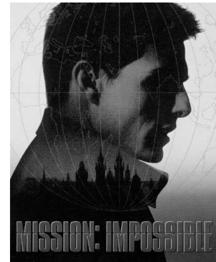
Introduzione al C

Dario Maggiorini
dario@dsi.unimi.it

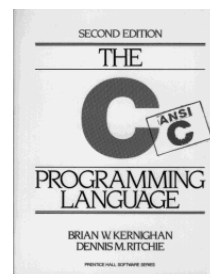
Imparare il C in 3 ore



Obiettivi

- Imparare la grammatica del C
- Saper trattare semplici strutture dati
- Scrivere programmi che utilizzino i canali di I/O
- Imparare a compilare
- Imparare a debuggare

La bibbia



Il linguaggio C

- Tipi base: costanti e variabili
- Strutture e unioni
- Operatori ed espressioni
- Assegnamenti e conversioni
- Strutture di controllo
- Vettori / Puntatori
- Stringhe
- Funzioni

Caratteristiche generali

- General purpose
- Nato per la programmazione di sistema operativo
- Efficiente (basso overhead)
- Procedurale

La filosofia

- Fidati del programmatore
- Non impedire nulla
- Mantieni il linguaggio sintetico e semplice
- Fornisci un modo solo per fare una operazione
- Fallo in modo efficiente, anche se la portabilità non è garantita

Espressioni

- Sono composte mediante :
  - variabili e costanti
  - chiamate di funzioni
  - operatori
  - Parentesi (graffe)
- Le espressione hanno sempre un valore di ritorno

Tipi di base

interi	int	
caratteri (byte)	char	
Virgola mobile	float	double

Modificatori

di dimensione	long short (non char)
di segno	signed unsigned

Dichiarazione di variabili

[modificatori] <tipo> <nome> [= <valore>];

```
int i0;
long int i1 = 12;
unsigned int i2;
unsigned long i3;
signed char c0;
char c1 = 'A';
unsigned long double l0 = 3.141592654;
```

Tipi derivati

vettori

```
<tipo base> <nome> [ [dimensione] ] ;
```

puntatori

```
<tipo base> *
```

strutture

```
struct <nome> {
        <variabile>
        ...
};
```

unioni

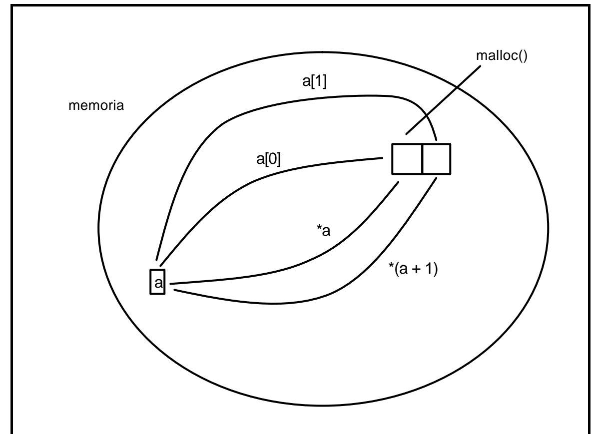
```
union <nome> {
        <variabile>
        ...
};
```

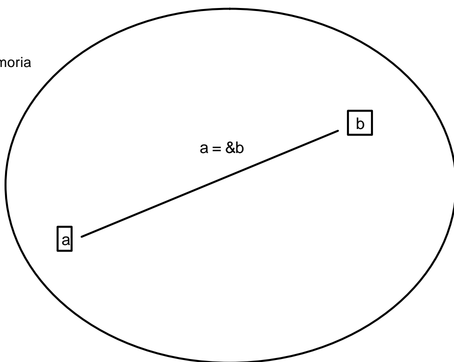
Dichiarazione di tipi derivati

```
int v0 [100];
unsigned long double v1 [50];
signed char v2 [];

int * p0;

struct s0 {
        int f0;
        char * f1;
};
struct s0 * p1;
```

vettori/puntatori/stringhe

- Sono tutte e tre lo stesso tipo di oggetto
- Un puntatore è una variabile che contiene l'indirizzo di un dato
- L'algebra dei puntatori ci permette di usare i vettori come scostamenti da un indirizzo base
- Le stringhe sono vettori di caratteri
- Definire un vettore vuol dire definire un puntatore già inizializzato ad un'area di memoria allocata

