Note sintetiche sul linguaggio C

Corso di algoritmi e strutture dati (I anno)
Anno Accademico: 1998-99

Prima versione completa: si prega di segnalare errori e punti oscuri

Indice

1 Premessa

Queste dispense non hanno la pretesa di sostituire il manuale di C consigliato [1]. Vogliono semplicemente sintetizzare gli aspetti principali, presentando ed arricchendo quanto visto a lezione con ulteriori approfondimenti. Inoltre, presuppongono la conoscenza degli elementi di programmazione introdotti nelle prime dispense del Prof. Puppo.

2 Tipi, operatori ed espressioni

Nel seguito verranno presentate le regole generali per costruire gli identificatori di un programma C. Quindi, verranno presentati i tipi e le espressioni ammissibili per il linguaggio.

2.1 Identificatori

Un *identificatore* rappresenta un nome di variabile/funzione/costante/tipo. Un identificatore *legale* è:

- una stringa non vuota
- sull'alfabeto $\{A, ..., Z\} \cup \{a, ..., z\} \cup \{0, ..., 9\} \cup \{\_\}$.

<u>Osservazioni</u>

- Non esiste limite sulla lunghezza degli identificatori. Tuttavia, sono significativi almeno i primi 31 caratteri per i nomi gestiti internamente dal sistema e i primi 6 caratteri per quelli esterni, utilizzati dall'utente (cioè quelli che usate nei vostri programmi).
- Le parole chiave sono riservate. Non possono cioè essere utilizzate come nomi di variabili.
- Le lettere maiuscole sono diverse dalle lettere minuscole. Quindi (`MAX` e `max` sono identificatori diversi).
- In generale: le costanti usano lettere *maiuscole*, le variabili/funzioni/tipi. *minuscole*
- Per i nomi locali si usano in genere stringhe corte, mentre per i nomi esterni si usano stringhe lunghe.
- Evitare di usare _ come primo carattere di un identificatore.

Esempio 1 *Alcuni esempi di identificatori:*

```
#define PI_GRECO 3.1415 extern int 68000; typedef char bool; int
get_op(int s);
```

2.2 Tipi

Esiste un ristretto numero di tipi di dato di base:

- `char`: un singolo byte, in grado di rappresentare uno qualsiasi dei caratteri del set locale;
- `int`: interi, riflettono l'ampiezza degli interi sulla macchina utilizzata;
- `float`: floating-point (razionali) in singola precisione;
- `double`: floating point (razionali) in doppia precisione.

Il tipo `bool` non esiste, le espressioni logiche valgono `0` se false e `1` (o comunque un valore diverso da `0`) se vere.

In aggiunta ai tipi introdotti in precedenza, si possono usare i *qualificatori*:

- `short`: può precedere `int`. `short int` indica in genere un intero di almeno 16 bit.
- `long`: può precedere `int` e `double`. Se precede `int`, rappresenta un intero di almeno 32 bit.

  I short non devono superare i long.
- `signed` e `unsigned`: possono precedere `char, short, int` e `long`.
  - `unsigned`: sempre positivi o nulli ed obbediscono alle leggi dell'aritmetica modulo 2^n, dove n è il numero di bit di tipo.
  - `signed`: positivi o negativi (complemento a due).

In generale ogni compilatore sceglie la dimensione appropriata ad un tipo in relazione all'hardware su cui opera.

Esempio 2 *Alcuni esempi di variabili con tipi diversi. I commenti rappresentano il range di valori che la variabile può assumere:*

```
signed char c;     /* -128 ... 127 */ unsigned char uc;  /* 0..
255 */ short i;           /* short int */ long j;        /* long
int */
```

2.3 Costanti

Nel seguito, verranno elencati tutti i tipi di costanti utilizzabili in C. Per ogni tipo di costante, verranno presentati alcuni esempi ed altre informazioni di carattere generale.

Intere

- Intera: `4875`;
- Intera long: suffisso `L`. Esempio: `4875932L`. Un intero troppo grande per essere considerato un `int`, verrà considerato un `long`.
- Costanti prive di segno (`unsigned int`): suffisso `u` o `U`. Esempio: 4875U.

Razionali

- Contengono il punto decimale, `125.26`, oppure un esponente, `1e-6`, oppure entrambi.
- Double: `123.4, 1e-2, 1.23e4`.
- Float: suffisso `f` o `F`. Esempio: `123.4F, 1e-2F, 1.23e4F`.
- Double long: suffisso `L`. Esempio: `123.4L, 1e-2L, 1.23e4L`.

Interi base 8/16

- Base 8: numero preceduto da 0. Esempio: `037` corrisponde al numero ottale 37.
- Base 16: numero preceduto da 0x o 0X. Esempio: `0X1F` corrisponde al numero esadecimale 1F.

- In entrambi i casi, si possono usare i suffissi u e L (unsigned o long).

Char

- Intero scritto sotto forma di carattere racchiuso tra apici *singoli*, come `´x´`.
- Il valore di una costante carattere è il valore numerico di quel carattere all'interno del set della macchina. Quindi *i caratteri sono interi*!!! Possono apparire in espressioni numeriche, anche se spesso usate per confronto con altri caratteri.
- `´0´` ha valore 48 su macchine con ASCII.
- `´\0´` è il carattere nullo. Vale 0 e viene usato come terminatore per le stringhe (vedi oltre).
- La seguente tabella elenca alcuni caratteri speciali:

Costante	Significato
\a	allarme
\b	backspace
\f	salto pagina
\n	new line
\r	ritorno carrello (return)
\v	tab verticale
\\	backslah
\?	punto interrogativo
\´	apice singolo
\"	apice doppio
\ooo	numero ottale
	\ooo sequenza di cifre ottali (0,...,7)
\xhh	numero esadecimale
	\xhh sequenza di cifre esadecimali (0,...,9,a,...,f,A,...,F)
\0	carattere con valore zero

Espressioni costanti

- Sono costituite da sole costanti e possono essere valutate al momento della compilazione.
- Possono essere inserite in ogni posizione nella quale si può trovare una costante.

Esempio 3 *Nella definizione del vettore che segue,* `MAXLINE + 1` *è una espressione costante:*

```
#define MAXLINE  1000 char line[MAXLINE + 1];
```

Stringhe

- Sequenza di zero o più caratteri racchiusi tra doppi apici e terminata dal carattere nullo `´\0´`.

  Esempio 4 *I seguenti sono alcuni esempi di stringhe:*
  `"\0". " Io sono una stringa"`
  `"Ciao" "a tutti"` *equivale a* `"Ciao a tutti"`.

- Tecnicamente: una costante stringa è un array di caratteri terminato da `´\0´`.

  Esempio 5 *Funzione per calcolare la lunghezza di una stringa:*

```
int strlen(char s[]) {
   int i;

   i=0;
   while (s[i] != '\0')
       ++i;
   return i;
}
```

- Si noti che 'x' è una costante di tipo `char` mentre `"x"` è una costante stringa.

Costanti enumerative

- In C è possibile definire un insieme di costanti numeriche a cui associare nomi simbolici. Ciò è possibile utilizzando la seguente dichiarazione:

  `enum nome-enumerazione {nome-1,...,nome-n}`

  Esempio 6 *Le costanti booleane possono essere definite utilizzando le costanti enumerative nel modo seguente:* `enum boolean {No,YES}`.

- Ad ogni costante in una enumerazione viene assegnato un valore numerico. Il primo nome in un'enumerazione ha valore zero, il secondo uno e così via, a meno che non vengano specificati valori espliciti.

- Esistono due modi per specificare i valori assegnati alle costanti:

  ```
  enum escape {BELL = '\a', BACKSPACE = '\b', TAB = '\t'};
  enum mesi { GEN = 1, FEB, MAR, APR, MAG, GIU, LUG, AGO, SETT, OTT, NOV, DIC };
  ```

  Nel primo caso, si specifica quale valore si vuole assegnare a ciascuna costante. Nel secondo caso, si specifica solo il primo. Tutti gli altri valore verranno assegnati in sequenza. Nel caso dell'esempio, a `FEB` verrà assegnato il valore 2 e così via. Se nessun valore viene specificato, il valore assegnato dal sistema alla prima costante è 0.

- È possibile dichiarare variabili di tipo `enum`, ma il compilatore non esegue il controllo di tipo su di esse.

2.4 Dichiarazioni

Tutte le variabili devono essere dichiarate prima di essere usate. La dichiarazione ne specifica il tipo. Vedremo nel seguito la differenza tra il concetto di *dichiarazione* e *definizione*.

- Due dichiarazioni alternative per variabili multiple:

  1. `int a,b,c;`
  2. `int a;`
     `int b;`
     `int c;`

- Una variabile può essere inizializzata all'interno di una dichiarazione (in realtà, nel contesto di una definizione). Si veda Sezione 4.6 per dettagli ulteriori.

  Esempio 7 *Alcuni esempi di inizializzazione di variabili:*


```
char esc='\\';
int i=9;
float eps= 1.0e-5;
```

- L'inizializzazione viene effettuata all'atto della compilazione.

- `const`: può essere applicato alla definizione di una qualunque variabile, per specificare che quel valore non verrà mai alterato. `const` può anche essere usato con vettori.

  Esempio 8 *Esempi di utilizzo di* `const`*:*

  ```
  const double e = 2.71828
  cost char msg[]="error";
  ```

  In entrambi i casi i valori assegnati non potranno essere modificati dal programma.

- `const` può anche essere usato per i vettori passati come argomenti ad una funzione, per indicare che la funzione chiamata non altera il vettore. Ad esempio, il prototipo `int strlen(const char[]);` specifica che la funzione `strlen` ha un parametro che non verrà modificato.

2.5 Operatori ed espressioni

Nel seguito verranno elencati gli operatori disponibili nel linguaggio C e le loro proprietà.

Aritmetici

- `+, -, *, /, %` sia su interi che su razionali.
- La divisione intera tronca la parte frazionaria.
- L'operatore `%` fornisce il resto della divisione intera. Non è applicabile ai float e ai double.
- Per i numeri negativi, il troncamento di `/` e il risultato di `%` dipendono dalla macchina.
- Precedenze (dalla più alta alla più bassa): `+, -` unari; `*,/,%`; `+,-`.
- Gli operatori sono associativi da sinistra a destra.

Relazionali e logici

- Precedenze (dalla più alta alla più bassa): `>, >=, <, <=`; `==, !=`; `&&` (AND); `||` (OR); `!` (NOT).
- Gli operatori relazionali hanno precedenza minore rispetto agli operatori aritmetici.
- Il valore numerico di un'espressioni logica o relazionale è 1 se l'espressione è vera, 0 se è falsa. Ad esempio `5 != 3` vale 1 (TRUE) mentre `5 <= 3` vale 0 (FALSE)
- `&&` e `||` sono valutati con *corto circuito*. Ciò significa che nel valutare l'espressione `exp1 && exp2`, si valuta prima `exp1`, se il valore è 0, allora `exp2` non viene valutata e si restituisce 0. Altrimenti, si restituisce il valore di `exp2`. `exp1 || exp2` viene valutata in un modo simile.

Incremento e decremento

- `++`: aggiunge 1 al suo operando.
- `--`: toglie 1 al suo operando.
- Vale sia la notazione prefissa (`++n`) che postfissa (`n++`). In entrambi i casi, l'effetto è l'incremento della variabile `n`. Tuttavia, l'espressione `++n` incrementa `n` prima di utilizzarne il valore mentre l'espressione `n--` incrementa `n` dopo che il valore è stato utilizzato. Ad esempio, se `n` vale 5, allora `x = n++;` assegna a `x` il valore 5 ma `x = ++n;` assegna a `x` il valore 6. In entrambi i casi, `n` diventa 6 dopo l'assegnazione.
- L'espressione a cui si applicano questi operatori deve denotare un indirizzo, quindi `(i+j)++` non è una espressione corretta.

