Puntatori a funzioni - 1

Un puntatore a funzione è una variabile che contiene un indirizzo (come per i puntatori a dati). L'indirizzo sarà il puntatore ad una zona di memoria che contiene il codice della funzione puntata.

Come per i puntatori ai dati, ci sono vari passi che coinvolgono l'uso dei puntatori a funzioni.

Primo: bisogna dichiarare la variabile che conterrà un puntatore a funzione. Tale dichiarazione è un pò complessa, ed è del tipo:

```
int (*pf)();
```

dichiara `pf` come un puntatore ad una funzione che ritorna un intero. Il simbolo `*` indica che si tratta di un puntatore e le parentesi `()` indicano che è coinvolta una funzione. La funzione a cui punta non ha argomenti.

Si noti che eliminando una coppia di parentesi, l'istruzione:

```
int *pf();    // non è un puntatore a funzione
```

dichiara una funzione che restituisce un puntatore ad intero.

Secondo: si possono assegnare dei valori al puntatore a funzione che sono l'indirizzo della funzione a cui deve puntare.

Definita una funzione `f1()`:

```
int f1() { ... }
```

possiamo far puntare pf alla funzione `f1()` tramite:

```
pf = f1;
```

La precedente assegnazione è equivalente a:

```
pf = &f1;
```

poichè il nome della funzione individua il suo indirizzo.

Puntatori a funzioni - 2

Terzo: un puntatore ad una funzione è usato per invocare una funzione, così dopo aver dichiarato e assegnato il puntatore a funzione possiamo invocare `pf()`

```
int k = pf();
```

questa istruzione è equivalente all'invocazione:

```
int k = f1();
```

oppure all'invocazione:

```
int k = (*pf)(); // con dereferenziazione
```

Quarto: un puntatore ad una funzione può essere passato come argomento ad una funzione:

```
void func( int (*pf)() )
```

La funzione `func()` è una funzione che prende come argomento `pf` cioè un puntatore a funzione che ritorna un intero.

Esempio:

Funzione che prende un `int` e ritorna un `int`

```
int func1(int x) {
   if (x<1) return 1;
   return x*(x-1);
}
```

Funzione che prende puntatore a funzione ed `int` e ritorna un `int`

```
int func2(int (*pf)(int), int n) {
   return pf(n*n);
}
```

Puntatori a funzioni - 3

Dichiarazione ed invocazione funzioni

```
int main() {
   int (*pf)(int);  // dichiarazione
   pf = func1;      // assegnazione
   printf("Restituito da func1: %d\n", func1(3));
   // richiamiamo func2 passando a questa func1
   printf("Restituito da func2: %d\n", func2(pf,3));
}
```

L'istruzione `typedef` ci aiuta a semplificare la notazione con i puntatori a funzioni:

```
typedef int (*pFun)(int);
```

definisce (non dichiara) un puntatore a funzione `pFun`

Possiamo quindi riscrivere il programma con PFun

```
int func2(pFun pf, int n) {
   return pf(n*n);
}

int main() {
   pFun pf;         // dichiarazione
   pf = func1;      // assegnazione
   printf("Restituito da func1: %d\n", func1(3));
   // richiamiamo func2 passando a questa func1
   printf("Restituito da func2: %d\n", func2(pf,3));
}
```

Esempio di Puntatori a funzioni - 1

Esempio: creiamo una funzione `sort()` che (1) ordina un array di stringhe, (2) ordina ignorando maiuscole e minuscole e (3) ordina considerando il valore delle stringhe quando esse rappresentano numeri.

Soluzione: dobbiamo disaccoppiare la funzione sort dal confronto tra stringhe.

```
void sort(char *strings[], int n, int (*cmpfunc)()) {
   int i, j;
   int didswap;
   do {
      didswap = 0;
      for(i = 0; i < n - 1; i++) {
         j = i + 1;
         if ((*cmpfunc)(strings[i], strings[j]) > 0) {
            char *tmp = strings[i];
            strings[i] = strings[j];
            strings[j] = tmp;
            didswap = 1;
         }
      }
   } while(didswap);
}
```

Quindi implementiamo delle funzioni di confronto

```
int ignoreCaseCmp(char *str1, char *str2) {
   while (1) {
      char c1 = *str1++;
      char c2 = *str2++;
      if (isupper(c1)) c1 = tolower(c1);
      if (isupper(c2)) c2 = tolower(c2);
      if (c1 != c2) return c1 - c2;
      if (c1 == '\0') return 0;
   }
}
```

Esempio di Puntatori a funzioni - 2

```
int numStrCmp(char *str1, char *str2) {
   int n1 = atoi(str1);
   int n2 = atoi(str2);
   if (n1 < n2) return -1;
   else if (n1 == n2) return 0;
        else return 1;
}
```

A seconda dell'ordinamento che vogliamo avere, passiamo alla funzione `sort()` l'opportuna funzione di confronto

```
int main() {
    char *array1[] = {"Zeppelin", "detto", "dado", "Charlie"};

     sort(array1, 4, StrCmp);

     sort(array1, 4, ignoreCaseCmp);

    char *array2[] = {"1", "234", "12", "3", "4", "24", "2"};

     sort(array2, 7, numStrCmp);
}
```

In una implementazione alternativa, senza puntatori a funzioni, dovremmo controllare all'interno della funzione `sort()` il tipo di confronto da effettuare, con lo svantaggio di complicare il codice di `sort()` e di legare la funzione `sort()` alle funzioni di confronto

Utilita' dei Puntatori a funzioni

Riassumendo, i puntatori a funzioni sono utili per:

- rendere il codice più “pulito” facendo a meno di istruzioni condizionali tipo `switch` ed `if`
- disaccoppiare le funzioni, rendono una funzione `f()` dipendente da un argomento (puntatore ad una funzione) anzichè da una funzione specificata all'interno del corpo di `f()`

Forniscono quindi maggiore flessibilità e potenza nell'esprimere le istruzioni di un programma e consentono di implementare alcuni costrutti in modo più sintetico (compatto).