# Puntatori a Funzioni e Callback

Massimo Benerecetti

# Puntatori a Funzione

- Un **«puntatore a funzione»** è un puntatore che punta all'indirizzo di memoria in cui è contenuto il **codice eseguibile di una funzione**.
- La memoria indirizzata da un **puntatore a funzione** si trova nel **text-segment** di memoria associata al processo (**memoria statica**).
- La memoria indirizzata da un **puntatore a funzione non deve**, quindi, essere allocata.
- **Importante**: un puntatore a funzione punta sempre a una funzione con una **specifica segnatura**.
- Lo **stesso puntatore** può indirizzare funzioni diverse ma con gli **stessi (tipi di) parametri** e **stesso tipo di valore di ritorno**.

# Funzioni e Puntatori a Funzione

- **Prototipo** di funzione che restituisce un intero e non riceve parametri:

```
int GetInputValue(void);
```

- **Dichiarazione** di un **puntatore a funzione** (i.e., **GetValue**) che restituisce un intero e non riceve parametri:

```
int (*GetValue)(void);
```

- **Dichiarazione** del tipo **FNINPUT** per **puntatori a funzione** che restituiscono un intero e non ricevono parametri:

```
typedef int (*FNINPUT)(void);
```

- **Dichiarazione** di una variabile di tipo **FNINPUT** (**puntatore a funzione**):

```
FNINPUT GetValue;
```

# Esempio

```
typedef int (*FNINPUT)(void);

int main(void)  {
  int ret;
  FNINPUT GetValue;
  ...
  GetValue = &GetInputValue; /* Assegnazione a variabile di tipo */
  GetValue = GetInputValue;  /* puntatore a funzione, assegnato */
                             /* all'indirizzo della funzione */
                             /* "GetInputValue()" */
  ...
  ret = GetValue();     /* Chiamata di funzione tramite puntatore */
  ret = (*GetValue)(); /* Chiamata di funzione tramite puntatore */
  ...
}
```

# Funzioni di Callback

- Una **funzione di callback** è una funzione chiamata tramite un puntatore a funzione.
- Supponiamo che a una funzione venga passato come argomento il puntatore (indirizzo) di un'altra funzione.
- Quando la pima funzione usa il puntatore a funzione per chiamare la seconda funzione, viene effettuata una **callback**.
- Le **callback** risultano particolarmente utili per rendere il codice più generale, **disaccoppiando** la funzione chiamante dalla funzione chiamata.
- Può essere utile nella definizione di **librerie** (es. libreria di liste generiche, di alberi, ordinamenti di generici elementi, etc.) e, in generale, per l'implementazione di ***Tipi di Dati Astratti***.
- Rendere il codice delle funzioni indipendente da dati specifici la cui tipologia non ha impatto sulla logica del loro funzionamento.

# Esempio di Callback 1

Si supponga di voler definire una funzione che, dato un array **A**, applichi ad ogni suo elemento l'operazione di ***elevamento a quadrato***.

```
Array_Square(int *A, int dim)
  for (i=0; i < dim; i++)
      A[i] = A[i] * A[i];
```

Supponiamo ora di voler definire una funzione che, dato un array **A**, applichi ad ogni suo elemento l'operazione di ***incremento***.

```
Array_Incr(int *A, int dim)
  for (i=0; i < dim; i++)
      A[i] = A[i] + 1;
```

# Esempio di Callback 1

Possiamo definire un'unica funzione **Array_Op** che applica una qualche operazione a tutti gli elementi dell'array **A**.

```
typedef int (*FN_op)(int);

int square(int a) { return (a * a);}

int increment(int a) { return (a + 1);}

int double(int a) { return (2 * a);}

Array_Op(int *A, int dim, FN_op op) {
  for (i=0; i < dim; i++)  A[i] = op(A[i]);
}

int main(void) {
   Array_op(A,n,&square);
   Array_op(A,n,&increment);
   Array_op(A,n,&double);
}
```

# Esempio di Callback 2

- Si supponga di voler definire una libreria di gestione liste di interi (o di qualche altro tipo).
- Il popolamento della lista può avvenire in vari modi:
  - ad es. tramite input diretto dell'utente o tramite generazione casuale di interi.
- Possiamo rendere la gestione dell'inserimento degli interi indipendente dalla modalità di generazione dei valori, utilizzando una callback **`GetValue()`**.

