19. Puntatori, array dinamici e liste

Andrea Marongiu
(andrea.marongiu@unimore.it)
Paolo Valente

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Da oggetti statici a dinamici

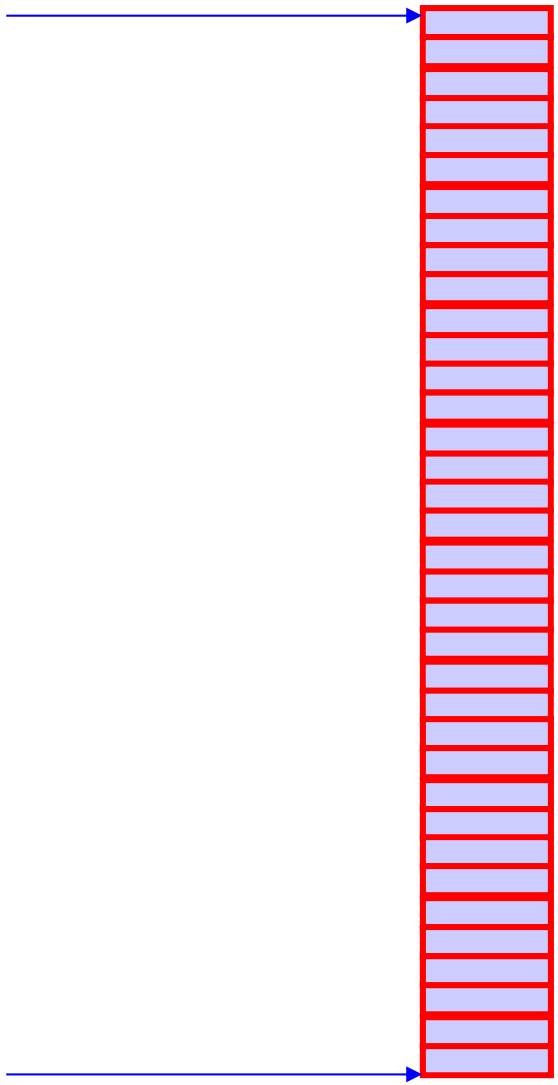
- La dimensione di tutti gli oggetti (concreti) considerati finora deve essere definita a tempo di scrittura del programma
- Talvolta però non è possibile sapere a priori la quantità di dati da memorizzare/gestire
- Per superare la rigidità della definizione statica delle dimensioni, occorre un meccanismo per allocare in memoria oggetti le cui **dimensioni sono determinate durante l'esecuzione del programma**
- Questo è esattamente quello che si riesce a fare mediante il meccanismo di allocazione dinamica della memoria

Memoria dinamica 1/2

- Prima dell'inizio dell'esecuzione di un processo, il sistema operativo riserva al processo un spazio di memoria di dimensioni predefinite
  - sono locazioni consecutive di memoria (tipicamente da un byte l'una)
  - è quello che finora abbiamo chiamato memoria del programma
- Questo spazio di memoria è a sua volta organizzato in segmenti (zone contigue di memoria) distinti
- Uno di questi segmenti è chiamato con vari nomi equivalenti:
  - memoria libera, memoria dinamica oppure heap
- E' possibile allocare oggetti di **dimensione arbitraria** all'interno della memoria dinamica in **momenti arbitrari** dell'esecuzione del programma
  - ovviamente finché lo spazio non si esaurisce

Memoria dinamica 2/2

Indirizzo prima locazione della memoria libera, fisso, ad es.: 0xFFFF0000



Memoria dinamica

Indirizzo ultima locazione della memoria libera, mobile (come vedremo in seguito)

Oggetti dinamici

- Gli oggetti allocati in memoria dinamica sono detti **dinamici**
- In questa presentazione considereremo solo
- **array dinamici**
  - Array allocati in memoria dinamica durante l'esecuzione del programma
  - Come vedremo, il numero di elementi di tali array non è vincolato ad essere definito a tempo di scrittura del programma

