Lezione 12 - 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, e per questo spiegati nel dettaglio.

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];
    char autore[50];
    int anno_pubblicazione;
    float prezzo;
};
```

```
//dichiarazione dell'istanza biblio
```

```
struct libro biblio;
```

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

```
// dichiarazione della struct e della variabile biblio
```

```
struct libro
{
    char titolo[100];
    char autore[50];
    int anno_pubblicazione;
    float prezzo;
} biblio;
```

mentre è possibile pre-inizializzare i valori, 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 può accedere 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_pubblicazione;
```

```
// stampa il titolo del libro
```

```
printf ("%s \n", biblio.titolo);
```

Nuovi tipi di dato

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

```
typedef struct libro
{
    char titolo[100];
    char autore[50];
    int anno_pubblicazione;
    float prezzo;
} t_libro;
```

```
t_libro guida={"Guida al C", "Fabrizio Ciacchi", 2003, 45.2};
```

In questo modo abbiamo definito un nuovo tipo di nome "t_libro", che non è altro che una struttura; "guida" è la variabile creata di tipo "t_libro"; come per ogni altro tipo di dato, anche con "t_libro" si possono creare degli array:

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

e, per accedervi, o per inizializzare i valori, è sufficiente utilizzare l'indice per identificare l'elemento dell'array ed il punto (.) per accedere alle variabili interne del 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);
```

Unioni

Il tipo di dato **union** serve per memorizzare (in istanti diversi) oggetti di differenti dimensioni e tipo, con, in comune, il ruolo all'interno del programma. Si alloca la memoria per la più grande delle variabili, visto che esse non possono mai essere utilizzate contemporaneamente (la scelta di una esclude automaticamente le altre), condividendo il medesimo spazio di memoria. In C una unione viene definita tramite la parola chiave **union**, come mostrato di seguito:

```
union numero
{
    short numero_piccolo;
    long numero_lungo;
    double numero_grande;
} numeri;
```

Una unione è molto simile ad una struttura, ed è medesimo il modo di accedervi. Inoltre, usata in combinazione con una struttura, si possono ottenere gradevoli effetti in cui viene selezionato automaticamente il tipo giusto della union:

```
typedef struct
{
    int max_passeggeri;
} jet;
```

```
typedef struct
{
    int max_altezza;
} elicottero;
```

```

typedef struct
{
    int max_carico;
} aereocargo;

typedef union
{
    jet un_jet;
    elicottero un_elicottero;
    aereocargo un_aereocargo;
} velivolo;

typedef struct
{
    tipo_velivolo tipo;
    int velocita;
    velivolo descrizione;
} un_velivolo;

```

In questo esempio viene definita un'unione di nome "velivolo" che può essere un jet, un elicottero o un aereocargo. Nella struttura "un_velivolo" c'è un oggetto che identifica il tipo e che permette di selezionare la struttura giusta al momento giusto.

Casting

Quando si lavora con tipi di dati diversi tra loro, che siano primitivi (int, char, float, ecc.) od avanzati, può essere necessario convertire valori da un tipo ad un altro. Questa operazione si chiama **casting**, ad assume il nome di **coercizione (coercion)** quando si forza la conversione esplicita di un tipo ad un altro tipo, conversione magari non prevista automaticamente. In C, per convertire esplicitamente un tipo ad un altro tipo, si usa l'operatore (), queste parentesi tonde prendono il nome di operatore di **cast**; all'interno delle parentesi bisogna mettere il nuovo tipo al quale vogliamo passare:

```
// Casting da float ad int
```

```
int numero;
float reale;
```

```
reale = 47.19;
numero = (int)reale; // vale 47
```

ad esempio è possibile forzare la conversione di tipo dal tipo float al tipo int, o dal tipo char al tipo int;

```
// Casting da int a float
```

```
int numero;
float reale;
```

```
numero = 18;
reale = (float)numero;
```

```
/* ----- */
```

```
// Casting da char ad int
```

```
int numero;
int lettera;
```

```
lettera = 'A';
numero = (int)lettera; // vale 65, il valore ASCII di A
```

alcune conversioni di tipo vengono eseguite automaticamente (casting implicito), generalmente quando si utilizzano tipi di altro tipo come int. E' possibile, e consigliabile, utilizzare l'operatore di casting anche quando si compiono operazioni sui valori come, ad esempio, la divisione:

