Programmazione 1

Lezione 10

Vincenzo Marra

vincenzo.marra@unimi.it

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

17 maggio 2017

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

[1] Il momento in cui ciò accade dipende dal campo di visibilità della variabile.

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

1. Quando il programma dichiara una variabile di tipo `T`, il sistema assegna (o alloca) al programma memoria centrale della macchina in quantità pari al numero N di byte necessari a rappresentare un dato di tipo `T`.

[1] Il momento in cui ciò accade dipende dal campo di visibilità della variabile.

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

1. Quando il programma dichiara una variabile di tipo `T`, il sistema assegna (o alloca) al programma memoria centrale della macchina in quantità pari al numero N di byte necessari a rappresentare un dato di tipo `T`.
2. Il numero N è costante: esso *non cambia* dal momento in cui la variabile è dichiarata, fino al momento in cui essa cessa di esistere.[1]

[1] Il momento in cui ciò accade dipende dal campo di visibilità della variabile.

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

1. Quando il programma dichiara una variabile di tipo `T`, il sistema assegna (o alloca) al programma memoria centrale della macchina in quantità pari al numero N di byte necessari a rappresentare un dato di tipo `T`.
2. Il numero N è costante: esso *non cambia* dal momento in cui la variabile è dichiarata, fino al momento in cui essa cessa di esistere.[1]
3. Non appena la variabile cessa di esistere, gli N byte di memoria sono automaticamente liberati, o deallocati, e ritornano nella disponibilità del sistema operativo.

[1] Il momento in cui ciò accade dipende dal campo di visibilità della variabile.

Allocazione statica *vs.* allocazione dinamica

Il C permette però anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

Allocazione statica *vs.* allocazione dinamica

Il C permette però anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.

Allocazione statica *vs.* allocazione dinamica

Il C permette però anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.
2. Se la richiesta può essere soddisfatta, il sistema assegna N byte al programma, che può usarli per memorizzare dati.

Allocazione statica *vs.* allocazione dinamica

Il C permette però anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.
2. Se la richiesta può essere soddisfatta, il sistema assegna N byte al programma, che può usarli per memorizzare dati.
3. Durante la sua esecuzione, il programma può anche chiedere che gli N byte ad esso riservati siano aumentati o diminuiti: si parla di riallocazione (dinamica) della memoria.

