UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento
di Fisica
e Astronomia
"Ettore Majorana"

L-30 CLASSE DELLE LAUREE IN SCIENZE E TECNOLOGIE FISICHE
CORSO DI LAUREA IN FISICA
PROGRAMMAZIONE AD OGGETTI E BIG DATA

SANDRO FIORETTO

REALIZZAZIONE IN C DELL'OGGETTO LISTA

SOMMARIO

L'OBIETTIVO DELLA RELAZIONE È PRESENTARE L'OGGETTO "LISTA", REALIZZATO IN C IN DUE VERSIONI: UNA CLASSICA IN CUI VIENE CREATO UN NUOVO ELEMENTO AD OGNI VALORE INSERITO E UNA PIÙ VELOCE PERCHÉ UTILIZZA, ASSIEME A TUTTE LE ALTRE LISTE DELLO STESSO TIPO, UN UNICO SPAZIO IN MEMORIA PREALLOCATO.

ANNO ACCADEMICO 2022/2023 - 3° ANNO

1 Cos'è la lista in informatica

La lista è una struttura dati dinamica. A differenza dell'array può facilmente cambiare numero di elementi e i suoi elementi non sono necessariamente contigui in memoria.
La complessità delle seguenti funzioni della lista non dipende dalla sua dimensione:

- inserimento e rimozione in testa alla lista
- inserimento in coda alla lista

ma vi dipende nel caso di:

- rimozione in coda alla lista
- inserimento e rimozione all'interno della lista
- ricerca di elementi interni alla lista

È perciò particolarmente utile quando il numero di elementi non è noto a priori e non si devono effettuare molte operazioni con elementi interni alla lista.

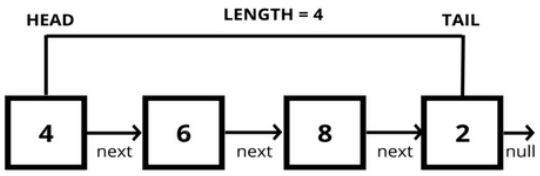


Figura 1: Schema della lista. Gli oggetti non sono contigui in memoria ma ciascuno contiene l'informazione su come raggiungere il successivo.

2 Tipi di lista

Le liste di tutti i tipi vengono istanziate attraverso la funzione $malloc\_list$, internamente tuttavia sono fornite 6 classi derivate, 3 per ciascun tipo. I due tipi di lista sono $list\_dynamic$ e $list\_table$.

La $list\_dynamic$ è la versione più classica della lista: la lista occupa una quantità di memoria proporzionale al numero di elementi contenuti. Ha il vantaggio rispetto alla versione successiva di occupare solamente la memoria necessaria per contenere i suoi elementi, tuttavia l'inserimento di ciascun elemento richiede un'allocazione in memoria, il ché rende la $list\_dynamic$ più lenta degli array o della sua controparte $list\_table$.

La $list\_table$ è una versione in cui viene preallocata una quantità di memoria arbitraria (detta $table$ o tabella) all'istanziamento della prima lista e nella quale vengono salvati gli elementi di tutte le liste della stessa natura. Ciascuna table infatti deve avere elementi della stessa dimensione quindi potrà essere condivisa solo da liste che che contengono lo stesso tipo di dato (sarà istanziata una table per le liste che contengono interi, una per i float etc.).
Il vantaggio di questa versione rispetto alla precedente è che lavorare con la table è più veloce che lavorare con puntatori; rispetto agli array è che le liste contenute nella table possono avere numero di elementi variabile.
Lo svantaggio è che quando la memoria preallocata si riempie la table deve essere ricopiata in una zona di memoria più grande, il ché rallenta temporaneamente il programma. Per decidere come ingrandire la table sono forniti due tipi di resize:

- $tpe\_resize\_default$: è il tipo di resize della table se non è stato specificato diversamente. La table si espande automaticamente quando si cerca di inserire elementi oltre la sua capienza.
- $type\_resize\_manual$: le funzioni di inserimento tornano errore se la tabella è già piena e questa va espansa manualmente attraverso la funzione $resize\_table$.

