

Linguaggio C (parte terza)

Le funzioni

Le funzioni sono uno degli strumenti più potenti del C; esse infatti permettono un notevole risparmio e riuso del codice, con ovvii vantaggi del programmatore. Le funzioni esistono più o meno in tutti i linguaggi e vengono chiamate anche **procedure** o **subroutine**. Alcuni linguaggi fanno distinzione tra funzioni che ritornano un valore e quelle che, invece, non ritornano valori; il C assume che ogni funzione ritorni un valore, questo accade utilizzando l'istruzione **return** seguita, eventualmente, da un valore.

Prendiamo ad esempio, la definizione di una funzione che ha come parametri un valore di tipo `double` ed un valore di tipo `int` e ritorna, come risultato un valore di tipo `double`. In particolare, la funzione di esempio esegue l'elevamento a potenza di una determinata base.

```
double elevamento_potenza(double base, int potenza)
{
    double valore_ritorno = 1.0;
    int i;

    for(i=0; i<potenza; i++)
    {
        valore_ritorno *= base;
    }
    return(valore_ritorno);
}
```

Analizzando questa funzione passo passo:

```
double elevamento_potenza(double base, int potenza)
```

Questa è la definizione della funzione, dove vengono dichiarati: il tipo del valore di ritorno (`double`), il nome della funzione (`elevamento_potenza`) e la lista degli argomenti. Per ogni argomento della funzione è necessario dichiarare il tipo (`double` e `int` nell'esempio) ed il nome della variabile (nell'esempio `base` e `potenza`) corrispondente.

```
return(valore_ritorno);
```

Questa è l'istruzione di ritorno della funzione. Quando si raggiunge un'istruzione "`return`", il controllo del programma ritorna a chi ha chiamato la funzione. Il valore ritornato è quello compreso tra le parentesi di `return`.

La parte di codice sottostante riporta l'utilizzo della funzione `elevamento_potenza` appena descritta.

```
double val = 100.0;
int pot = 3;
double risultato = 0.0;

risultato = elevamento_potenza(val, pot);
```

In questo esempio, la funzione calcola il `val` elevato alla potenza `pot` ed il risultato ritornato viene assegnato alla variabile `risultato`.

Esistono anche funzioni che non ritornano alcun valore, in questo caso si parla di funzioni che ritornano `void`. Una funzione `void` non deve avere necessariamente un'istruzione "return" anche se è buona regola terminare la funzione con un'istruzione di return senza alcun parametro.

Va fatta una piccola nota riguardante la **prototipazione** delle funzioni, ovvero la creazione di prototipi. Un prototipo di una funzione non è altro che la sua *dichiarazione*, senza specificare il corpo della funzione stessa (**implementazione**); si scrive, quindi, solo la dichiarazione iniziale comprendente il nome ed il tipo restituito dalla funzione, e gli argomenti passati come parametro; questo avviene perché ogni funzione è utilizzabile solamente quanto è stata dichiarata, quindi se in un pezzo di codice, prima di dichiarare la funzione, voglio usare tale funzione, non posso farlo, perché "fisicamente" non è ancora stata creata. Creando dei prototipi si ovvia a questo problema e si hanno dei vantaggi anche in termini di funzionalità ed organizzazione del codice stesso. Ecco due esempi, il primo errato, il secondo corretto:

```
/* questo non funziona */
#include <stdio.h>

int main(void)
{
    int var = 5;
    stampa_doppio(var);
    return(0);
}

void stampa_doppio(int variabile)
{
    printf("Il doppio di %d è %d ", variabile, 2*variabile);
}

/* ----- */

/* questo funziona */
#include <stdio.h>

void stampa_doppio(int variabile); /* dichiarazione del prototipo */

int main(void)
{
    int var = 5;
    stampa_doppio(var);
    return(0);
}

void stampa_doppio(int variabile) /* implementazione funzione */
{
    printf("Il doppio di %d è %d ", variabile, 2*variabile);
}
```

I Puntatori

I puntatori sono uno strumento molto potente nella programmazione in C. I puntatori permettono di lavorare "a basso livello" (cioè agendo su singole istruzioni del processore), mantenendo però una praticità unica nel suo genere.

Un puntatore non è altro che una variabile che contiene l'indirizzo di memoria di un'altra variabile. Quando dichiariamo una variabile, a questa verrà riservato un indirizzo di memoria, ad esempio la posizione 1000; un puntatore a questa variabile contiene, appunto, l'indirizzo di memoria di tale variabile (quindi il valore 1000). L'importanza risiede nel fatto che si possono manipolare sia il puntatore che la variabile puntata (cioè la variabile memorizzata a quell'indirizzo di memoria).