Operatore new

- Un array dinamico può essere allocato mediante l'operatore **new**

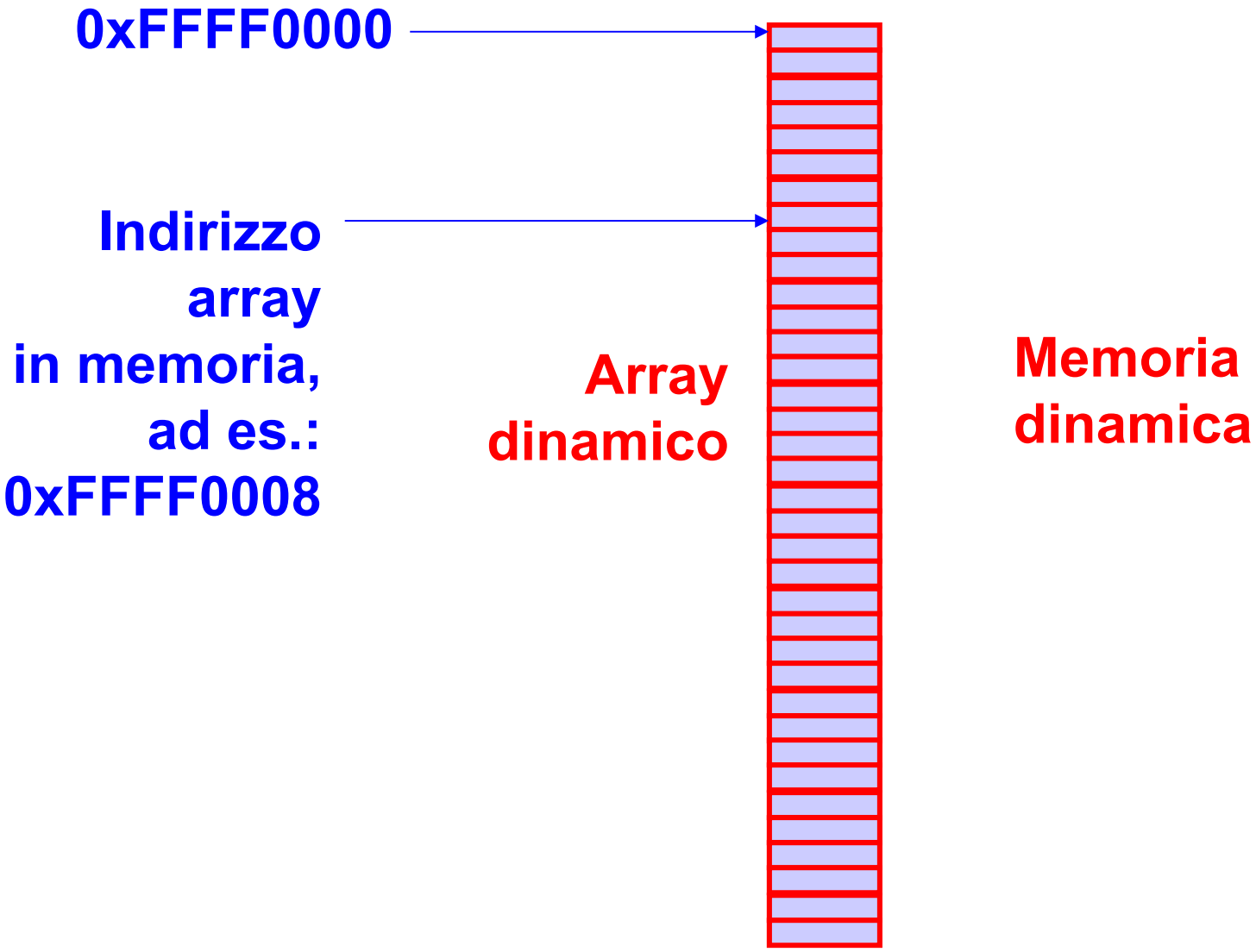
new <*nome_tipo*> **[**<*num_elementi*>**] ;**

Alloca un array di <*num_elementi*> oggetti di tipo <*nome_tipo*>, non inizializzati (valori casuali)
<*num_elementi*> può essere un'**espressione aritmetica qualsiasi**

Ad esempio:

```
int a ; do cin>>a ; while (a <= 0) ;
new int[a] ; // alloca un array dinamico
             // di a elementi
```