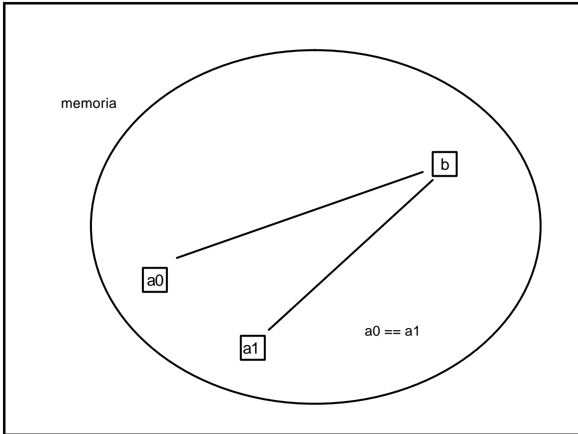


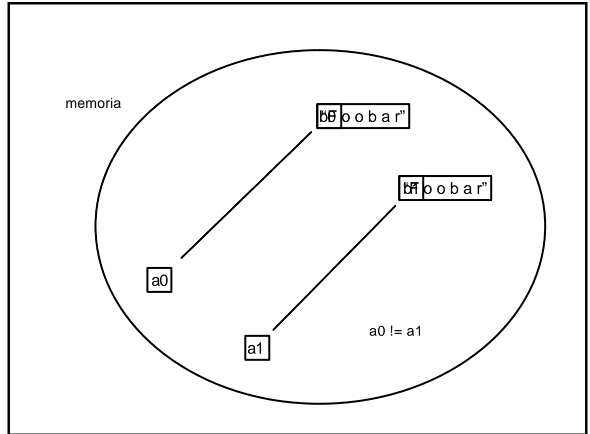


Confronto tra puntatori

Due puntatori non sono MAI uguali a meno che non puntino allo stesso dato

Al massimo potremmo voler confrontare tra loro i dati indirizzati da due puntatori diversi

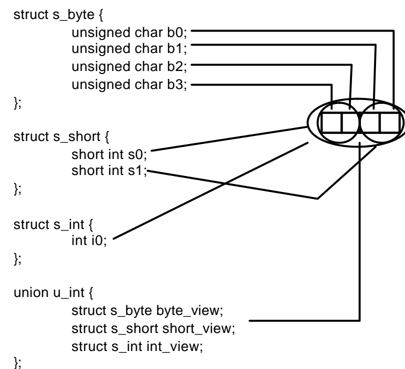




Confronti tra stringhe

- Non possiamo confrontare i puntatori

- Confronto lessicografico int strcmp(s1, s2)
- Lunghezza int strlen(s1)
- Copia char * strcpy(s1, s2)
- Concatenamento char * strcat(s1, s2)



Costanti

Numeriche	decinali	1	10	100
	otali	040	045	065
	esadecimali	0x34	0xff	0x453f
	floating point	12.2	10.2e3	
Caratteri		'a'	'b'	'\n'
Stringhe		"foobar"		

Sequenze di escape (caratteri)

\n	a capo (newline)
\t	tabulazione orizzontale
\v	tabulazione verticale
\b	backspace
\r	ritorno a inizio riga (carriage return)
\f	salto pagina
\\	backslash
\'	apice
\"	virgolette
\0	carattere NULL
\a	allarme (bell)
\ooo	numero ottale
\xhh	numero esadecimale

Chiamate a funzioni

```
nume_funzione([parametri]);

exit(1);

sqrt(2);
```

Operatori

ARITMETICI:	+ - * / %
RELAZIONALI:	> >= < <= == !=
LOGICI:	&& || !
INCREMENTO/ DECREMENTO:	++ --
BIT A BIT:	& | ^ << >> ~
CONDIZIONALI:	? :
ASSEGNAMENTO:	= -= *= /= %= <<= >>= &= ^= |=
CAST:	(tipo) espressione
PUNTATORI E VETTORI:	* & [ ]
ACCESSO A STRUTTURE:	-> .
VIRGOLA:	,

Operatori relazionali

> >= < <= == !=

Producono un valore intero che indica vero o falso.

- FALSO: il valore 0
- VERO : qualsiasi altro valore non nullo

Operatori logici

- Si applicano generalment al risultato di operatori relazionali
  - AND &&
  - OR ||
  - NOT !
- Possono comuqnue essere applicati a qualunque tipo di espressione

Incremento/Decremento

- Si applicano solo a variabili, non a espressioni
- Possono essere prefisso o postfisso

```
a = 1;
x = a++;        /* x = 1    a = 2 */
x = ++a;        /* x = 2    a = 2 */
```

Operatori di assegnamento

+= -= *= /= %=

- Tutti gli operatori aritmetici hanno il corrispondente operatore di assegnamento.

```
a += 5;      /* a = a + 5; */
```

Strutture di controllo

- Selezione: if, switch
- Iterazione: for, while – do, do – while
- Salto: break, continue, return

if

```
if (<espressione>)
        [espressione];
else
        [espressione];
```

if

```
if (a == 10) {
        printf("OK");
}
else {
        printf("failed");
        exit();
}
```

switch

```
switch (<espressione>) {
        case <costante>: [espressione];
        ...
        default : [espressione];
}
```

switch

```
switch (a) {
        case 1 : printf("1");          break;
        case 2 : printf("2");          break;
        case 3 : printf("3");          break;
        default : printf("> 3");
}
```

break

- Interrompe la struttura di controllo che lo contiene
- Serve ad *interrompere* switch e while
- Se usato correttamente riduce il quantativo di codice da scrivere

for

inizializzazione

permanenza

incremento

```
for (<esp1>; <esp2>; <esp3>)
        [espressione4];
```

for

```
for (a = 0; a < 10; a += 1) {
        array[a] = 0;
}

for (; b != 100; c++) {
        b += c;
        if (b >= 1000) break;
        if (b <= 0) continue;
        c = my_func(b);
}
```