Dimensione e tipo di resize della table possono essere modificati manualmente attraverso le funzioni $resize\_table$ e $change\_resize\_table$, o specificati in fase di allocazione (qualora la tabella non esistesse già) attraverso $malloc\_list\_specify\_table$.

Le 3 classi derivate per ciascun tipo sono:

- lista di elementi singoli
- lista di array
- lista generica

infatti queste tre categorie differiscono nel codice tra di loro ma racchiudono tutti i tipi di lista istanziabili (int, float, array_float, generic, ...).

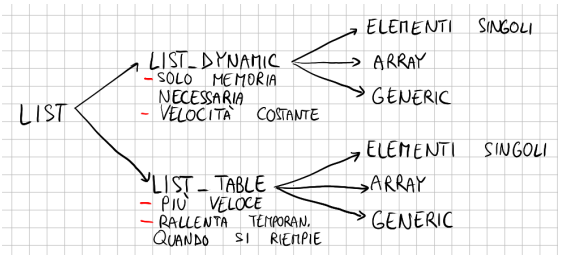


Figura 2: Schema dei tipi di lista. Tutte le liste sono comunque istanziate con la funzione $malloc\_list$ mentre il differenziamento tra tipi è interno.

2.1 Tipi di dati contenuti

I tipi che l'oggetto può contenere, stabiliti all'istanziamento di ciascuna lista, sono "CHAR", "INT", "LONG", "FLOAT", "DOUBLE", loro array e il tipo "GENERIC".
"GENERIC" è un tipo di dato con dimensione variabile individuato da un puntatore all'indirizzo di memoria in cui è contenuto e un intero unsigned in cui è salvata la sua dimensione, ottenibile ad esempio con la funzione standard di C $sizeof$.

Lavorare con array di dati permette di non dover traversare la lista un elemento alla volta, ma N alla volta, con N numero di elementi di ciascun array. All'istanziamento della lista, è fornita la possibilità di scegliere il numero di valori contenuti in ciascuno dei suoi elementi. Si noti che questo numero è fissato all'istanziamento della lista e deve essere lo stesso per tutti gli array in essa contenuti.

È importante sottolineare che quando un elemento viene inserito all'interno della lista viene creata una sua copia locale. Ciò permette di mantenere invariato l'elemento salvato nella lista pur modificando la variabile attraverso cui è stato inserito (presa ad esempio in input da $insert\_first$).

Poiché C non permette l'overloading delle funzioni, sarebbero richieste tante funzioni di inserimento ed estrazione quanti sono i tipi contenuti nella lista. Piuttosto si è scelto di utilizzare la union all_type, che contiene tutti i tipi di dato conservabili nell'oggetto lista.
Le funzioni di inserimento e di estrazione prendono in input elementi di tipo all_type, per cui è importante castare ad all_type tali variabili.
Si noti che:

- la union non contiene tutti i tipi di puntatore ma solo void*, per cui quando vogliamo fornire alla funzione un puntatore dobbiamo prima castarlo a void* e poi ad all_type;
- castare una variabile a union dà warning quando si compila con –pedantic; tuttavia si è scelto di mantenere questa scelta per la sua facilità di utilizzo.

Ad esempio nel caso di inserimento ed estrazioni di valori di tipo float, si avranno le seguenti funzioni:

```
insert_first(plista, (all_type) f, ...)
extract_first(plista, (all_type)((pvoid)&f), ...)
```

3 Manuale d'uso della lista

Di seguito sono riportate le funzioni membro fornite e come si utilizzano.

3.1 Istanziamento della lista

Esempi di allocazione di liste:

- list_dynamic che contiene int singoli:

```
plist=malloc_list(type_list_dynamic, "INT", 1);
```

- list_dynamic che contiene array di 5 float:

```
plist=malloc_list(type_list_dynamic, "FLOAT", 5);
```

- list_table che contiene generic:

```
plist=malloc_list(type_list_table, "GENERIC", ...);
```