Allocazione nello heap



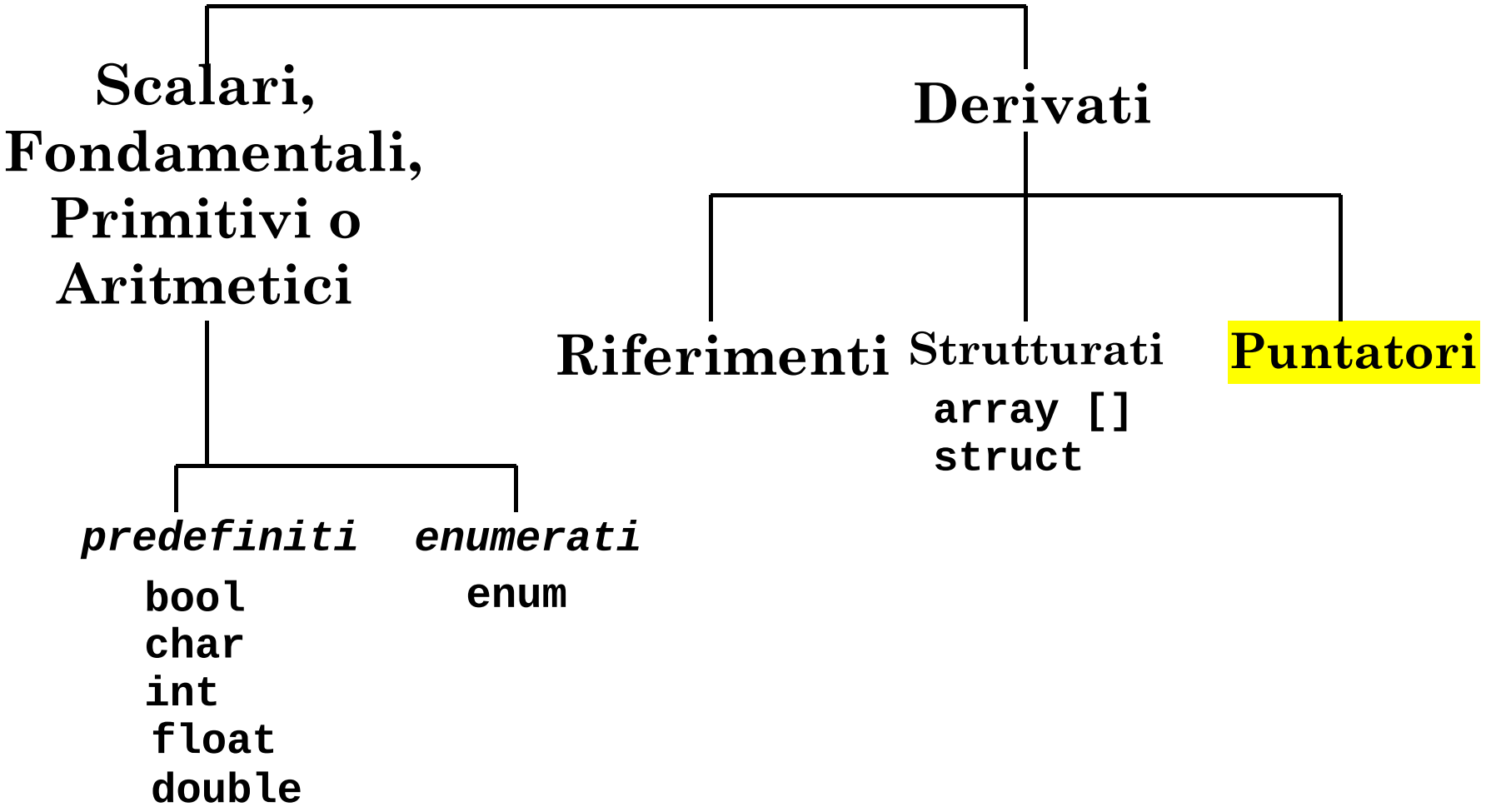
Oggetti senza nome

- L'operatore **new** può essere utilizzato per allocare oggetti dinamici di ogni tipo, ma per ora noi vedremo solo il caso degli array dinamici
- Gli elementi dell'array dinamico hanno **valori casuali**
- L'operatore **new** non ritorna un riferimento (nel senso di nome o sinonimo) all'oggetto allocato
- Gli oggetti dinamici sono **oggetti senza nome**
- In generale, quale informazione ci serve per accedere ad un oggetto?
  - **Il suo indirizzo**

Ritorno operatore new

- L'operatore **new** ritorna proprio l'**indirizzo dell'oggetto allocato**
- Possiamo quindi accedere all'oggetto tramite tale indirizzo
- Ma per farlo dobbiamo prima memorizzare tale indirizzo da qualche parte
- Dove lo memorizziamo?
  - Ci serve un oggetto di tipo puntatore

Tipi di dato


Scalari, Fondamentali, Primitivi o Aritmetici
Derivati
Riferimenti
Strutturati
array []
struct
Puntatori
predefiniti
bool
char
int
float
double
enumerati
enum

Puntatori

- Un oggetto di tipo puntatore ha per valore un indirizzo di memoria (che non è altro che un numero naturale)
- Le definizione di un oggetto puntatore ha la seguente forma

[**const**] <*tipo_oggetto_puntato*>
    * [**const**] <*identificatore*> [= <*indirizzo*>] **;**

- Il primo qualificatore **const** va inserito se il puntatore deve puntare ad un oggetto non modificabile
- Il secondo **const** va inserito se il valore del puntatore, una volta inizializzato, non deve più essere modificato
- Per definire un puntatore inizializzato con l'indirizzo di un array dinamico di *N* elementi di tipo **int**:

```
int * p = new int [N] ;
```

Indirizzo array dinamico

Accesso agli elementi

- Accesso agli elementi di un array dinamico
  - Possibile in modo **identico** agli array statici
    - selezione con indice mediante parentesi quadre
    - gli indici partono da 0

Proviamo ...

- Scrivere un programma che
  - Allochi un array dinamico di interi, di dimensioni lette da *stdin*
  - Lo inizializzi (con i valori che preferite)
  - Lo stampi
- Soluzione: parte dell'esempio seguente ...

Esempio accesso agli elementi

```
main()
{
  int N ;
  cin>>N ;

  int * p = new int [N] ;

  for (int i = 0 ; i < N ; i++)
     p[i] = i ; // inizializzazione

  cout<<p[0]<<endl ; // stampo primo elemento

  cin>>p[N] ; // Esempio di: ERRORE LOGICO
              // e DI ACCESSO ALLA MEMORIA
}
```

Puntatore ad array costante

```
main()
{
  int N ;
  cin>>N ;

  int * p = new int [N] ;

  int * q = p ; // q punta allo stesso array

  const int * r = q ; // r punta allo stesso array,
                      // ma tramite r non lo si potrà
                      // modificare

  cin>>q[0] ; // corretto

  cin>>r[0] ; // errore segnalato a tempo di
              // compilazione: non si può utilizzare
              // r per cambiare valore all'array
}
```

Puntatore costante

```
main()
{
  int N ;
  cin>>N ;

  int *p = new int [N] ;

  int * const s = p ; // s punta allo stesso array
                      // e non potrà cambiare valore

  p = new int [N] ; // d'ora in poi p punta ad un
                    // diverso array, l'unico
                    // riferimento al precedente è
                    // rimasto s

  s = p ; // ERRORE: s non può cambiare valore
}
```

Valori, operazioni, tempo di vita

- Un oggetto di tipo puntatore
  - Ha per valori un sottoinsieme dei numeri naturali
    - un puntatore che contenga 0 (NULL in C) viene detto **puntatore nullo**
  - Prevede operazioni correlate al tipo di oggetto a cui punta
    - A PARTE L'ASSEGNAMENTO E L'ACCESSO AGLI ELEMENTI DI UN ARRAY DINAMICO, NON VEDREMO ALTRE OPERAZIONI CON I PUNTATORI
  - Segue le stesse regole di tempo di vita di un qualsiasi altro tipo di oggetto
- Il riferimento di default ad un oggetto di tipo puntatore, ossia il nome dell'oggetto, segue le **stesse regole di visibilità di qualsiasi altro identificatore**

Tempo di vita array dinamico

- Torniamo agli array dinamici
- **NON CONFONDETE UN PUNTATORE CON L'ARRAY A CUI PUNTA !!!**
- Il puntatore serve solo a mettere da parte l'indirizzo dell'array per poter poi accedere ALL'ARRAY STESSO
- Una volta allocato, un array dinamico esiste fino alla fine del programma (o fino a che non viene deallocato, come stiamo per vedere)
  - L'array continua ad esistere anche se non esiste più il puntatore che contiene il suo indirizzo !!!