Per definire un puntatore è necessario seguire la seguente sintassi:

```
/* variabile normale */
int variabile;

/* puntatore */
int *puntatore;
```

L'asterisco * viene chiamato operatore di **indirizzamento** o **deferenza** e restituisce il contenuto dell'oggetto puntato dal puntatore; mentre l'operatore & (già visto nell'utilizzo della `scanf`) restituisce l'indirizzo di memoria della variabile e viene normalmente usato per assegnare ad un puntatore l'indirizzo di una variabile nel seguente modo:

```
/* assegno al puntatore l'indirizzo di variabile */
puntatore = &variabile;
```

Da notare che un puntatore deve essere dello stesso tipo della variabile puntata. Ad esempio un puntatore ad una variabile di tipo `double` dovrà essere dichiarato come:

```
double *puntatore;
```

Per fare un esempio pratico, assumiamo di avere una variabile, `alfa` ed un puntatore di nome `pointer`; assumiamo anche che `alfa` risieda alla locazione di memoria "100" e `pointer` alla locazione "1000" e vediamo, eseguendo il codice sotto proposto, il risultato ottenuto:

```
#include <stdio.h>
int main(void) {
    int alfa = 4; /* dichiarazione ed inizializzazione variabile */
    int *pointer; /* dichiarazione puntatore */

    pointer = &alfa; /* al puntatore viene assegnato l'indirizzo di
    .                memoria della variabile alfa */

    printf("L'indirizzo di memoria di alfa è: %d\n", pointer);
    /* in questo modo stampiamo l'indirizzo di memoria della
    variabile alfa che in questo caso vale 100. */

    printf("Il contenuto di alfa è: %d\n", *pointer);
    /* in questo modo stampiamo il contenuto della variabile
    alfa che in questo caso vale 4 */

    *pointer = 5;
    /* in questo modo assegnamo al contenuto della variabile alfa
    il valore 5 quindi modifichiamo la variabile alfa,
    è come se avessimo scritto alfa = 5; */
```

```

printf("il valore di alfa è %d\n", alfa);
/* adesso la printf stamperà: il valore di alfa è 5 */
return(0);
}

```

Esiste anche un aritmetica base dei puntatori ai quali è possibile aggiungere o togliere dei valori, che generalmente vengono rappresentati come blocchi di memoria; per intendersi, un puntatore esiste sempre in funzione del tipo di oggetto puntato, se creo un puntatore ad `int`, il blocco di memoria cui punta ha una dimensione di 4 byte, un puntatore a `char`, invece, punta ad un blocco di memoria di 1 byte.

Puntatori e Funzioni

I puntatori risultano molto utili anche usati con le funzioni. Generalmente vengono passati alle funzioni gli argomenti (le variabili) per valore e si ottiene un risultato ancora per valore (utilizzando `return`). Ma questo modo di passare gli argomenti, non modifica gli argomenti stessi e quindi potrebbe risultare limitativo quando, invece, vogliamo che vengano modificate le stesse variabili che passiamo alle funzioni.

Pensiamo ad un caso esemplare, uno su tutti, l'uso di una funzione chiamata `swap` (alfa, beta) che scambia il valore di alfa con beta e viceversa; in questo caso il valore restituito dalla funzione (con `return`) non va minimamente ad intaccare i valori delle variabili alfa e beta, cosa che vogliamo, invece, accada per poter effettivamente fare lo scambio.

In questo caso è necessario passare alla funzione, non i valori delle variabili, ma il loro indirizzo di memoria così da poter effettivamente scambiare i loro valori.

Facciamo un esempio pratico per chiarire:

```

#include <stdio.h>

void swap(int *apt, int *bpt); /* prototipo della funzione swap
                               notate che gli argomenti sono
                               del tipo puntatore a variabili
                               intere */

int main(void)
{
    int alfa = 5; /* dichiarazione ed inizializzazione variabili */
    int beta = 13;

    printf("alfa vale %d, beta vale %d\n", alfa, beta);

    swap(&alfa, &beta); /* uso la funzione sulle variabili
                        notare che alla funzione viene passato
                        l'indirizzo di memoria delle le varibili */

    printf("alfa vale %d, beta vale %d\n", alfa, beta);
    return(0);
}

void swap(int *apt, int *bpt) /* implementazione della funzione */
{
    int temp; /* variabile intera temporanea per memorizzare il
              valore di una variabile durante lo scambio */
    temp = *apt; /* a temp viene assegnato il contenuto della
                 variabile puntata da apt cioè alfa */
    *apt = *bpt; /* al contenuto della variabile puntata da apt viene
                 assegnato il contenuto della variabile puntata da
                 bpt cioè beta */
    *bpt = temp; /* al contenuto della variabile puntata da bpt viene
                 assegnato il valore memorizzato in temp */
}

```

Puntatori ed Array

Una digressione su array e puntatori potrebbe essere essa stesso l'argomento di un unico corso di linguaggi di programmazione. In questo ambito ci limitiamo ad accennare alla correlazione tra i due strumenti a disposizione nel linguaggio C.