Allocazione statica *vs.* allocazione dinamica

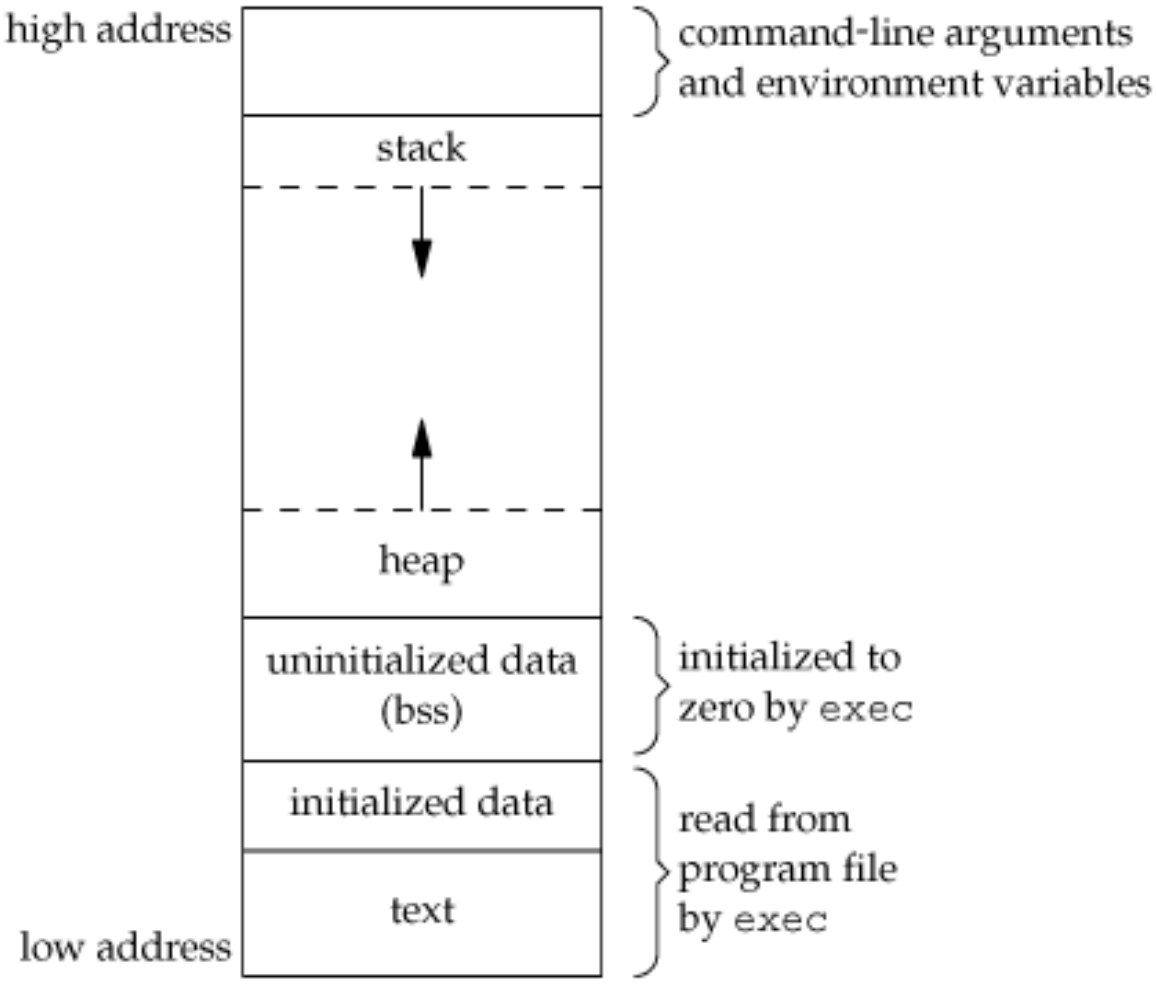
Il C permette però anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.
2. Se la richiesta può essere soddisfatta, il sistema assegna N byte al programma, che può usarli per memorizzare dati.
3. Durante la sua esecuzione, il programma può anche chiedere che gli N byte ad esso riservati siano aumentati o diminuiti: si parla di riallocazione (dinamica) della memoria.
4. Quando il programma non necessità più degli N byte, comunica al sistema che essi possono essere deallocati.

Cenno al *layout* della memoria

Abbiamo già visto che le funzioni di un programma (inclusa la funzione `main`) hanno la loro memoria locale privata, ove risiedono, ad esempio, le loro variabili locali (anche dette automatiche). D'altro canto, le variabile globali del programma, accessibili da tutte le funzioni, risiedono nella memoria globale del programma. Lo spazio allocato dalla funzione `malloc` (o `calloc` ecc.) risiede nella memoria globale del programma, ed è quindi accessibile da tutte le funzioni che costituiscono il programma. In particolare:

Nota Bene. La memoria allocata dinamicamente non è mai automaticamente liberata, durante l'esecuzione del programma: essa è liberata solo quando il programmatore ne fa richiesta. Quindi è di fondamentale importanza evitare che il codice conduca all'allocazione di blocchi di memoria non più utilizzabili perché non più raggiungibili da un puntatore, ma tuttavia non ancora liberati (*memory leakage*).



Schema di layout della memoria di un programma C

Il file di intestazione `stdlib.h`

Il file d'intestazione `stdlib.h` contiene, *inter alia*, le dichiarazioni delle funzioni dedicate alla gestione dinamica della memoria.

Il file di intestazione `stdlib.h`

Il file d'intestazione `stdlib.h` contiene, *inter alia*, le dichiarazioni delle funzioni dedicate alla gestione dinamica della memoria.

- `void *malloc(size_t N)`
- `void *calloc(size_t n, size_t dim)`
- `void realloc(void *ptr, size_t M)`
- `void free(void *ptr)`

Il file di intestazione `stdlib.h`

Il file d'intestazione `stdlib.h` contiene, *inter alia*, le dichiarazioni delle funzioni dedicate alla gestione dinamica della memoria.

- `void *malloc(size_t N)`
- `void *calloc(size_t n, size_t dim)`
- `void realloc(void *ptr, size_t M)`
- `void free(void *ptr)`

Prima di parlare di queste funzioni discutiamo i due nuovi tipi che in esse figurano:

```
size_t
void *
```

Il tipo `size_t` e l'operatore `sizeof`

Il tipo `size_t` è definito nel file di intestazione `stddef.h`, che è già incluso da `stdlib.h`. I dati di tipo `size_t` rappresentano la dimensione in byte di oggetti che risiedono in memoria centrale. Ne segue che i valori rappresentati da questo tipo sono interi non negativi, o come anche si dice in programmazione, interi privi di segno (*unsigned integers*).

