Lezione 32 C e assembly 10 + Memoria dinamica

venerdì 1 marzo 2024 14:30

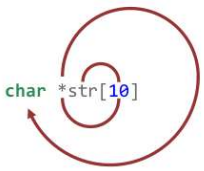
Riepilogando invece per quanto riguarda i qualificatori di tipo ed i puntatori : i primi servono a specificare i tupi di puntatori utilizzati e l'area di memoria puntata. Mentre nel secondo servono a specificare il puntatore stesso ed il tipo di dato puntato . Inoltre i primi (qualificatori di tipo) servono anche per interpretare la scrittura del codice ed i tipi di dato , visto che rendono molto versatile il linguaggio . Vediamo ora degli esempi :

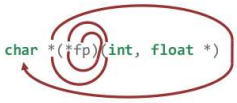
`int *ptr`: un puntatore a intero
`int const *ptr`: un puntatore a intero costante
`const int *ptr`: un puntatore a intero costante
`int * const ptr`: un puntatore costante a intero
`int const * const ptr`: un puntatore costante a intero costante
`const int * const ptr`: un puntatore costante a intero costante
`int **ptr`: un puntatore a un puntatore a intero
`int ** const ptr`: un puntatore costante a un puntatore a intero
`int * const * ptr`: un puntatore a un puntatore costante a intero
`int const **ptr`: un puntatore a un puntatore a un intero costante
`int * const * const ptr`: un puntatore costante a un puntatore costante a intero

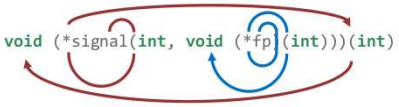
Ma da notare che in generale è un bel rompicapo capire costa si voglia dire attraverso queste istruzioni :

`char **argv`: un puntatore a puntatore a carattere
`const char * const * const ptr`: un puntatore costante a un puntatore costante a un carattere costante
`volatile char * const ptr`: un puntatore costante a un carattere volatile
`int *ptr[13]`: un vettore di 13 puntatori a intero
`int (*ptr)[13]`: un puntatore a un vettore di 13 interi
`void *ptr()`: una funzione che restituisce un puntatore generico
`void (*ptr)()`: un puntatore a una funzione senza valore di ritorno
`char (*(*x())[])()`: una funzione che restituisce un puntatore a un vettore di puntatori a funzione che restituiscono un carattere
`char (*(*x[3])())[5]`: un vettore di tre puntatori a funzione che restituiscono un puntatore ad un vettore di cinque caratteri

Quindi , vista la complessità della traduzione si usa la famosa **regola della spirale** : si parte dal nome e si procede in senso orario :

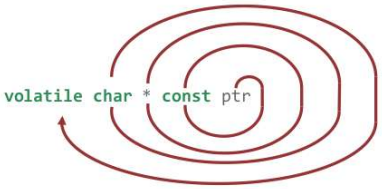
<table>
<tr>
<td> 

str è un vettore di 10 puntatori a carattere</td>
<td> 

fp è un puntatore a una funzione che accetta un intero e un puntatore a float che restituisce un puntatore a carattere</td>
</tr>
<tr>
<td> 

signal è una funzione che accetta un intero e un puntatore a una funzione che accetta un intero e non restituisce nulla che restituisce un puntatore a una funzione che accetta un intero e non restituisce nulla</td>
<td> 

x è un vettore di tre puntatori a funzione che restituiscono un puntatore ad un vettore di cinque caratteri</td>
</tr>
</table>



ptr è un puntatore costante a un carattere volatile

Il linguaggio c , consente inoltre di dichiarare **funzioni variadiche** : ovvero funzioni che accetta un numero arbitrario di parametri , mentre la stampa lo è intrinsecamente .

Funzioni variadiche

file: variadic.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

int somma(int n, ...)
{
        int nSum = 0;

        va_list int_ptr;
        va_start(int_ptr, n);

        for (int i = 0; i < n; i++)
                nSum += va_arg(int_ptr, int);

        va_end(int_ptr);

        return nSum;
}

int main(int argc, char **argv) {
        printf("10 + 20 = %d\n", somma(2, 10, 20));
        printf("10 + 20 + 30 = %d\n", somma(3, 10, 20, 30));
        printf("10 + 20 + 30 + 40 = %d\n", somma(4, 10, 20, 30, 40));

