Pointers, the best worst feature of C

Nao Pross

Le basi

I puntatori sono variabili grandi 32 o 64 bit a dipendenza del sistema operativo che contengono un *indirizzo in memoria*. I puntatori sono dichiarati usando un `*` preposto (e attaccato!) al nome della variabile.

```
int *foo;
```

L'indirizzo di una variabile si prende con l'operatore `&`. Si noti che il puntatore ha lo stesso tipo (ma puntatore) della variabile di cui si prende l'indirizzo.

```
int bar = 0;
int *foo = &bar;
```

Per accedere alla regione di memoria a cui *punta* il puntatore, si utilizza l'operatore di *dereferenza* `*`, che è convenientemente uguale alla dichiarazione per creare confusione nella mente di coloro che cercando di apprendere questo soggetto per la prima volta.

```
int bar = 0;
int *foo = &bar;

printf("%p contains %d\n", foo, *foo);
// console: 0x7ffd63f125e4 contains 0
```

Se un puntatore non punta a niente si assegna il valore `NULL`. Se si dereferenza un puntatore nullo, il programma crasha.

Operazioni aritmetiche sui puntatori (da non usare mai)

Sommando o sottraendo ad un puntatore un valore `n` il linguaggio C incrementa o diminuisce il puntatore di `n * sizeof(T)` in cui `T` è il tipo del puntatore. Ciò ha un senso (davvero).

Perché? Mettiamo di avere un puntatore `int *p`, che punta ad un certo indirizzo contente ovviamente un `int`. Un `int` solitamente sono 4 bytes (`sizeof(int)` = 4) e il puntatore `p` punta al primo di questi.

Se incrementando il puntatore questo non aumentasse di `sizeof(int)`, ossia 4, ma invece aumentasse di 1, si andrebbe a sovrascrivere il secondo byte dei 4 dell'`int`. Ergo ha unicamente senso incrementare il puntatore di 4 alla volta, per poter accedere al prossimo `int`, ammesso che ci sia, in memoria.

La verità sugli array

Gli array in C non sono che un blocco di memoria contiguo allocato da qualche parte nella heap con le celle allineate una dopo l'altra.

Un modo efficiente per accedere ai dati dell'array ti tipo `T` consiste nel salvare l'indirizzo della prima cella, poi per accedere all'elemento `n`, si prende l'indirizzo della prima cella e si somma `n * sizeof(T)`.

Ossia esattamente come funziona un puntatore, infatti la sintassi `arr[i]` è solamente una notazione più comoda dell'operazione di puntatori sottostante.

```
// arr is a pointer to the first element!
int arr[10];

// same operation
arr[3] = 42;
*(arr + 3) = 42;
```

La notazione `arr[i]` ha il vantaggio che viene controllata dal compiler, mentre l'aritmetica dei puntatori no. Quindi la seconda non è da usare, *mai*.

Puntatori `const`

Perché i puntatori non erano già sufficientemente difficili, si può mettere il `const`!

Sintassi	Puntatore	Contenuto
`char *v`	Mutabile	Mutabile
`const char *v`	Immutabile	Mutabile
`char * const v`	Mutabile	Immutabile
`const char * const v`	Immutabile	Immutabile

E per complicare ancora il tutto `const char *` e `char const *` sono equivalenti.

Degli esempi:

```
int a = 10, b = 32;
```

Con puntatore mutabile si intende che è possibile riassegnarlo:

```
int *v = &a;
const int *w = &a;
```

```
v = &b; // OK
w = &b; // Not OK
```

Il contenuto mutabile si intende poter cambiare il valore dopo la dereferenza.

```
int * x = &a;
int * const y = &a;
```

```
*x = b; // OK
*y = b; // Not OK
```

Funzioni con argomenti puntatori

Sorprendentemente i puntatori servono a qualcosa oltre a rendere difficile la vita agli studenti. Un esempio facile:

```
void increment(int *value) {
    // ALWAYS check a pointer before using
    if (value == NULL)
        return;

    (*value)++;
}
```

In questo caso la funzione incrementa la variabile fuori dal suo scopo.

```
int a = 10;
increment(&a);
// a is now 11
```

Un esempio utile:

```
uint8_t xor_checksum(uint8_t * const data, size_t size) {
    uint8_t chksum = 0;

    // ALWAYS check a pointer before using
    if (data == NULL)
        return 0;

    while (size--) {
        chksum ^= *(data++);
    }

    return chksum;
}
```

È importante notare che il puntatore (la variabile contenente l'indirizzo) è passata per *copia*, mentre il contenuto (ossia a cosa punta) è per *referenza*.

Puntatori `void`

I puntatori di tipo `void *`, sono un caso speciale per indicare un qualsiasi puntatore. Siccome il tipo `void` non ha dimensione, non è ammesso deferenziare

ne applicare dell'aritmetica ai puntatori di questo tipo. Per fare qualsiasi cosa lo si deve prima convertire in un altro puntatore.

Un esempio utile potrebbe essere per fare il checksum di un qualsiasi blocco di memoria (dall'esempio precedente):

```
uint8_t xor_checksum(void * const ptr, size_t size) {
    uint8_t * const data = (uint8_t * const) ptr;
    // same as before
}
```

Allocazione di memoria dinamica

In C i puntatori sono utilizzati soprattutto per allocare dinamicamente della memoria. Perché mai si utilizza la memoria dinamica? Prova a creare un'array dalla grandezza data da una variabile mutabile e vedrai.

Dalla standard library si ha le funzioni `malloc` e `free` (e tante altre per casi specifici) che permettono di ricevere della memoria dal sistema operativo. I prototipi di `malloc` e `free`:

```
void *malloc(size_t size); // size in bytes!
void free(void *ptr);
```

Quando si alloca della memoria per delle variabili di tipo `T`, si deve sempre allocarne un multiplo dello spazio occupato da `T` (`sizeof(T)`). Inoltre dopo l'utilizzo la memoria deve essere liberata con `free` e il puntatore annullato.

```
// allocate 128 integers
int *block = (int *) malloc(128 * sizeof(int));
...
// deallocate memory
free(block);
block = NULL;
```

Puntatori a puntatori

Siccome si può prendere l'indirizzo di una qualsiasi variabile, anche di un puntatore!

```
short val = 12;
short life = 42;
```

```
// foo points to val
short *foo = &val;
// bar points to foo
short **bar = &foo;
```

```
// foo now points to life
*bar = &life;
```

```
// life is now 12
**bar = 12;
```

Perché?? Per esempio per avere un puntatore di stringhe (puntatori di caratteri), ossia un array di stringhe.

```
// messages is actually a char **
char *messages[] = {
    "The meaning of life is 42",
    "No, actually life has no meaning",
    "What is i to the power of i?"
};
```

È possibile anche creare puntatori a puntatori di puntatori: `int ***v;`, o puntatori di puntatori di puntatori di puntatori: `char ****why;`.

`const`-hell

Puntatore a puntatori costanti di interi:

```
int * const *v;
```

Puntatore a puntatori di costanti `float`:

```
float ** const p;
```

Puntatore costante di puntatori di puntatori di puntatori di `char`:

```
const char ***strings;
```

Più precisamente, l'esempio sopra:

```
// messages is a char * const *
char *messages[] = { ... };
```

Già, buona fortuna.

Puntatori a funzioni

Un altro giorno magari, sappi che esistono ma non usarli mai. Esempio:

```
int sum(int x, int y) { return x + y; }
```

```
int (*intop)(int, int) = &sum;
int result = (*intop)(1, 2); // 3
```