Il tipo `size_t` e l'operatore `sizeof`

Il tipo `size_t` è definito nel file di intestazione `stddef.h`, che è già incluso da `stdlib.h`. I dati di tipo `size_t` rappresentano la dimensione in byte di oggetti che risiedono in memoria centrale. Ne segue che i valori rappresentati da questo tipo sono interi non negativi, o come anche si dice in programmazione, interi privi di segno (*unsigned integers*).

Il linguaggio C comprende l'operatore unario:

`sizeof`

la cui valutazione risulta pari alla dimensione del suo argomento espressa in byte e rappresentata come valore del tipo `size_t`. Se l'argomento è il nome di un tipo racchiuso fra parentesi, il valore in questione è il numero di byte necessario a memorizzare un dato di quel tipo. Si noti che ciò vale sia per i tipi primitivi che per i tipi definiti, incluse le strutture.

Per esempio,

```
size_t N = sizeof (double);
```

dichiara una variabile `N` di tipo `size_t`, e le assegna il numero di byte necessari a memorizzare un `double`. Questo valore dipende dalla specifica implementazione del compilatore C.

Si supponga invece di aver dichiarato, per esempio:

```
1 struct punto //Etichetta (opzionale) della struttura
2 {
3         double x; //Primo membro
4         double y; //Secondo membro
5 };
```

Allora la riga:

```
7 struct punto q = {-0.1, 2}; //Inizializzazione con lista
```

assegna ad `N` il numero di byte necessary a memorizzare un valore del tipo di dato strutturato non primitivo: `struct punto`.

sizeof.c

```c
#include <stdio.h>
int main(void)
{
        // typedef rinonima un tipo: typedef Tipo NuovoNome;
        // Cfr. Laboratorio.
        typedef struct
                {
                        int a;
                        double b;
                } esempio;

        // %lu per stampare unsigned long restituito da sizeof
        printf("Dim. tipo esempio: %lu\n", sizeof (esempio));
        printf("Dim. tipo int: %lu\n", sizeof (int));
        double d;
        printf("Dim. tipo double: %lu\n", sizeof d);
        return 0;
}
```

Uso di `sizeof`.

Il tipo `void *`

Il tipo "puntatore a `void`", ossia:

```
void *
```

è usato per un puntatore che fa riferimento a una zona di memoria contenente dati il cui tipo non è specificato. Esso è stato introdotto per risolvere il problema che le funzioni usate per allocare zone di memoria in realtà non sanno quali tipi di dati saranno memorizzati in una data zona di memoria. Ne segue che non sanno quale tipo di puntatore restituire al chiamante.

Per esempio, potremmo voler allocare memoria per un `int`:

```
malloc( sizeof (int) );
```

oppure per un `double`:

```
malloc( sizeof (double) );
```

Cosa deve restituire `malloc`? Il tipo `int *`, oppure il tipo `double *`?

Per risolvere questo problema, le versioni moderne del C permettono di usare la sintassi seguente:

```
int *pt1 = malloc( sizeof (int) );
double *pt2 = malloc( sizeof (double) );
```

Queste istruzioni chiedono di allocare memoria sufficiente per un `int` e un `double`, rispettivamente. Se la chiamate vanno a buon fine, `pt1` puntano alle zone di memoria allocate a questi fini, rispettivamente.

Per risolvere questo problema, le versioni moderne del C permettono di usare la sintassi seguente:

```
int *pt1 = malloc( sizeof (int) );
double *pt2 = malloc( sizeof (double) );
```

Queste istruzioni chiedono di allocare memoria sufficiente per un `int` e un `double`, rispettivamente. Se la chiamate vanno a buon fine, `pt1` puntano alle zone di memoria allocate a questi fini, rispettivamente.

La variabile `pt1` è un puntatore a `int`, ma la funzione `malloc` restituisce `void *`. La conversione dall'uno all'altro tipo è eseguita automaticamente. Similmente per la variabile `pt2`, che è un puntatore a `double`.

Per risolvere questo problema, le versioni moderne del C permettono di usare la sintassi seguente:

```
int *pt1 = malloc( sizeof (int) );
double *pt2 = malloc( sizeof (double) );
```

Queste istruzioni chiedono di allocare memoria sufficiente per un `int` e un `double`, rispettivamente. Se la chiamate vanno a buon fine, `pt1` puntano alle zone di memoria allocate a questi fini, rispettivamente.