Esempio 9 *La seguente funzione elimina da una stringa* `s` *tutte le occorrenze del carattere* `c`*.*

```
void squeeze(char s[],int c) {
   int i,j;
   for (i=j=0;s[i] != ´\0´;i++)
       if (s[i] !=c)
           s[j++] = s[i];
   s[j] = ´\0´;
}
```

Ogni volta che si incontra un carattere diverso da `c`*, tale carattere viene copiato nella posizione corrente* `j` *e soltanto allora* `j` *viene incrementato per disporsi sul carattere successivo diverso da* `c`*. Il programma precedente è equivalente a :*

```
void squeeze(char s[],int c) {
   int i,j;
   for (i=j=0;s[i] != ´\0´;i++)
       if (s[i] !=c)
           {
               s[j] = s[i];
               j++;
           }
   s[j] = ´\0´;
}
```

Bit a bit

- Operatori per la manipolazione di bit, applicabili solo ad operandi interi (`char`, `short`, `int` e `long`).
- Gli operatori sono illustrati nella seguente tabella:

Simbolo	Significato
&	AND bit a bit
|	OR inclusivo bit a bit
^	OR esclusivo bit a bit
<<	shift a sinistra
>>	shift a destra
~	complemento a uno

- Gli operatori `&` e `|` effettuano rispettivamente l'and e l'or bit a bit dei loro operandi (quindi, considerano la rappresentazione binaria degli operandi). Quindi, ad esempio, `02 & 04` è uguale a `0` mentre `02 | 04` è uguale a `06`.

- L'operatore ^ effettua l'or escusivo degli operandi, cioè restituisce vero se e solo se uno degli operandi è vero.
- L'operatore ~ cambia il valore di ciascun bit.
- Gli operatori di shift `<<` e `>>` spostano, verso sinistra e verso destra rispettivamente, il loro operando sinistro di un numero di bit pari al valore del loro operando destro, che deve essere positivo.
- Eseguendo uno shift a destra di una quantità `unsigned`, i bit rimasti liberi vengono posti a zero. Eseguendo invece lo shift a destra di una quantità `signed`, il risultato dipende dalla macchina: su alcuni sistemi i bit rimasti liberi vengono posti uguali al bit di segno (shift aritmetico) mentre su altri vengono posti a zero (shift logico).
- Da non confondere con gli operatori logici `&&`,`||`,`!`

Esempio 10 *L'istruzione* `n = n & 0177;` *maschera i bit di* `n` *lasciando visibili solo i 7 meno significativi.*

Assegnazione ed espressioni

- Se `espr1` e `espr2` sono espressioni, allora

  `espr1 op= espr2` equivale a `espr1 = espr1 op espr2`.

  Nel primo caso, `espr1` viene valutato una volta soltanto.
- Questo meccanismo è applicabile agli operatori `+`,`-`,`*`,`/`,`%`,`<<`,`>>`,`&`,`^`,`|`
- `x *= y+1` equivale a `x = x*(y+1)` e non a `x = x*y + 1`.

  Quindi in `expr1 op= expr2` il valore da assegnare a `expr1` corrisponde all'espressione `(expr1) op (expr2)`.
- `expr1` viene valutata solo una volta e deve denotare un indirizzo.

Esempio 11 *La seguente funzione conta il numero di bit positivi all'interno del suo argomento, che deve essere un intero.*

```
int bitcount(unsigned x) {
  int b;
  for (b=0;x!=0;x>>=1)
     if (x&01)
        b++;
  return b;
}
```

Espressioni condizionali

L'istruzione:

```
if espr1
    espr2;
else
    espr3;
```

può anche essere scritta nel modo seguente:

```
espr1 ? espr2 : espr3
```

Esempio 12 *L'espressione condizionale permette di scrivere codice molto compatto. Si consideri il seguente ciclo:*

```
for (i=0; i<n;i++)
   printf("%6d%c, a[i], (i%10 == 9 || i == n-1) ? ´\n´ : ´´);
```

Vengono stampati `n` *elementi di un vettore, 10 per linea, separando ogni colonna con uno spazio bianco, e terminando ogni linea con un new line.*

Esempio 13 *Esempio di espressione condizionale per il calcolo del massimo tra due valori:*

```
z = (a >b) ? a: b;  /* z = max(a,b) */
```

2.5.1 Regole di precedenza e ordine di valutazione

La seguente tabella mette in evidenza le regole di precedenza tra gli operatori. In genere sono impossibili da ricordare: tenetevi la tabella sottomano quando scrivete i programmi.

<table>
<tr><th>Operatore</th><th>Associatività</th></tr>
<tr><td>() []</td><td>da sx a dx</td></tr>
<tr><td>!= - ++ − + = * & (tipo) sizeof</td><td>da dx a sx</td></tr>
<tr><td>/ + -</td><td>da sx a dx</td></tr>
<tr><td><< >></td><td>da sx a dx</td></tr>
<tr><td>&</td><td>da sx a dx</td></tr>
<tr><td>^</td><td>da sx a dx</td></tr>
<tr><td>|</td><td>da sx a dx</td></tr>
<tr><td>&&</td><td>da sx a dx</td></tr>
<tr><td>||</td><td>da sx a dx</td></tr>
<tr><td>?:</td><td>da dx a sx</td></tr>
<tr><td>= += -= *= /= ,</td><td>da sx a dx</td></tr>
</table>

Come nota generale è opportuno ricordare che non ci sono regole nell'ordine di valutazione degli operandi ad eccezione di `&&`,`||`,`?:` e `,`.

Esempio 14 *Nelle seguenti istruzioni:*

```
x = f() + g(); a[i] = i++;
```

non si sa in quale ordine vengono valutate le varie (sotto)espressioni.

Consiglio. Usare parentesi e variabili temporanee per evitare pasticci.

2.6 Conversioni di tipo

Anche se tipato, il C è molto permissivo (non è *strongly typed*). Nel seguito verranno elencate alcune regole applicate per la conversione dei tipi.

Tipi di base

Il tipo `char` può essere usato liberamente in ogni espressione aritmetica (un valore di tipo `char` è un numero intero).

Esempio 15 *La funzione* `atoi` *converte una stringa di caratteri che rappresentano numeri nel numero corripondente.*

```
int atoi(char s[]) {
   int i,n;
   n=0;
   for (i=0;s[i] >= ´0´ && s[i] <= ´9´; i++)
         n = 10 * n + (s[i] - ´0´);
   return n;
}
```

La seguente funzione converte un carattere alfabetico nella corrispondente lettera maiuscola.

```
int lower(int c); {
   if (c>= ´A´ && c <= ´Z´)
      return c + ´Z´ -´A´;
   else
      return c;
}
```

Per i tipi aritmetici, esistono delle conversioni implicite quando si passa a tipi che occupano più spazio in memoria.

Esempio 16 *Esempio di conversione implicita:*

```
int i; float f; ... ... i + f ... /* conversione implicita da int
a float */
```

Le seguenti regole di conversione valgono per operandi `signed`:

- Se almeno un operando è di tipo:
  - long double: conversione a long double;
  - double: conversione a double;
  - float: conversione a float.
- Altrimenti:
  - conversione char/short in int;
  - conversione a long se almeno un operando è long.

In presenza di operandi `unsigned`, le regole si complicano.

Assegnazione

Il valore dell'espressione di destra viene convertito nel tipo di sinistra.

Esempio 17 *Alcuni esempi di conversione:*

```
int i; char c; i = c; /* conversione char -> int */ c = i; /*
conversione int -> char */
```

Alcune regole

- Quando si passa ad interi più corti la conversione elimina i bit più "alti".
- La conversione da float a int causa il troncamento della parte frazionale.
- La conversione da double a float può generare troncamento o approssimazione a seconda del compilatore.

Chiamata di funzione

Il tipo dei parametri attuali viene convertito in quello dei parametri formali.

Esempio 18 *Esempio di conversione durante il passaggio di parametri:*

```
double sqrt(double); int n; sqrt(n); /* conversione int -> double
*/
```

Casting

In qualsiasi espressione è possibile forzare particolare conversioni di tipo, tramite l'operatore di *cast*, utilizzando la seguente sintassi:
`(nome_tipo) espressione`
Dopo questa operazione, il valore di `espressione` viene convertito nel tipo `nome_tipo`.

Esempio 19 *Esempio di casting:*

```
int n; ...
printf("%f\n",(float)n);
```

L'argomento della `printf` *viene convertito da* `int` *a* `float`.

Considerazioni aggiuntive

- Espressioni che possono provocare una perdita di informazione, come l'assegnamento di un intero ad un short, o quello di un float ad un intero, producono al più un messaggio di warning.
- Espressioni prive di senso, come l'uso di un float come indice, non sono consentite.
- È opportuno ricordare che il risultato arbitrario di una computazione assegnato ad una variabile di tipo `char` potrebbe essere negativo. Meglio specificare `signed` o `unsigned`.
- Header <ctype.h> definisce una famiglia di funzioni che forniscono meccanismi di controllo e conversione indipendenti dal set di caratteri.

3 Strutture di controllo

Nel seguito verrà illustrata la sintassi e la semantica delle istruzioni supportate dal C.

Istruzione base

Espressione seguita da ';'.

Esempio 20 *Le seguenti sono esempi di istruzioni semplici.*


```
x =0; i++; printf("Ciao!");
```

Si deve ricordare che ';' è un terminatore e non un separatore.

Istruzione a blocchi

Una sequenza di istruzioni, possibilmente preceduta da una sequenza di dichiarazioni, viene considerata come una singola istruzione se delimitata da { }. Non ci vuole ';' dopo }. I blocchi vengono utilizzati nel corpo delle funzioni e nelle istruzioni multiple dopo `if`, `else`, `while`, e `for`.

If-else

```
if (espressione)
     istruzione1
else
     istruzione2
```

Se `espressione` ha valore diverso da 0 viene eseguita `istruzione1` altrimenti viene eseguita `istruzione2`, nel caso in cui ci sia. Altrimenti si prosegue in sequenza. Quindi:

```
if (espressione)
```

equivale a

```
if (espressione != 0)
```

La parte `else` è opzionale. L'else viene associato all'if più interno che ne è privo.

Esempio 21 *Nel seguente programma, l'*`else` *si riferisce all'*`if` *più interno. L'indentazione non conta.*

```
if (n>=0)
    for (i=0;i<n;i++)
         if (s[i] > 0)
           {
             ...
           }
else
    printf("error: n is negative\n");
```

Per eliminare l'ambiguità messa in evidenza dall'esempio precedente è opportuno usare { e }.

Esempio 22 *Nel programma seguente, l'*`else` *viene associato all'*`if` *più intenno.*

```
if (n>0)
   if (a>b)
      z=a;
   else
      z=b;
```

Se vogliamo associarlo a quello più esterno, usiamo le parentesi:

```
if (n>0)
   {if (a>b)
      z=a;
   }
else
   z=b;
```

Else-if

'if' in cascata

```
if (espressione_1)
     istruzione1
else if (espressione_2)
     istruzione2
       .
       .
       .
else
     istruzione
```

Viene sempre eseguita solo una delle n+1 istruzioni. L'istruzione i-sima viene eseguita sse `espressione_1`, ...,`espressione_n-1` sono false (cioè valgono 0) ed `espressione_n` ha un valore diverso da 0.

Esempio 23 `/* bin search */`

```
#include <stdio.h> #define MAX 100

int binsearch(int,int*,int);

int main() {
     int v[MAX];
     int i,n,num;

    /* leggo la dimensione del vettore */

         printf("fornisci il numero di elementi del  vettore: ");
         scanf("%d",&n);

     if (n < 0)
     {
                printf("errore!");
                return -1;
     }

     /* leggo il primo vettore */

     for (i=0; i < n;i++)
     {      printf("\nfornisci valore %d:  ",i+1);
            scanf("%d",&v[i]);
```



```
     }

     /* leggo l´elemento da cercare */

     printf("\nfornisci il numero da cercare: ");
     scanf("%d",&num);

     /*  stampo il risultato della funzione di ricerca */

     printf("\n\nil numero cercato e´ in posizione %d",binsearch(num,v,n)+1);
     printf("\n(0 significa non c´e´)\n\n ");
     return 0;
}


int binsearch(int x, int v[], int n) {
  int low = 0, high = n - 1, mid;
  while (low <= high)
  {
      mid = (low + high) / 2;
      if (x < v[mid])
           high = mid - 1;
      else if (x > v[mid]);
           low = mid + 1;
      else
       return mid; /* trovato */
    }
  return -1; /* non trovato */
}
```

Switch

Scelta plurima che controlla se un'espressione assume un valore all'interno di un certo insieme di costanti intere.

```
switch (espressione) {
  case espr-costante:    istruzione1
  case espr-costante:    istruzione2
  default:               istruzione3
}
```

- È costituito da un'espressione ed un blocco dove le istruzioni possono essere etichettate. La prima istruzione del blocco ad essere eseguita è quella etichettata con il valore di `espressione`. Nel caso in cui non esistesse, l'esecuzione inizia da `default`, se c'è, altrimenti il blocco intero viene saltato.
- Le espressioni che determinano le etichette devono essere *costanti* e tutte *differenti*.
- L'ordine delle etichette è irrilevante (`default` compresa).
- L'etichetta `default` è opzionale.