L'aspetto che accomuna i puntatori e gli array è sostanzialmente il fatto che entrambi sono memorizzati in locazioni di memoria continue, ed è quindi possibile agire su un array (se lo si vede come una serie di blocchi ad indirizzi di memoria sequenziali) come se si stesse agendo su un puntatore (e viceversa); ad esempio se dichiariamo un array "alfa" ed un puntatore "pointer":

```
/* dichiaro un array ed una variabile intera */
int alfa[20];
int x;

/* dichiaro un puntatore a intero */
int *pointer;

/* assegno al puntatore l'indirizzo di memoria della prima cella
dell'array cioè alfa[0] */
pointer = &alfa[0];
```

Se volessi scorrere tutte le caselle dell'array adesso mi basterebbe incrementare il puntatore di un indice *i* cioè:

pointer + *i* "è equivalente a" *a*[*i*]. Facciamo un esempio con il codice:

```
for(i = 0; i < 10; i++)
{
    alfa[i] = 0;
}
```

equivale a:

```
for(i = 0; i < 10; i++)
{
    *(pointer + i) = 0;
}
```

Allocazione dinamica della Memoria

Il C è, per natura, un linguaggio molto flessibile, e la sua gestione della memoria contribuisce a renderlo ancora più flessibile. A differenza di altri linguaggi (come il C++ o il Java), il C permette di assegnare la giusta quantità di memoria (solo e solamente quella necessaria) alle variabili del programma. Fino a questo momento abbiamo visto come allocare la memoria destinata alle variabili semplicemente dichiarandole (`int variabile = 0;`). Questo modo di allocare la memoria viene detto **statico** in quanto l'allocazione avviene all'inizio del programma ed in particolare è compito del compilatore assegnare la giusta memoria alle variabili. In questo modo la quantità di memoria utilizzata dal programma è nota a priori e non può più cambiare durante l'esecuzione. Questo può essere un limite nel caso in cui solamente durante l'esecuzione del programma sapremo effettivamente quanta memoria ci servirà. Prendete ad esempio un visualizzatore di immagini. Questo, fin quando non ha "aperto" il file di immagine non sa quanta memoria dovrà destinare alla visualizzazione.

In linguaggio C per far fronte a questa necessità è possibile allocare della memoria in maniera **dinamica** cioè solo al momento opportuno.

Utilizzare queste caratteristiche del C permette di creare programmi altamente portabili, in quanto utilizzano di volta in volta la giusta quantità di memoria per ogni piattaforma.

Una piccola nota su come è organizzata la memoria di un processo (programma in esecuzione) è doverosa: la memoria è divisa sostanzialmente in due parti, una statica, che contiene tutto quello che sappiamo verrà allocato staticamente (come una variabile `int`) e che si chiama **Stack**, ed una dinamica, cioè in cui la dimensione di memoria per rappresentare qualche elemento del programma può cambiare durante l'esecuzione del programma, che si chiama **Heap**.

Le funzioni utilizzate per gestire la memoria dinamica sono principalmente `malloc()` e `calloc()`, adibite all'allocazione della memoria, `free()` che, come si intuisce, serve per liberare la memoria allocata, e `realloc()` la cui funzione è quella di permettere la modifica di uno spazio di memoria precedentemente allocato. Infine, un comando particolarmente utile risulta essere `sizeof`, che restituisce la dimensione del tipo di dato da allocare.

Le funzioni appena nominate vengono implementate nella libreria `malloc.h`, che deve essere inclusa ogni qualvolta le si intenda utilizzare.

La funzione `malloc()`

La funzione `malloc()` come già accennato permette di allocare della memoria durante l'esecuzione di un programma. La sintassi della funzione `malloc()` è la seguente:

```
*void malloc(int size)
```

In particolare tale funzione vuole come argomento un parametro di tipo `int` che rappresenta la dimensione in byte dello spazio di memoria contiguo da allocare e ritorna un puntatore di tipo `void` alla prima posizione della memoria appena allocata. Nel caso in cui l'allocazione non vada a buon fine (la memoria sulla macchina non è sufficiente ad allocare lo spazio richiesto) la funzione `malloc` ritorna un puntatore nullo (`NULL`) indicando un errore.

La funzione `sizeof()`

La funzione `sizeof()` permette di determinare la dimensione di un tipo di dato rappresentato sull'architettura corrente. Tale funzione vuole come argomento un tipo di dato e ritorna un valore intero che rappresenta il numero di byte con cui tale tipo di dato viene rappresentato sull'architettura corente. Quindi su un processore della famiglia x86 l'istruzione `sizeof(double)`, ad esempio, ritornerà il valore intero 8 che corrisponde ad una rappresentazione a 8 byte = 64 bit per le variabili di tipo `double`.

La funzione `free()`

La funzione `free()` permette di liberare una zona di memoria preventivamente allocata, ad esempio con una chiamata `malloc()`. La funzione `free()` vuole come argomento il puntatore alla prima locazione di memoria da liberare.

