Lezione 31 C e assembly 9

giovedì 29 febbraio 2024 14:53

Andiamo ora a vedere un altro tipo di struttura in c: la **union** : valore che può assumere una qualsiasi configurazione in una stessa porzione di memoria. In altre parole un'unione è la sovrapposizione di tipi di dati differenti nella stessa area di memoria , quindi con una variabile posso rappresentare più tipi di dato , ma questo tipo di dato è uno solo (assomiglia al polimorfismo di java):

```c
#include <stdio.h>

typedef enum type{
        INT,
        FLOAT,
        DOUBLE
}type_t;

typedef union operand{
        //nella struttura ci puo essere un solo tipo
        int i;
        float f;
        double d;
}operand_t;

void sum(type_t t, operand_t op1, operand_t op2)
{
        switch(t) {
        case INT:
        printf("%d + %d = %d\n", op1.i, op2.i, op1.i + op2.i);
        break;
        case FLOAT:
        printf("%.02f + %.02f = %.02f\n", op1.f, op2.f, op1.f + op2.f);
        break;
        case DOUBLE:
        printf("%.02lf + %.02lf = %.02lf\n", op1.d, op2.d, op1.d + op2.d);
        break;
        default:
        printf("Unexpected operand types.\n");
        }
}

int main(void)
{
        // Designated union initializers
        operand_t
        op11 = { .i = 3 },
        op12 = { .i = 5 },
        op21 = { .f = 2.18f },
        op22 = { .f = 8.79f },
        op31 = { .d = 1.53 },
        op32 = { .d = 4.71 };
        sum(INT, op11, op12);
        sum(FLOAT, op21, op22);
        sum(DOUBLE, op31, op32);
        printf("\nsizeof(int) = %zu\n", sizeof(int));
        printf("sizeof(float) = %zu\n", sizeof(float));
        printf("sizeof(double) = %zu\n", sizeof(double));
        printf("sizeof(operand_t) = %zu\n", sizeof(operand_t));

        return 0;
}
```

Con il seguente output :

```
3 + 5 = 8
2.18 + 8.79 = 10.97
1.53 + 4.71 = 6.24

sizeof(int) = 4
sizeof(float) = 4
sizeof(double) = 8
sizeof(operand_t) = 8
```

Notiamo che con queste strutture possiamo fare diversi cast :

```c
#include <stdio.h>
#include <stdint.h>

union binary_float_t {
        float real;
        uint32_t integer; // Assumes float is 32 bits wide
};
int main(void)
{
        union binary_float_t f;
        f.real = 3.141592F;
        printf("Hex representation of %f is %#04x\n", f.real, f.integer);
        return 0;
}
```

Con output seguente :

```
Hex representation of 3.141592 is 0x40490fd8
```

Sempre usando le struct o meglio le union, come possiamo rappresentare dati minori di 1 byte(8 bit)?? Usando la tecnica del **bit fielding** : rappresento gruppi di n-bit all'interno dei quali ne uso solamente una porzione :

```c
union flt {
        struct ieee754 {
                uint32_t mantissa: 23;
                uint32_t exponent: 8;
                uint32_t sign: 1;
        } raw;
        float f;
};

number.raw.sign = 1;
number.raw.exponent = 120;
number.raw.mantissa = 1685475;

printf("\fConverting %d %s %s to float:\n", number.raw.sign, exponent,
        mantissa);
printf("\t%f\n", number.f);
```

Quindi più in generale , esistono dei meccanismi per forzare un qualunque valore ad un qualunque valore : **Maschera di bit** : o setto il bit più significativo a 0 oppure a 1 . Nel caso dell' "1" si usa solo in assembler con l'istruzione : **orl $0x80000000, %eax** , mentre per settare la maschera di bit a "0" si usa **andl $0x7FFFFFFF, %eax** . Per quanto riguarda invece il procedimento (invero ultimo bit) uso **xorl $0x80000000, %eax** . Vediamo un esempio :

Supponiamo di avere un numero a 32 bit e di voler estrarre il valore dei 3 bit meno significativi

Si costruisce una maschera di bit del tipo 000....00111, equivalente a 7

Estrazione dei bit: `andl $7, %eax`

Per verificare se i bit sono a zero: `testl $7, %eax`

Che cosa fa l'istruzione `testq %rax, %rax`?

