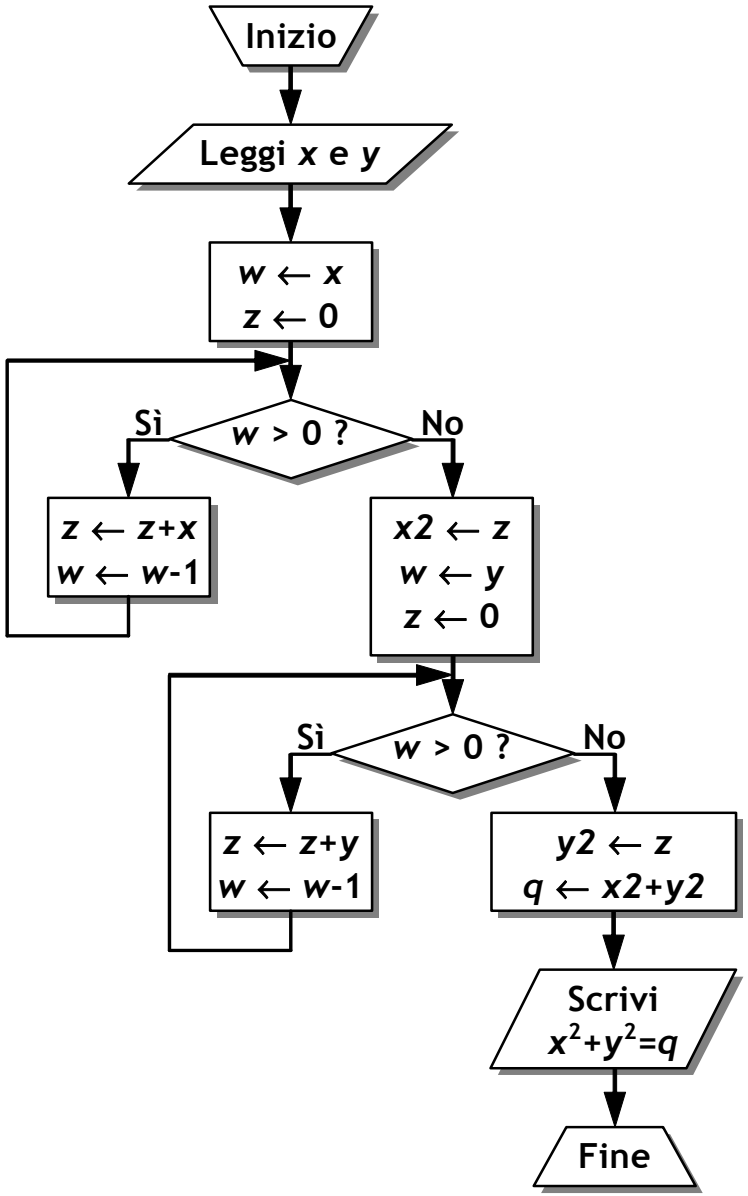
```
quot /* quoziente */
rem /* resto */
```