# Popolamento di una lista

```
lista *PopolaLista(lista *L, int nelem) {
    /* Popola una lista ordinata L inserendo nelem
       elementi utilizzando una 'generica' funzione
       GetValue() per ricevere i valori da inserire. */
   int elem, i;
   for (i=0; i < nelem; i++) {
      elem = GetValue();
      L = InserisciOrd(L,elem);
   }
   PrintLista(L);
   return(L);
 }
```

# Esempio Callback 2

```
int GetInputValue(void) {
    int k;
    printf("Inserisci un valore intero: ");
    scanf("%d", &k);
    return(k);
 }

int GetRandomValue(void) {
     return( rand()%100 ); /* valore pseudo-casuale
                              tra 0 e 99 */
 }
```

# Esempio Callback 2

```
typedef struct List {
   int key;
   struct List *next;
} lista;

int main(void) {
    lista *L1=NULL;
    lista *L2=NULL;
    …
    L1 = PopolaLista(L1, 20, &GetInputValue);
    PrintLista(L1);
    L2 = PopolaLista(L2, 20, &GetRandomValue);
    PrintLista(L2);
    …
}
```

# Esempio Callback 2

```
/* Dichiarazione di tipo puntatore a funzione */
typedef int (*FNINPUT)(void);

lista *PopolaLista(lista *L, int nelem, FNINPUT GetValue) {
    /* Funzione che popola una lista L inserendo nelem
       elementi utilizzando la callback GetValue() per
       ricevere i valori da inserire. */
   int elem, i;
   for (i=0; i < nelem; i++) {
      elem = GetValue();     /* oppure (*getValue)() */
      L = InserisciOrd(L,elem);
    }
   PrintLista(L);
   return(L);
 }
```

# Dati generici e puntatori “void”

Spesso può essere utile dichiarare variabili che possano **contenere tipi di dati diversi**.

Il **C** non ammette, però, variabili di tipo generico **void**:

```
void a;    /* dichiarazione non valida */
```

Ammette, invece, **puntatori** a dati di tipo generico **void**:

```
void *p;        /* dichiarazione valida */
```

Un puntatore **void** può essere usato per indirizzare qualsiasi tipo di dato

- `int x;` `void* p = &x;` `/* p punta a un int */`
- `float f;` `void* p = &f;` `/* p punta a un float */`

L’aritmetica dei puntatori tratta tutti i puntatori a **void** come puntatori a singoli **byte**, quindi:

- se `int *x;` e `void* p = x;` allora $(p + i) \neq (x + i)$

# Dereferenziazione di puntatori a void

I puntatori a **void** non possono essere dereferenziati direttamente

- `void* p; printf ("%d",*p); /* non valido */`

Devono essere sempre sottoposti a **cast** prima di accedere al loro contenuto:

- `void* p; printf ("%d",*((int *)p); /* valido */`

Analogamente, l'assegnamento di un valore al contenuto di un puntatore a **void** necessita di **cast**:

- `int x; void* p = &x; *p = 5;  /* non valido */`
- `int x; void* p = &x; *((int *)p) = 5; /* valido */`

In alternativa, è possibile usare la funzione di libreria `memcpy(dest,source,n_byte)` per copiare un numero dato di ***byte contigui*** da un qualsiasi indirizzo **sorgente** a un indirizzo **destinazione**, ***indipendentemente dal tipo di dati contenuti***:

- ```
  int x; int y = 5; void* p = &x;
  memcpy(p,&y,sizeof(int)); /* valido */
  ```
  ```

# Esempio di Callback 3

- Si supponga di voler definire una funzione generale per il popolamento di un qualsiasi array, che possa funzionare per ogni tipo di dato.
- Tipi di dati diversi dovranno essere generati in modo differente (ad. interi, float, record/struct ecc.).
- Per i valori numerici (int/float) vogliamo permettere anche una generazione di valori casuali.
- Possiamo definire una funzione **`PopolaArray`** che utilizza una callback **`GetValue()`** per generare gli elementi specifici dell'array che deve popolare, indipendentemente dal tipo di dato dell'array.

# Esempio di Callback 3