Esempio di programma che alloca memoria dinamicamente

Presentiamo un esempio per chiarire meglio l'uso di queste funzioni, in particolare allocheremo la memoria per un array con `malloc()`, lavoreremo sull'array stesso ed, infine, libereremo la memoria con `free()`.

```

#include <stdio.h>
#include <malloc.h> /* inclusione della libreria malloc.h */

int main(void)
{
    int numero;
    int *array; /* dichiaro un puntatore ad intero che mi
                rappresenterà un array */

    int i;
    int allocati;
    numero = 100; /* rappresenta la dimensione dell'array che
                  voglio allocare dinamicamente */

    array = (int *)malloc(sizeof(int) * numero);
    /* questa istruzione alloca memoria necessaria a contenere 100
       elementi di tipo intero */

    if(array == NULL) /* controlla se c'era memoria sufficiente per
                      allocare l'array */
    {
        printf("Memoria esaurita\n");
        exit(1);
    }

    for(i=0; i<numero; i++)
    {
        array[i] = 0; /* assegna a tutte le caselle dell'array il
                      valore 0 */
    }

    printf("\n\nNumero elementi %d\n", numero);
    printf("Dimensione elemento %d\n", sizeof(int));
    printf("Bytes allocati %d\n", sizeof(int) * numero);

    free(array); /* libero la memoria associata all'array che ora
                 non serve più */
    printf("\nMemoria Liberata\n");

    return 0;
}

```

In questo programma possiamo notare l'utilizzo della funzione `malloc()` che ritorna un puntatore a di tipo `void`, corrispondente al punto di inizio, in memoria, della porzione riservata della dimensione "intera" passata come argomento; se la memoria richiesta non è presente sulla macchina (memoria insufficiente) , ritorna un puntatore nullo.

Nel caso citato si può notare che è stato usata la funzione `sizeof` per specificare il numero esatto di byte, mentre è stata usata la **coercizione** (`int *`) per convertire il tipo di dato "puntatore a void" a "puntatore ad int", questo per garantire che i puntatori aritmetici vengano rappresentati correttamente. Esiste, inoltre, un legame molto forte tra puntatori ed array per trattare la memoria riservata come un array, ed è per questo che in realtà la variabile "array" non è stata definita inizialmente come array, ma come puntatore ad int.

Introduzione Input e Output su file

Abbiamo già accennato alle operazioni di input/output limitandoci a quelle che coinvolgevano la tastiera come dispositivo di input e lo schermo come dispositivo di output. Come abbiamo visto, per poter operare correttamente è necessario includere l'header file `<stdio.h>` che contiene tutte funzioni per le operazioni di l'input/output standard, comprese quelle che operano sui file.

In C le operazioni di I/O vengono semplificate attraverso l'uso degli **stream**, altro non sono che delle astrazioni rappresentative di un file o di un dispositivo fisico, che vengono manipolate attraverso l'uso di puntatori. L'enorme vantaggio di usare gli stream è quello di potersi riferire ad un identificatore senza preoccuparsi di come questo venga implementato; generalmente le operazioni che si compiono su uno stream sono tre: apertura, accesso (lettura o scrittura) e chiusura. L'altra importante caratteristica da ricordare è che lo stream è bufferizzato, ovvero viene riservato un buffer (memoria temporanea) per evitare ritardi o interruzioni nella fase di lettura e scrittura. Il contenuto del buffer non viene mandato al dispositivo (che si tratti di un dispositivo fisico o di un file non ha importanza) fino a quando non viene svuotato o chiuso.

La funzione `fopen()`

I file sono la parte più importante degli stream perché, come già detto, sono un elemento essenziale per permettere al programmatore di fare applicazioni interattive. Come menzionato prima, la prima cosa da fare è aprire uno stream su file. Per fare ciò si usa la funzione `fopen()`, la cui sintassi è la seguente:

```
FILE *fopen(char *nome, char *modo);
```

Com'è possibile notare, la funzione `fopen()` prevede due argomenti di tipo puntatore a carattere (cioè una stringa di testo). Il primo argomento identifica il nome del file che vogliamo aprire mentre il secondo argomento indica la modalità di apertura del file e può assumere i seguenti valori:

- "r" - lettura;
- "w" - scrittura;
- "a" - scrittura in fondo al file (append).

