Il linguaggio C

Puntatori, puntatori a funzioni, liste generiche

Liste e alberi in C

- Usando i puntatori

```
typedef struct elem {
   struct elem * next;
   int key;
} elem_t;

typedef struct nodo {
   struct nodo * left;
   struct nodo * right;
   int key;
} nodo_t;
```

Liste e alberi in C (2)

- Aggiungere in testa

```
elem_t* insert (elem_t* head, int k) {
  elem_t* tmp;
  tmp = malloc(sizeof(elem_t));
  if ( tmp == NULL ) return head;
  tmp -> next = head;
  tmp -> key = k;
  return tmp;
}
```

Liste e alberi in C (3)

- Ricercare .....

```
elem_t* find (elem_t* head, int k) {
  if ( head == NULL ) return NULL;
  if  ( head -> key  == k )
      return head;
  return find ( head -> next , k );
}
```

Liste e alberi in C (4)

- Come realizzare una lista ‘generica’ in C ?
  - In cui le chiavi possono essere di tipo qualsiasi (non solo **`int`**) ?
  - Si usano
    - puntatori a void (void *) per essere in grado di puntare qualsiasi tipo nel campo chiave
    - Si usano i puntatori a funzione per gestire correttamente il confronto e la copia di elementi di tipo void*

tipo puntatore generico: `void*`

- Puo' contenere indirizzi di variabili di tipi diversi
  - Non si può dereferenziare
  - É necessario un cast prima di manipolare la variabile puntata.
  - Es :

```
void * c;
int a;
c = &a;
*c = 5;  /* scorretto*/
*(int *)c = 5;    /* corretto*/
```

tipo puntatore generico: `void*` (2)

- Serve a scrivere funzioni ‘polimorfe’ in modo un po’ brutale
- Es :
  - il tipo della malloc() è

```
void * malloc (unsigned int
 size);
```

  - quando scrivo

```
int * a;
a = malloc(10*sizeof(int));
```

  - viene effettuato un cast implicito a (int *)

tipo puntatore generico: `void*` (3)

- Tipi delle altre funzioni di allocazione e deallocazione

```
void * calloc (unsigned int
 size);
void * realloc (void * ptr,
                unsigned int
 size);
void free (void * ptr);
```

Liste generiche in C

- Usando i puntatori

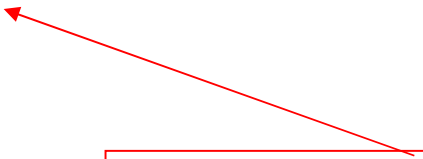
```
typedef struct elem_t {
   struct elem_t * next;
   void * key;
} elem_t;
```

Non é più noto il tipo della chiave

Liste generiche in C (2)

- Aggiungere in testa

```
elem_t* insert (elem_t* head, void* k) {
  elem_t* tmp;
  tmp = malloc(sizeof(elem_t));
  if ( tmp == NULL ) return head;
  tmp -> next = head;
  tmp -> key = k;
  return tmp;
}
```



Questo non si puo' piu' fare

Liste generiche in C (3)

- Aggiungere in testa

```
elem_t* insert (elem_t* head, void* k, funz
 copiakey ) {
  elem_t* tmp;
  tmp = malloc(sizeof(elem_t));
  if ( tmp == NULL ) return head;
  tmp -> next = head;
  tmp -> key = copiakey(k);
  return tmp;
}
```

I puntatori a funzione

- Consideriamo la funzione :

