Il linguaggio C

Vincenzo Calabrò

Sommario

- Caratteristiche del linguaggio C
- Un esempio per cominciare
- Struttura di un programma C
- Regole sintattiche fondamentali
- Tipi di dato elementari
- Operatori
- Funzioni di libreria per l'input/output
- Strutture di controllo
- Array matrici e puntatori
- Funzioni e passaggio di parametri
- Array, puntatori e argomenti delle funzioni
- Compilazione di un programma C in ambiente UNIX
- Bibliografia

Caratteristiche del linguaggio C

- È un linguaggio **procedurale**.
- È un linguaggio **etremamente versatile** che si presta facilmente ad ogni compito.
- È un linguaggio **più vicino alla macchina** rispetto ad altri linguaggi di alto livello, quindi produce un **codice più compatto ed efficiente**.
- Fornisce al programmatore una **visibilità molto elevata sulla memoria** della macchina e fornisce degli strumenti che consentono di **ottimizzare fin nel sorgente il codice eseguibile**, condizionando il compilatore.
- La sintassi del C è molto compatta e per certi aspetti meno comprensibile di quella adottata da altri linguaggi: con la pratica però la sintesi del C si fa molto apprezzare.
- Il compilatore non esegue uno stretto controllo sui tipi di dato, come ad esempio in Pascal, ma demanda al programmatore il compito di verificare la correttezza semantica delle espressioni e la gestione di eventuali errori generati da espressioni non corrette.
- Consente lo sviluppo di **programmi facilmente *portabili*** da una piattaforma ad un'altra; esiste una normativa ANSI che stabilisce le caratteristiche standard di un compilatore C e la modalità standard di programmazione in C.

Un esempio per cominciare

Questo breve programma C legge in input *n* numeri interi e li memorizza in un array (funzione "`leggi_dati`"); quindi individua il massimo valore inserito (funzione "`massimo`") e lo stampa (funzione "`main`" che richiama le altre due).

```
#include <stdlib.h>
#include <stdio.h>
#define MAXN 100

int input_dati(int *v) {
  int i, n;
  printf("Numero elementi:");
  scanf("%d", &n);
  for (i=0; i<n; i++) {
    printf("elemento n.%d: ");
    scanf("%d", v+i);
  }
  return(n);
}
```

```
int massimo (int *v, int n) {
  int i, max;
  max = *v;
  for (i=1; i<n; i++) {
    if (max < *(v+i))
      max = *(v+i);
  }
  return(max);
}

int main(void) {
  int n, max, vett[MAXN];
  n = input_dati(vett);
  max = massimo(vett, n);
  printf("massimo=%d\n", max);
  return(1);
}
```

Struttura di un programma C 1/5

- Inclusione degli *header* delle librerie

  Gli *header* sono dei file (estensione ".h") che contengono i "prototipi" (le dichiarazioni) delle funzioni di libreria e la definizione di costanti. La parola chiave "`include`" non è una istruzione del linguaggio C, ma una "direttiva per il precompilatore".

  Esempio:

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

Struttura di un programma C 2/5

- Dichiarazione di variabili globali e costanti

Le variabili dichiarate al di fuori delle funzioni hanno una validità globale sull'intero programma e possono essere utilizzate da tutte le funzioni. **L'uso di *variabili globali* contribuisce a diminuire la portabilità e la riutilizzabilità delle funzioni ed è pertanto sconsigliato.**

Le **costanti** vengono gestite dal compilatore in modo assai diverso rispetto alle variabili (il valore delle costanti viene sostituito nel sorgente all'inizio della compilazione del programma) e consentono di rendere più efficiente il programma stesso. La parola chiave "`define`" è una direttiva del precompilatore e non una istruzione del linguaggio C.