La variabile `pt1` è un puntatore a `int`, ma la funzione `malloc` restituisce `void *`. La conversione dall'uno all'altro tipo è eseguita automaticamente. Similmente per la variabile `pt2`, che è un puntatore a `double`.

Il tipo `void *` ha qualche altro uso, sebbene quanto appena visto sia il suo scopo primario. Non ne parleremo oltre.

```
void *malloc(size_t N)
```

```
void *malloc(size_t N)
```

1. Restituisce un puntatore a una zona di memoria di `N` byte (non necessariamente fisicamente contigui) nella memoria centrale, o `NULL` se la richiesta non può essere esaudita.

`void *malloc(size_t N)`

1. Restituisce un puntatore a una zona di memoria di `N` byte (non necessariamente fisicamente contigui) nella memoria centrale, o `NULL` se la richiesta non può essere esaudita.
2. È un errore grave non testare il valore restituito da `malloc`. Uso corretto:

```
1 int *pt = malloc(sizeof (int));
2 if (pt != NULL)
3         printf("Allocazione riuscita\n!");
4 else
5         printf("Allocazione fallita\n!");
```

`void *malloc(size_t N)`

1. Restituisce un puntatore a una zona di memoria di `N` byte (non necessariamente fisicamente contigui) nella memoria centrale, o `NULL` se la richiesta non può essere esaudita.
2. È un errore grave non testare il valore restituito da `malloc`. Uso corretto:

```
1  int *pt = malloc(sizeof (int));
2  if (pt != NULL)
3          printf("Allocazione riuscita\n!");
4  else
5          printf("Allocazione fallita\n!");
```

3. È pure un errore grave, in questo esempio, usare una costante nell'argomento di `malloc`, per quanto plausibile possa apparire la costante — per esempio, 4: infatti, la dimensione dei dati di tipo `int` è dipendente dall'implementazione.

```
void *calloc(size_t n, size_t dim)
```

```
void *calloc(size_t n, size_t dim)
```

1. Come `malloc`, ma alloca un totale di `N=n*dim` byte e li inizializza tutti a zero. Restituisce `NULL` se non è possibile esaudire la richiesta.

`void *calloc(size_t n, size_t dim)`

1. Come `malloc`, ma alloca un totale di `N=n*dim` byte e li inizializza tutti a zero. Restituisce `NULL` se non è possibile esaudire la richiesta.
2. Quindi, per esempio,

```
calloc(10, sizeof (int))
```

e

```
malloc(sizeof (int[10]))
```

sono equivalenti, eccetto che la prima chiamata inizializza la memoria.

```
void *realloc(void *ptr, size_t M)
```

```
void *realloc(void *ptr, size_t M)
```

- Restituisce un puntatore a una zona di memoria di `M` byte che estende o riduce la zona di memoria di `N` byte puntata da `ptr`. Il puntatore `ptr` deve essere stato restituito da una precedente chiamata a `malloc`, `calloc` o `realloc`. I dati presenti nella regione di `N` byte sono preservati, fino alla lunghezza di `M` byte. Se `M>N`, i nuovi `M-N` byte allocati non sono inizializzati. Se `N>M`, i rimanenti `N-M` byte precedentemente allocati vanno persi. Le chiamate con `M` pari a `0` hanno effetto indefinito. Se `ptr` è `NULL`, equivale a `malloc(M)`.

`void *realloc(void *ptr, size_t M)`

- Restituisce un puntatore a una zona di memoria di `M` byte che estende o riduce la zona di memoria di `N` byte puntata da `ptr`. Il puntatore `ptr` deve essere stato restituito da una precedente chiamata a `malloc`, `calloc` o `realloc`. I dati presenti nella regione di `N` byte sono preservati, fino alla lunghezza di `M` byte. Se `M>N`, i nuovi `M-N` byte allocati non sono inizializzati. Se `N>M`, i rimanenti `N-M` byte precedentemente allocati vanno persi. Le chiamate con `M` pari a `0` hanno effetto indefinito. Se `ptr` è `NULL`, equivale a `malloc(M)`.
- Restituisce il puntatore alla zona di memoria opportunamente estesa o ridotta, oppure `NULL` se non può esaudire la richiesta. In questo caso, `ptr` e l'allocazione precedente sono invariate.