Puntatore ed array in memoria

Un puntatore e l'array a cui tale puntatore può puntare hanno **tempi di vita indipendenti**, ed occupano **zone di memoria diverse**.
In particolare, se non deallocato, un array dinamico continua ad esistere anche se e quando il puntatore utilizzato per memorizzare il suo indirizzo non esiste più

Deallocazione array dinamico

- Si può deallocare esplicitamente un array dinamico, ossia liberare lo spazio da esso occupato nella memoria dinamica, mediante l'operatore

`delete[]` <*indirizzo_oggetto_da_deallocare*>

Operatore **`delete[]`**

- Prende per argomento l'indirizzo dell'array da deallocare

Esempio:

```
int * p = new int[10] ;
delete [] p ;   // dealloca l'array dinamico
                // a cui punta p
```

- Può essere applicato solo all'indirizzo di un array dinamico allocato con l'operatore **`new`**
  - Altrimenti si ha un errore di gestione della memoria
  - Se si è fortunati, l'errore è segnalato durante l'esecuzione del programma

- Può essere applicato anche al puntatore nullo, nel qual caso non fa nulla e non genera errori

Esempio

```
main()
 {
   int vector[15]; // spazio per 15 interi
   int *dynVect;   // spazio per il puntatore, non l'array !

   int k ;
   cout<<"Inserire la dimensione desiderata del vettore\n";
   cin>>k ;
   dynVect = new int [k];

   // ora è possibile usare liberamente sia vector sia
   // dynVect come array, lunghi 15 e k, rispettivamente

   for (int i=0;i<k;i++)
      dynVect[i] = i*i;
   for (int i=0;i<15;i++)
      vector[i] = 2*i;

   delete [] dynVect; // necessaria?
 }
```

Esercizio 1

- Scrivere un programma che crei dinamicamente un array di int, di dimensioni inserite dall'utente da tastiera, e che lo riempia con valori anch'essi inseriti dall'utente. Quindi il programma stampa il contenuto dell'array.

- A questo punto, il programma conserva in memoria SOLO una parte degli elementi iniziali dell'array. In particolare chiede all'utente di inserire un numero naturale n non maggiore delle dimensioni dell'array, crea un nuovo array di dimensione n e ci copia dentro i primi n valori dal primo array.

- Quindi stampa il sotto-array conservato in memoria, dealloca tutti gli array e termina.

Esercizio 1

Esempio di output:

Dimensioni array? 4

Valore elemento 1/4? 3
Valore elemento 2/4? 2
Valore elemento 3/4? 1
Valore elemento 4/4? 2
Valore elemento 1/4: 3
Valore elemento 2/4: 2
Valore elemento 3/4: 1
Valore elemento 4/4: 2

Numero di elementi da preservare? 2

Sotto-array:
Valore elemento 1/2: 3
Valore elemento 2/2: 2

Esercizio 1

```
main(){
  int dim = -1 ;

  cout<<"Dimensioni array? " ; cin>>dim ;
  int *p = new int[dim] ; // allocazione memoria
  for (int i = 0 ; i < dim ; i++) { // memorizzazione valori
    cout<<"Valore elemento "<<i<<"/"<<dim<<"? " ;
    cin>> p[i] ;
  }         cout<<endl ;
  for (int i = 0 ; i < dim ; i++) // stampa array
    cout<<"Valore elemento "<<i<<"/"<<dim<<": "<<p[i]<<endl ;

  cout<<"Numero di elementi da preservare? " ; cin>>dim ;
  int * p2 = new int[dim] ;

  for (int i = 0 ; i < dim ; i++) { // copia valori
    p2[i] = p[i] ;
  delete[] p ; // deallocazione memoria vecchio array

  cout<<"\nSotto-array:"<<endl ;
  for (int i = 0 ; i < dim ; i++) // stampa sotto-array
    cout<<"Valore elemento "<<i<<"/"<<dim<<": "<<p2[i]<<endl ;

  delete [] p2 ;
}
```

Passaggio alle funzioni 1/2

- Passaggio di un array dinamico ad una funzione:
  - Stessa sintassi utilizzata per gli array statici
    - Oltre che
    - [**const**] <*nome_tipo*> <*identificatore*> **[]**
    - il parametro formale può essere dichiarato così

      [**const**] <*nome_tipo*> ***** <*identificatore*>
  - Queste due sintassi sono perfettamente equivalenti
  - Entrambe si possono utilizzare sia per gli array statici che per gli array dinamici

Passaggio alle funzioni 2/2

- Le dimensioni dell'array passato come parametro attuale non sono implicitamente note alla funzione chiamata
  - Il passaggio dell'array è **per riferimento**
  - Usare il qualificatore **`const`** se si vuole evitare modifiche

Domanda

- Quale importante fatto possiamo provare a dedurre dall'equivalenza delle due possibili sintassi per un parametro formale di tipo array?