Esempio:

```
#define PI 3.1415
#define raggio 10.001
int matrice[100][100], n, m;
float vettore[1000], x, y, z;
```

Struttura di un programma C 3/5

- Definizione delle funzioni

La possibilità di definire funzioni consente di frammentare il programma in tanti piccoli sottoprogrammi; se questa suddivisione rispecchia la frammentazione del problema in sotto-problemi più semplici ed elementari (secondo la metodologia *top-down*), il programma assume una architettura modulare e la riutilizzabilità delle funzioni è più semplice.

Per ogni funzione è necessario elencare tutti i parametri indicandone il tipo (*dominio* della funzione) ed il tipo del valore restituito dalla funzione stessa (*codominio* della funzione).

Se il dominio della funzione è nullo (non richiede parametri) si deve indicare "`void`"; analogamente nel caso in cui la funzione non restituisca alcun valore (codominio nullo) il tipo della funzione è "`void`".

Il *corpo* della funzione descrive, utilizzando le istruzioni del linguaggio C o richiamando altre funzioni definite precedentemente o presenti in qualche libreria, l'algoritmo che implementa la funzione stessa.

Struttura di un programma C 4/5

- Definizione delle funzioni

Esempi:

```
float massimo(float a, float b) {
  float c;
  if (a>b) {
    c = a;
  } else {
    c = b;
  }
  return(c);
}

void saluti(void) {
  printf("Ciao\n");
  return;
}
```

Struttura di un programma C 5/5

- Definizione della funzione “*main*”

La funzione “`main`” è una funzione esattamente come tutte le altre, ma con la caratteristica che viene eseguita automaticamente dal programma quando questo viene lanciato (è il punto di inizio del programma).

Il valore restituito dalla funzione `main` (tipicamente un intero, come richiesto dallo standard *ANSI-C*) può essere utilizzato come codice di controllo da un eventuale programma chiamante.

Esempio:

```
int main(void) {
  int a, b;
  a=3;
  b=5;
  printf(“massimo = %d\n”, max(a, b));
  return(1);
}
```

Regole sintattiche fondamentali

- Il linguaggio C è *case-sensitive*, parole scritte con lettere minuscole o maiuscole vengono interpretate come differenti.
- Ogni istruzione è terminata da un punto-e-virgola (“`;`”).
- Le variabili e le funzioni devono essere dichiarate prima di poter essere utilizzate.
- I nomi delle funzioni e delle variabili devono iniziare con una lettera dell’alfabeto o con il simbolo di sottolineatura (*underscore*: “`_`”) e sono costituiti solo da lettere alfabetiche, numeri ed il simbolo *underscore*. Es.: “`A`”, “`casa`”, “`Casa`”, “`_nome`”, “`Nome17`”.
- I parametri passati ad una funzione vengono specificati tra parentesi tonde e separati tra loro mediante la virgola. La chiamata di funzioni senza il passaggio di parametri (dominio di tipo *void*) deve comunque essere effettuata riportando le parentesi tonde dopo il nome della funzione. Es.: “`stampa_informazioni( );`”, “`A = maggiore(C, D);`”.
- Blocchi di istruzioni possono essere racchiusi da parentesi graffe (“`{`” e “`}`”).
- I commenti sono racchiusi tra “`/*`” e “`*/`”.
- Non è possibile definire funzioni all’interno del corpo di altre funzioni.

Tipi di dato elementari 1/2

- I principali tipi di dato elementari sono rappresentati nella seguente tabella, insieme al numero di bit utilizzati per la loro rappresentazione in memoria in una tipica implementazione del compilatore C:

Tipo	Descrizione	Bit utilizzati
char	Caratteri ASCII	8 bit
int	Numeri interi tra -32768 e 32768	16 bit
long	Numeri interi tra -2.147.483.648 e 2.147.483.648	32 bit
float	Numeri in singola precisione	32 bit
double	Numeri *floating point* in doppia precisione	64 bit

- Per dichiarare il tipo delle variabili si deve riportare il nome del tipo seguito dai nomi delle variabili separati da virgole. Esempio:

```
int a, b, c;
float x, y;
```