- Più etichette possono etichettare un'unica istruzione.
- Tutte le istruzioni che seguono quella etichettata dal valore dell'espressione vengono eseguite (a meno di uso di istruzioni di salto).
- Il caso in cui nessuna etichetta viene selezionata non rappresenta un errore.
- Per uscire: `break`

Esempio 24 *Programma per contare il numero di caratteri numerici, bianchi e non numerici e non bianchi digitati da tastiera:*

```
#include <stdio.h> main() {
        int c,i,nwhite,nother,ndigit[10];
        nwhite =nother =0;
        for (i=0;i<10;i++)
           ndigit[i] =0;
        while ((c=getchar()) != EOF )
           {
              switch(c)
              { case ´0´:...case ´9´: ndigit[c-´0´]++;
                                      break;
               case ´ ´:case ´\n´:case ´\t´: nwhite++;
                                             break;
               default: nother++;
                        break;
            }
        }
        printf("digits =");
        for (i=0;i<10;i++)
            printf("%d",ndigit[i]);
        printf(",white spaces = %d, other = %d\n",nwhite,nother);
        return 0;
```

Se non avessimo usato `break` *nel caso di carattere numerico, se il carattere numerico fosse stato un numero, il programma avrebbe proseguito in sequenza, incrementando comunque anche* `nwhite`.

While e for

```
while (espressione)
  istruzione
```

`istruzione` viene ripetuto finchè `espressione` non viene valutato in 0. Poichè, `espressione` viene valutata prima, `istruzione` può anche essere eseguita 0 volte.

```
for (espr1; espr2; espr3)
   istruzione
```

equivale a

```
espr1; while (espr2)
  istruzione
  espr3
```


In generale: `espr1` e `espr3` sono assegnamenti o chiamate di funzione, mentre `espr2` è un'espressione relazionale. Se manca `espr1` o `espr3`, il codice corrispondente non viene eseguito. Se manca `espr2`, il compilatore assume che il test sia sempre vero. Quindi, il ciclo `for(;;)` è infinito (non termina).

L'indice e la condizione limite di un ciclo for possono essere modificati all'interno del ciclo stesso e la variabile usata mantiene il suo valore anche al termine del ciclo.

Esempio 25 *Un'altra possibile implementazione della funzione* `atoi`*:*

```
#include <ctype.h>
int atoi(char s[])
{
  int i,n,sign;
  for (i=0;ispace(s[i]);i++)
    ;  /* istruzione vuota */
  sign = (s[i] == ´-´) ? -1 : 1;
  if (s[i] == ´+´ || s[i] == ´-´)
    i++;
  for (n=0;isdigit(s[i]);i++)
    n = 10 *n + (s[i]-´0´);
  return sign * n;
}
```

`espr1` e `espr3` possono essere composte da più istruzioni separate da virgola. In questo caso la valutazione viene effettuata da sinistra a destra. Il tipo e il valore dell'espressione complessiva sono quelli dell'ultima espressione eseguita.

La virgola non è un operatore.

Esempio 26 *Funzione per rovesciare una stringa:*

```
#include <string.h>

void reverse (char s[]) {
  int c,i,j;
  for (i=0,j=strlen(s)-1 ; i<j ; i++,j--)
    c=s[i], s[i] =s[j], s[j] = c;
}
```

Do-while:

```
do
   istruzione
while (espressione);
```

Simile a `while`, ma in questo caso `istruzione` viene eseguita prima della valutazione di `espressione`.

Esempio 27 `La seguente funzione converte un intero in un array di caratteri.`

```
void itoa(int n, char s[]) {
   int i,sign;
   if ((sign = n) < 0)
    n = -n;
```

```
    i=0;
    do
     s[i++] = n % 10 + ´0´;
    while ((n/=10)>0);
    if (sign <0)
     s[i++] = ´-´;
    s[i]= ´\0´;
    reverse(s);
}
```

Break e continue

Per uscire dal ciclo o dallo switch più interni, in modo incondizionato, si può usare `break`.

Esempio 28 *Funzione che rimuove gli spazi, i tab e i new line dalla fine di una stringa:*

```
int trim(chat s[]) {
  int n;
  for (n = strlen(s) - 1; n>=0;n--)
    if (s[n] != ´ ´ && s[n] != ´\t´ && s[n] != ´\n´)
        break;
  s[n+1] = ´\0´;
  return n;
}
```

Al contrario, `continue` forza l'inizio dell'iterazione successiva di un ciclo (non si applica allo switch). Essa cioè provoca l'esecuzione immediata della parte di controllo del ciclo.

Esempio 29 *Nella seguente porzione di programma, quando* `a` *diventa minore di 0, si esce passa al ciclo immediatamente successivo.*

```
for (i=0; i<n;i++) {
   if (a[i]< 0)
    continue;
   ...
}
```

Goto ed etichette

Esiste anche una forma di `goto`, per ritornare ad una certa etichetta (label), ma è preferibile non usarla.

Esempio 30 *Esempio di applicazione di* `goto`*:*

```
for (...)
  for (...)
   {
     if (disaster)
       goto error;
     ...
   }
... error: ...
```

Eseguendo l'istruzione `goto error`*, il programma prosegue l'esecuzione a partire dall'istruzione con label* `error`*.*

4 Funzioni e struttura dei programmi

L'obiettivo di questa sezione è introdurre la struttura di un programma C e le problematiche relative alla definizione e alla dichiarazione di variabili. Tali problematiche verranno ulteriormente approfondite nella Sezione 8, quando si introdurrà la compilazione separata.

4.1 Concetti di base

Funzione: unità di programmazione principale.
Programma: insieme di file sorgente.
File sorgente: insieme di definizioni di variabili e funzioni.

L'uso di più file sorgente verrà discusso con maggior dettaglio nella Sezione 8.
Una buona strutturazione di un programma in funzioni favorisce:

- la sua *comprensibilità*, dato che i dettagli inutili possono essere nascosti e l'organizzazione logica risulta più chiara;
- le sue possibili *modifiche*, che diventano più circoscritte e facilmente localizzabili;
- la *riutilizzazione* del codice.

Ogni definizione di funzione ha la forma:

```
tipo ritornato nome-funzione(dichiarazione argomenti) {
    dichiarazioni ed istruzioni
}
```

La funzione che non fa nulla e non ritorna alcun valore (funzione minimale) è la seguente:

```
dummy() {}
```

Osservazioni aggiuntive

- Se viene omesso il tipo del valore di ritorno (come nel caso precedente) esso viene considerato automaticamente `int`.
- Il tipo di una funzione che non ritorna alcun valore è `void`.
- La comunicazione tra le funzioni avviene tramite gli argomenti, i valori di ritorno e le variabili dichiarate esternamente ad ogni funzione (si vedano le dispense del Prof. Puppo).
- Il programma può essere suddiviso in diversi file. Una funzione deve trovarsi all'interno di un singolo file.
- L'istruzione `return` permette alla funzione di restituire un valore al chiamante. Quest'istruzione può essere seguita da una qualsiasi espressione:

  `return espressione;`

  Quando si incontra l'istruzione `return`, l'esecuzione della funzione viene immediatamente interrotta, il controllo passa al punto da cui era stata chiamata e il valore restituito è quello di `espressione`. Se necessario, l'espressione verrà convertita nel tipo del valore di ritorno della funzione.
- `return` non è obbligatorio e neppure l'espressione che lo segue.

- La funzione chiamante è libera di ignorare il valore ritornato (tipico nell'uso di `printf`, che restituisce sempre un valore quasi mai considerato dal programma chiamante). In tal caso, il chiamante non riceverà alcun valore di ritorno.
- Un altro caso in cui il chiamante riprende il controllo dell'esecuzione è quello in cui l'esecuzione della funzione chiamata termina per il raggiungimento della parentesi graffa di chiusura. In questo caso, la funzione si comporta come se ci fosse `return`.
- Una funzione per cui non è specificato il valore restituito, restituirà un valore indeterminato casuale.
- Non è ammessa la definizione di una funzione dentro l'altra.

4.2 Funzioni che ritornano valori non interi

- Una funzione può ritornare anche valori non interi (double, float).
- Per garantirsi che il programma chiamante conosca il tipo della funzione, prima di utilizzarla è necessario dichiararne le sue proprietà (la sua segnatura) utilizzando un *prototipo*. Il prototipo di una funzione ha la seguente forma:

  ```
  tipo ritornato nome-funzione(dichiarazione argomenti)
  ```

  Il prototipo specifica solo le proprietà della funzione, non il codice ad essa associata. Tale codice potrà essere contenuto nello stesso file del prototipo (magari anche dopo la funzione che lo ha invocato - si veda la Sezione 4.3 per quanto riguarda le regole di scope -) oppure in un altro file (si veda la Sezione 8).

  Esempio 31 *Esempio di prototipo:*

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

- Il prototipo e la definizione della funzione devono corrispondere. Tuttavia il compilatore si accorge di eventuali problemi solo se il prototipo e la definizione della funzione sono contenuti nello stesso file (si veda la Sezione 8 per il caso con più file sorgente).
- Senza prototipo la funzione viene dichiarata implicitamente dalla sua prima occorrenza in un'espressione e le viene assegnato un tipo di ritorno intero.

  Non viene invece fatta alcuna assunzione sul tipo dei parametri.
- La presenza del prototipo permette di applicare le corrette conversioni di tipo implicite, nonchè la corretta invocazione di funzioni definite in altri file (si veda Sezione 8).
- La dichiarazione e la chiamata di una funzione devono essere consistenti (i tipi degli argomenti devono essere consistenti).
- È importante osservare che: `char getchar();` indica che non si vuole fare nessuna assunzione sui parametri, mentre `char getchar(void)` indica espressamente che la funzione non ha alcun parametro.
- Nei casi dubbi, è sempre buona norma mettere i prototipi.

4.3 Variabili esterne ed automatiche

Le variabili dichiarate all'interno di un programma possono essere di due tipi (si veda la Sezione 8 per ulteriori approfondimenti):

- **Variabili interne**: parametri e variabili definiti nel blocco di una funzione. Sono anche dette *variabili automatiche*.

  Le variabili interne nascono (cioè vengono allocate) quando viene iniziata l'esecuzione della funzione nella quale sono state dichiarate. Devono quindi essere opportunamente inizializzate ad ogni chiamata.
- **Variabili esterne**: variabili definiti al di fuori di ogni funzione, potenzialmente utilizzabili da più funzioni. Vengono allocate una volta sola al momento della compilazione del programma.

<u>Osservazioni</u>

- Non è possibile annidare la definizione di funzioni. Quindi, tutte le funzioni sono sempre esterne (in questo senso, il C non è propriamente un linguaggio a blocchi).
- Se non esplicitamente specificato, variabili e funzioni esterne sono visibili all'esterno del loro file sorgente, utilizzando opportune dichiarazioni per quanto riguarda le variabili (si veda oltre).
- Non scomparendo al termine dell'esecuzione di una certa funzione, le variabili esterne sono in grado di conservare il loro valore anche dopo la terminazione della funzione che lo ha alterato. Sono quindi un'alternativa al parametri/valore restituito per realizzare la comunicazione tra funzioni.
- Ogni funzione può usare una variabile esterna accedendovi tramite il nome.
- Le variabili esterne sono permanenti e sopravvivono a qualsiasi chiamata di funzione.

  Sono utili se esistono funzioni che devono condividere dei dati. L'uso delle variabili esterne implica tuttavia una perdita di astrazione per cui i programmi diventano meno gestibili e controllabili.
- È buona norma utilizzare il minor numero possibile di variabili esterne.