Andl ritorna gli ultimi tre bit meno significativi !! . Solo 3 bit perché abbiamo scelto di leggere solo "7" in codifica binaria. Con testl verifico se almeno uno degli n-bit vale 1 , con aggiornamento dello zf (zero-flag). Con l'ultima istruzione calcolo and bit-a-bit !! . **Se 0 and 0 lo zero flag vale 1.** Andiamo ora a vedere come risolvere il seguente problema : per accedere ad un bit , devo "percorrere la gerarchia di livelli" , e se questa gerarchia è grande , aumenta il tempo : usiamo le union / struct anonime: strutture come se fossero a livello inferiore:

```c
#include <stdio.h>

struct scope
{
        // Anonymous structure
        struct
        {
        char alpha;
        int num;
        };
};
int main()
{
        struct scope x;
        x.num = 65;
        x.alpha = 'A';
        printf("x.alpha = %c, x.num = %d\n", x.alpha, x.num);

        return 0;
}
```

Con output il seguente:

```
x.alpha = A, x.num = 65
```

Il programmatore può dire al compilatore come trattare le variabili , cosi in modo che il compilatore possa effettuare maggiore ottimizzazione :

const (C89): indica che i dati sono in sola lettura.

volatile (C89): indica che un valore può cambiare tra due accessi diversi, anche se apparentemente nulla l'ha cambiato (previene ottimizzazioni del compilatore).

restrict (C99): si utilizza solo con dichiarazioni di puntatori. Indica al compilatore che, per tutta la vita del puntatore, solo questo avrà accesso all'oggetto puntato.

Analogamente a quanto detto per le variabili , ovvero si usano i puntatori per salvarne il valore, esistono anche **puntatori a funzione** : variabili a cui possiamo assegnare il valore dell'indirizzo in memoria delle funzioni :

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define ELEMENTS 6

int values[] = { 40, 10, 100, 90, 20, 25 };
// puntatore a funzione
typedef int (*compare_t)(const void *, const void *);
        /**
        * retval meaning
        * <0 The element pointed to by p1 goes before the element pointed to by p2
        * 0 The element pointed to by p1 is equivalent to the element pointed to by p2
        * >0 The element pointed to by p1 goes after the element pointed to by p2
        */
int compare(const void *a, const void *b) {     return (*(int *)a - *(int *)b) ; }

void shuffle(int *array, size_t n)
{
        //ordino a casaccio il vettore
        if (n > 1) {
                size_t i;
                for (i = 0; i < n - 1; i++) {
                        size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
                        int t = array[j];
                        array[j] = array[i];
                        array[i] = t;
                        }
        }
}
void print_array(char *header, int *array, size_t n)
{
        int i;
        printf("%s: ", header);
        for (i = 0; i < n; i++)
        {
                printf("%d ", values[i]);
                puts("");
        }
}

int main(int arcg, char *argv[])
{
        compare_t compare_f1 = compare; // indirizzo della prima istruzione della funzione. Attenzione al tipo
        int (*compare_f2)(const void *, const void *) = compare; // dichiarazione esplicita
        srand(time(0));
        print_array("Original", values, ELEMENTS);
        qsort(values, ELEMENTS, sizeof(int), compare);
        print_array("Sorted with f.name", values, ELEMENTS);
        shuffle(values, ELEMENTS);
        print_array("Shuffled", values, ELEMENTS);
        qsort(values, ELEMENTS, sizeof(int), compare_f1);
        print_array("Sorted with f.ptr 1", values, ELEMENTS);
        shuffle(values, ELEMENTS);
        print_array("Shuffled", values, ELEMENTS);
        qsort(values, ELEMENTS, sizeof(int), compare_f2);
        print_array("Sorted with f.ptr 2", values, ELEMENTS);
        return 0;
}
```

Il cui output è il seguente :

```
Original: 40
10
100
90
20
25
Sorted with f.name: 10
20
25
40
90
100
Shuffled: 90
100
25
40
20
10
Sorted with f.ptr 1: 10
20
25
40
90
100
Shuffled: 100
20
25
90
40
10
Sorted with f.ptr 2: 10
20
25
40
90
100
```

Riprendendo il discorso dei puntatori in C , andiamoli ora a vedere in assembler : il loro uso è molto facile , in quanto il passaggio dei parametri viene effettuato molto velocemente , grazie alle calling convenctions. Quindi nel caso di funzione parametrica in assembler basta fare :

```
movq $function, %rax
call *%rax
```