La funzione `fopen()` restituisce un puntatore al tipo di dato descritto dall'identificativo privato `FILE`. Il puntatore restituito servirà, dopo l'apertura, per poter effettuare le operazioni di lettura e scrittura sul file stesso. Nel caso in cui si verificano problemi nell'apertura del file viene restituito un puntatore nullo (`NULL`). Qui di seguito proponiamo un semplice programma per poter leggere un file, ad esempio, di nome `miofile.txt`.

```
#include <stdio.h>

int main(void)
{
    /* dichiarazione del puntatore al file */
    FILE *f_pointer;

    /* apre lo stream del file in sola lettura */
    f_pointer = fopen("miofile.txt", "r");

    /* controlla se il file è stato aperto correttamente */
    if (f_pointer == NULL)
    {
        printf("ERRORE: impossibile aprire il file\n");
        exit(1);
    }else
    {
        printf("Il file è stato aperto correttamente\n");
    }
    return(0);
}
```

Le funzioni fprintf e fscanf

Una volta che si è aperto un file con la funzione `fopen()`, possiamo usare due funzioni per accedervi, queste sono la `fprintf()` e la `fscanf()` che operano come la `printf()` e la `scanf()` con la sola differenza che agiscono su uno stream aperto da `fopen()`. La sintassi delle due funzioni è la seguente:

```
int fprintf(FILE *f_pointer, char *formato, argomenti ...);
int fscanf(FILE *f_pointer, char *formato, argomenti ...);
```

Come si può intuire la `fprintf()` scrive sul file mentre la `fscanf()` legge. L'unica differenza rispetto alla `printf()` e `scanf()` che abbiamo già visto sta nel fatto che come primo argomento prevedono gli venga passato un puntatore a file.

Le funzioni fflush() e fclose()

Infine gli stream, qualunque uso ne sia stato fatto, devono essere prima "puliti" e poi chiusi, utilizzando le funzione `fflush()` che svuota il buffer dello stream pulendolo e `fclose()` che chiude lo stream. La sintassi delle funzioni è la seguente:

```
fflush(FILE *f_pointer);
fclose(FILE *f_pointer);
```

Il seguente programma chiarirà meglio l'uso delle funzioni spiegate per operare su file. Nell'esempio acquisiremo un dato da tastiera e lo stamperemo su un file di testo.

```
#include <stdio.h>

int main(void)
{
    int dato;
    FILE *puntatore_file; /* dichiarazione del puntatore a file */

    /* apertura del file in modalità append */
    puntatore_file = fopen("miofile.txt", "a");

    /* controllo della validità del puntatore a file */
    if(puntatore_file == NULL)
    {
        printf("Impossibile aprire il file\n");
        exit(1);
    }

    /* acquisizione del dato da tastiera */
    printf("Digita il dato da scrivere su file\n");
    scanf("%d", &dato);

    /* scrittura del dato sul file */
    fprintf(puntatore_file, "Il dato e': %d\n", dato);

    /* svuotamento del buffer dello stream */
    fflush(puntatore_file);

    /* chiusura del file */
    fclose(puntatore_file);
    return(0);
}
```

Tipi di Dati avanzati

Una volta presa dimestichezza con le principali caratteristiche del C si possono utilizzare dei tipi di Dati avanzati. Qui di seguito presentiamo i tipi di dati avanzati del C, ovvero tipi non semplici nella loro rappresentazione.

Strutture

Le strutture del C sono simili ai record del Pascal e sostanzialmente permettono un'aggregazione di variabili, molto simile a quella degli array, ma a differenza di questi non ordinata e non omogenea (una struttura può contenere variabili di tipo diverso). Per denotare una struttura si usa la parola chiave `struct` seguita dal nome identificativo della struttura, che è opzionale. Nell'esempio sottostante si definisce una struttura "libro" e si crea un'istanza di essa chiamata "biblio":

```
/* dichiarazione della struct */
struct libro
{
    char titolo[100]; /* array di caratteri conterrà il titolo */
    char autore[50]; /* array di caratteri conterrà l'autore */
    int anno_publicazione; /* l'anno di pubblicazione */
    float prezzo; /* il prezzo del libro in euro */
};

/* dichiarazione dell'istanza chiamata biblio */
struct libro biblio;
```

La variabile "biblio" può essere dichiarata anche mettendo il nome stesso dopo la parentesi graffa:

```
/* dichiarazione della struct e dell'istanza biblio */
struct libro
{
    char titolo[100];
    char autore[50];
    int anno_publicazione;
    float prezzo;
} biblio;
```

è possibile pre-inizializzare i valori, della struttura alla dichiarazione, mettendo i valori (giusti nel tipo) compresi tra parentesi graffe:

```
struct libro biblio = {"Guida al C", "Fabrizio Ciacchi", 2003, 45.2};
```

Per accedere alle variabili interne della struttura si usa l'operatore "."; una volta che si ha accesso ad una variabile interna questa può essere trattata e/o manipolata come qualsiasi altra variabile:

```
/* assegna un valore al prezzo del libro */
biblio.prezzo = 67.32;

/* assegna ad una variabile int l'anno di pubblicazione del libro */
int anno = biblio.anno_publicazione;

/* stampa il titolo del libro */
printf ("%s \n", biblio.titolo);
```

Nuovi tipi di dato

Per definire nuovi tipi di dato viene utilizzata la parola chiave `typedef`. Con `typedef` e l'uso di `struct` è possibile creare tipi di dato molto complessi, come mostrato nell'esempio seguente:

```
/* definizione del tipo di dato t_libro */
typedef struct libro
{
    char titolo[100];
    char autore[50];
    int anno_pubblicazione;
    float prezzo;
} t_libro;
```

D'ora in poi il tipo di dato `t_libro` viene a tutti gli effetti riconosciuto come un nuovo tipo di dato. L'identificativo `t_libro` ora può essere utilizzato proprio come gli identificativi standard `int`, `double`, ecc.

```
t_libro guida; /* dichiarazione di una variabile di tipo t_libro */

/* inizializzazione della variabile */
guida = {"Guida al C", "Fabrizio Ciacchi", 2003, 45.2};
```

In questo modo, dopo avere definito un nuovo tipo di dato chiamato "`t_libro`", abbiamo dichiarato una variabile "`guida`" di questo tipo e l'abbiamo inizializzata.