Esempio 32 *Implementazione di stack con variabile esterna:*

```
#include <stdio.h> # define MAX 100

int sp =0; double val[MAX];

void push(double f) {
  if (sp < MAX)
     val[sp++]=f;
  else
     printf("errore:stach pieno\n");
}


double pop(void) {
```

```
  if (sp>0)
    return val[--sp];
  else
    {
      printf("vuoto\n");
      return 0.0;
    }
}
```

Regole di scope

- Lo *scope* di un nome è la porzione di programma all'interno del quale tale nome può essere correttamente usato.
- Per una variabile automatica, lo scope è il blocco all'interno del quale è stata definita. Lo stesso vale per i parametri. Ciò significa che usando i corrispondenti identificatori all'interno della funzione (in generale, al di fuori del blocco, si veda oltre) nel quale sono definiti, ci si riferisce sempre alla stessa area di memoria durante l'esecuzione della funzione (del blocco in generale, si veda oltre).
- Lo scope di una variabile esterna o di una funzione va dal punto in cui essa compare al termine del file sorgente in cui si trova.

Esempio 33 *Si considerino le seguenti dichiarazioni:*

```
main() {...} int sp =0; double val [MAXVAL]; void push(double f)
{...} double pop(void){...}
```

Secondo le dichiarazioni precedenti, `sp, val, push, pop` *non sono visibili nel main in quanto la dichiarazione del* `main` *precede la dichiarazione delle altre variabili e funzioni.*

Estensioni scope

Per utilizzare una variabile esterna fuori dal suo scope (cioè dal file in cui è contenuta), la variabile deve essere dichiarata come `extern` nel file in cui la si vuole utilizzare. Al contrario, le funzioni sono sempre visibili globalmente a tutto il programma, quindi per potere utilizzare una funzione al di fuori del suo scope è sufficiente specificarne il prototipo. Implicitamente verrà assunta la dichiarazione di `extern`.

Esempio 34 *Esempi di dichiarazione di variabili extern:*

```
extern int sp; extern double val[];
```

Da tenere ben presente:

- **Definizione di variabile**: causa allocazione di memoria, oltre a dichiarare le caratteristiche della variabile.
- **Dichiarazione di variabile**: nessuna allocazione.

Quindi:

- l'inizializzazione ha senso solo per le definizioni;
- la dimensione dell'array è obbligatoria per le definizioni, opzionale per le dichiarazioni.

Esempio 35 *Si supponga di utilizzare due file sorgente per uno stesso programma che gestisce degli array. Si supponga che i file siano i seguenti:*

```
FILE 1:

extern int sp;
extern double val[];
void push(double f) {...}
double pop(void) {...}
```

```
FILE 2:

int sp =0;
double val[MAXVAL];
```

Il primo file dichiara `sp` *e* `val` *ma non alloca alcuna area di memoria. Al contrario, il secondo file alloca e inizializza* `sp`*. Inoltre alloca lo spazio necessario al vettore* `val`*. Si noti che al momento della definizione è necessario specificare la dimensione massima del vettore.*

4.4 Variabili static

Le variabili esterne sono implicitamente visibili dall'esterno, cioè possono essere utilizzate al di fuori del loro scope (parte dal file sorgente dal punto in cui vengono definiti alla fine del file), quindi in un altro file, utilizzando la parola chiave `extern` all'atto della dichiazione della variabile nel file in cui la si vuole usare (si veda Sezione 8). Se si vuole evitare questa possibilità, è necessario utilizzare la parola chiave `static`. In questo modo, lo scope della variabile non potrà essere esteso in alcun modo. Come conseguenza di questo fatto, variabili static contenute in file diversi possono quindi avere lo stesso nome.

Esempio 36 *Si consideri la seguente definizione:*
`static char buf[BUFSIZE];`
`buf` *è una variabile esterna visibile solo all'interno del file in cui è definita.*

`static` può essere usato anche con le definizioni e dichiarazioni di funzioni, con lo stesso significato. Nel caso in cui sia applicato ad una dichiarazione, vuol dire che il prototipo in questione dovrà essere associato ad una definizione di funzione contenuta nel file in oggetto.

Esempio 37 *Si considerino le seguenti definizioni:*

```
static getch(void) {...} static void ungetch(int c){...}
```

`getch` *e* `ungetch` *sono visibili solo nel file in cui sono definite.*

`static` può infine essere utilizzato anche con le variabili interne, ma in questo caso il suo significato è profondamente diverso. In particolare, se una variabile interna viene dichiarata `static`, il suo valore non viene alterato da una chiamata all'altra della funzione della quale è definita. Ciò significa che il suo comportamente coincide con quello delle variabili esterne, anche se lo scope è diverso.

Esempio 38 *Si consideri la seguente definizione di funzione:*

```
int f(...) {
  static int x=1;
}
```

La variabile interna `x` *non è automatica ma statica. Ciò significa che la prima volta che viene eseguita la funzione, la variabile viene inizializzata a 1. Nell'esecuzioni successive,* `x` *mantiene il valore che aveva al termine dell'esecuzione precedente.*

Quindi il significato di `static` applicato ad una variabile esterna è profondamente diverso dal significato di `static` applicato ad una variabile interna: nel primo caso, limita la visibilità della variabile al file sorgente in cui è definita; nel secondo caso, ne modifica sostanzialmente il comportamento.

Nota informativa: le variabili Register

- Una variabile `register` viene collocata nei registri della macchina. L'uso di tali variabili può consentire di ottenere programmi più veloci.
- I compilatori, sono liberi di ignorare tale avvertimento.
- La dichiarazione `register` può essere applicata soltanto alle variabili automatiche ed ai parametri formali di una funzione.

  Esempio 39 *Esempio di uso di variabili register:*

  ```
  f(register unsigned m, register long n) {
     register int i;
      ...
  }
  ```

- Non è possibile conoscere l'indirizzo di una variabile register.

4.5 Blocchi

Le variabili possono essere definite dopo una qualsiasi parentesi graffa aperta '{' che introduce una istruzione composta (un blocco). Queste variabili vengono dette *automatiche*. A differenza dalle funzioni, i blocchi possono essere annidati.

Esempio 40 *Si consideri il seguente blocco:*

```
if (n<0) {
  int i;
  for (i=0;i<n;i++)
  ....
}
```

La variabile `i` *è visibile solo nel blocco.*

Le differenze principali tra le variabili esterne e le variabili automatiche sono le seguenti:

- il ciclo di vita è limitato alla singola esecuzione del blocco/funzione;
- l'inizializzazione esplicita e l'allocazione di memoria viene ripetura per ogni entrata nel blocco/funzione (si veda la Sezione 4.6);
- il sistema non applica nessuna inizializzazione implicita (si veda Sezione 4.6).

Lo scope delle variabili automatiche è il blocco nel quale sono definite. Nascondono quindi variabili con nomi uguali dichiarate in blocchi più esterni.

Esempio 41 *Si consideri il seguente stralcio di programma:*

```
f(int y) {...} int x;

g(float f) {
   int x=1;
   ...
   if (x >0)
     {
        double f = 1.0;
     }
}
```

La dichiarazione del parametro `f` *nella definizione della funzione* `g` *nasconde la funzione* `f`*. La dichiarazione della variabile automatica* `x` *nasconde la dichiarazione della variabile esterna* `x`*. Infine, la dichiarazione della variabile double* `f` *nasconde il parametro* `f`*.*

L'uso di molte variabili con lo stesso nome è da evitare in quanto rende i programmi molto meno leggibili.

4.6 Inizializzazione

Al momento della definizione delle variabili, è possibile assegnare alle stesse un valore, che verrà inserito nello spazio di memoria allocato all'atto della compilazione. Esistono due tipi distinti di inizializzazione: esplicita, cioè richiesta dall'utente, o implicita, cioè realizzata dal sistema quando l'utente non richiede alcuna inizializzazione esplicita. Nel seguito, verranno analizzati entrambi i tipi di inizializzazione.

Inizializzazioni implicite

- *Variabili esterne e static*: vengono inizializzate a zero.
- *Variabili automatiche e register*: valori indefiniti.

Inizializzazioni esplicite

1. **Variabili scalari**
   - L'inizializzazione esplicita delle variabili scalari avviene al momento della loro definizione, postponendo al loro nome un segno di uguale ed un'espressione:

     ```
     int x=1; char squote= '\''; long day=1000L*60L*60L*24L;
     ```

   - *Esterne e static*: inizializzazione deve essere una costante e viene eseguita prima dell'esecuzione del programma.
   - *Automatiche e register*: viene eseguita ogni volta che inizia l'esecuzione della funzione o del blocco di interesse.

     Il valore può non essere una costante. Possono comparire valori definiti in precedenza ed anche chiamate di funzione (inizializzazione come modo compatto per eseguire assegnamenti).
2. **Vettori**

- Si possono inizializzare postponendo alla sua dichiarazione una lista di valori iniziali, tra parentesi e separati da virgole.
  `int days[]={31,28,31,30,31,30,31,31,30,31,30,31}`
- La dimensione del vettore, se omessa, viene calcolata dall'inizializzazione.
- Se i valori specificati sono minori rispetto a quelli richiesti da tipo, gli altri elementi del vettore vengono inizializzati a zero, se il vettore è una variabile esterna o static, oppure rimangono indefiniti, se il vettore è una variabile automatica.
- Se la lista è troppo lunga rispetto al tipo, si genera un errore.

5 Puntatori e vettori

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile. I puntatori in C vengono utilizzati molto spesso, perchè in parte rappresentano l'unico modo possibile di esprimere un calcolo, ed in parte perchè consentono di ottenere codice particolarmente compatto ed efficiente. nel seguito, dopo avere introdotto le nozioni di base sui puntatori, analizzeremo le analogie tra puntatori e vettori, quindi introdurremo i puntatori a carattere, i vettori multidimensionali e i vettori di puntatori o di vettori.

5.1 Puntatori ed indirizzi

Informazioni generali

- Una macchina possiede un vettore di celle di memoria numerate in modo consecutivo. Queste celle possono essere manipolate singolarmente o a gruppi. Ad esempio, un byte viene considerato come un `char`.
- Un *puntatore* è un gruppo di celle (spesso due o quattro) che può contenere un indirizzo di una locazione di memoria contenente valori di un particolare tipo.
- Se `c` è un elemento di un certo tipo `T` e `p` è un puntatore a `c`, allora
  `p = &c`
  assegna l'indirizzo di `c` alla variabile `p` e si dice che `p` punta a `c`.
- `&` si applica solo agli oggetti della memoria: variabili ed elementi dei vettori. Non può essere applicato alle espressioni, alle costanti o alle variabili `register`. Quindi, l'espressione `&(3+i)` non è corretta.
- L'operatore `*` è l'operatore di deferimento. Può essere applicato ad una qualsiasi espressione di tipo puntatore e, come risultato, accede l'oggetto puntato. Quindi, in riferimento all'esempio precedente:
  `*p` è uguale a `c`.
- Dichiarazione di un puntatore:
  `tipo *tp`
  dichiara che `tp` è un puntatore di tipo `tipo`.

  Esempio 42 `int *p` *dichiara che* `p` *è un puntatore ad oggetti interi.* `int *f(char *)` *dichiara che* `f` *è una funzione che prende come parametro un puntatore ad un carattere e restituisce un puntatore ad un intero.*

Esempio 43 *Si analizzimo le seguenti dichiarazioni:*

```
int x=1, y=2, z[10]; int * ip;   /* ip e´ un puntatore a int */ ip
= &x;    /* ip punta a x */ y = *ip;    /* poiche´ x vale 1, dopo
questa assegnazione anche y vale 1 */ *ip = 0;    /* x vale 0 */
ip = &z[0]; /* ip punta a z[0] */
```

Si noti che `*&x` *coincide con* `x` *e che* `&*p` *denota lo stesso valore di* `p`.

È importante osservare che nella dichiarazione `int *p`, `*p` è a tutti gli effetti una variabile di tipo `int`. Se `ip` punta ad un intero `x`, allora `*ip` può ricorrere in ogni contesto nel quale può comparire `x`.

- I puntatori sono delle variabili. Quindi si possono usare anche senza deferimento.

  Esempio 44 `iq = ip` *assegna il contenuto di* `ip` *a* `iq`. *Se* `iq` *e* `ip` *sono due puntatori, il risultato è quello di fare puntare* `iq` *all'oggetto puntato da* `ip`.