Tipi di dato elementari 2/2

- Le variabili possono essere anche di tipo **puntatore**: non conterranno un intero, un carattere o un numero in virgola mobile, ma l'indirizzo di memoria di una variabile di tipo intero, *char* o *floating point*. In questo caso si parla di variabili *puntatore*. Non è possibile definire delle variabili che siano dei puntatori "universali": è necessario dichiarare il tipo di variabile a cui un certo puntatore dovrà indirizzare.
- Per indicare che una variabile è un puntatore, nella sua dichiarazione si antepone al nome della variabile un asterisco ("*").
- Esempio:

```
int *pa, *pb, *pc;
float *x, *y;
```

Operatori 1/2

- **Operatori aritmetici**: “+” somma, “-“ sottrazione, “*” prodotto, “/” divisione, “%” modulo (resto della divisione intera).
- **Operatore di assegnazione in forma estesa**: “=” (es.: “`a=b*c;`”, “`x=-27.4`”).
- **Operatori aritmetici di assegnazione in forma compatta**: “++”, “--“ (es.: “`x--;`” decrementa di una unità il valore di `x`, è equivalente a “`x=x-1;`”, ma molto più efficiente), “+=”, “-=”, “*=”, “/=” (es.: “`x += 7;`” è equivalente a “`x = x+7;`”).
- **Operatori logici di confronto**: “==” (es.: “`a == b`” è vero se `a` e `b` hanno lo stesso valore, falso altrimenti), “<” (minore) “<=” (minore o uguale), “>” (maggiore), “>=” (maggiore o uguale), “`!=`” (diverso).
- **Operatori logici**: L’operatore booleano *unario* “*not*” è rappresentato dal simbolo “`!`”. L’operatore “*and*” è rappresentato da “`&&`” e l’operatore “*or*” è rappresentato da “`||`”. Le espressioni logiche vengono valutate da sinistra verso destra e la valutazione di una espressione si interrompe non appena è possibile stabilirne con certezza il valore.

Operatori 2/2

- Esempio:

```
if (a>=0 && b!=3) …
```

In questo caso se `a` è minore di zero allora l'intera espressione è sicuramente falsa (c'è una congiunzione logica "*and*") e quindi la condizione "b!=3" viene trascurata; se `a` è invece maggiore o uguale a zero, allora viene anche valutata la condizione "`b!=3`".

- **Operatori unari sugli indirizzi**: "&" restituisce l'indirizzo di memoria in cui è memorizzata una variabile; "*" restituisce il contenuto della variabile a cui fa riferimento un puntatore.
- Esempio:

```
int a, b, *pa;
a=3;      /* a contiene 3 */
pa = &a; /* pa contiene l'indirizzo di a */
b = *pa; /* b contiene 3 (il valore della variabile a cui punta pa) */
```

Funzioni di libreria per l'input/output 1/2

- Esistono numerosissime funzioni di libreria dedicate alle operazioni di I/O: molte però sono state specializzate per operare esclusivamente su una determinata piattaforma hardware.
- Le funzioni di I/O indipendenti dal tipo di hardware su cui opererà l'utente (terminale) sono le seguenti:
  - **printf**("*formato*", *espressione*, *espressione*, …);

    stampa sul terminale le espressioni in accordo con la formattazione espressa dalla stringa di formato. La stringa "\n" indica che il cursore deve andare a capo (*new line*); "%d" indica che in quella posizione andrà visualizzato il valore di una certa espressione numerica in formato decimale.

    Esempi:

```
printf("ciao\n");
printf("A=%d\n", a);
printf("%2d + %2d = %3d\n", a, b, a+b);
```

Funzioni di libreria per l'input/output 2/2

- **scanf**("*formato*", *indirizzo di memoria*, *indirizzo di memoria*, …);

  legge in input una riga dal terminale e memorizza i valori immessi dall'utente (separati da spazi o da caratteri di fine riga) nelle variabili il cui tipo è descritto nella stringa di formato ed il cui indirizzo di memoria è indicato nell'elenco successivo.