        return EXIT_SUCCESS;
}
```

Da notare l'header <stdarg.h>: permette di accedere ad un nuovo tipo che si chiama va_list (lista argomenti variadici) , la quale permette al programma di tenere traccia tra un numero di argomenti , di sapere qual è l'ultimo usato. *Questa variabile viene inizializzata con il numero dei parametri*. Andiamo a vederlo in assembler : i primi 6 parametri passati per registro , mentre gli altri per comunicazione con stack : quindi vista la complessità nell'accesso allo stack , va_start effettua una copia di tutti quanti i parametri dai registri allo stack : da notare che se i parametri maggiori di 6 , quelli dopo il sesto sono ad indirizzo più basso di quello del base pointer (ricordiamoci che gli indirizzi nello stack partono da valori altri ed arrivano a valori bassi). Al termine della funzione , i parametri vengono eliminati dallo stack(per i primi 6 ci pensa va_start), mentre per quelli eccedenti il programmatore. Andiamo a vedere un esempio riassuntivo : in assembly abbiamo la funzione **movs**: copia di buffer di memoria in un altro buffer; mentre in c si ha la **memcpy(char *from, char *to, int count)**:

```c
int memcpy(char *from, char *to, int count)
{
        int i = 0;
        for (i = 0; i < count; i++) {
                to[i] = from[i];
        }
        return i;
}
```

Nota : questa rappresentazione fa schifo !! Andiamo a vedere come renderlo più performante :

```c
int duff_device(char *from, char *to, int count)
{
        int n = (count + 7) / 8;
        switch (count % 8) {
                case 0:
                do {
                        *to++ = *from++;
                case 7:
                        *to++ = *from++;
                case 6:
                        *to++ = *from++;
                case 5:
                        *to++ = *from++;
                case 4:
                        *to++ = *from++;
                case 3:
                        *to++ = *from++;
                case 2:
                        *to++ = *from++;
                case 1:
                        *to++ = *from++;
                } while (--n > 0);
        }
        return count;
}
```

Andiamo a vedere ora la **memoria dinamica** : area di memoria dalla quale possiamo recuperare dei buffer nel momento in cui ci servono. Questa area di memoria prende il nome di HEAP : ovvero aria di memoria a noi sconosciuta in quanto viene allocata a tempo di compilazione, quindi fornisce a tempo di esecuzione tali blocchi. Quindi il programmatore deve "richiedere" buffer di memoria e rilasciarli quando non ci servono più. In C si fa ciò usando due funzioni di libreria :

`void *malloc(size_t size)`: alloca una quantità di memoria di dimensione `size` byte e restituisce un puntatore alla memoria allocata, o `NULL` in caso di errore. La memoria non è inizializzata. Se `size` è zero, `malloc()` restituisce `NULL` o un puntatore che può essere successivamente passato a `free()`.
`void free(void *ptr)`: libera l'area di memoria puntata da `ptr`, che deve essere stato precedentemente restituito da `malloc()` (o varianti).

Vediamo un esempio :

```c
int *ptr = malloc(10 * sizeof(int));
if (!ptr) {
    /* Manage the error here */
} else {
    /* Allocation successful. Do whatever you want! */
    free(ptr);  /* When memory is not needed anymore, you free it. */
    ptr = NULL; /* Set the pointer to NULL, to avoid a "dangling" pointer */
}
```

Nota la malloc non inizializza l'area di memoria , quindi se ne deve occupare il programmatore. Questa possibilità di allocare aree di memoria dinamicamente permette di avere gli **array flessibili** ovvero un vettore di cui non si specifica la dimensione (concettualmente di dimensione 0) :

```c
struct list {
        struct list *next;
        size_t size;
        unsigned char payload[];
}
```

Quindi per allocare una struttura del genere :

```c
malloc(sizeof(struct list) + size)
```

Vediamo ora una combinazione delle precedenti due :

`void *calloc(size_t nmemb, size_t size)`: restituisce un'area di memoria tale da contenere un vettore di `nmemb` elementi, ciascuno di dimensione size. La memoria è inizializzata a zero. Viene verificato se `nmemb * size` provoca overflow, caso in cui viene generato un errore.
`void *realloc(void *ptr, size_t size)`: viene "modificata" la quantità di memoria puntata da `ptr`, effettuando una nuova allocazione e spostando `size` byte da `ptr` al nuovo buffer allocato. Il buffer puntato da `ptr` viene liberato con `free(ptr)`. Il nuovo buffer viene restituito dalla funzione.

Facciamo attenzione al **memory leak** : non rilascio la memoria , aumentando cosi' la possibilità di undefined behaviour:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool leak(void)
{
        char *s = malloc(4096);

        if(s == NULL) {
                return false;
        } else {
                s[0] = 'A';
                return true;
        }
}

int main(void)
{
        while(leak());
        return 0;
}
```