Fuzioni di stringhe

- Utilizzabili con **#include <string.h>**

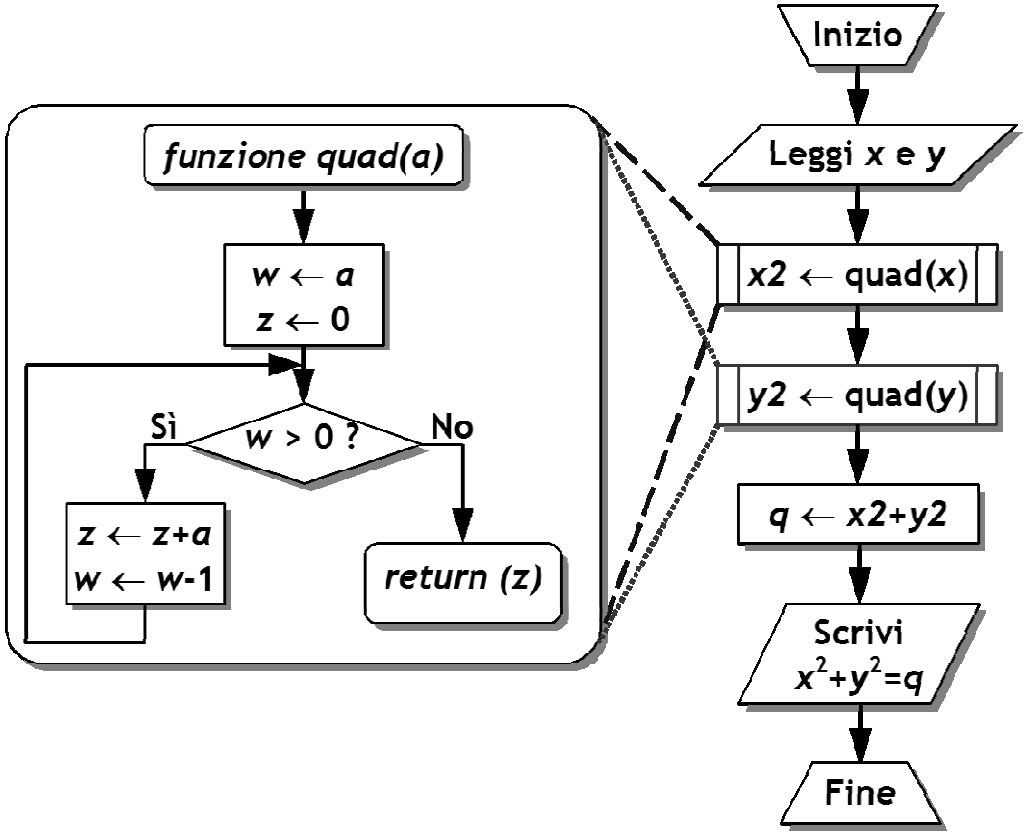
funzione	*definizione*
`char* strcat (char* s1, char* s2);`	*concatenazione s1+s2*
`char* strchr (char* s, int c);`	*Puntatore al primo c*
`int strcmp (char* s1, char* s2);`	*Confronto s1 e s2*
`char* strcpy (char* s1, char* s2);`	*Copia s2 in s1*
`int strlen (char* s);`	*lunghezza di s*
`char* strncat (char* s1,char* s2,int n);`	*concat. n car. max*
`char* strncpy (char* s1,char* s2,int n);`	*copia n car. max*
`char* strncmp(char* dest,char* src,int n);`	*confron. n car. max*

$X^2 + Y^2$ (Senza sottoprogrammi)




```
main()  /* q = x² + y² */
{ int x,y,x2,y2,q,w,z;
  scanf("%d %d",&x,&y);
  w = x;
  z = 0;
  while (w > 0) {
      z = z + x;
      w = w – 1; }
  x2 = z;
  w = y;
  z = 0;
  while (w > 0) {
      z = z + y;
      w = w – 1; }
  y2 = z;
  q = x2+y2;
  printf("%d", q);
}
```

$X^2 + Y^2$ (Con sottoprogrammi)