Riferimenti e puntatori

- A livello di linguaggio, il passaggio di un array (statico o dinamico) non è per riferimento, ma per valore
  - Il parametro formale contiene infatti, per la precisione, una copia dell'indirizzo dell'array
- Ma proprio siccome il parametro formale contiene (una copia de) l'indirizzo dell'array, allora tramite il parametro formale si accede esattamente all'array il cui indirizzo è passato come parametro attuale
- Quindi ogni modifica effettuata all'array puntato dal parametro formale si riflette sull'array di cui si è passato l'indirizzo
- Ecco perché a livello logico possiamo affermare che si tratta a tutti gli effetti di un passaggio per riferimento di un array

Domanda

- Abbiamo scoperto che una funzione a cui passiamo un array si aspetta di fatto l'indirizzo del primo elemento dell'array
- Cosa ne deduciamo quindi sul contenuto del parametro attuale che utilizziamo quando passiamo un array?

Risposta

- Tale parametro attuale deve contenere quindi l'indirizzo del primo elemento dell'array
- Questo vale
  - sia che si tratti di un array dinamico, e quindi passiamo come parametro attuale un puntatore al primo elemento
  - sia che si tratti di un array statico, e quindi passiamo come parametro attuale il nome dell'array
- Cosa ne possiamo dedurre sull'uso del nome di un array?

Nome array statico ed indirizzo

- Scrivere il nome di un array statico come parametro attuale in corrispondenza di un parametro formale ti tipo puntatore
  - equivale a scrivere l'indirizzo del primo elemento dell'array

Ritorno da parte di funzioni

- Una funzione può ritornare l'indirizzo di un array dinamico
- Il tipo di ritorno deve essere

  [**const**] <*nome_tipo*> *****

Domanda

- Considerate una funzione che alloca un array dinamico e ne memorizza l'indirizzo in una variabile locale di tipo puntatore
- Tale puntatore viene deallocato al termine della funzione
- Alla luce di questa considerazione tale funzione può comunque ritornare l'array dinamico senza problemi?

Risposta

- Sì
- Perché l'array dinamico non viene deallocato alla fine della funzione
  - Il tempo di vita del puntatore in cui viene memorizzato l'indirizzo dell'array dinamico all'interno della funzione
  - Non ha niente a che fare col tempo di vita dell'array dinamico
- L'array dinamico rimane in memoria fino a quando non viene esplicitamente deallocato (o fino alla fine del programma stesso)

Esercizio 2

- Scrivere un programma che utilizzi una funzione per leggere da *stdin* un numero di valori di tipo **`int`** fornito a tempo di esecuzione del programma, ed inserisca tali valori in un array allocato dinamicamente dalla funzione stessa

- La funzione deve restituire al **`main()`** l'indirizzo del primo elemento dell'array dinamico creato. Stampare poi l'array nel **`main()`**

- Vediamo assieme un prima proposta di algoritmo, struttura dati e programma

Esercizio 2

- Algoritmo
  - Si chiede il numero di valori che si vogliono inserire
  - Si alloca un array dinamico della dimensione richiesta
  - Si inseriscono i valori nell'array

- Struttura dati
  - Serve un puntatore a **`int`** sia nella funzione sia nel **`main()`**
  - Serve una variabile **`int`** per memorizzare la dimensione presa da input
  - Serve un **`int`** come indice per scandire l'array

Esercizio 2

```
int* creaVett(void)
{
  int num ;
     cout<<"Quanti valori? ";
     cin>>num; // trascuriamo il controllo per brevità
     int *v = new int[num] ;
     for (int i=0; i<num; i++)
      { cout<<"v["<<i<<"]="; cin>>v[i] ; }
     return v;
}

main()
{
    int *pv;
    pv = creaVett();
    // come si fa
    // a stampare l'array?
    delete [] pv ;
}
```

Esercizio 2

```
int* creaVett(void)
{
  int num ;
     cout<<"Quanti valori? ";
     cin>>num;
     int *v = new int[num] ;
     for (int i=0; i<num; i++)
      { cout<<"v["<<i<<"]="; cin>>v[i] ; }
     return v;
}

main()
{
    int *pv;
    pv = creaVett();
    // come si fa
    // a stampare l'array?
    delete [] pv ;
}
```

Al termine della funzione non si sa più quanti elementi abbia l'array. Il *main()* e altre eventuali funzioni non potrebbero utilizzare l'*array* senza sapere la dimensione. Per poter usare l'array, il programma va esteso ...

Esercizio 2

```
int* creaVett(int &num)
{
     cout<<"Quanti valori? ";
     cin>>num;
     int *v = new int[num] ;
     for (int i=0; i<num; i++)
         { cout<<"v["<<i<<"]="; cin>>v[i] ; }
     return v;
}

main()
{
    int *pv, dim;
    pv = creaVett(dim);
    for (int i=0; i<dim; i++)
        cout<<pv[i]<<endl ;
    delete [] pv ;
}
```

In questo modo, il **`main()`** può accedere correttamente agli elementi dell'array

Riferimento a puntatore

- Come sappiamo, il riferimento è un tipo derivato
  - Dato un tipo di partenza, si può definire un riferimento a tale tipo
- Se il tipo di partenza è un puntatore, allora un riferimento ad un oggetto di tipo puntatore si definisce come segue:

[**const**] <*nome_tipo*> *** &** <*identificatore*> **;**

Esempio 1/2

- Come esempio vediamo un modo alternativo di scrivere il precedente programma
- Per ritornare l'indirizzo dell'array allocato nella funzione, memorizziamo tale indirizzo in un parametro di uscita
- Per contenere tale indirizzo il parametro deve essere un puntatore
- Ma per poter modificare il valore del parametro attuale passato alla funzione e memorizzarvi dentro l'indirizzo dell'array
  - il parametro formale dovrà essere di tipo riferimento
- In definitiva, il parametro formale deve essere proprio un riferimento a puntatore

