Esercitazione con puntatori e `sizeof`

Filippo Cugini

Bubble sort con passaggio per riferimento

Riconsideriamo il programma per bubble sort per l'ordinamento

Introduciamo l'uso di due funzioni:

- `bubbleSort`
- `swap`

La funzione `bubbleSort` ordina l'array e chiama la funzione `swap` per scambiare tra loro gli elementi `array[j]` e `array[j + 1]`

Bubble sort con passaggio per riferimento

```c
#include <stdio.h>
#define SIZE 10

// prototipo
void bubbleSort(int * const array, size_t size);

int main(void)
{
   int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   puts("Data items in original order");

   // effettua un ciclo lungo l'array a
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }
```

Bubble sort con passaggio per riferimento

```
   bubbleSort(a, SIZE); // ordina l'array

   puts("\nData items in ascending order");

   // effettua un ciclo lungo l'array a
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

   puts("");
}
```

Bubble sort con passaggio per riferimento

```
void bubbleSort(int * const array, const size_t size)
{  // ordina un array di interi usando bubble sort
   // prototipo
   void swap(int *element1Ptr, int *element2Ptr);

   // ciclo di controllo delle iterazioni
   for (unsigned int pass = 0; pass < size - 1; ++pass) {
      // ciclo di controllo durante ogni iterazione
      for (size_t j = 0; j < size - 1; ++j) {
         // scambia gli elementi adiacenti
         // se non sono in ordine
         if (array[j] > array[j + 1]) {
            swap(&array[j], &array[j + 1]);
         }
      }
   }
}
```

Bubble sort con passaggio per riferimento

```
// scambia i valori delle locazioni di memoria
// alle quali element1Ptr ed element2Ptr
// rispettivamente puntano
void swap(int *element1Ptr, int *element2Ptr)
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}
```

Bubble sort con passaggio per riferimento

Output del programma:

```
Data items in original order
   2    6    4    8   10   12   89   68   45   37
Data items in ascending order
   2    4    6    8   10   12   37   45   68   89
```

Bubble sort - analisi

Ricordiamo che il C applica il *principio di occultamento* delle informazioni alle interazioni tra funzioni

Per questo motivo la funzione `swap` non avrebbe accesso agli elementi individuali dell'array in `bubbleSort` per default

Tuttavia `bubbleSort` vuole che `swap` abbia accesso agli elementi dell'array da scambiare

Per questo, `bubbleSort` passa per **riferimento** ognuno di questi elementi a `swap`

L'indirizzo di ogni elemento dell'array è passato esplicitamente

Bubble sort - analisi

L'**indirizzo** di ogni elemento dell'array è passato da `bubbleSort` a `swap` esplicitamente

```
if (array[j] > array[j + 1]) {
   swap( &array[j], &array[j + 1] );
}
```

Bubble sort - analisi

Gli array nella loro interezza sono passati automaticamente per riferimento

Tuttavia, i loro elementi individuali sono considerati scalari e sono ordinariamente passati per valore

Pertanto, `bubbleSort` usa l'operatore di indirizzo (`&`) con ognuno degli elementi dell'array nella chiamata di `swap` per effettuare il passaggio per riferimento

```
swap( &array[j], &array[j + 1] );
```

Bubble sort - analisi

Quando `swap` accede a `*element1Ptr`, sta in realtà facendo riferimento ad `array[j]` in `bubbleSort`

Analogamente, quando `swap` accede a `*element2Ptr`, sta in realtà facendo riferimento ad `array[j + 1]` in `bubbleSort`

```
swap( &array[j], &array[j + 1] );
```

```
void swap(int *element1Ptr, int *element2Ptr) {
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}
```

Bubble sort - analisi

L’intestazione della funzione dichiara `array` come

```
int * const array
```

invece che come

```
int array[]
```

per indicare che `bubbleSort` riceve un array unidimensionale come argomento

```
void bubbleSort(int * const array, const size_t size)
```

Bubble sort - analisi

Il parametro `size` è dichiarato `const` per applicare il principio del privilegio minimo

Sebbene il parametro `size` riceva una copia di un valore in `main` e una modifica della copia non possa cambiare il valore in `main`, `bubbleSort` non ha bisogno di alterare `size`

La dimensione dell'array rimane fissa durante l'esecuzione della funzione `bubbleSort`

Pertanto, `size` è dichiarata `const` per essere sicuri che non venga modificata

```
void bubbleSort(int * const array, const size_t size)
```