Chiamate per valore e per riferimento

Il C passa alle funzioni gli argomenti per valore. Quindi, una chiamata di funzione non può modificare direttamente il valore delle variabili passate come parametri (i parametri vengono considerati come variabili locali).

L'uso dei puntatori permette di simulare la chiamata per riferimento (cioè con la possibilità di modificare i parametri). Questo è possibile passando come parametri non le variabili ma gli indirizzi corrispondenti. Anche tali indirizzi verranno considerati come contenuti in variabili locali, ma accedendo le locazioni individuate da tali indirizzi è possibile modificare i valori di partenza.

Esempio 45 *Si consideri la seguente funzione, che "vorrebbe" scambiare il contenuto di due variabili passate come argomento:*

```
void swap(int x, int y) {
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Supponiamo adesso di volere cambiare il contenuto di due variabili `a` *e* `b`, *dichiarate come segue:* `int a= 1, b= 2;`. *Si potrebbe pensare di chiamare la funzione* `swap(a,b)`, *tuttavia questo approccio non funziona, in quanto alla funzione non vengono passati gli indirizzi delle locazioni di memoria corrispondenti ad* `a` *e* `b`, *necessari per andare a cambiare i valori, ma solo i valori contenuti in tali locazioni. Quindi la chiamata non altera i valori associati a* `a` *e* `b` *nel programma chiamante. Per effettuare effettivamente lo swap, il passaggio per riferimento deve essere simulato nel modo seguente:*

```
void swap(int *x, int *y) {
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

A questo punto, per effettuare lo swap di `a` *e* `b` *è necessario effettuare la seguente chiamata di funzione* `swap(&a,&b)`*. In questo modo, alla funzione vengono passati gli indirizzi delle locazioni e lo swap viene effettivamente eseguito.*

5.2 Puntatori e vettori

In C esiste una stretta relazione tra puntatori ed array. In particolare, ogni operazione eseguibile indicizzando un vettore può essere eseguita anche tramite l'uso di puntatori. In generale, la versione con i puntatori è più veloce. Si consideri ad esempio la seguente dichiarazione di vettore: `int a[10]`. Per definizione, il valore di `a` è l'indirizzo della locazione con indice `0` del vettore.

Esempio 46 *Si considerino le seguenti dichiarazioni ed istruzioni:*

```
int a[10];

int *pa;

pa = & a[0];
```

Dopo questa operazione, `pa` *e* `a` *hanno valori identici. Quindi, l'operazione* `pa = & a[0]` *coincide con* `pa = a`. `pa +1` *rappresenta l'indirizzo dell'elemento del vettore con indice* `1`*, cioè* `pa + 1` *coincide con* `&a[1]`. `*(pa + 1)` *rappresenta l'elemento del vettore con indice* `1`*, cioè* `*(pa + 1)` *coincide con* `a[1]`.

In generale, se `pa` è un puntatore, `pa + i` punta a i elementi dopo quello puntato da `pa`. Il calcolo dell'indirizzo può essere effettuato conoscendo il tipo a cui punta `pa`, in quanto si conosce il numero di byte occupati da ciascuna locazione.

A seguito delle analogie precedentemente discusse: `a[i]` coincide con `*(a+i)`, `&a[i]` coincide con `(a+i)`, `pa[i]` coincide con `*(pa+i)`.

<u>Osservazioni</u>

- È opportuno ricordare che un puntatore è una variabile, quindi può comparire in ogni contesto in cui una variabile può comparire (espressioni, assegnamenti, ...). Un vettore non è una variabile, quindi, ad esempio, costruzioni come `a=pa` o `a++` non sono legali.
- Quando il nome di un vettore viene passato ad una funzione, ciò che viene passato è la posizione dell'elemento iniziale. All'interno della funzione chiamata, questo argomento è una variabile locale. Quindi, un nome di vettore passato come parametro è in realtà un puntatore, ovvero una variabile che contiene un indirizzo.

  Un parametro di funzione può quindi essere dichiarato in due modi equivalenti:

  `char s[];`

  `char *s;`

  Meglio la seconda forma. In entrambi i casi, si può decidere se manipolare il parametro come puntatore o come vettore. In ogni caso, gli array sono sempre passati come puntatori, e quindi sono sempre passati per riferimento.
- Al momento della chiamata, si può passare un vettore completo o solo una parte.

Esempio 47 *Le seguenti sono alcuni modi alternativi per chiamare una funzione, di prototipo* `int f(int vet[])` *su un array definito come* `int a[20]`*;*
`f(a)`*: viene passato tutto il vettore,*
`f(&a[2])`*: viene passato il vettore a partire dalla terza posizione;*
`f(a+2)`*: viene passato il vettore a partire dalla terza posizione.*

5.3 Aritmetica degli indirizzi

Le operazioni consentite sui puntatori sono le seguenti:

- assegnamento fra puntatori dello stesso tipo;
- addizione e sottrazione tra puntatori ed interi;
- sottrazione e confronto fra due puntatori ad elementi di uno stesso vettore.

Tutte le altre operazioni sono illegali. In particolare, è illegale:

- sommare due puntatori;
- moltiplicare due puntatori;
- applicare operazioni di shift;
- applicare altri operatori bit a bit;
- sommare a puntatori quantità float o double;
- ad eccezione per il tipo `void *`, è illegale assegnare ad un puntatore di un tipo, un puntatore di un tipo diverso, senza applicare l'operatore di cast.

Esempio 48 *Si consideri la seguente definizione della funzione* `strlen` *per calcolare la lunghezza di una stringa. Si confronti questa funzione con quella presentata nell'Esempio 5.*

```
int strlen(char *s) {
    char *p=s;
    while (*p != ´\0´)
       p++
    return p-s;
}
```

5.4 Puntatori a caratteri

- Una stringa costante è un vettore di caratteri. Nella rappresentazione interna, il vettore è terminato dal carattere nullo '`\0`'.

  Quando in un programma compare una stringa di caratteri, l'accesso ad essa avviene tramite un puntatore al primo elemento della stringa.

  Esempio 49 *Si consideri la seguente dichiarazione di puntatore e successiva assegnazione:*

  ```
  char *pmessage;

  pmessage = "eccoci qui!";
  ```

  L'ultima istruzione assegna a `pmessage` *un puntatore ad un array* costante *di caratteri.*

- Esiste una importante differenza tra l'inizializzazione di vettori di caratteri e puntatori a caratteri. Si consideri il seguente esempio:

  ```
  char amessage[] = "eccoci qui!";  /* un vettore */

  char *pmessage = "eccoci qui!";   /* un puntatore */
  ```

Nel primo caso, `amessage` è un vettore sufficientemente grande da contenere la stringa considerata. Si potrà cambiare il contenuto delle varie locazioni del vettore (*stringa modificabile*), ma l'area di memoria referenziata rimarrà sempre la stessa.

Nel secondo caso, `pmessage` è un puntatore ad una stringa *costante*, quindi tramite nuove assegnazioni, potrà puntare a locazioni diverse da quella a cui punta inizialmente. Tuttavia, la stringa in questo caso *non è modificabile*.

Come conseguenza di questo fatto, non è possibile assegnare ad un puntatore a carattere una stringa costante. Ad esempio, l'istruzione

```
scanf("%s",pmessage)
```

produce un errore in esecuzione. Infatti, leggendo la stringa in input e memorizzandola a partire dalla posizione puntata da `pmessage`, poichè nessuna locazione di memoria era stata precedentemente allocata, è possibile accedere locazioni già occupate. Affinchè non si generi alcun errore, è necessario allocare dinamicamente la memoria. Si veda a questo proposito la Sezione 5.7.

- *Inizializzazione*. Due possibili dichiarazioni:

```
char pattern[] = "pippo";

char pattern[] = {´p,´i´,´p´,´p´,´o´, ´\0´};
```

5.5 Vettori multidimensionali

- Un vettore multidimensionale può essere pensato come una tabella multidimensionale. Ad esempio, se la dimensione è 2, il vettore si può pensare come una matrice bidimensionale.

- *Dichiarazione*:

  `tipo nome[MAX1][MAX2]`

  dove: `tipo` è il tipo degli elementi del vettore, `nome` è il suo nome, `MAX1` è il numero delle righe e `MAX2` è il numero di colonne.

- Gli elementi di un vettore multidimensionale vengono memorizzati per righe.

- *Inizializzazione*: Si supponga di volere definire un vettore con due righe e 12 colonne. Si supponga che la prima riga contenga il numero di giorni presenti nei vari mesi in un anno non bisestile mentre la seconda riga contenga il numero di giorni presenti nei vari mesi in un anno bisestile. Il vettore può essere dichiarato e inizializzato nel modo seguente:

```
char daytab[2][12] = {
      {31,28,31,30,31,30,31,31,30,31,30,31},
      {31,29,31,30,31,30,31,31,30,31,30,31}
 };
```

- *Riferimento*:

  `daytab[i][j]`

  dove `i` rappresenta la riga e `j` la colonna.

  La seguente notazione invece è sbagliata:

  `daytab[i,j]`

- Se un vettore bidimensionale viene passato ad una funzione, la dichiarazione del parametro nella funzione deve comprendere il numero di colonne; il numero di righe non è rilevante, perchè ciò che viene passato è un puntatore ad un vettore di righe, nel quale ogni riga è un vettore di una certa dimensione. Quindi, se un vettore bidimensionale è parametro di una funzione, la funzione può essere definita come segue:

  `f(int daytab[][13]) {...}`

  oppure

  `f(int daytab[2][13]) {...}`

  oppure

  `f(int (* daytab)[13]) {...}`

  In questa ultima notazione, `int (* daytab)[13]` rappresenta un puntatore ad un array di 13 interi. Si noti la differenza con:

  `int * daytab[13]`

  che rappresenta un array di 13 puntatori ad interi.

- Più in generale, soltanto la prima dimensione di un vettore multidimensionale è libera; tutte le altre devono essere specificate.

- In generale, la notazione con puntatori, anche nel caso di vettori multidimensionali, è più flessibile. Si considerino ad esempio le seguenti dichiarazioni:

```
char a[10][20];

char *b[10];
```

  Entrambe le definizioni possono essere utilizzate per memorizzare un vettore bidimensionale. La differenza sta nel fatto che nel primo caso il vettore occuperà sempre 200 locazioni carattere, mentre nel secondo caso, lo spazio occupato dipende dalla lunghezza delle stringhe effettivamente memorizzate.

5.6 Vettori di puntatori e vettori di vettori

È possibile dichiarare che

- `int *ptr[MAX]`

  dichiara che `ptr` è un vettore di `MAX` elementi, tale che ciascun elemento è un puntatore ad un intero. Viene quindi allocato lo spazio necessario a contenere `MAX` puntatori.

  Se al contrario scriviamo:

  `int ptr[MAX][MAX1]`

  viene allocato lo spazio necessario a contenere `MAX * MAX1` interi.

- *Osservazione*.

  Si consideri la seguente dichiarazione:

  `char *name[] = {"Charlie","Lucy","Snoopy","Linus"}`

  La precedente dichiarazione asserisce che `name` è un array di 4 puntatori a carattere. Tale vettore viene anche inizializzato.

Si consideri adesso la seguente dichiarazione:

`char aname [][8] ={"Charlie","Lucy","Snoopy","Linus"}`

In questo caso, `aname` è una matrice di caratteri, tale che ogni riga è lunga 8, cioè ci sono 8 colonne. Questo significa che le stringhe che possiamo inserire in `aname` hanno dimensione fissata, mentre quelle che possiamo inserire in `name` possono avere dimensione variabile.

5.7 Array dinamici

In tutti gli esempi visti in precedenza, affinchè il sistema fosse in grado di allocare uno spazio di memoria sufficiente a mantenere gli elementi di un vettore, era necessario specificare la dimensione del vettore. Se poi veniva inserito un numero inferiore di elementi, alcune posizioni del vettore, benchè allocate, non venivano utilizzate.

In C, grazie all'analogia tra vettori e puntatori, è possibile gestire un vettore in modo *dinamico*, allocando cioè una quantità di memoria nota solo a run time. Questo è possibile utilizzando opportune funzioni di libreria. La libreria che contiene queste funzioni è `stdlib.h` e le istruzioni sono le seguenti:

`void * malloc(int num);`

dove `num` rappresenta il numero di byte che devono essere globalmente allocati.