`void *realloc(void *ptr, size_t M)`

- Restituisce un puntatore a una zona di memoria di `M` byte che estende o riduce la zona di memoria di `N` byte puntata da `ptr`. Il puntatore `ptr` deve essere stato restituito da una precedente chiamata a `malloc`, `calloc` o `realloc`. I dati presenti nella regione di `N` byte sono preservati, fino alla lunghezza di `M` byte. Se `M>N`, i nuovi `M-N` byte allocati non sono inizializzati. Se `N>M`, i rimanenti `N-M` byte precedentemente allocati vanno persi. Le chiamate con `M` pari a `0` hanno effetto indefinito. Se `ptr` è `NULL`, equivale a `malloc(M)`.
- Restituisce il puntatore alla zona di memoria opportunamente estesa o ridotta, oppure `NULL` se non può esaudire la richiesta. In questo caso, `ptr` e l'allocazione precedente sono invariate.
- Nota. Il ridimensionamento della zona originariamente puntata da `ptr` può comportare uno spostamento: il puntatore restituito può avere valore diverso da `ptr`.

```
void free(void *ptr)
```

```
void free(void *ptr)
```

- Dealloca lo spazio di memoria precedentemente allocato tramite `malloc`, `calloc` o `realloc`, cui punta `ptr`. Se l'argomento è `NULL` non ha alcun effetto.

`void free(void *ptr)`

- Dealloca lo spazio di memoria precedentemente allocato tramite `malloc`, `calloc` o `realloc`, cui punta `ptr`. Se l'argomento è `NULL` non ha alcun effetto.
- L'argomento deve essere un puntatore che è stato precedentemente restituito da una chiamata a `malloc`, `calloc` o `realloc`.

`void free(void *ptr)`

- Dealloca lo spazio di memoria precedentemente allocato tramite `malloc`, `calloc` o `realloc`, cui punta `ptr`. Se l'argomento è `NULL` non ha alcun effetto.
- L'argomento deve essere un puntatore che è stato precedentemente restituito da una chiamata a `malloc`, `calloc` o `realloc`.
- Nota. L'uso scorretto delle funzioni di allocazione e deallocazione della memoria può portare al cosiddetto `memory leakage`, ossia *perdita* o *fuoriuscita di memoria*. Il caso più ovvio si ha quando si alloca memoria, si sovrascrive per errore il puntatore `ptr` restituito dalla funzione di allocazione, e quindi non si ha più la possibilità di fare riferimento alla zona di memoria allocata — né, in particolare, di deallocarla. Si tratta di errori di programmazione gravi.

ex1.c

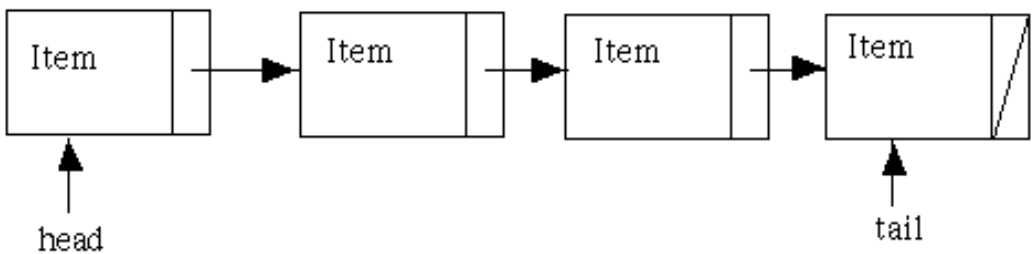
```c
#include<stdio.h>
#include<stdlib.h> //Per malloc, realloc e free
int main(int argc, char *argv[])
{
        if (argc==1)
        {
                printf("Inserisci almeno un numero intero come parametro.\n"); return -1;
        }

        int *pt = calloc(argc-1, sizeof (int));
        if (pt == NULL)
        {
                printf("Errore nell'allocazione della memoria.\n");
                return -1;
        }
        for (int i=0; i<argc-1; i++)
                pt[i]=atoi(argv[i+1]);
        printf("Inserisci un altro intero.\n");
        int *tmp = realloc(pt, (sizeof (int))*argc); //ridimensionamento
        if ( tmp != NULL)
        {
                pt = tmp;
                scanf("%d", pt+argc-1); getchar();
        }
        else
        {
                printf("Errore nell'allocazione della memoria.\n"); return -1;
        }
        printf("Ecco:\n");
        for (int i=0; i<argc; i++)
                printf("%d\n",pt[i]);
        free(pt);
        printf("Ho deallocato la memoria. Addio.\n");
        return 0;
}
```

Esempio: Qualche cenno alle liste

1. Una lista di tipo T è una struttura di dati ad allocazione tipicamente dinamica, costituita da una successione finita di elementi, anche detti nodi, ciascuno dei quali contiene un dato di tipo T e un puntatore al prossimo nodo della lista.