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

  - se proviamo ad eseguire

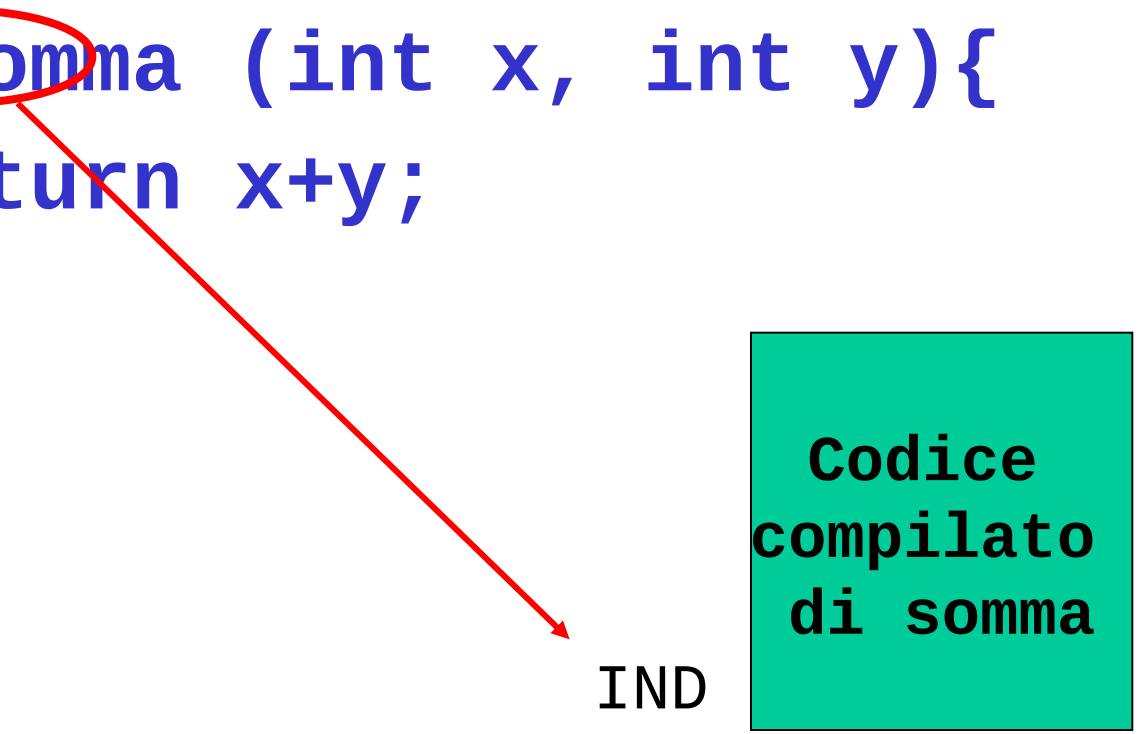
```
printf("%p",somma);
```

    otteniamo un valore esadecimale che rappresenta un indirizzo legale del nostro programma
  - ??????????????????????????

I puntatori a funzione (2)

- Consideriamo la funzione :

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



somma è un puntatore costante con valore pari a **IND**

I puntatori a funzione (3)

- Consideriamo la funzione :

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

/* variabile di tipo funzione
 (int,int)->int */
int (*fun) (int,int);
int a;

fun = somma;
a = fun(3,5);
```

I puntatori a funzione (4)

- Consideriamo la funzione :

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

/* variabile di tipo funzione
 (int,int)->int */
int (*fun) (int,int);
int a;

fun = somma;
a = fun(3,5);
```

Ma a che serve ???????????

I puntatori a funzione (5)

- Serve a definire funzioni che prendono come argomenti altre funzioni (*di ordine superiore*) :

```
void map (int (*fun) (int),
          int x[], int l){
    for(i=0;i<l;i++)
     x[i] = fun(x[i]);
}
```

  - è un iteratore che applica la funzione **`fun`** a tutti gli elementi dell'array **x**

I puntatori a funzione (6)

- Esempio di uso della map :

```
int piu_uno (int x){
   return x+1;}
int quad (int x){
   return x*x;}
...
int a[3] = {3,4,5};
map(piu_uno,a,3); /* somma uno a   tutti
 gli elementi */
map(quad,a,2); /* eleva al quadrato i primi
 due elementi */
```

Genericita': esempio qsort

```
void qsort (void* base,
size_t nmemb,
size_t size,
int (*compar) (void*,void) );
```

– Ordina una array di tipo qualsiasi (**`base`**) di **`nmemb`** elementi ognuno di ampiezza **`size`** utilizzando la funzione **`compar`** per confrontare gli elementi