- list_table, specificando le caratteristiche della tabella:

```
plist=malloc_list_specify_table("INT",
                                3,
                                type_resize_manual,
                                100);
```

a.
```
pvoid malloc_list(type_list type_list,
                  pchar type_string,
                  unsi dim_array);
```

crea nuove liste e ne restituisce l'indirizzo come void*. Può essere usata per creare **liste di tutti i tipi**, dipendentemente dai parametri inseriti in input. Gli argomenti sono rispettivamente:

- type_list: tipo di lista da istanziare, da scegliere tra $type\_list\_dynamic$ e $type\_list\_table$, con le differenze riportate in Sezione 2;
- type_string: è una stringa che corrisponde al tipo del dato da contenere. Deve essere una tra: "CHAR", "INT", "LONG", "FLOAT", "DOUBLE", "GENERIC";
- dim_array: numero di valore che contiene ciascun elemento della lista. Si inserisca 1 per liste con valori singoli e un numero maggiore di uno per liste che contengono array.
  Si noti che la dimensione scelta in fase di allocazione deve essere la stessa per tutti gli array inseriti successivamente nella lista;
- return: puntatore alla nuova lista, NULL altrimenti.

b.
```
pvoid malloc_list_specify_table(pchar type_string,
                                unsi dim_array,
                                type_resize type_resize,
                                unsi dim_table);
```

che alloca **esclusivamente** liste di tipo ***list_table*** e permette di specificare, nel caso in cui la tabella non fosse già esistente (per evitare errori con altre liste che vi apparterrebbero):

- type_resize: tipo di resize da impostare per la tabella. Va scelto tra $type\_resize_default$ e $type\_resize\_manual$, con le differenze riportate in Sezione 2;
- dim_table: numero massimo di elementi delle liste contenute nella tabella;
- return: puntatore alla nuova lista, NULL altrimenti.

Se si vuole cambiare il tipo di resize di una tabella già esistente si può usare $change\_resize\_table$, mentre per cambiarne la dimensione sono fornite $expand_table$, $shrink_table$ (vedi Sezione 3.2 per maggiore informazioni).

3.2 Modifica della tabella

Esempi di modifiche della tabella:

- cambio il tipo di resize:

```
change_resize(plist, type_resize_manual);
```

- ridimensiono la tabella se c'è abbastanza spazio:

```
unsigned* n_occupied;
get_info_table(plist,
               NULL, /* non ci interessa al momento */
               &n_occupied);
resize_table(plist, n_occupied + 10);
```

La precedente funzione $malloc\_list\_specify\_table$ agisce specifica solo le caratteristiche di tabelle non ancora create. Se ho precedentemente istanziato una lista con $malloc\_list$ e adesso voglio cambiare le caratteristiche della tabella in cui è contenuta, posso usare:

a.
```
int change_resize_table(pvoid plist,
                        type_resize type_resize);
```

cambia il tipo di resize della tabella che contiene plist.

- plist: lista contenuta nella tabella che si vuole modificare;
- type_resize: tipo di resize da impostare per la tabella che contiene plist. Va scelto tra $type\_resize\_default$ e $type\_resize\_manual$, con le differenze riportate in Sezione 2;
- return: 1 se modificato correttamente, 0 altrimenti.

b.
```
int resize_table(pvoid plist, unsi n_entries);
```

ridimensiona la tabella se possibile.

- plist: lista contenuta nella tabella che si vuole modificare;
- n_entries: numero di elementi complessivi della tabella, dopo che è stata ridimensionata;
- return: 1 se ridimensionata correttamente, 0 altrimenti, ad esempio se n_entries è minore del numero di elementi delle liste contenute.

c.
```
int get_info_table(pvoid plist,
                   punsi pn_entries,
                   punsi pn_occupied);
```

fornisce delle informazioni sulla tabella che contiene plist. Può essere usata ad esempio per sapere quanto la tabella può essere rimpicciolita.