Esempio 2/2

```
void creaVett(int * &v, int &num)
{
   cout<<"Quanti valori? ";
   cin>>num;
   v = new int[num] ;
   for (int i=0; i<num; i++)
     { cout<<"v["<<i<<"]="; cin>>v[i] ; }
}

main()
{
    int *pv, dim;
    creaVett(pv, dim);
    for (int i=0; i<dim; i++)
        cout<<pv[i]<<endl ;
    delete [] pv ;
}
```

Versione (concettualmente più complessa) con due parametri che riportano sia l'array sia la sua dimensione.
La funzione deve restituire l'array attraverso un parametro passato per riferimento. Poiché il tipo dell'array è un puntatore a int (cioè, int *), il tipo del parametro è un riferimento a puntatore a int

Esercizio per casa

- Scrivere una funzione che prenda in ingresso un array di interi, ne crei un altro uguale, e ritorni (l'indirizzo del) secondo array mediante un parametro di uscita (un parametro quindi di tipo riferimento a puntatore).
- La funzione non legge niente da *stdin* e non scrive niente su *stdout*.
- Se ci riuscite, realizzate la funzione dichiarandola con tipo di ritorno **`void`**

Programma

```
void vett_copy(const int* v1, int num,
               int*& v2)
{
    v2 = new int[num] ;
    for (int i=0; i<num; i++)
        v2[i] = v1[i];
}

main()
{
    int vettore[] = {20,10,40,50,30};
    int* newVet = 0 ;
    vett_copy(vettore, 5, newVet);
    delete [] newVet ;
}
```

Flessibilità e problemi seri

- Più di un puntatore può puntare allo stesso oggetto
  - Quindi possono esservi effetti collaterali
- Una variabile di tipo puntatore è come una variabile di un qualsiasi altro tipo
  - Quindi può essere utilizzata anche se non inizializzata !!!!
    - Errore logico e di accesso/gestione della memoria
- Inoltre può essere (ri)assegnata in ogni momento
- Queste caratteristiche dei puntatori possono portare ad alcuni degli errori di programmazione più difficili da trovare

Problema: dangling reference

- **Dangling reference** (**pending pointer**)
  - Si ha quando un puntatore punta ad una locazione di memoria in cui non è presente alcun oggetto allocato
  - Tipicamente accade perché il puntatore non è stato inizializzato, o perché l'oggetto è stato deallocato
- Problema molto serio
  - Se si usa un pending pointer si hanno errori di gestione della memoria che possono portare ad un comportamento impredicibile del programma

Puntatore non inizializzato

```
main()
{
  int N ;
  cin>>N ;
  int *p ; // p contiene un valore casuale

  cin>>p[0] ; // ERRORE LOGICO E DI GESTIONE DELLA
              // MEMORIA: p non è stato
              // inizializzato/assegnato
              // all'indirizzo di alcun array
              // dinamico
}
```

Oggetto deallocato

```
main()
{
  int N ;
  cin>>N ;
  int * p = new int [N] ;
  delete [] p ;
  cout<<p[0]<<endl ; // ERRRORE LOGICO
                     // E DI ACCESSO ALLA MEMORIA
}
```

Per ridurre i problemi

- Ovunque possibile, utilizzare perlomeno puntatori costanti

Esempio:

```
int dim ;
cin>>dim ;
int * const p = new int[dim] ;
```

- Così siamo costretti ad inizializzarli

- Inoltre non possiamo riassegnarli ad altri array

Esaurimento memoria

- In assenza di memoria libera disponibile, l'operatore **new** fallisce
  - viene generata una eccezione
  - se non gestita, viene stampato un messaggio di errore ed il programma è terminato forzatamente
- Se si vuole, si può
  - gestire l'eccezione
- oppure
  - *agganciare* il fallimento dell'operatore ad una propria funzione
    - Tale funzione verrà invocata in caso di fallimento
- Non vedremo nessuna delle due soluzioni, quindi i nostri programmi semplicemente termineranno in caso di esaurimento della memoria

Prova

- Compilare ed eseguire il seguente programma

```
main()
{
      while(true) {
            int *p = new int[100000] ;

            for (int i = 0 ; i < 100000 ; i++)
                  p[i] = i ;
      }
}
```

Prova

- Il precedente programma esaurisce abbastanza rapidamente tutta la memoria del calcolatore
- A causa dell'importante errore descritto nella prossima slide

Problema serio: memory leak

- **Memory leak**
  - Esaurimento inaspettato della memoria causato da mancata deallocazione di oggetti dinamici non più utilizzati
- Spesso correlato con la perdita dell'indirizzo degli oggetti stessi

Esempio memory leak

```
void fun()
{
  int N ;
  cin>>N ;
  int * p = new int [N] ;
}

main()
{
  fun() ;
  // una volta invocata fun(), l'array rimane in memoria,
  // ma nel main p non è visibile, per cui, una
  // volta terminata fun(), si è perso ogni
  // modo di accedere all'array, quindi, tra l'altro, non
  // si può più deallocarlo !
  ...
}
```

Tipo elementi array dinamici

- E’ possibile allocare array dinamici di oggetti di qualsiasi tipo
  - Come si alloca una stringa dinamica?
  - Come si alloca una array di struct?
  - Come si alloca un array di array, ossia una matrice dinamica?