Come per ogni altro tipo di dato, anche con "`t_libro`" si possono creare degli array:

```
t_libro raccolta[5000];
```

Per accedere, o per inizializzare i valori, è sufficiente utilizzare l'indice per identificare l'elemento dell'array ed il punto (.) per accedere alle variabili interne al tipo "`t_libro`";

```
/* assegna un valore al prezzo del 341° libro */
raccolta[340].prezzo = 67.32;

/* assegna ad una variabile int l'anno di pubblicazione del 659°
libro */
int anno = raccolta[658].anno_pubblicazione;

/* stampa il titolo del 23° libro */
printf ("%s \n", raccolta[22].titolo);
```

Tipo enumerazione

Il tipo enumerazione è abbastanza particolare, perché permette di associare a delle costanti letterali, un valore intero; in questo modo possiamo utilizzare tali nomi per identificare il loro valore; facciamo un esempio utilizzando i giorni della settimana:

```
#include <stdio.h>

int main(void) {
    /* definiamo una enumerazione giorni contenente i giorni della
       settimana ed una variabile settimana di tipo enum giorni */
    enum giorni { lun, mar, mer, gio, ven, sab, dom } settimana;

    /*inizilizziamo la variabile settiman a martedì */
    settimana = mar;

    /* una struttura switch ci dirà che giorno è in base al valore
       della variabile settimana */

    switch(settimana) {
        case dom:
            printf("E' domenica\n");
            break;
        case lun:
            printf("E' lunedì\n");
            break;
        case mar:
            printf("E' martedì\n");
            break;
        case mer:
            printf("E' mercoledì\n");
            break;
        case gio:
            printf("E' giovedì\n");
            break;
        case ven:
            printf("E' venerdì\n");
            break;
        case sab:
            printf("E' sabato\n");
            break;
        default:
            printf("Non e' un giorno della settiman\n");
            break;
    }
    return(0);
}
```

In questo caso abbiamo definito una nuova variabile di nome "settimana" di tipo enumerazione "giorni"; l'identificatore "lun" assume il valore 0, "mar" assume il valore 1, e così via; in poche parole si ha un indice iniziale "0" e gli altri assumono una numerazione progressiva. Questo ci può essere molto utile se dobbiamo scrivere un programma che operi sui giorni della settimana (come un calendario); se non esistesse il tipo enumerazione il programma non potrebbe assegnare alcun "valore" ad un determinato giorno e quindi sarebbe molto più difficile (e spendioso in termini di codice) lavorare in tal senso.

E' possibile, però, assegnare alle costanti anche valori iniziali diversi da 0, o valori non numerici, come spiegato nei due esempi:

```
/* valori non numerici */
enum seq_escape { suono = 'a', cancella = '\b', tab = '\t', invio =
'\r' };

/* indice iniziale diverso da 0 */
enum mesi { gen = 1, feb, mar, apr, mag, giu, lug, ago, set, ott,
nov, dic };
```

Il Pre-processore C e le Direttive di inclusione

Una direttiva, come abbiamo già visto per `#include`, inizia sempre con il carattere cancelletto `#`.

Il file sorgente contenente le direttive, viene tradotto dal **pre-processore** in quella che viene chiamata la **translation unit** (anch'essa formata solamente da codice sorgente), la quale viene poi compilata in codice binario, dando origine al corrispondente file oggetto; tutti i file oggetto, infine, vengono collegati dal linker, generando un unico programma eseguibile.

Direttive di Inclusione

Le direttive di inclusione sono quelle usate maggiormente, semplicemente perché vengono usate per inserire le librerie standard del linguaggio; la forma sintattica corretta della direttiva, che peraltro conosciamo benissimo, per includere i file è la seguente:

```
#include <file>
```

mettendo il nome del file da includere tra il simbolo di minore (<) e quello di maggiore (>); in questa forma il compilatore andrà a cercare i file presenti nelle directory di include prescelta; generalmente questa directory è predefinita, ed è quella che contiene i file delle librerie standard, come ad esempio `stdio.h`, `stdlib.h`, `math.h`, `string.h`, `time.h` e così via.