- plist: lista contenuta nella tabella di cui si vogliono conoscere le informazioni;
- pn_entries: indirizzo in cui scrivere il numero di elementi complessivi della tabella;
- pn_occupied: indirizzo in cui scrivere il numero di elementi occupati della tabella;
- return: 1 se tutto va bene, 0 altrimenti

3.3 Inserimento in lista

Esempi di inserimento in lista:

- Inserimento in lista float:

```
float f=3.2;
insert_first(plist, (all_type)f,...)
```

- Inserimento in lista array di int:

```
int array[5]={1,2,3,4,5};
insert_first(plist, (all_type)((pvoid)array),...)
```

- Inserimento in lista generic:

```
insert_first(plist,
(all_type)((pvoid)&var),
sizeof(var));
```

Le funzioni di inserimento prendono in input variabili di tipo all_type, che è una union che contiene tutti i tipi di dato contenibili.
È importante castare le variabili ad all_type. Si noti che:

- la union non contiene tutti i tipi di puntatore ma solo void*, per cui quando vogliamo fornire alla funzione un puntatore dobbiamo prima castarlo a void* e poi ad all_type;
- castare una variabile a union dà warning quando si compila con –pedantic; tuttavia si è scelto di mantenere questa scelta per la sua facilità di utilizzo.

a. `int insert_first(pvoid plist, all_type value, unsi size);`

   inserisce un elemento in cima alla lista.

   - plist: indirizzo della lista al cui inizio inserire l'elemento
   - value: valore dell'elemento da inserire, da castare ad all_type
   - size: deve essere rispettivamente: - dato "GENERIC": dimensione del dato da inserire - altri: non ha importanza
   - return: 1 se tutto va bene, 0 altrimenti.

3.4 Estrazione dalla lista

Esempi di estrazione dalla lista:

- Estrazione da lista float:

```
float* pf;
extract_first(plist, (all_type)((pvoid)pf),...)
```

- Estrazione da lista array di int:

```
int* parray;
extract_first(plist, (all_type)((pvoid)&parray),...)
```

- Estrazione da lista generic:

```
void* pvar;
unsi  size;
extract_first(plist,
(all_type)(&pvar),
&size);
```

  dove in questo caso dobbiamo fornire un void* perché non sappiamo a priori la dimensione dell'elemento estratto quindi ci pensa il programma ad allocare la memoria necessaria e salvarla in pvar.

Le funzioni di estrazione prendono in input variabili di tipo all_type, che è una union che contiene tutti i tipi di dato contenibili.
È importante castare le variabili ad all_type. Si noti che:

- la union non contiene tutti i tipi di puntatore ma solo void*, per cui quando vogliamo fornire alla funzione un puntatore dobbiamo prima castarlo a void* e poi ad all_type;
- castare una variabile a union dà warning quando si compila con –pedantic; tuttavia si è scelto di mantenere questa scelta per la sua facilità di utilizzo.

a. `int extract_first(pvoid plist, all_type pvalue, punsi psize);`

   estrae un elemento dalla cima della lista.

   - plist: indirizzo della lista al cui inizio inserire l'elemento
   - pvalue: indirizzo in cui scrivere il valore dell'elemento estratto, da castare a pvoid se è un puntatore di tipo diverso e ad all_type
   - size: indirizzo in cui sarà scritta rispettivamente: - dato "GENERIC": dimensione del dato da inserire - altri: non ha importanza
   - return: 1 se tutto va bene, 0 altrimenti.

3.5 Altre funzioni utili

a. `int search_first(pvoid plist,`
`all_type value_searched, unsi size_searched,`

```
all_type pvalue_found,    punsi psize_found,
pcustom_compare pinput_compare);
```

trova la prima occorrenza dell'elemento cercato e la scrive in pvalue_found, in particolare:

- value_searched: valore con cui confrontare l'elemento da cercare, da castare ad all_type
- size_searched: dimensione dell'elemento da cercare, utile per cercare elementi di tipo generic
- pvalue_found: indirizzo in cui scrivere il valore dell'elemento trovato
- psize_found: indirizzo in cui scrivere la dimensione dell'elemento trovato (nel caso di liste con tipo di dato generic)
- pinput_compare: funzione del tipo

  ```
  int pcustom_compare(all_type value1, unsi size1,
                      all_type value2, unsi size2);
  ```

  con cui ciascun elemento della lista viene confrontato (come value1) con l'elemento da cercare (come value2). La funzione deve tornare 0 quando i due elementi sono uguali, e in quel caso l'elemento trovato viene scritto in pvalue_found e psize_found.
- return: 1 se lo ha trovato, 0 altrimenti

b. `int print_list(pvoid plist, pcustom_print pinput_print);`

stampa informazioni sulla lista nel formato:

```
type_list: ...
type_data: ...
Numero di elementi: ...
```

e i primi 5 elementi della lista se è fornita pinput_print. In particolare:

- plist: lista da stampare
- pinput_print: funzione del tipo

  ```
  int pcustom_print(all_type value, unsi size);
  ```

  con cui stampare un elemento della lista. Deve tornare 1 se tutto va bene, 0 altrimenti.
- return: 1 se tutto va bene, 0 altrimenti

4 Manuale tecnico

4.1 Tipi di lista

Abbiamo visto in Sezione 2 che l'oggetto consiste di 6 classi derivate. In realtà le classi sono molto più di queste, e in particolare esiste una classe derivata per ogni tipo di dato contenibile.

La creazione di queste classi è realizzata in modo programmatico da uno script Bash a partire dalle classi: $list\_dynamic\_BASETYPE$, $list\_dynamic\_array\_BASETYPE$, $list\_dynamic\_generic$, $list\_table\_BASETYPE$, $list\_table\_array\_BASETYPE$, $list\_table\_generic$.
Lo script ricopia le classi con $BASETYPE$ tante volte quanti sono i tipi base da contenere (char, int, float, ...) e sostituisce nel codice il termine BASETYPE con il nome del tipo base e MEMBERTYPE con il nome del rispettivo membro della union. Ad esempio creerà la classe $list\_dynamic\_char$ con "char" al posto di BASETYPE e "c" al posto di MEMBERTYPE.

Per questioni di velocità si è scelto di evitare chiamate di funzioni a cascata, ma per ciascuna funzione esiste un array di puntatori a funzioni che collega il tipo di lista alla rispettiva funzione nella classe derivata.

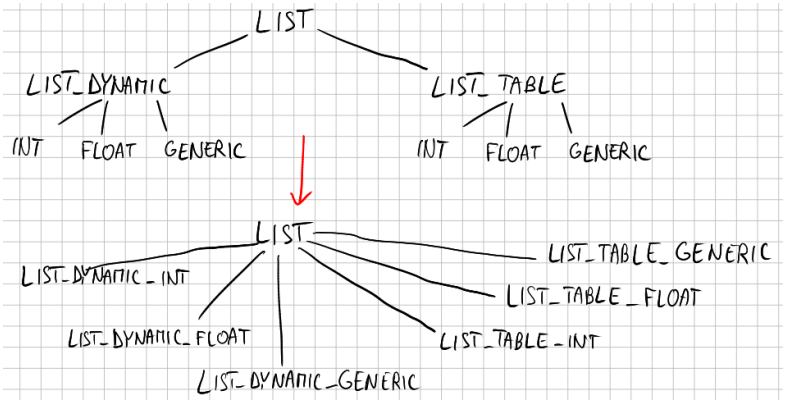


Figura 3: Schema di chiamata delle funzioni. Si è scelto il metodo in basso per risparmiare una chiamata a funzione. Ciò implica avere array di funzioni più lunghi ma sempre dell'ordine della decina di funzioni, il ché non dovrebbe essere un problema nella maggior parte dei casi.

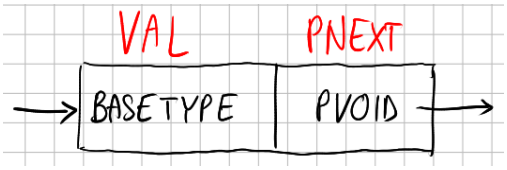
4.2 Lista dinamica

Illustrerò di seguito la struttura della lista e come sono programmate le funzioni base. Siccome le classi derivate sono molto simili tra di loro, prenderò come esempio la classe $list\_dynamic\_BASETYPE$, sottolineando le differenze con le altre classi quando necessario.