  Esempi:

```
scanf("%d", &n);
scanf("%d %d %d", &a, &b, &c);
```

- **getchar**( );

  restituisce il prossimo carattere in input.

  Esempio:

```
char c;
c = getchar();
```

Strutture di controllo 1/4

Il C è un linguaggio che favorisce l'uso delle tecniche della *programmazione strutturata*. Per questo mette a disposizione le seguenti strutture di controllo:

- Struttura di controllo condizionale:

  if (*condizione*) { *istruzioni* } **else** { *istruzioni* }

  La parola chiave "`else`" ed il blocco di istruzioni che la segue possono essere omessi. Se il blocco di istruzioni si riduce ad una sola istruzione allora possono essere omesse le parentesi graffe che lo racchiudono. Esempi:

```
if (a > b && a%2 == 0) {
  a=a/2;
  c = c+1;
} else {
  printf("%d", b);
}
```

```
if (a == 0) a=1;
```

```
if (a>b || (b>=c && a!=c)) c=a; else c=b;
```

- Strutture di controllo iterativo:

do { *istruzioni* } **while** (*condizione*);

Ripete il blocco di istruzioni fintanto che la condizione risulta essere vera; quando la condizione diventa falsa esce dal ciclo. È importante osservare che, visto che la condizione viene valutata a fine ciclo, il programma esegue sempre almeno una volta il blocco di istruzioni.

Esempio (stampa i numeri da 0 a 9):

```
i=0;
do {
  printf("%d", i);
  i=i+1;
} while (i<10);
```

while (*condizione*) { *istruzioni* }

Ripete il blocco di istruzioni fintanto che la condizione risulta verificata; quando la condizione diventa falsa esce dal ciclo. In questo caso se la condizione è falsa già prima di iniziare il ciclo, questo non viene iterato neanche una volta.

Esempio: (stampa i numeri da 0 a 9):

```
i=0;
while (i<10) {
  printf("%d", i);
  i=i+1;
}
```

Strutture di controllo 4/4

for (*assegnazione iniziale*; *condizione finale*; *incremento*) { *istruzioni* }

Generalmente l'iterazione mediante un *ciclo for* viene implementata utilizzando una variabile "contatore": il ciclo viene ripetuto fino a quando la variabile non supera una certa soglia, incrementandone il valore ad ogni iterazione.

Esempio (stampa i numeri interi da 0 a 9):

```
for (i=0; i<10; i++) {
  printf("%d ", i);
}
```

La formulazione dell'istruzione "`for`" può anche essere più articolata. Il seguente esempio stampa solo i numeri 10 e 9, perché per `i=8` la condizione "`i%4==0`" risulta verificata e quindi `flag` viene impostato a 1 ed il ciclo a termine:

```
for (i=10; i>=0 && flag!=1; i--) {
  if (i%4 == 0) flag=1;
  printf("%d", i);
}
```

Array, matrici e puntatori 1/3

- Sono delle collezioni di variabili, tutte dello stesso tipo, individuate da un nome comune e da un indice (nel caso degli array, o vettori) o da una *n*-pla di indici (nel caso di matrici con *n* dimensioni).
- Gli indici vengono indicati racchiudendoli fra parentesi quadre. Quando si dichiara una matrice è necessario indicarne tutte le dimensioni. Gli indici hanno una variabilità da 0 alla dimensione meno uno.
- Esempio:

```
int a[10];
int M[4][3];
```

Il primo esempio definisce un *array* (vettore) di 10 numeri interi individuati dagli indici 0, 1, …, 9. Il terzo elemento del vettore è "`a[2]`".

Il secondo esempio definisce una matrice di numeri interi con 4 righe e 3 colonne. Il primo elemento della matrice è "`M[0][0]`", l'ultimo è "`M[3][2]`".