Stringhe dinamiche

- Stringa di 10 caratteri:

```
char * const str  =  new char[11] ;
```

- Stringa di dimensioni definite da *stdin*:

```
int dim ;
do cin>>dim ; while (dim <= 0) ;
char * const str  = new char[dim+1];
```

Array dinamici di **`struct`**

```
struct persona
{
      char nome_cognome[41];
      char codice_fiscale[17];
      float stipendio;
} ;

main()
{
      int dim ;
      do cin>>dim ; while(dim <= 0) ;
      persona * const t = new persona[dim] ;
      ...
}
```

Matrici dinamiche

- Una matrice è un array di array
- Quindi una matrice dinamica è un array dinamico di array
  - Ogni elemento dell'array dinamico è a sua volta un array
  - Ci concentriamo solo sul caso in cui sia un array statico
  - Le sue dimensioni devono essere quindi specificate a tempo di scrittura del programma
- Esempio di puntatore ed allocazione matrice bidimensionale di **n** righe e 10 colonne:

```
int (*p)[10] = new int[n][10] ;
```

- Deallocazione:

```
delete [] p ;
```

Passaggio e ritorno

- Per passare una matrice dinamica bidimensionale occorre un parametro della forma:

[**const**] <*nome_tipo*> **(*** <*identificatore*>**) [**<*espr_costante*>**]**

- Ad esempio, per passare una matrice dinamica da 10 colonne:

```
void fun(int (*p)[10]) { … }
```

  - Nel caso si ometta il nome del parametro in una dichiarazione, la sintassi diviene

```
void fun(int (*)[10]) ;
```

- La stessa forma si usa per ritornare una matrice dinamica. Ad esempio:

```
int (*fun(...))[10] ;
```

Accesso agli elementi

- Si accede agli elementi di una matrice dinamica utilizzando la stessa sintassi che si utilizza per una matrice statica

Esercizio 3

- Una pila è una struttura dati che prevede due operazioni:
- 1) push: inserimento di un elemento di un dato valore (da passare alla funzione) in cima alla pila. La pila ha una dimensione massima predefinita; se la si supera l'operazione di push fallisce;
- 2) pop: estrazione di un elemento dalla cima della pila; se la pila è vuota l'operazione fallisce altrimenti viene ritornato il valore dell'elemento estratto (una volta estratto l'elemento, è necessario modificarne il valore all'interno della pila?)
- Scrivere un programma che implementi una pila di elementi di tipo int.

Esercizio 3

In particolare il programma deve offrire all'utente la possibilità di scegliere tra le seguenti operazioni:

1. inserimento nella pila da stdin: si aggiunge un elemento in cima alla pila e si stampa il nuovo contenuto dell'intera pila

2. estrazione e stampa dalla pila su stdout: si chiede all'utente quanti elementi si vuole estrarre dalla pila mediante una successione di pop, e se ne stampa il valore

- Come al solito potete definire la pila come variabile locale al main, o come variabile globale. La prima soluzione è un po' più complessa, ma più pulita, mentre la seconda è più semplice ma è peggiore in termini di visibilità delle variabili ed effetti collaterali.

Esercizio 3

Finite le precedenti due operazioni, realizzare l'operazione:

3) Riserva memoria: si passa il numero di elementi che si intende riservare a partire dalla cima (come se si fossero fatte delle push, ma senza cambiare valore agli elementi)

Aggiungere la possibilità per l'utente di riservare memoria con la precedente operazione. In particolare, in conseguenza della richiesta di riservare memoria, stampare il valore degli elementi riservati

Testare il programma nel suo insieme facendo un po' di inserimenti, seguiti da altrettante estrazioni. Scegliere infine di riservare memoria, e controllare il contenuto degli elementi riservati. Sono valori fissi o dipendono dalla precedente sequenza di inserimenti ed estrazioni?

Esercizio 3

```
main(){
  int dim = -1 ;

  cout<<"Dimensioni array? " ; cin>>dim ;
  int *p = new int[dim] ; // allocazione memoria
  for (int i = 0 ; i < dim ; i++) { // memorizzazione valori
    cout<<"Valore elemento "<<i<<"/"<<dim<<"? " ;
    cin>> p[i] ;
  }         cout<<endl ;
  for (int i = 0 ; i < dim ; i++) // stampa array
    cout<<"Valore elemento "<<i<<"/"<<dim<<": "<<p[i]<<endl ;

  cout<<"Numero di elementi da preservare? " ; cin>>dim ;
  int * p2 = new int[dim] ;

  for (int i = 0 ; i < dim ; i++) { // copia valori
    p2[i] = p[i] ;
  delete[] p ; // deallocazione memoria vecchio array

  cout<<"\nSotto-array:"<<endl ;
  for (int i = 0 ; i < dim ; i++) // stampa sotto-array
    cout<<"Valore elemento "<<i<<"/"<<dim<<": "<<p2[i]<<endl ;

  delete [] p2 ;
}
```

Strutture dati dinamiche

- Vi sono problemi risolvibili efficacemente mediante algoritmi che fanno uso di **strutture dati dinamiche**
  - Ossia strutture dati che cambiano dimensione durante l'esecuzione dell'algoritmo

Problema

- Supponiamo di dover memorizzare e ristampare una successione di valori il cui numero non sia noto a priori
- Supponiamo inoltre che, oltre ad inserirli, sia necessario di tanto in tanto estrarre alcuni valori

Array dinamico 1/2

- Possibile soluzione: array dinamico riallocato ogni volta che si renda necessario
- Ogni riallocazione ha costo *O(N)*
  - Bisogna ricopiare tutti i valori nella nuova locazione
- Comunque si fa “ogni tanto”, per cui l'inserimento ha costo *ammortizzato O(1)*

Array dinamico 2/2

- Però ad ogni estrazione di un elemento che non sia l'ultimo bisogna ricompattare l'array se non si vogliono lasciare 'buchi'
- Questo costa *O(N)* **tutte le volte**

Domanda

- Vi viene in mente una soluzione migliore?
- In merito, considerate che, anche se non abbiamo visto come, con l'operatore **new** si può anche allocare un solo oggetto anziché un array di oggetti

Proposta

- Perché ogni volta che dobbiamo aggiungere un elemento non lo allochiamo in memoria **da solo**?