```
typedef void (*GETIN)(void *);

typedef struct STRNOME {
    char nome[30];
    char cognome[30];
} NOME;

int main(void)  {
    NOME array1[10];
    float array2[10];
    int array3[10];

    PopolaArray(array1,10, sizeof(NOME), &GetInputNome);
    PopolaArray(array2,10, sizeof(float), &GetRandomFloat);
    PopolaArray(array3,10, sizeof(int), &GetRandomInt);
}
```

# Esempio di Callback 3

```
void PopolaArray(void *A, int nel, int dimel, GETIN GetValue) {
   int elem, i;
   void *elemento = malloc(dimel);
   for (i=0; i<nel; i++) {
        GetValue(elemento);
        CopiaDato(A,i,elemento,0,dimel);
    }
   free(elemento);
 }
void CopiaDato(void *dest,int dpos, void *src,int spos, int dim) {
   void *dst_addr = dest+(dpos*dim);
   void *src_addr = src+(spos*dim);
   memcpy(dst_addr,src_addr,dim);
 }
```

# Esempio di Callback 3

```
void GetInputInt(void *k)
{
   printf("Inserisci un valore intero: ");
   scanf("%d", (int *)k);
}

void GetRandomInt(void *k) {
   /* intero casuale 0 ≤ k < 100 */
   *((int *) k) = rand()%100;
}
```

# Esempio di Callback 3

```
void GetInputFloat(void *k)   {
     printf("Inserisci un valore float: ");
     scanf("%f", (float *)k);
}


void GetRandomFloat(void *k)   {
     /* float casuale 0 ≤ k < 100 */
     float d = (float)rand()/(float)RAND_MAX;
    *((float *) k) = (rand()%100) + d;
}
```

# Esempio di Callback 3

```
void GetInputNome(void *elem) {
    NOME *temp = (NOME *)elem;

    printf("Inserisci un nome (max 30): ");
    scanf("%s", temp->nome);
    printf("Inserisci un cognome (max 30): ");
    scanf("%s", temp->cognome);
}
```

# Esempio di Callback 4

```
typedef int (*COMPAREFN)(void *,void *);

int ConfrontaInt(void *a, void *b) {
    int inta = *((int *)a);
    int intb = *((int *)b);
    return ((inta > intb)? 1 :((inta < intb)? -1 : 0));
}

int main(void) {
    int array[100];
    PopolaArray(array,100,sizeof(int),&GetRandomInt);
    InsertionSort(array,100,sizeof(int),&ConfrontaInt);
}
```

# Ordinamento per array generico

- Si supponga di voler generalizzare l'algoritmo di `InsertSort` per ordinare array che possono contenere ***vari tipi di dati***.
- Tipi di dati diversi hanno, infatti, diverse logiche di confronto (ad. interi, stringhe, ecc.) .
- Possiamo definire `InsertSort` in modo che utilizzi una callback `Compare()` per confrontare gli elementi del generico array che deve ordinare.
- `Compare()` punterà a diverse funzioni di confronto, a seconda del tipo di dati correntemente contenute nell'array

# Algoritmo di InsertionSort

```
InsertSort(A)
   for j = 1 to Lenght(A)-1 do
      Key = A[j]
      i = j-1
      while (i ≥ 0 and A[i] > Key) do
            A[i+1] = A[i]
            i=i-1
       A[i+1] = Key
```

# Esempio di Callback 4