Se volessimo includere un file di libreria non standard generato da noi e che non si trovi nella directory predefinita ma nella attuale directory di lavoro dovremmo usare una forma alternativa, come segue:

```
#include "file"
```

Le Direttive di definizione

Le direttive di definizione sono sostanzialmente due, `#define` e `#undef`, che, come vedremo sotto, possono essere usate congiuntamente per definire o meno qualcosa in modo condizionale. La direttiva di `#define` ha la seguente sintassi:

```
#define identificatore espressione
```

Sostanzialmente il pre-processore sostituisce tutte le occorrenze di "identificatore" con l'espressione corrispondente (che può contenere anche spazi o virgolette); generalmente il nome di una costante definita con tale direttiva, oltre ad essere tutto maiuscolo, viene preceduto dal carattere underscore (`_`). Quando la direttiva serve per definire una macro ha una sintassi leggermente diversa:

```
#define identificatore(argomenti) espressione
```

in questo caso, invece, si suppone che l'espressione contenga gli argomenti come variabili sulle quali operare, il pre-processore non fa altro che sostituire il codice di espressione,

modificando gli argomenti definiti nella macro con quelli attuali, ad ogni occorrenza dell'identificatore.

Il vantaggio che si ha ad usare le direttive di definizione è quello di non appesantire il codice con chiamate o allocazioni di memoria, poiché si sostituisce il codice della macro con le occorrenze dell'identificatore nel codice sorgente e solo dopo lo si passa la compilatore.

Ovviamente `#undef` serve solo per annullare il compito di ciò che abbiamo definito con l'eventuale `#define`.

Per fare un esempio presentiamo un pezzo di codice che fa uso delle direttive di definizione:

```
#include <stdio.h>

#define NUMERO 5          /* definizione di una costante simbolica */
#define QUADRATO(a) (a)*(a) /* definizione di un MACRO */

int main(void)
{
    int x;

    printf("Numero : %d \n", NUMERO);
    x = QUADRATO(NUMERO);
    printf("Quadrato: %d \n", x);

    #undef NUMERO
    #define NUMERO 7

    printf("Numero : %d \n", NUMERO);
    x = QUADRATO(NUMERO);
    printf("Quadrato: %d \n", x);

    return 0;
}
```

Questo semplice programma stamperà a video:

```
Numero : 5
Quadrato : 25
Numero : 7
Quadrato : 49
```

Le Direttive condizionali

Le direttive condizionali permettono di definire una serie di istruzioni che verranno **compilate** solo in determinate condizioni, questo tipo di direttive viene usato spesso per compilare il medesimo sorgente su diversi sistemi operativi (cambiando dei parametri che in Linux funzionano ed in Windows no, e viceversa); le direttive condizionali sono sei, ma non sono difficili da ricordare:

- `#if` - Se il valore di ciò che sta a destra è zero (quindi FALSO) allora il pre-processore escluderà dalla compilazione tutto il codice del corpo dell'`#if`. Se il valore è diverso da zero (TRUE), allora si compilerà il codice sotto l'`#if` fino a che non si incontrerà un `#elif`, un `#else` o un `#endif`.
- `#ifdef` - Riferendosi all'identificatore passato come parametro, se è già stato definito (TRUE) verrà compilato il codice sottostante, altrimenti (FALSE) si salta al prossimo `#elif`, o `#else` o un `#endif`.
- `#ifndef` - Complementare al precedente; infatti in questo caso si compila il codice sottostante solo se l'identificatore passato non è stato definito.

- `#elif` - Corrisponde al costrutto del ELSE IF, e quindi verifica la propria condizione ed agisce di conseguenza; può essere usato per definire scelte multiple (tranne l'ultima).
- `#else` - Nell'ultima voce delle scelte condizionali bisogna mettere l'else che copre la casistica non ricoperta dall'`#if` e dagli `#elif` precedenti. Tutto il codice che sta dopo l'else verrà compilato fino a quando non si incontra un `#endif`.
- `#endif` - Chiude la direttiva `#if` . (obbligatoria).

Di seguito facciamo un semplice esempio su come possiamo usare queste direttive per selezionare quale codice compilare in diverse occasioni, cosa molto utile, lo ripetiamo, quando vogliamo compilare il medesimo programma su sistemi operativi diversi.

```
#include <stdio.h>
#include "miofile.h" /* direttiva di inclusione di un file
                    presente nella directory di lavoro */

#ifdef SISTEMA
#define SISTEMA 1
#endif

/* SISTEMA = 1 -> Sistema Linux */
/* SISTEMA = 0 -> Sistema Windows */

int main(void)
{

    #if SISTEMA==1
        printf("Sistema operativo: Linux\n");
    #elif SISTEMA==0
        printf("Sistema operativo: Windows\n");
    #else
        printf("Sistema operativo: sconosciuto\n");
    #endif

    return 0;
}
```

In questo semplice pezzo di codice abbiamo due scenari, o l'identificatore `SISTEMA` è già stato definito, ad esempio, in `miofile.h`, con un valore che può assumere 0 (Windows), 1 (Linux), 2 (sconosciuto), oppure viene definito all'interno del codice con valore di default pari ad 1. A questo punto gli strumenti fino ad ora spiegati servono per compilare una delle tre `printf()` nel codice finale. Il codice è volutamente un po' "forzato" per mostrare tutte le dichiarazioni condizionali citate.