`void * calloc(int num-pos,int num-byte-per-pos);`

dove `num-pos` rappresenta il numero di locazioni da allocare mentre `num-byte-per-pos` rappresenta il numero di byte necessari a memorizzare ciascuna locazione.

Esempio 50 *Si supponda di volere leggere n numeri e calcolarne il massimo. Per risolvere questo problema è necessario memorizzare i valori letti (supponiamo che siano interi) in un vettore. Se si utilizza un vettore statico (dimensione costante), il vettore deve essere dichiarato come segue:*

`int numeri[MAX];`

dove `MAX` *rappresenta il numero massimo di numeri che possono essere considerati. Dovrà poi essere dichiarata una variabile* `n`*, corrispondente alla lunghezza della sequenza di numeri considerata in ciascuna esecuzione. Il valore di* `n` *verrà quindi richiesto all'utente.*

Supponiamo che `MAX` *sia 100 e* `n` *sia 4. In questo caso verranno allocate 100 locazioni di memoria, benchè all'atto dell'esecuzione ne vengano utilizzate solo 4. Per risolvere questo problema, è possibile allocare la memoria dinamicamente. In questo caso sarà sufficiente dichiarare un puntatore ad interi:*

`int *numeri;`

Quindi, la memoria potrà essere allocata durante l'esecuzione del programma mediante l'esecuzione della seguente istruzione:

`numeri = (int *) calloc(n, sizeof(int));`

dove:

- `n` *è un valore dato in input;*
- `sizeof` *è una funzione che prende come argomento l'identificatore di un tipo e restituisce il numero di byte necessari a memorizzare un valore di quel tipo;*
- *l'operazione di casting* `(int *)` *è necessaria poichè la funzione* `calloc` *restituisce un valore di tipo* `void *` *mentre a noi serve il tipo* `int *`*. Poichè* `void *` *è il tipo puntatore più generale, il casting è corretto.*

La chiamata a `calloc` *può essere rimpiazzata con la seguente chiamata a* `malloc`*:*

`numeri = (int *) malloc(n * sizeof(int));`

Le locazioni di memoria allocate con `calloc` vengono inizializzate a 0 mentre la memoria inizializzata con `malloc` non viene inizializzata (non si conosce il numero di locazioni).

La memoria allocata durante l'esecuzione può essere rilasciata (cioè deallocata,) permettendo un successivo utilizzo delle stesse locazioni, mediante la funzione `free`. Tale funzione prende come argomento un puntatore che punta ad una zona di memoria precedentemente allocata con `calloc` o `malloc` e la rilascia al sistema. Nell'esempio precedente, si può quindi scrivere `free(numeri)`.

6 Strutture

Nel seguito verranno introdotti alcuni meccanismi per definire tipi di dato complessi in C.

6.1 Fondamenti

- Una struttura è una collezione di una o più variabili, raggruppate da un nome comune per motivi di maneggevolezza.
- In generale una struttura viene definita come segue:

```
struct nome-struttura {
    tipo1 campo1;
     .
     .
     .
    tipon campon;
  };
```

  Il `nome-struttura` è opzionale.

  Esempio 51 *La seguente dichiarazione introduce un tipo di dato per la rappresentazione dei punti:*

```
struct point {
    int x;
    int y;
  };
```

- Una dichiarazione `struct` definisce un tipo (il tipo `struct nome-struttura`). Quindi, la parentesi graffa di chiusura può essere seguita da una lista di variabili:

  `struct  { ... } x,y,z;`

- Una dichiarazione di struttura non seguita da una lista di variabili non riserva alcuna area di memoria. Se la dichiarazione specifica un *tag* (`nome-struttura`) questo può essere utilizzato in dichiarazioni successive, come di seguito specificato:

```
struct nome-struttura {
    tipo1 campo1;
     .
     .
     .
    tipon campon;
```

```
};
```

```
struct nome-struttura nome-var1,...,nome-varm;
```

- Una variabile di tipo struttura può essere inizializzata specificando un valore per ogni campo della struttura. Ad esempio:

  `struct point maxpt ={320,200};`

- Il membro di una particolare struttura viene riferito usando la notazione:

  `nome-struttura. campoi`

  `point.x`

- Le strutture si possono innestare, cioè una struttura può essere utilizzata per definire un'altra struttura.

Esempio 52 *Il tipo struttura rettangolo può essere definito a partire dal tipo punto:*

```
struct rectangle {
    struct point pt1;
    struct point pt2;
  };
```

6.2 Operazioni su strutture

- Una struttura può essere copiata, assegnata come un unico oggetto, indirizzata tramite l'operatore `&`, oppure manipolata tramite l'accesso ai suoi membri.
- Le strutture non possono essere confrontate.
- Non esistono conflitti tra nomi di variabili e nomi di campi di strutture coincidenti.
- È possibile definire puntatori a strutture nel modo usuale. In riferimento agli esempi precedenti, possiamo avere:

  `struct point *pt`

  `(*pt).x`

- Si noti che la precedenza dell'operatore `.` è superiore a quella di `*`.
- I puntatori alle strutture sono usati talmente spesso che sono state introdotte delle notazioni alternative.

  `(*pt).x` può essere scritto come `pt -> x`.

- Sia `.` che `->` sono associativi da sinistra a destra.
- La precedenza di `.` e `->` è massima e quindi analoga alla precedenza di `()` e di `[]`.

Esempio 53 *Si considerino i tipi struttura punto e rettangolo precedentemente definiti:*


```
struct point {
    int x;
    int y;
    };

struct rectangle {
    struct point pt1;
    struct point pt2;
    };
```

Si consideri adesso la seguente funzione che restituisce un punto, date le sue componenti:

```
struct point makepoint(int x, int y) {
   struct point temp;
   temp.x = x;
   temp.y = y;
   return temp;
}
```

La funzione precedente può ad esempio essere utilizzata per generare un rettangolo come segue:

```
struct rect screen;
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX,YMAX);
```

6.3 Vettori di strutture

È possibile dichiarare un array le cui componenti sono strutture. Si supponga ad esempio di volere creare un array di punti. La dichiarazione in questo caso è la seguente:

```
struct point {
   int x;
   int y;
} vet[MAXP];
```

Nel caso in cui i punti debbano anche essere inizializzati, la dichiarazione sarà la seguente:

```
struct point {
   int x;
   int y;
} vet[]= {
    {1,2},
    {5,8},
    {45,8},
    {5,87}
   };
```

6.4 Strutture ricorsive

È possibile definire una struttura in termini di se stessa. Si consideri a questo proposito il tipo di dato albero. La struttura C che permette di rappresentare un albero è la seguente:

```
struct tnode {
   char *info;
   struct tnode *left;
   struct tnode *right;
};
```

Si noti che `tnode` contiene un campo dichiarato come puntatore a `tnode` e per questo motivo la definizione è valida. Non sarebbe valida se contenesse un campo di tipo `struct tnode`.

6.5 Typedef

- È possibile creare nuovi nomi per tipi di dato con la seguente sintassi:

  `typedef descrizione-tipo nuovo-nome;`

  Esempio 54 *La seguente dichiarazione introduce il nuovo tipo* `Lenght`*, equivalente a* `int`*:*
  `typedef int Length;`

- Se si vuole utilizzare la typedef con le strutture bisogna porre attenzione. Si consideri ad esempio la seguente dichiarazione:

  `typedef struct tpoint{int x; int y} Point;`

  Questa dichiarazione diche che `Point` è un nuovo identificatore per il tipo `struct tpoint`. Si noti che, se l'identificatore `tpoint` non è rilevante, può anche essere omesso, scrivendo:

  `typedef struct {int x; int y} Point;`

- `typedef` non crea un nuovo tipo ma aggiunge un nuovo nome ad un tipo già esistente.
- L'uso di `typedef` permette spesso di aumentare la leggibilità dei programmi.

6.6 Union

- Una *union* è una variabile che può contenere in istanti diversi oggetti di tipo e dimensioni different. In pratica, le union permettono di manipolare dversi tipi di dati all'interno di un'unica area di memoria. Ad esempio, la seguente dichiarazione introduce il tipo union `u_tag`, che può essere, in relazione all'uso, un intero, un float o un puntatore a carattere.

  ```
  union u_tag{
      int ival;
      float fval;
      char *sval;
   } u;
  ```

- La variabile `u` è sufficientemente grande da contenere il più ampio dei tre tipi.
- Ognuno di questi tre tipi può essere assegnato alla variabile `u` ed utilizzato nelle espressioni, purchè l'utilizzo sia consistente.
- È responsabilità del programmatore tenere traccia dell'ultimo tipo memorizzato. Se ciò che è memorizzato è diverso da ciò che viene estratto, il risultato dipende dall'implementazione.
- L'accesso avviene secondo il seguente costrutto:

  `nome-union.membro`

  `puntatore-union -> membro`

- Una `union` può venire inizializzata fornendo soltanto il valore del tipo del suo primo membro.

7 Input e output

Nel seguito verranno analizzate in dettaglio le principali funzioni di input/output da/su video/tastiera e da/su file.

7.1 Input e output di base

Ogni file sorgente che utilizza una qualsiasi funzione di input-output della libreria standard deve contenere la linea:

`#include <stdio.h>`

Il più semplice meccanismo di input consiste nel leggere un carattere per volta dallo standard input, normalmente la tastiera, utilizzando la funzione:

`int getchar(void);`

Ogni chiamata di `getchar` restituisce il successivo carattere in input, oppure il valore `EOF`, se incontra la fine del file.

Analogamente, per l'output esiste la funzione:

`int  putchar(int);`

`putchar(c)` invia il carattere `c` allo standard output, normalmente il video. `putchar` ritorna il carattere scritto, o `EOF` se si verifica un errore.

7.2 Output formattato

- `printf` ha il seguente formato:

  `int printf(char *format, arg1,arg2, ...)`

  `printf` converte, formatta e stampa sullo standard output i suoi argomenti sotto il controllo di `format`. Restituisce il numero dei caratteri stampati.

- La stringa di formato contiene due tipi di oggetti:
  - caratteri ordinari, che vengono semplicemente copiati sull'output;
  - specifiche di conversione, ognuna delle quali provoca la conversione e la stampa del successivo argomento di `printf`.

- Ogni specifica di conversione inizia con un `%` e termina con un carattere di conversione (`d` per gli interi, `f` per i float, -si veda il libro per informazioni più particolareggiate-). Tra `%` e il carattere di conversione possono comparire i seguenti simboli:
  - '`-`': allineamento a sinistra del testo stampato;
  - `numero`: ampiezza minima del campo, se necessario vengono lasciati degli spazi bianchi;
  - '`.`': separatore tra quanto detto sopra e quanto segue;
  - `numero`: indica la precisione e il significato preciso varia in relazione al tipo dell'argomento stampato:
    - stringhe: indica il massimo numero di caratteri stampabili;
    - float: numero cifre dopo la virgola;
    - interi: numero minimo di cifre stampate.

  Esempio 55 *Si considerino i seguenti formati e il corrispondente output prodotto (il carattere ":" serve solo a delimitare le stringhe):*

```
:%s:       :salve, mondo:
:%10s:     :salve, mondo:
:%.10s:    :salve, mon:
:%-10s:    :salve, mondo:
:%.15s:    :salve, mondo:
:%-15s:    :salve, mondo   :
:%15.10s:  :     salve, mon:
:%-15.10s: :salve, mon     :
```

- `printf` usa il suo primo argomento per decidere quanti e di che tipo sono gli argomenti successivi. Se gli argomenti non sono sufficienti o se sono di tipo scorretto, `printf` produce stampe diverse da quelle che ci si potrebbe aspettare.
- La funzione

  `int sprintf(char *string, char *format, arg1,arg2, ...)`

  effettua conversioni uguali a `printf` ma memorizza il suo output in una stringa, che rappresenta il suo primo argomento.