Genericita': esempio qsort (2)

- Come si usa qsort ? (vedi man qsort)

```
int array[N];

int cmpint (void* a, void* b){
  int ai = * (int*) a;
  int bi = * (int*) b;
  return ai - bi;
}
…
qsort(array,N,sizeof(int),cmpint);
```

Genericita' esempio qsort (3)

```
int compar_string (void* a, void* b) {
    char* ai = (char *) a;
    char* bi = (char *) b;
    return strcmp(ai,bi)};

char* string[MAX];
qsort(string,MAX,sizeof(char*),compar_string);
```

- Ordina l'array **`string`** usando la funzione **`compar_string`** passata come argomento

Genericita': esempio qsort (4)

- Come si implementa qsort ? (cenni)

```
void qsort (void* base, size_t nmemb,
size_t size, int (*compar) (void*,void*) ) {
/* accedo agli elementi */
unsigned char * pfirst = (char*) base;
unsigned char * psecond;
pfirst; /* puntatore primo elemento */
psecond = pfirst + size; /*aritmetica dei
  puntatori mi creo il puntatore al secondo
  elemento */
……
/* confronto gli elementi */
 compar((void*)pfirst,(void*)psecond);
```

Liste generiche in C (3)

- Aggiungere in testa

```
elem_t* insert (elem_t* head, void* k, funz
 copiakey ) {
  elem_t* tmp;
  tmp = malloc(sizeof(elem_t));
  if ( tmp == NULL ) return head;
  tmp -> next = head;
  tmp -> key = copiakey(k);
  return tmp;
}
```

Liste generiche in C (4)

- Aggiungere in testa

```
elem_t* insert (elem_t* head, void* k, void*
 (*copiakey) (void *) ) {
  elem_t* tmp;
  tmp = malloc(sizeof(elem_t));
  if ( tmp == NULL ) return head;
  tmp -> next = head;
  tmp -> key = copiakey(k);
  return tmp;
}
```

Liste generiche in C (5)

- Ricercare .....

```
elem_t* find (elem_t* head, void* k) {
  if ( head == NULL ) return NULL;
  if  ( head -> key  == k )
      return head;
  return find ( head -> next , k );
}
```

Questo non funziona più
Con un qualsiasi tipo

Liste generiche in C (6)

- Ricercare .....

```
elem_t* find (elem_t* head, void* k,
int (*compare) (void*, void*)) {
  if ( head == NULL ) return NULL;
  if  ( compare( head -> key, k )==0 )
      return head;
  return find ( head -> next , k );
}
```

Liste generiche in C (7)

- Per non dover passare sempre molti parametri:
  - possiamo prevedere una struttura che contenga le funzioni rilevanti per ciascuna istanza della lista generica

```
typedef struct {
  elem_t* head; /*testa*/
  int (*compare) (void*, void*);
  void* (*copiakey) (void*);
} lista_t;
```

Liste generiche in C (8)

- Per non dover passare sempre molti parametri:
  - Poi passiamo **`lista_t`** alle varie funzioni

```
elem_t* find (lista_t* l, void*
 k ) {…}
lista_t* insert (lista_t* l, void*
 k ) {…}
```

  - L'implementazione la sviluppiamo nelle ore di esercitazione

Liste generiche in C (9)

- Come si definisce dalla lista generica una lista di interi ?
  - Definendo due funzioni per la copia e il confronto e allocando una nuova struttura lista
  - La **`compareint`** è la stessa del qsort
  - La **`copiaint`** si può definire come

```
void* copiaint (void* a) {
  int * tmp;
  if ( ( tmp=malloc(sizeof(int))) == NULL)
       return NULL;
  *tmp= * (int *) a
  return (void*) tmp;
}
```

Liste generiche in C (10)

- La lista di interi si ottiene con

```
lista_t * string_list;
string_list=malloc(sizeof(lista_t);

string_list->head=NULL;
string_list->compare= compareint;
string_list->copiakey= copiaint;
```