4.2.1 Struttura

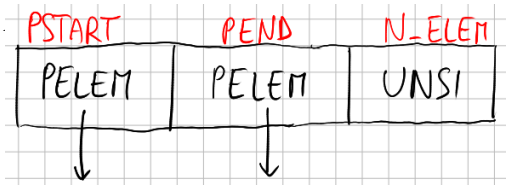
La struct $elem\_BASETYPE$ (o $elem\_generic$ in modo molto simile):

```
typedef struct _elem_BASETYPE {
  BASETYPE val;
  struct _elem_BASETYPE *pnext;
} elem_BASETYPE;
```



sono i "mattoncini" di cui è fatta la lista. Ciascuno contiene un valore di tipo BASETYPE e il puntatore all'elemento successivo.
La struct $list\_dynamic\_BASETYPE$:

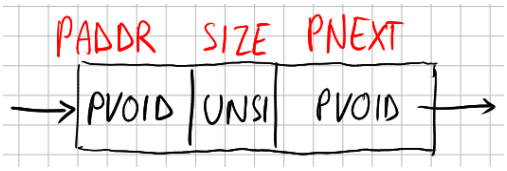
```
typedef struct _list_dynamic_BASETYPE
  pelem_BASETYPE pstart;
  pelem_BASETYPE pend;
  unsi n_elem;
} list_dynamic_BASETYPE;
```



contiene le informazioni sulla lista vera e propria, cioè l'indirizzo del primo e dell'ultimo elemento e il numero di elementi contenuti.

Nel caso di $elem\_generic$ cambia leggermente la struct:

```
typedef struct _elem_generic {
  pvoid paddr;
  unsi size;
  struct _elem_generic *pnext;
} elem_generic;
```



ma il principio è del tutto analogo.

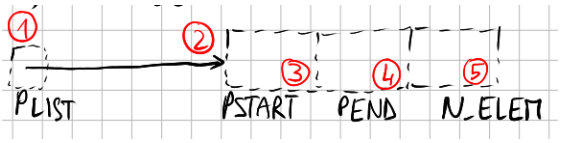
Nel caso di $elem\_array\_BASETYPE$ l'implementazione è più complicata perché gli elementi hanno size variabile in base alla dimensione degli array contenuti, quindi non possiamo usare le struct di C. Invece la memoria è gestita manualmente, e invece di allocare memoria pari al $sizeof(elem\_BASETYPE)$ allochiamo $size\_array * sizeof(BASETYPE) + sizeof(pvoid)$.
$elem\_BASETYPE->pnext$ è sostituito invece dalla macro GET_PNEXT:

```
#define GET_PNEXT(pvalue) \
    (*((ppvoid)(((pBASETYPE) pvalue )+size_array)))
```

che partendo dall'inizio dell'elemento ($pvalue$) si sposta di $sizeof(BASETYPE)*size\_array$ e legge il pezzo di memoria successivo come void*.

4.2.2 malloc_list

L'allocazione di questo tipo di lista è molto semplice; basta avere memoria sufficiente per la sua struct:



1 creo pvoid plist

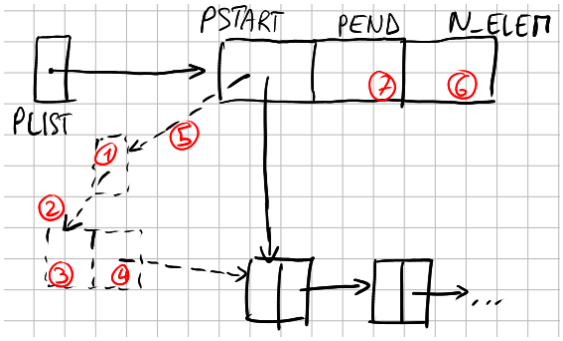
2 plist = malloc(sizeof(list_dynamic_BASETYPE)

3 pstart = NULL

4 pend = NULL

5 n_elem = 0

4.2.3 insert_first



1 creo buco temporaneo pvoid ptmp

2 ptmp = malloc(sizeof(elem_dynamic_BASETYPE)

3 scrivo il valore preso in input in ptmp- val

4 ptmp- pnext = plist- pstart

5 plist- pstart = ptmp

6 n_elem++

7 if(n_elem == 1) plist− >pfine = ptmp