continue

- Torna alla valutazione della condizione di permanenza per la struttura di controllo che lo contiene
- Serve a *ri-valutare* for e while
- Se usato correttamente riduce il quantitativo di codice da scrivere

while-do

```
while (<espressione1>)
        [espressione2];
```

(permanenza: <espressione1>)

while-do

```
while (enough_cash(account)) {
        withdraw(amount);
}
```

while-do

```
do
        [espressione2]
while (<espressione1>)
```

(permanenza: <espressione1>)

break/continue/return

ATTENZIONE

- Per ridurre considerevolmente gli errori in fase di programmazione è bene che:
  - Le funzioni abbiano un solo punto di uscita
  - Non ci sia codice *irraggiungibile*
- A parte il caso dello switch è sempre possibile riprodurre break e continue con degli if

Operatore triatico

```
espressione1 ? espressione2 : espressione3;

if (espressione1)
   risultato = espressione2;
else
   risultato = espressione3;
```

Funzioni

- Servono a modularizzare il codice
- In pratica, se ripeto lo stesso codice due volte o più, allora dovrei metterlo in una funzione e richiamare quella

Definizione di funzione

```
<tipo> <nome> ([parametri]) {
        [definizione variabili locali];
        <espressione>
}
```

Dove i parametri sono definiti secondo il formalismo delle variabili

Tutti i parametri vengono passati per valore
(è per questo motivo che esistono i puntatori)

```
int compare(int a, int b) {
        if (a == b) return 0;
        if (a > b ) return 1;
        if (a < b) return -1;
}
```

return

- Termina la funzione attualmente chiamata
- Il “parametro” verrà usato come valore di ritorno

Prototipi

```
<tipo> <nome> ([tipo parametri]);
```

Serve ad “*avvisare”* il compilatore che prima o poi sarà definita una funzione con quel nome ed attributi

Il prototipo non è obbligatorio se l’uso avviene solo dopo la definizione

Costruiamo un programma

- Direttive per il compilatore
- Variabili globali
- Protopi di funzione
- Funzioni definite dal programmatore
- Funzione main

main

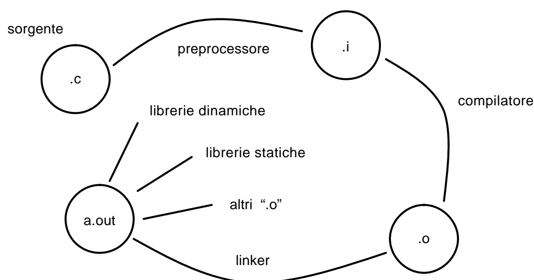
```
int main()
int main(int argc, char *argv[])
int main(int argc, char *argv[], char * envp[]);
```

- È la funzione che verrà chiamata dal sistema operativo al momento dell'esecuzione
- Il suo valore di ritorno deve essere intero e verrà restituito alla shell

Compilatore

- Traduce testo in codice eseguibile
- Il testo deve seguire una grammatica con specifiche convenzioni (un protocollo ?)
  - La grammatica e le convenzioni prendono il nome di "linguaggio di programmazione"

Il compilatore C



Preprocessore

- Lavora a livello di testo
- Accetta direttive interne al codice
  - #define
  - #if / #else / #endif
  - #undef
  - #error
  - #include <file.h>
- Accetta parametri
  - -D key
  - -D key=value

Preprocessore

- Predefinisce dei valori per indicare
  - Il compilatore
  - Il sistema operativo
  - Il linguaggio
  - Il processore usato
- NON sostituisce testo all'interno di stringhe
- Ci si *limita* al preprocessore con il parametro "-E"

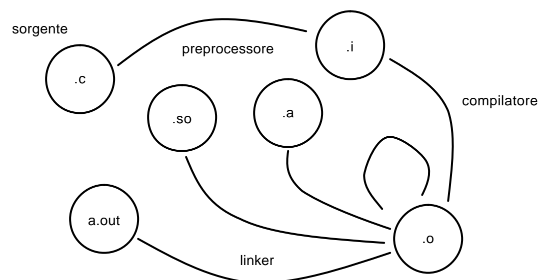
Compilatore (quello vero)

- Parte dal testo preprocessato e produce del codice binario non eseguibile
- Il compilatore si occupa di effettuare:
  - Controllo lessicale
  - Controllo sintattico
  - Ottimizzazione (parametro "-O")
  - Inserimento codici di debugging (parametro "-g")

Linker

- Prende un insieme di file oggetto e librerie e può creare:
  - Un nuovo file oggetto oggetto
  - Una libreria
  - Un eseguibile

Il compilatore C



nm

- Serve ad “esplorare” file binari, estrae i nomi di:
  - Variabili
  - Funzioni
  - Chiamate

definite all’interno del sorgente.

debugger

Abbiamo due opzioni

- gdb
  - Sempre presente
  - Piccolo
  - A caratteri
  - Decisamente scomodo
- ddd
  - Deve essere installato
  - Abbastanza pesante
  - Ambiente grafico
  - Semplice da usare