Array, matrici e puntatori 2/3

È importante sapere come vengono memorizzate le matrici sulla macchina per poterle usare correttamente sfruttando a fondo le caratteristiche del C:

- Gli array vengono memorizzati in locazioni di memoria contigue: se `A` è un array di interi (supponiamo che ogni intero occupi 2 byte) e l'elemento `A[0]` viene memorizzato a partire dalla locazione di memoria 1000, allora l'elemento `A[1]` sarà memorizzato a partire dall'indirizzo 1002, `A[2]` all'indirizzo 1004, ecc.
- Le matrici sono considerate *array di array*. Vengono memorizzate disponendo le righe (che sono array e quindi memorizzati su locazioni di memoria contigue) una di seguito all'altra. Quindi se `M` è una matrice di 4 righe e 3 colonne di interi ogni elemento occuperà 2 byte di memoria. Se `M[0][0]` viene allocato nella posizione 5000, allora `M[0][1]` sarà allocato nella posizione 5002, `M[0][2]` nella posizione 5004. `M[1][0]` (il primo elemento della seconda riga) sarà memorizzato nella locazione successiva a quelle occupate dall'ultimo elemento della riga precedente (`M[0][2]`): la locazione 5004+2=5006.

Array, matrici e puntatori 3/3

- Se `M` è una matrice di *n* righe ed *m* colonne, la formula per individuare la locazione di memoria dell'elemento `M[i][j]` (di qualunque tipo sia la matrice) è la seguente: "`&M[0][0]+i*m+j`" (l'indirizzo di memoria del primo elemento, più *i* volte il numero di elementi presenti su ogni riga, più *j*).
- Le operazioni aritmetiche sui puntatori tengono conto del tipo di dato a cui punta la variabile. Esempio: se `A` è una variabile intera memorizzata a partire dalla locazione di memoria 2000, e `PA` è un puntatore ad `A` ("`PA = &A;`"), allora `PA+1` punterà alla locazione di memoria 2002 (e non a 2001).

Funzioni e passaggio di parametri 1/2

- Esistono tre modalità fondamentali:
  1. Mediante variabili “globali”.

     È un metodo fortemente sconsigliato: non consente di sviluppare funzioni autonome e riutilizzabili.
  2. Passando il **valore** di variabili locali.

     È il metodo corretto quando la funzione deve fare uso del valore di una variabile definita in un’altra funzione, senza la necessità di modificare tale valore nella variabile originale e senza dover accedere a locazioni di memoria adiacenti.
  3. Passando l’**indirizzo** di variabili locali.

     È il metodo corretto quando la funzione deve modificare il valore di una variabile definita in un’altra funzione e quando deve accedere a locazioni di memoria adiacenti a quella della variabile ricevuta come parametro.

Funzioni e passaggio di parametri 2/2

- In realtà il C passa gli argomenti alle funzioni soltanto tramite “chiamate per valore”, quindi se si intende passare un indirizzo lo si deve esplicitare indicandolo con l’operatore “&” o utilizzando una variabile puntatore di cui verrà passato il valore.
- Esempio **errato** (passaggio per *valore*):

```
…
int a, b;
…
if (a>b) scambia(a,b);
…

void scambia(int x, int y)
{
  int app;
  app = x;
  x = y;
  y = app;
  return;
}
```

- Esempio **corretto** (passaggio per *indirizzo*):

```
…
int a, b;
…
if (a>b) scambia(&a, &b);
…

void scambia(int *x, int *y)
{
  int app;
  app = *x;
  *x = *y;
  *y = app;
  return;
}
```

Array, puntatori e argomenti delle funzioni 1/2

Gli array ed i puntatori hanno una strettissima parentela in C: possono essere considerati quasi come la stessa cosa. Nella rappresentazione interna di array e matrici, il compilatore infatti si riconduce sempre ad una notazione che fa uso di puntatori, anche dove nel programma era stato fatto uso di una notazione "con indici".

L'uso dei puntatori al posto degli indici rende il programma più efficiente.