Bubble sort - analisi

Il prototipo per la funzione `swap` è incluso nel corpo della funzione `bubbleSort`

Il motivo è che `bubbleSort` è l'unica funzione che chiama `swap`

Mettere il prototipo in `bubbleSort` riduce le chiamate appropriate di `swap` solo a quelle fatte da `bubbleSort` (o qualsiasi funzione che appaia dopo `swap` nel codice sorgente)

```
void bubbleSort(int * const array, const size_t size)
{
   void swap(int *element1Ptr, int *element2Ptr);
```

Bubble sort - analisi

La funzione `bubbleSort` riceve la dimensione dell'array come parametro

La funzione deve conoscere la dimensione dell'array per metterlo in ordine

Quando un array viene passato a una funzione, la funzione riceve l'indirizzo di memoria del primo elemento dell'array

```
void bubbleSort(int * const array, const size_t size)
{
   void swap(int *element1Ptr, int *element2Ptr);
```

Bubble sort - analisi

L’indirizzo, naturalmente, non riporta il numero degli elementi nell’array

Pertanto, si deve passare alla funzione la dimensione dell’array

Ciò consente al programmatore di riusare la funzione in qualunque programma che debba ordinare array interi con singolo indice di qualsiasi dimensione

```
void bubbleSort(int * const array, const size_t size)
```

Sizeof

Il C fornisce lo speciale operatore unario `sizeof` per determinare la dimensione in byte di un array (o di un qualunque altro tipo di dati)

Questo operatore viene applicato al momento della compilazione, a meno che l'operando sia un array di lunghezza variabile

Quando è applicato al nome di un array, l'operatore `sizeof` restituisce il numero totale di byte nell'array come tipo `size_t`

Sizeof

```c
#include <stdio.h>
#define SIZE 20

size_t getSize(float *ptr); // prototipo

int main(void)
{
   float array[SIZE]; // crea l'array

   printf("Number of bytes in the array is");
   printf("%u\n", sizeof(array));
   puts("Number of bytes returned by getSize is");
   printf("%u\n", getSize(array));
}
```

Sizeof

```
// restituisce la dimensione di ptr
size_t getSize(float *ptr)
{
    return sizeof(ptr);
}
```

Output:

```
Number of bytes in the array is 80
Number of bytes returned by getSize is 4
```

Sizeof

Le variabili di tipo `float` sul nostro computer sono memorizzate in 4 byte di memoria e `array` è definito di 20 elementi

Vi sono quindi in totale 80 byte in `array`

Sizeof

Il numero degli elementi in un array si può anche determinare con `sizeof`

Ad esempio:

```
double real[ 22 ];
```

Le variabili di tipo `double` sono normalmente memorizzate in 8 byte di memoria

Pertanto, l'array `real` contiene un totale di 176 byte

Sizeof

Per determinare il numero degli elementi nell'array si può usare:

```
sizeof(real) / sizeof(real[0])
```

L'espressione determina il numero di byte nell'array `real` e divide quel valore per il numero dei byte usati per memorizzare il primo elemento dell'array (un valore `double`)

Sizeof

Anche se la funzione `getSize` riceve un array di 20 elementi come argomento, il parametro `ptr` della funzione è semplicemente un puntatore al primo elemento dell'array

Quando usate `sizeof` con un puntatore, esso restituisce la dimensione del puntatore, non la dimensione dell'elemento al quale punta

```
\\[..]
   printf("Number of bytes returned is %u\n", getSize(array));
}

// restituisce la dimensione di ptr
size_t getSize(float *ptr){
   return sizeof(ptr);
}
```

Portabilità

Il numero di byte usati per memorizzare un particolare tipo di dati può variare tra i sistemi di computer

Quando scrivete programmi che dipendono dalle dimensioni dei tipi di dati e che saranno eseguiti su diversi sistemi, usate `sizeof` per determinare il numero di byte usati per memorizzare i vari tipi di dati

Sizeof

L’operatore `sizeof` può essere applicato a un qualsiasi nome di variabile, tipo o valore (compreso il valore di un’espressione)

Quando è applicato al nome di una variabile (che non è il nome di un array) o a una costante, viene restituito il numero di byte usati per memorizzare il tipo specifico della variabile o della costante

Le parentesi sono necessarie quando l’operando di `sizeof` è un tipo di dati