7.3 Input formattato

- La funzione `scanf` è l'analogo di `printf` per la lettura formattata dell'input.
- Tale funzione legge dallo standard input, interpreta i dati letti in base al formato stabilito e memorizza il risultato di queste operazioni negli argomenti successivi.
- Gli argomenti devono essere puntatori e indicano dove registrare l'input convertito.
- Il valore di ritorno di `scanf` è il numero di elementi in input letti e registrati correttamente.
- Al termine dei valori in input, `scanf` ritorna il valore `EOF`. Tale valore è diverso da 0.
- 0 viene restituito quando il carattere in input è in contrasto con la specifica di controllo presente nel formato.
- Anche per `scanf` esiste la versione `sscanf` che legge una particolare stringa, anzichè lo standard input.
- Ogni chiamata a `scanf` riprende dal carattere immediatamente successivo all'ultimo convertito.
- La stringa di formato, può contenere:
  - spazi e tabbing: vengono semplicemente ignorati;
  - caratteri normali: ci si aspetta che corrispondano al prossimo carattere non bianco dell'input;
  - specifiche di conversione: si aspettano un valore del tipo corrispondente come prossimo carattere in input.
- Per ulteriori dettagli, si veda il libro.

Esempio 56 *Si considerino le seguenti dichiarazioni e istruzioni:*


```
int day, year; char monthname [20];

scanf("%d %s %d", &day,monthname,&year);
printf("%d %s %d", day,monthname,year);
```

Si noti che nella `scanf`*, visto che* `monthname` *è un puntatore non specifichiamo* `&`*, mentre è necessario utilizzarlo per gli operandi interi perchè è necessario recuperare l'indirizzo.*

Se vogliamo semplificare l'utente nel digitare la data, possiamo imporre che digiti anche i caratteri separatori "/". In questo caso otteniamo la seguente istruzione:

`scanf("%d / %s / %d", &day,monthname,&year);`

7.4 Accesso a file

Un *file di testo* è una sequenza di linee terminate dal carattere `newline`, che risiede su memoria secondaria.

Affinchè sia possibile operare su un certo file, è necessario eseguire le seguenti operazioni:

- *apertura*: prima di usare un file è necessario dichiarare al sistema la nostra intenzione di usarlo e il modo con cui lo vogliamo fare;
- *utilizzo*: dopo avere aperto un file è possibile leggerlo e scriverlo in modo sequenziale;
- *chiusura*: dopo avere utilizzato un file, è necessario chiuderlo, cioè dichiarare al sistema che non dobbiamo più usarlo. Questa operazione è utile perchè in genere il sistema operativo pone un limite sul numero di file che possono essere aperti contemporaneamente in un certo istante.

Nel seguito, le operazioni precedenti verranno illustrate con maggior dettaglio.

7.4.1 Apertura di un file

Prima di potere essere letto o scritto, un file deve essere aperto con la funzione di libreria `fopen`.

`fopen` legge un nome esterno, esegue alcune operazioni di carattere gestionale ed alcune trattative con il sistema operativo, e restituisce un puntatore che deve essere utilizzato nelle successive letture e scritture del file. Il puntatore si chiama *file pointer* e punta ad una struttura che contiene informazioni sul file (es. se il file è stato aperto in lettura o scrittura). La struttura a cui punta il file pointer è `FILE`, dichiarata nell'header `<stdio.h>`.

La dichiarazione necessaria per usare un file pointer è la seguente:

```
FILE *fp;
```

Il tipo `FILE` è definito tramite una `typedef`. I prototipo della funzione `fopen` è il seguente:

`FILE *fopen(char *name, char *mode);`

Nella chiamata a `fopen`, il secondo parametro è una stringa che indica in che modo si intende utilizzare il file. I modi possibili sono: lettura (`"r"`), scrittura (`"w"`), aggiunta (`"a"`).

Se il file non esiste e viene aperto in modo `a` o `w`, viene creato. Se si verifica un errore nell'apertura, ad esempio non esistono i permessi per aprirlo o si vuole leggere un file inesistente, `fopen` restituisce NULL.

7.4.2 Lettura/scrittura di un file

Una volta aperto, il file può essere letto e/o scritto. Le operazioni più semplici con le quali unfile può essere letto/scritto sono le seguenti:

```
int getc(FILE *fp);
int putc(int c,FILE *fp);
```

`putc` ritorna il carattere scritto oppure `EOF` se si verifica un errore. `getc` ritorna il prossimo carattere dal file `fp`, oppure `EOF` se si verifica un errore.

Quando un programma C entra in esecuzione, l'ambiente di sistema operativo si incarica di aprire tre file e di fornire i rispettivi puntatori. I file sono: standard input, standard output e standard error. I puntatori corrispondenti sono chiamati `stdin`, `stdout`, e `stderr`. In genere `stdin` è associato alla tastiera, mentre `stdout` e `stderr` sono associati al video. `stdin` e `stdout` possono essere rediretti su altri file. Si veda il libro per ulteriori informazioni su questo argomento.

Per l'input e l'output formattati, si possono usare le funzioni:

```
int fscanf(FILE *fp, char *format, ...);
int fprintf(FILE *fp,char *format, ...);
```

Queste funzioni agiscono come `printf` e `scanf`, ad eccezione del fatto che leggono (scrivono) dal (sul) file, il cui file pointer è specificato nella chiamata (`fp`).

Altre funzioni utili sono le seguenti:

```
int ferror(FILE *fp); int feof(FILE *fp):
```

Entrambe ritornano un valore non nullo se si è verificato un errore o, nel secondo caso, se si è raggiunta la fine del file considerato.

Nei casi in cui sia necessario effettuare molte operazioni sul contenuto di un file, può essere conveniente copiarne il contenuto in memoria. Questo è possibile ad esempio utilizzando un vettore di strutture.

7.4.3 Chiusura di un file

Quando le operazioni su un file sono terminate, il file deve essere chiuso, per rompere il legame tra file pointer e file. Questa operazione è fondamentale nel caso in cui il sistema consenta di aprire solo un numero limitato di files.

`int fclose(FILE *fp)`

Se un programma termina correttamente, `fclose` viene chiamata direttamente su tutti i file aperti.

7.5 Input e output di linee

La libreria standard fornisce alcune funzioni per la lettura e la scrittura di linee. La funzione per leggere una linea di un file è definita come segue:

```
char *fgets(char *line, int maxline, FILE *fp);
```

`fgets` ritona una linea di input (compreso il `newline`) dal file `fp` e la registra nel vettore `line`. Essa legge al più `maxline-1` caratteri. La linea risultante termina con `´\0´`. Al termine del file oppure alla rilevazione di un errore, la funzione ritorna `NULL`.

La funzione per scrivere una linea in un file è definita come segue:

```
char *fputs(char *line,  FILE *fp);
```

Restituisce `EOF` se si verifica un errore, e zero altrimenti.

7.6 Argomenti alle linee di comando

È possibile specificare con quali argomenti eseguire un certo programma, quindi quali argomenti passare alla funzione `main`. Tali argomenti possono essere specificati dopo il nome del file eseguibile durante l'esecuzione dello stesso dalla shell dei comandi (per eseguirlo, basta digitare il nome del file eseguibile seguito dagli eventuali argomenti).

In generale, il `main` viene sempre chiamato con due argomenti:

- `argc`: numero degli argomenti con i quali il programma è invocato; `argc` è sempre almeno 1 in quanto il primo argomento è sempre il nome del programma in esecuzione.
- `argv`: puntatore ad un vettore di stringhe di caratteri che contengono gli argomenti, uno per stringa. Tale vettore conterrà almeno una stringa, corrispondente al nome del programma che deve essere eseguito.

Il `main` dovrà essere definito nel modo seguente:
`main(int argc, char *argv[])`

Esempio 57 *Si consideri la seguente esecuzione del file eseguibile di nome* `nome-file`*:*
`nome-file arg1  arg2`
Il `main` *relativo al programma* `nome-file` *viene invocato con* `argc = 3` *e* `argv[] = {"nome-file", "arg1", "arg2"}`.

Esempio 58 *Si consideri adesso questo esempio più complesso in cui si vuole realizzate il comando* `cat`*, che prende i nomi di alcuni file e li stampa a video, uno di seguito all'altro. Quindi, se* `x.c` *e* `y.c` *sono due file, il comando* `cat x.c y.c` *deve stampare a video il contenuto del file* `x.c` *seguito dal contenuto del file* `y.c`.

Il programma per implementare questo comando è il seguente.

```
#include <stdio.h> main (int argc,char *argv[]) {
  FILE *fp;
  void filecopy(FILE *,FILE *);
  if (argc ==1) /* non sono stati forniti nomi di file */
       filecopy(stdin,stdout); /* copia l´input da tastiera in output su video
                                 */
  else
     while (--argc>0)
        if ((fp=fopen(*++argv,"r")==NULL)
            {
                printf("Problemi in apertura del file %s \n",*argv);
                return (1);
            }
        else
            {
               filecopy(fp,stdout);
               fclose(fp);
            }
}


void filecopy(FILE* ifp,FILE * ofp) {
  int c;
```

```
  while ((c = getc(ifp)) != EOF)
    putc(c,ofp);
}
```

8 La compilazione separata

Nel seguito introdurremo i principi alla base della compilazione separata. Per prima cosa, chiariremo le differenze tra dichiarazione e definizione e le diverse classi di memoria utilizzabili in un programma C. Quindi introdurremo il ruolo del preprocessore ed infine la compilazione separata.

8.1 Le classi di memoria

Per comprendere come viene generato il file eseguibile a partire da più file sorgente, è opportuno puntualizzare quando e come viene allocato spazio di memoria per una variabile o viene generato del codice associato ad un identificatore di funzione.

La prima distinzione da fare è tra *definizione di variabile o funzione* e *dichiarazione di variabile o funzione*:

- *Definizione di variabile*: operazione che provoca allocazione di memoria per le variabili in oggetto. Può essere seguita da una espressione di inizializzazione. Può essere effettuata al di fuori di ogni blocco (*variabile esterna*) oppure all'interno di un blocco (*variabile locale*).
- *Definizione di funzione*: operazione che provoca generazione di codice associato all'identificatore di funzione. È sempre seguita dal blocco che rappresenta il corpo della funzione e non può comparire all'interno di un blocco (la definizione di una funzione è sempre esterna).
- *Dichiarazione di variabile*: non provoca allocazione di memoria ma specifica solo il tipo della variabile. Deve essere preceduta dalla parola chiave `extern` (si veda Sezione 4) e può comparire sia a livello esterno che all'interno di un blocco. Non può essere seguita da un valore di inizializzazione.
- *Dichiarazione di funzione*: è il prototipo, con il quale non viene specificato il corpo della funzione (quindi non provoca generazione di codice) ma se ne indica solo la segnatura. Può essere preceduta da `extern` o `static` e può comparire sia a livello esterno che in blocco.

Dalla discussione precedente segue che variabili e funzioni possono essere classificate rispetto al luogo in cui compare la loro definizione:

- *Variabili*: sono dette *globali* o *esterne* se vengono definite all'esterno di ogni blocco. Sono invece dette *locali* o *automatiche* se sono definite all'interno di un blocco.
- *Funzione*: sono sempre globali. Non è infatti possibile definire funzioni all'interno del blocco di un'altra funzione.

Mettendo insieme le considerazioni precedenti con la discussione presentata in Sezione 4, possiamo sintetizzare le proprietà di variabili e funzioni con la Tabella 8.1. L'obiettivo della tabella è quello di identificare quali specifiche aggiuntive (extern, static, si veda Sezione 4) possono essere associate a dichiarazioni/definizioni di variabili/funzioni. La tabella riporta in orizzontale i possibili casi di definizione/dichiarazione mentre riporta in verticale specifiche aggiuntive per tali dichiarazioni/definizioni. In ogni casella sono quindi riportate le proprietà fondamentali della classe di memoria così identificata, se esiste, altrimenti i motivi per i quali non esiste.

8.2 Il preprocessore C

Il C fornisce alcune particolari funzionalità per mezzo di un processore che, concettuamente, costituisce la prima fase della compilazione. Il preprocessore supporta due funzionalità principali: inclusione di file e sostituzione delle macro.

Inclusione file

- Per ogni linea di codice

  `#include <nome-file>`

  `#include "nome-file"`

  `nome-file` viene sostituito dal contenuto del file corrispondente. Se `nome-file` è racchiuso tra apici, la ricerca del file da includere inizia, normalmente, dalla directory nella quale è stato trovato il file sorgente. In caso contrario, la ricerca prosegue secondo regole dipendenti dall'implementazione.
- Il file incluso può a sua volta contenere nuove istruzioni di `include`.