```
int primo;
```

```
int secondo;
float ris_div;
```

```
ris_div = (float)primo / (float)secondo;
```

Comunque la regola dice che se si è nel dubbio bisogna sempre mettere l'operatore di cast, ovvero bisogna sempre convertire esplicitamente i tipi. Il casting è e rimane un'operazione potente che, se ben utilizzata, può apportare notevoli benefici ad un programma.

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:

```
enum giorni { lun, mar, mer, gio, ven, sab, dom } settimana;
```

In questo caso abbiamo definito una nuova variabile di nome "settimana" e 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 dispendioso 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 };
```

che implica "feb" uguale a 2, "mar" uguale a 3, e così via.

Variabili statiche

Le variabili hanno la possibilità di essere visibili solo in una funzione (variabili locali) o in tutto il programma (variabili globali). Le variabili locali possono avere un'altra caratteristica, possono essere **statiche**, ovvero, essere definite una volta, ed avere vita propria all'interno di una funzione, con il vantaggio di mantenere il loro valore intatto, cioè il valore che una variabile statica ha all'uscita della funzione è il medesimo quando si rientra (richiamandola) nella funzione stessa. La parola chiave che identifica le variabili statiche è **static** e deve essere messa prima del tipo della variabile. Come mostrato nell'esempio, vengono definite due variabili in una funzione, una statica ed una non:

```
include <stdio.h>
```

```
// prototipo della funzione
```

```
void stampa();
```

```
int main()
```

```
{
    int i;
    for(i=0; i<5; i++)
    {
        stampa();
    }
}
```

```

void stampa()
{
    int alfa = 0;
    static int beta = 0;
    printf("(int) alfa = %d, (static int) beta = %d \n", alfa, beta);
    ++alfa;
    ++beta;
}

```

L'esecuzione del programma genere il seguente output a video:

```

(int) alfa = 0, (static int) beta = 0
(int) alfa = 0, (static int) beta = 1
(int) alfa = 0, (static int) beta = 2
(int) alfa = 0, (static int) beta = 3
(int) alfa = 0, (static int) beta = 4

```

ESEMPIO PRATICO

All'interno del programma abbiamo dovuto creare un nuovo tipo, di nome "t_contatto", che contenesse i dati sensibili (nome, cognome, telefono, email, sitoweb); questo è stato possibile grazie all'uso congiunto di typedef e di una struttura. Le variabili all'interno sono tutte stringhe (array di caratteri), ma potevano esserci tranquillamente int, float ed altri tipi di dato.

```

28 // Creo la struttura per contenere i contatti
29 typedef struct
30 {
31     char nome[50];
32     char cognome[50];
33     char telefono[30];
34     char email[100];
35     char sitoweb[200];
36 } t_contatto;

```

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. In particolare l'uso della memoria allocata dinamicamente risulta utile con gli array. Utilizzare queste caratteristiche del C permette di creare programmi altamente portabili, in quanto utilizzano di volta in volta i valori giusti per la piattaforma.

Una piccola nota su **come è organizzata la memoria** è doverosa: la memoria è divisa sostanzialmente in due parti, una statica, che contiene tutto quello che sappiamo verrà allocato (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()** (calloc() è stata rinominata dall'ANSI), 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. Un comando particolarmente utile risulta essere **sizeof**, che restituisce la dimensione del tipo di dato da allocare.