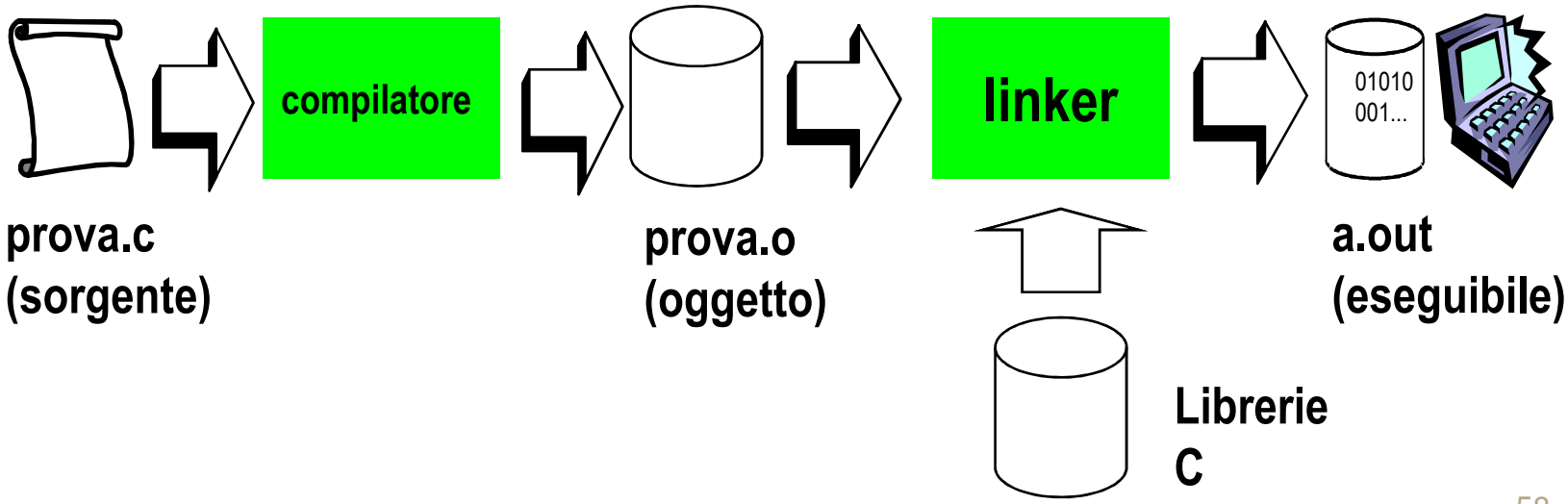



```
int quad (int a) {
    /* restituisce a2 /
    int w, z;
    w = a;  z = 0;
    while (w > 0) {
        z = z + a;
        w = w – 1;  }
    return (z);
}

main() /* q = x2 + y2 */
{ int x,y,x2,y2,q;
  scanf("%d %d",&x,&y);
  x2 = quad(x);
  y2 = quad(y);
  q = x2+y2;
  printf("%d", q);
}
```

Le librerie

- Quando un file viene compilato, dopo la fase di "LINK" (**linker**) ho a disposizione l'eseguibile, per il sistema operativo in cui sto lavorando!
- L'eseguibile che ottengo può essere **monolitico** (o **statico**), ovvero contenere tutto il codice delle librerie necessario per l'esecuzione, oppure **dinamico** cioè contiene solo i riferimenti alle "funzioni di libreria".
- Se l'eseguibile è "linkato" **dinamicamente** è necessario che siano presenti tutte le librerie richieste dall'eseguibile al momento dell'esecuzione del programma.



Le librerie

Librerie dinamiche: **.so**
Librerie statiche: **.a**

Con il seguente comando è possibile vedere le librerie dinamiche richieste da un eseguibile.:

ldd

Esempio:

```
ldd tar
  libpthread.so.0 => /lib/i686/libpthread.so.0 (0x4002f000)
  librt.so.1 => /lib/librt.so.1 (0x40044000)
  libc.so.6 => /lib/i686/libc.so.6 (0x40056000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Le librerie

Con il comando:

nm

E' possibile vedere il contenuto (simboli) delle librerie dinamiche e statiche.

Con il comando:

ar

E' possibile creare una libreria statica (.a) mettendo assieme un insieme di file **.o**

Le librerie

Le librerie dinamiche devono essere "trovate", esistono due sistemi di configurazione:

- ✓ /etc/ld.so.conf
- ✓ ld_library_path

gcc –l<nome>: specifica il link con la libreria con nome **lib<nome>.so**

Il compilatore C

- **`gcc prova.c`**
  - Genera a.out come eseguibile
- **`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.so`**

`Per eseguire il file ottenuto:`

`./prova  (oppure ./a.out)`

Il debugging

E' possibile eseguire/provare un programma passo per passo.

Il GDB (gnu debug) è il programma che ci consente di provare il nostro eseguibile, ha un'interfaccia a caratteri.

Esistono numerosi front-end grafici per il GDB, il più famoso è sicuramente il DDD.

Per poter analizzare un eseguibile lo si deve compilare con:

- **`gcc -g`**: genera le info per il *debugging*
- **`gcc -ggdb`**: genera le info per il *debugging GDB*