```
void InsertSort(Array_T A, int dim, int el_dim, COMPFN Compare)
{
    void *key = malloc(el_dim);
    for (j = 1; j < dim; j++) {
        CopiaDato(key,0,A,j,el_dim);     /* key = A[j] */
        i = j-1;
        while(i>=0 && Compare(A+(i * el_dim), key)>0) {
            CopiaDato(A,i+1,A,i,el_dim);     /* A[i+1] = A[i] */
            i--;
         }
        CopiaDato(A,i+1,key,0,el_dim);      /* A[i+1] = key */
     }
   free(key);
}
```

# Implementazione di ADT

- Tramite il meccanismo dei ***puntatori*** a tipo ***void*** e delle ***callback*** è, quindi, possibile definire in ***C Tipi di Dati Astratti*** (**ADT**).

- Come esempio, potremmo definire un **ADT array** nel seguente modo:

```
struct Array {
    void *elements[MAX_ELEMENTS];
    int  num_elements;
};
```

- e associare ad esso i relativi operatori.

# Esempio ADT

```
typedef struct Array *Array_T;

Array_T Array_new(void) {
    Array_T array = malloc(sizeof(struct Array));
    array->num_elements = 0;
    return array;
}

void Array_free(Array_T array) {
    free(array);
}

int Array_length(Array_T array) {
    return array->num_elements;
}
```

# Esempio ADT

```
void *Array_data(Array_T array, int index) {
   return array->elements[index];
}

void Array_append(Array_T array, void *datap) {
  int index = array->num_elements;
  if (index < MAX_ELEMENTS) {
     array->elements[index] = datap;
     array->num_elements++;
  } }

void Array_replace(Array_T array, int index, void *datap) {
  array->elements[index] = datap;
}
```

# Esempio ADT

```
void Array_insert(Array_T array, int index, void *datap) {
  int i;
  if (array->num_elements < MAX_ELEMENTS && index >= 0 &&
      index < MAX_ELEMENTS) {
      for (i = array->num_elements; i > index; i--)
            array->elements[i] = array->elements[i-1];
      array->elements[index] = datap;
      array->num_elements++;
 } }
void Array_remove(Array_T array, int index) {
  int i;
  if (array->num_elements > 0 index >= 0 && index < MAX_ELEMENTS) {
    for (i = index+1; i < array->num_elements; i++)
       array->elements[i-1] = array->elements[i];
    array->num_elements--;
  } }
```

# Esempio ADT

```
void Array_InsertSort(Array_T array, COMPFN Compare)
{
   void *key;
   int j,i;
   for (j = 1; j < array->num_elements; j++) {
      key = array->elements[j];
      i = j-1;
      while(i>=0 && Compare(array->elements[i], key) > 0) {
           array->elements[i+1] = array->elements[i];
           --i;
      }
    array->elements[i+1] = key);
    }
}
```

# Esempio ADT

```
#include "array.h"
#include <stdio.h>
#include <string.h>
int main() {
   int i;
   Array_T array = Array_new();
   Array_append(array, (void *)"IS");
   Array_append(array, (void *)"FUN");
   Array_append(array, (void *)"COS217");
   Array_InsertSort(array, &strcmp);
   for (i = 0; i < Array_length(array); i++) {
         char *str= (char *)Array_data(array, i);
         printf("%s ",str);
    }
   Array_free(array);
}
```

# Esercizio

- Definire una libreria per lo **ADT** Albero Binario di Ricerca.
- Un elemento dell'ADT è un ABR i cui nodi possono contenere un dato di tipo qualsiasi, sia un dato di tipo predefinito che di tipo definito dall'utente.

1. Estendere la libreria di ARB nell'esercizio 1 del corso, in modo che possa gestire ARB generici.
2. Definire una struttura per il nodo della lista in grado di contenere un qualsiasi tipo di dato.
3. Definire diverse funzioni di costruzione e distruzione del dato, in base al tipo di dato da costruire (interi, reali, stringhe, record composti,…) da utilizzare come ***callback***.
4. Definire le funzioni di confronto tra due elementi, dipendenti dal tipo dei dati da confrontare e da utilizzare come ***callback***.