Per far funzionare il programma dobbiamo includere la libreria **malloc.h**, senza la quale queste funzioni non potrebbero fare il loro dovere. 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 <stdlib.h>
#include <malloc.h>

int main()
{
    int numero, *array, i;
    char buffer[15];
    int allocati;

    numero = 100;
    printf("Numero di elementi dell'array: %d", numero);

    array = (int *)malloc(sizeof(int) * numero);

    if(array == NULL)
    {
        printf("Memoria esaurita\n");
        exit(1);
    }

    allocati = sizeof(int) * numero;

    for(i=0; i<numero; i++)
    {
        array[i] = i;
    }

    printf("\nValori degli elementi\n");
    for(i=0; i<numero; i++)
    {
        printf("%6d%c", array[i], i%10 == 9 ? '\n' : ' ');
    }

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

    free(array);
    printf("\nMemoria Liberata\n");

    return 0;
}
```

In questo programma possiamo notare l'uso della funzione **malloc()** che ritorna un puntatore a carattere, corrispondente al punto di inizio, in memoria, della porzione riservata della dimensione "intera" passata come argomento; se la memoria richiesta non può essere allocata, ritorna un puntatore nullo.

Nel caso citato si può notare che è stata usata la funzione **sizeof** per specificare il numero esatto di byte, mentre è stata usata la **coercizione** per convertire il tipo di dato "puntatore a carattere" 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 (vedere lezione 13).

Adesso proporranno l'uso della funzione **realloc()** per definire un **array flessibile**, ovvero un array cui viene riservata della memoria suddivisa in blocchi di dimensione arbitraria. Una volta "saturato" il primo blocco, utilizziamo **realloc()** per allocare il blocco successivo. La suddivisione in blocchi avviene durante la fase di lettura, che essendo, appunto, dinamica, permette di minimizzare le chiamate a **realloc()** e di avere un dimensionamento abbastanza preciso.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main()
{
    char buffer[20];
    int i=0, n=0, x, *array, nb;

    /* byte allocati */
    int allocati;
    /* byte in un blocco */
    int dimbloc;
    /* byte in un intero */
    int dimint;
    /* byte contenenti interi */
    int usati;

    nb = 1;
    printf("Elementi in un blocco: %d\n", nb);

    dimint = sizeof(int);
    dimbloc = nb * dimint;
    usati = 0;

    array = (int *)malloc(dimbloc);
    if(array == NULL)
    {
        printf("Memoria insufficiente\n");
        exit(1);
    }

    allocati = dimbloc;
    printf("Allocati: %d bytes\n", allocati);
    printf("Input di interi terminati da # :\n");

    while(scanf("%d", &x))
    {
        usati += dimint;
        if(usati>allocati)
        {
            allocati += dimbloc;
            array = (int *)realloc(array, allocati);
            if(array == NULL)
            {
                printf("Memoria insufficiente\n");
                exit(1);
            }
        }
        i++;
    }
}
```

```

/* in questo modo vengono letti n interi */
    array[n++] = x;
}

printf("\n");
printf("Allocati: %d bytes\n", allocati);
printf("Dim. blocchi: %d bytes\n", dimbloc);
printf("Dim. intero: %d bytes\n", dimint);
printf("Usati: %d bytes\n", usati);
printf("Chiamate realloc: %d\n", i);
printf("Numeri: %d\n", n);
printf("\nEcco i numeri\n");

for(i=0; i<n; i++)
{
    printf("%5d%c", array[i], i%10 == 9 ? '\n' : ' ');
}

printf("\n");

return 0;
}

```

La sintassi della funzione **realloc()** ha due argomenti, il primo riguarda l'indirizzo di memoria, il secondo specifica la nuova dimensione del blocco; il tipo restituito è un tipo puntatore a void, il quale nel programma è stato castato a puntatore ad intero, per verificare che il suo valore fosse NULL.

ESEMPIO PRATICO

Facciamo notare come, all'interno del programma, venga usata la direttiva "malloc" per allocare la giusta memoria al puntatore. Tale comando permette, nel suo uso semplificato, di ottimizzare notevolmente il programma e di rendere più veloce l'esecuzione. Inoltre, allocando prima la memoria, il puntatore è già pronto per contenere le informazioni, in poche parole è stato inizializzato nel migliore dei modi.

```

265     /* creazione primo elemento */
266     // Alloco la memoria necessaria
267     p = (struct elemento *)malloc(sizeof(struct elemento));
268     // Metto daInserire nell'informazione del puntatore
269     p->inf = daInserire;
270     // p punta a NULL, ovvero il marcatore di fine lista
271     p->pun = NULL;

```