Errori comuni e regole di stile in C

Errori comuni per chi è ai primo approcci con il linguaggio C possono essere così riassunti:

- **Assegnazione (=) al posto del confronto (==)**. Bisogna porre attenzione, quando, utilizzando una struttura condizionale come `if-else`, scriviamo l'operazione di confronto (`==`), poiché essa può, per un errore di battitura, diventare un assegnamento (`=`); se ciò dovesse accadere, ad esempio, cercando di confrontare se due numeri sono uguali, potremmo trovarci nella spiacevole situazione in cui, invece di controllare se `a == b`, che restituisce TRUE solamente quando le due variabili hanno lo stesso valore, poniamo `a = b`, che è TRUE praticamente sempre, il suo valore è FALSE solamente quando `b` vale 0.
- **Mancanza di () per una funzione** - Il programmatore inesperto tende a credere che una funzione alla quale non si passano parametri (`void`) non debba avere le parentesi tonde; ciò è errato, le parentesi tonde, anche senza parametri, devono essere sempre messe.
- **Indici di Array** - Quando si inizializzano o si usano gli array, bisogna porre attenzione agli indici utilizzati (e quindi alla quantità di elementi) poiché se si inizializza un array con `N` elementi, il suo indice deve avere un intervallo tra 0 (che è il primo elemento) ed `N-1` (che è l'`n`-esimo elemento).
- **Il C è Case Sensitive** - Il linguaggio C (come il C++ ed il Java) fa distinzione tra lettere maiuscole e minuscole, interpretandole come due caratteri diversi; bisogna quindi stare attenti ad utilizzare i nomi giusti per le variabili.
- **Il ";" chiude ogni istruzione** - E' un errore tanto comune che non può non essere citato, ogni istruzione deve essere chiusa con un punto e virgola; questa facile dimenticanza (Nda: che colpisce anche i più esperti :), segnalata dal compilatore, può far perdere del tempo prezioso ed appesantisce inutilmente il lavoro del programmatore.

Buone regole di programmazione

Utilizzando poche regole di buona programmazione possiamo scrivere del codice facilmente leggibile da altri programmatori e più facilmente debuggabile da noi stessi. Ecco un elenco di alcune di queste regole:

- **I nomi** delle variabili, strutture, costanti e funzioni devono essere significativi e normalmente scritti con caratteri minuscoli. Inoltre sarebbe buona norma, se il nome è composto da più parole, evidenziare le due parole separandole da un carattere di underscore "_". Le costanti, invece, è buona regola che siano scritte tutte in maiuscolo e separate, se formate da più di una parola, dal carattere underscore "_".
- **Uso, funzione e posizione dei commenti** - I commenti devono accompagnare quasi tutte le istruzioni, per spiegare il significato di ciò che stiamo facendo, inoltre devono essere sintetici e devono essere aggiornati appena modifichiamo un'istruzione. I commenti devono essere più estesi, invece, se devono spiegare un determinato algoritmo o se accompagnano una funzione.
- Ogni **sorgente** dovrebbe contenere, nell'ordine:
 - Commento iniziale (con nome del file, nome dell'autore, data, testo del problema ed eventuali algoritmi usati);
 - Istruzioni `#include` per tutti i file da includere (bisogna includere solo i file necessari);
 - Dichiarazioni di costanti, strutture e tipi enumerativi;
 - Definizione di variabili;
 - Prototipi delle funzioni;
 - Definizione delle funzioni (con la funzione **main** messa come prima funzione).
- **Indentazione:** tutte le istruzioni contenute in un blocco vanno indentate rispetto all'allineamento del blocco stesso (che sia la `{` di apertura o il `DO`), tranne per le dichiarazioni di variabili locali che vanno allineate sotto la `{` (o il `DO`). Per evitare di

avere sorgenti che eccedano la larghezza dello schermo è preferibile indentare di un paio di spazi, invece che di un tab.

- **Uso e commento delle funzioni** - Una funzione deve eseguire un compito e non è semanticamente corretto scrivere funzioni che contengano il codice per fare molte cose assieme. E', piuttosto, conveniente scrivere le funzioni per i singoli compiti ed una ulteriore funzione che svolga il compito complesso, richiamando le funzioni precedenti. Prima di ogni funzione, eccetto che per il main, è necessario fare un commento che contenga:
 - Scopo della funzione;
 - Significato e valori leciti per ognuno dei parametri;
 - Valori di ritorno nei vari casi;
 - Eventuali parametri passati per riferimento modificati all'interno della funzione;
 - Eventuali effetti collaterali sulle variabili globali

Ref: <http://programmazione.html.it/c/>