Notazione con indici	Notazione con puntatori
`int a[10];`	`int *a;` `a= malloc(sizeof(int)*10));`
`a[0]`	`*a`
`&a[0]`	`a`
`a[5] = 7;`	`*(a+5) = 7;`

Nel passaggio di un array ad una funzione è possibile soltanto passare l'indirizzo di un elemento dell'array stesso: indipendentemente dal tipo di notazione utilizzata.

Array, puntatori e argomenti delle funzioni 2/2

- Esempio (le funzioni "`main`" e "`main2`" sono tra loro perfettamente equivalenti; lo stesso vale per "`calc`" e "`calc2`"):

```
int main(void) {
 int vett[100];
 calc(vett, 30);
 …
}

int main2(void) {
 int *vett;
 vett =
   malloc(sizeof(int)*100);
 calc(vett, 30);
 …
}
```

```
void calc (int *v, int n) {
 int i;
 for (i=0; i<n; i++) {
   *(v+i) = i * i;
 }
 return;
}

void calc2(int v[], int n) {
 int i;
 for (i=0; i<n; i++) {
   v[i] = i*i;
 }
 return;
}
```

Compilazione di un programma C su UNIX 1/2

- La compilazione di un programma C richiede il passaggio attraverso le tre consuete fasi di traduzione da codice sorgente a codice eseguibile, passando attraverso il formato intermedio detto codice oggetto. Gli step di traduzione del programma sono i seguenti:
  1. Mediante un *editor* di testo ASCII (es.: *vi*, *emacs*, *pico*, ecc.) si produce il file in formato sorgente (estensione "`*.c`").
  2. Il *compilatore* riceve in input il programma in formato sorgente e lo traduce in formato oggetto.
  3. Il *linker* collega fra loro più moduli in formato oggetto e produce un unico file in formato eseguibile.
- È possibile inglobare in un'unica fase la compilazione ed il *linking* del programma: in questo modo si traduce direttamente da codice sorgente a codice eseguibile, senza accorgersi del passaggio attraverso il codice oggetto.
- È possibile distribuire il sorgente su più file sorgente: la funzione "`main`" dovrà però essere definita in uno solo di questi. Al momento della compilazione dovremo specificare il nome di tutti i file sorgente da compilare per produrre il file eseguibile desiderato.

Compilazione di un programma C su UNIX 2/2

- In ambiente UNIX il compilatore C viene invocato con il comando "`cc`" (C Compiler) o anche, in molti casi, "`gcc`" (GNU C Compiler). L'opzione "`-o`" serve per specificare il nome del file eseguibile di output. L'opzione "`-l`" serve per specificare i nomi di librerie aggiuntive da *linkare* al programma.
- Esempi

  Compila "`prova.c`" generando il file eseguibile "`prova`":

```
$ gcc prova.c -o prova
```

  Compila "`primo.c`" e "`secondo.c`" generando il file eseguibile "`pippo`":

```
$ cc primo.c secondo.c -o pippo
```

  Compila "`primo.c`" e "`secondo.c`" collegando anche la libreria matematica (il sorgente contiene la direttiva "`#include <math.h>`") e produce in output il file eseguibile "`pippo`":

```
$ gcc primo.c secondo.c -o pippo -lm
```

Bibliografia

- Brian W. Kernighan, Dennis M. Ritchie, *Linguaggio C*, Gruppo Editoriale Jackson, 1985.
- BrianW. Kernighan, Robert Pike, *UNIX*, Zanichelli, 1985.
- Herbert Schildt, *Linguaggio C - La guida completa*, McGraw-Hill, 1995.
- Mitchell Waite, Stephen Prata, Donald Martin, *C Primer Plus*, SAMS, 1984.
- M. J. Rochkind, *Advanced Unix programming*, Prentice-Hall, 1985.
- Richard W. Stevens, *UNIX - Sviluppo del software di networking*, Gruppo Editoriale Jackson, 1991.