Sostituzione delle macro

- Per ogni linea di codice

  `#define nome testo`

  tutte le occorrenze di `nome` nel testo che segue verranno sostituite con `testo`. `name` è un identificatore `testo` è un testo arbitrario che termina con la fine della linea. Può occupare più linee, purchè sia terminato con \.
- Lo scope di una `define` è la parte di file nella quale è contenuta, dalla sua definizione fino alla fine del file.
- Una definizione può anche usare definizioni precedenti.
- Le sostituzioni non hanno luogo in stringhe racchiuse tra apici.

Esempio 59 *Si considerino le seguenti definizioni:*

```
#define YES 1
...
enum day{YESTERDAY,TODAY,TOMORROW};
...
printf("YES");
```

Nel codice precedente, il nome `YES` *è contenuto sia nella* `printf` *che nella definizione del tipo enumerazione* `day`*. Tuttavia, durante il preprocessing, il rimpiazzamento di* `YES` *con 1 avviene solo nell'argomento della* `printf`*, in quanto* `YESTERDAY` *è una stringa.*

Esempio 60 *Una definizione non necessariamente associa un valore ad un nome. Il testo assegnato al nome può anche essere più complesso come nel caso seguente:*

```
#define FOREVER for(;;)
```

- È anche possibile definire macro con argomenti. Ad esempio:

  `#define max(A,B)   ((A) > (B)  ? (A) : (B))`

  In questo caso, bisogna porre attenzione all'uso delle parentesi e delle precedenze.

  Se nel codice si scrive `x = max(p+q,r+s)`, dopo il rimpiazzamento l'istruzione precedente diventa `x=((p+q)>(r+s)?(p+q):(r+s))`. Si noti che in questo ultimo caso le espressioni che rappresentano gli argomenti della definizione vengono valutate due volte. È quindi necessario porre molta attenzione alle macro con parametri. Ad esempio, se scriviamo `max(i++,j++)`, a seguito del rimpiazzamento, `i` e `j` vengono incrementati due volte.

- I parametri formali di una macro vengono espansi in una stringa tra `" "` se sono preceduti da `#`

  Esempio 61 *Si consideri la seguente definizione:*

  `#define dprint(expr) printf(#expr "=%g\n", expr)`

  Si consideri adesso la seguente istruzione:

  `dprint(x/y);`

  Questa istruzione viene espansa nel modo seguente:

  `printf("x/y" "=%g\n", x/y);`

- *Tipici errori con le define.*

  – Alla fine di una definizione, il punto è virgola deve essere omesso, quindi `define MAX 1000;` è una definizione errata. Infatti, in presenza della dichiarazione `int a[MAX-1];`, dopo il rimpiazzamento questa dichiarazione diventerebbe `int a[1000;-1];`.

  – È necessario porre molta attenzione alle macro con parametri. Ad esempio, riferendoci all'esempio proposto per le macro con parametri, se scriviamo `max(i++,j++)`, a seguito del rimpiazzamento, `i` e `j` vengono incrementati due volte.

  – Un altro problema relativo all'uso delle macro con parametri è dato dall'uso delle parentesi. Se scriviamo:

    `#define square(x) x*x`

    e poi

    `square(z+1)`

    dopo il rimpiazzamento questa istruzione diventa

    `z+1 * z+1`

    che non calcola il quadrato di `z+1`. In questi casi devono essere usate le parentesi:

    `#define square(x) ((x)*(x))`

8.3 La compilazione separata

Il C permette che un programma sia contenuto in più file diversi, con estensione .c o .h. Ciò significa che verrà generato un unico programma eseguibile (estensione .exe) a partire dall'insieme di file considerato. Ad esempio, il Visual C++ genera un unico file eseguibile a partire da tutti i file contenuti in un progetto.

Si consideri ad esempio un programma C contenuto nei seguenti file (si consideri cioè il seguente progetto): A.c, B.h, C.c, E.h, D.c. Chiaramente, uno solo dei file dovrà contenere

la funzione `main`, in quanto l'esecuzione di questa funzione rappresenta il punto di partenza dell'eseguibile. Se esistono più file contenenti una funzione main, l'eseguibile non può essere generato.

La generazione dell'eseguibile avviene attraverso i seguenti passi:

1. *Compilazione*: i file .c vengono compilati. Ciò significa che inizialmente vengono sottoposti all'operazione di preprocessing che include i file (quindi, i file per i quali esiste una include vengono inclusi nel file corrispondente) e rimpiazza le espressioni corrispondenti alle costanti dichiarate con `#define`.

   Il risultato di questa operazione è un insieme di file con estensione .o (oppure .obj, dipende dal compilatore). Questi file sono file binari e rappresentano la traduzione dei file di partenza in linguaggio macchina. Quindi, nel nostro esempio verranno generati i file A.o, C.o e D.o.

2. *Link*: dopo la compilazione deve essere risolto un ulteriore problema: i file potrebbero riferirsi a variabili o funzioni contenute in altri file. Quindi i file oggetto generati dalla compilazione devono essere "messi insiemi", risolvendo i riferimenti e generando un unico file eseguibile. Il link viene generato utilizzando le regole presentate in Tabella 8.1 Il risultato di questa fase è un unico file eseguibile che tipicamente ha estensione .exe.

8.4 File Header

Si riconoscono dall'estensione `.h` e contengono informazioni globali utili a più parti di un programma. Un file header (ad esempio `stack.h`) può essere incluso in un file sorgente nel modo seguente:

`#include <stack.h>`

oppure

`#include "stack.h"`

Di norma un file header non contiene codice ma:

- definizioni di costanti (`#define`);
- definizioni di tipo (`typedef, enum`);
- dichiarazioni di funzione (prototipi);
- dichiarazioni di variabili (`extern`).

Non contiene invece definizioni di variabili e funzioni.

L'uso dei file header garantisce che tutti i file sorgente condividano le stesse definizione e dichiarazioni di variabile evitando così alcuni tipici errori difficilissimi da individuare.

In un file di libreria, l'header corrisponde alla sua interfaccia. Ad esempio, nelle librerie C esiste il file `stdio.o` che fornisce varie funzioni e costanti necessarie ad effettuare le operazioni di input/output. Questo file è un file binario, quindi se provate ad aprirlo il contenuto non sarà leggibile. A fronte di questo file, esiste il file `stdio.h`, il quale contiene invece un insieme di definizioni di tipi, di costanti e dichiarazioni di variabili e funzioni esterne. Quindi, il file `stdio.h` specifica le proprietà degli oggetti contenuti nel file `stdio.o`.

Quando all'interno di un file noi scriviamo:

`#include <stdio.h>`

durante il preprocessing questa linea di codice viene rimpiazzata dal contenuto del file `stdio.h`. Le dichiarazioni contenute in tale file vengono usate durante la compilazione per conoscere le proprietà di variabili e funzioni, come ad esempio `printf`, `scanf` o `getchar`.

Si noti che, se non includiamo `stdio.h` e chiamiamo ad esempio la funzione `getchar` in modo non corretto, il compilatore non si accorge dell'errore, in quanto una funzione, in assenza di prototipo, viene dichiarata implicitamente dalla sua prima occorrenza. A tale funzione, viene assegnato un tipo di ritorno intero. L'errore sarebbe tuttavia rilevato in fase di linking. Se invece viene utilizzata una variabile che non è stata ancora dichiarata, questo comporta un errore in compilazione, in quanto in questo caso non viene effettuata alcuna dichiarazione implicita.

9 Regole pratiche per scrivere codice leggibile

Il C, benchè sia un linguaggio ad alto livello, come abbiamo visto mette a disposione alcuni concetti di basso livello. Si pensi ad esempio a tutte le funzionalità messe a disposizione per l'accesso alla memoria tramite puntatori. Ciò spesso porta a scrivere codice poco leggibile. Quando si scrive un programma (lo vedrete meglio nei corsi nei prossimi anni) è molto importante che il programma funzioni correttamente ma è altrettanto importante che sia semplice da leggere e quindi da capire. Ciò aiuta a comprendere il codice scritto da altri, il proprio codice letto a distanza di tempo ed aiuta a trovare i possibili errori, in caso di esecuzioni non rispondenti ai requisiti per i quali il programma è stato scritto. A questo proposito, vale la pena di elencare alcune semplici regole che conviene seguire per generare codice leggibile e per evitare alcuni tipici errori:

- Cercate di usare lettere maiuscole per le costanti e minuscole per gli altri identificatori.
- Cercate di non usare identificatori come `pippo, pluto, paperino`. Questi identificatori possono andare bene quando si fanno esempi generali. Non vanno bene quando si scrive del codice per risolvere un certo problema.
- Evitare di usare il carattere `_` come primo carattere di un identificatore. Usatelo invece per separare parole distinte nel contesto dello stesso identificatore.
- Cercate di usare poche variabili esterne.
- In caso di dubbio sulle precedenze, usate le parentesi tonde.
- Usate i prototipi delle funzioni. Anche se non servono rendono più leggibile il programma.
- Cercate di non abusare di espressioni troppo compatte. Il C permette di scrivere istruzioni molto complicate, ma non sempre facilmente leggibili.
- Occhio alla `scanf`: gli argomenti devono sempre essere puntatori.
- L'errore più comune è quello relativo all'accesso ad una locazione su cui non si ha diritto. Questo accade quando appunto non viene passato il giusto indirizzo alla `scanf` e in moltissimi altri casi. Se durante l'esecuzione si genera un errore di questo tipo, controllate quindi tutte le `scanf` nonchè tutti i puntatori.
- Se avete delle stringhe, vi conviene sempre dichiararle come vettore. In caso contrario, dovrete ricorrere ad una allocazione dinamica del puntatore dichiarato prima di assegnargli una stringa non costante (ad esempio letta da input).
- Ricordatevi di liberare la memoria allocata quando non serve più.
- Ricordatevi che i file vanno aperti prima dell'uso e poi chiusi.
- Se nel contesto dello stesso programma dovete effettuare molte operazioni sul contenuto di un file, vi conviene portarne il contenuto in memoria centrale, salvando i dati in un array del tipo opportuno.
- COMMENTATE I VOSTRI PROGRAMMI!

Riferimenti bibliografici

[1] B.W. Kernighan e D.M. Ritchie. Linguaggio C. Gruppo Editoriale Jackson.

Tipo di classe	**Nessuna specifica**	**Static**	**Extern**
Def. var. globale	1) viene allocata una sola volta all'atto della compilazione 2) visibile nel file in cui è definita a partire dal punto in cui compare la definizione e negli altri file eventualmente esistenti nei quali è stata dichiarata come `extern` 3) può essere inizializzata con una espressione costante	1) viene allocata una sola volta all'atto della compilazione 2) visibile solo nel file in cui è stata definita a partire dal punto in cui compare la definizione 3) è possibile effettuare un'inizializzazione con un'espressione costante	è una definizione, non posso usare `extern`
Def. var. locale	1) variabile automatica: viene allocata ad ogni esecuzione del blocco e deallocata al termine dell'esecuzione del blocco 2) visibile solo nel blocco in cui è stata definita 3) può essere inizializzata con una espressione anche non costante	1) allocazione unica 2) mantiene il valore tra chiamate successive alla stessa funzione (non viene deallocata al termine di una chiamata) 3) inizializzazione costante	non è possibile usare `extern` con una definizione
Dich. variabile	non possibile	non possibile	1) specifica solo le proprietà 2) link alla variabile globale definita nel file corrente o in un altro file
Def. funzione	1) viene generato il codice associato all'identificatore della funzione 2)visibile nel file in cui è definita a partire dal punto in cui compare la definizione e negli altri file eventualmente esistenti	1) viene generato il codice associato all'identificatore della funzione 2)visibile solo nel file in cui è stata definita a partire dal punto in cui compare la definizione	è una definizione, non posso usare `extern`
Dich. funzione	prototipo, viene assunto `extern`	1) prototipo 2) link alla funzione con lo stesso nome definita all'interno del file	link alla funzione definita nel file o link ad un altro file se definita in un altro file rispetto a quello in cui è contenuta la dichiarazione

Tabella 1: Sintesi delle classi di memoria utilizzate dal linguaggio C