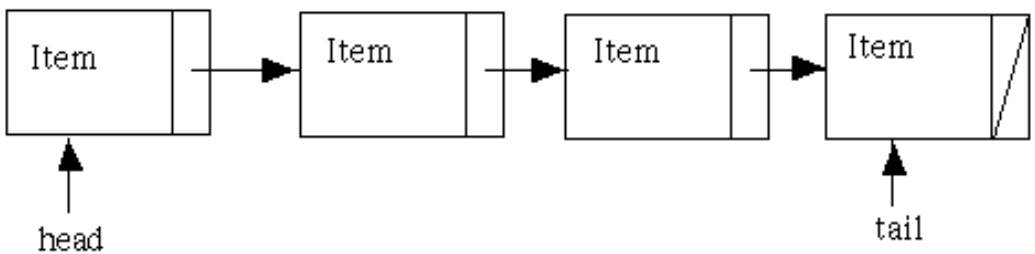
Esempio: Qualche cenno alle liste

1. Una lista di tipo T è una struttura di dati ad allocazione tipicamente dinamica, costituita da una successione finita di elementi, anche detti nodi, ciascuno dei quali contiente un dato di tipo T e un puntatore al prossimo nodo della lista.
2. Il primo nodo della lista è detto testa della lista, l'ultimo; coda.



Esempio: Qualche cenno alle liste

1. Una lista di tipo T è una struttura di dati ad allocazione tipicamente dinamica, costituita da una successione finita di elementi, anche detti nodi, ciascuno dei quali contiente un dato di tipo T e un puntatore al prossimo nodo della lista.
2. Il primo nodo della lista è detto testa della lista, l'ultimo; coda.



3. Vediamo ora come si possono implementare le liste in C. Usiamo liste di reali. La trattazione è minimale, e sarà sviluppata un po' in laboratorio se il tempo a disposizione lo permetterà.

```c
/* Implementazione di una lista di double */

#include <stdio.h>
#include <stdlib.h>

typedef struct elem {
        double d;
        struct elem *next;
} elem;
```

Struttura dati: lista di `double`.

Come già accennato, l'istruzione

```
typedef Tipo NuovoNome
```

rinomina il tipo `Tipo` col nuovo nome `NuovoNome`. Il vecchio nome `Tipo` rimane disponibile. Il riferimento ricorsivo a `struct elem` all'interno della definizione di `struct elem` è ammesso.

```
75  void list_print(elem *list)
76  {
77          elem *curr=list;
78          while(curr->next!=NULL) {
79                  printf("%g -> ",curr->d);
80                  curr=curr->next;
81          }
82          printf("%g\n",curr->d);
83  }
```

Stampare la lista.

La fine della lista è segnalata da un nodo il cui campo `next` valga `NULL`. Le funzioni che elaborano la lista, come in questo esempio della stampa, ricevono in ingresso la testa della lista, e scandiscono la lista fino al punto richiesto dall'elaborazione specifica. Si noti che, a differenza degli array, le liste non ammettono accesso diretto all'elemento in posizione i-esima.

```
60  int list_append(elem *list, double d)
61  {
62          if (list==NULL) return -1;
63
64          elem *p, *new;
65          if ( ( new=malloc(sizeof(elem)) ) == NULL )
66                  return 0;
67          new->d=d; //Assegna valore al nodo
68          new->next=NULL; //Nuovo nodo sara' coda della lista
69          //Scandisce fino alla vecchia coda...
70          for (p=list; (p->next)!=NULL; p=p->next);
71          p->next=new; //...e linka nuova coda alla lista
72          return 1;
73  }
```

Accodare nodi alla lista.

Si noti però che questa funzione non è in grado di accodare un nodo alla lista vuota. Ossia, se `list` è `NULL` la funzione non fa nulla. Serve una funzione separata di inizializzazione della lista vuota.

```
85  elem *list_init(double d)
86  {
87          elem *new=malloc(sizeof(elem));
88          if (new==NULL)
89                  return NULL;
90          new->d=d;
91          new->next=NULL;
92          return new;
93  }
```

Inizializzare la lista.

Per finire la lezione vedremo adesso il dettaglio il codice completo al computer.