- Se e quando dobbiamo estrarlo lo dealloceremo, di nuovo **da solo**

Problemi

- Dove memorizziamo l'indirizzo dei vari elementi?
- Cominciamo dal primo ...

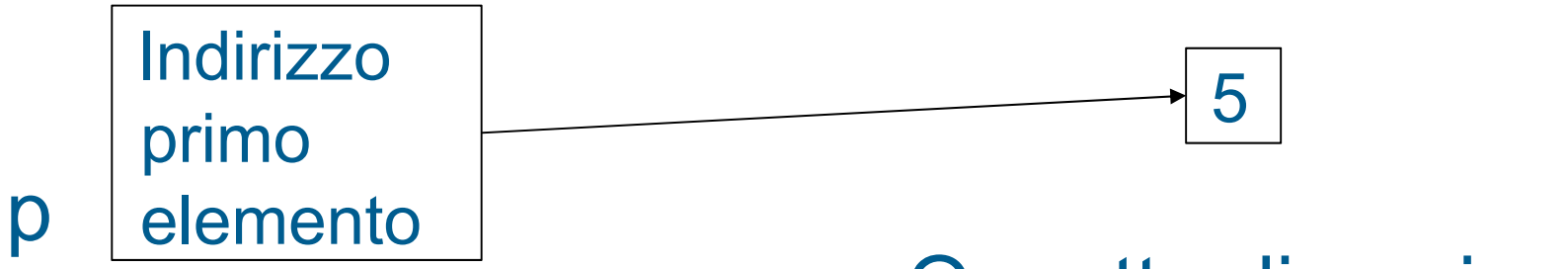
Puntatore al primo elemento

- Potremmo memorizzare in una variabile di tipo puntatore l'indirizzo di tale elemento

Puntatore al primo elemento

- Supponiamo che il primo valore sia 5
  - Allochiamo in memoria spazio per un intero e memorizziamo il valore
  - Ne memorizziamo l'indirizzo in una variabile **p** di tipo puntatore

Puntatore al primo elemento



Oggetto dinamico

Variabile locale o globale: oggetto automatico o statico

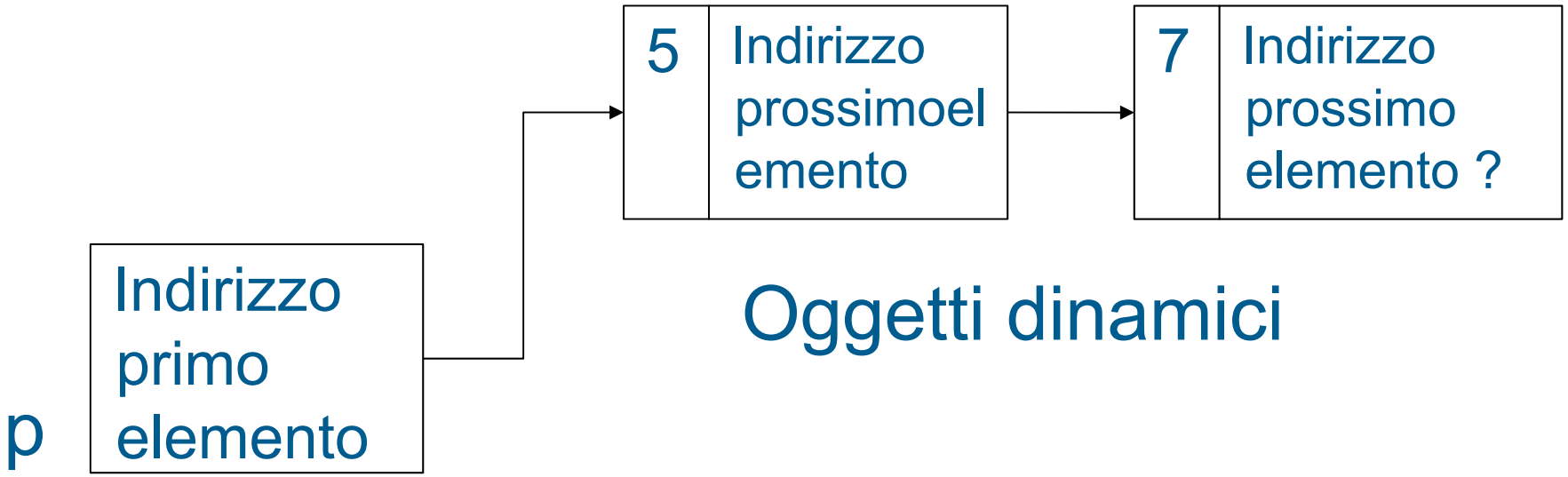
Elementi successivi

- Supponiamo di inserire un altro valore, diciamo 7
- Come facciamo per memorizzare l'indirizzo del secondo elemento, ed in generale l'indirizzo del prossimo elemento ogni volta che ne aggiungiamo uno?

Puntatore al successivo

- Per ciascun valore, potremmo allocare spazio in memoria
  - sia per il valore dell'elemento,
  - che per un puntatore che punti al prossimo elemento
- Così, una volta raggiunto un elemento, abbiamo le informazioni necessarie per accedere al prossimo

Puntatore al successivo



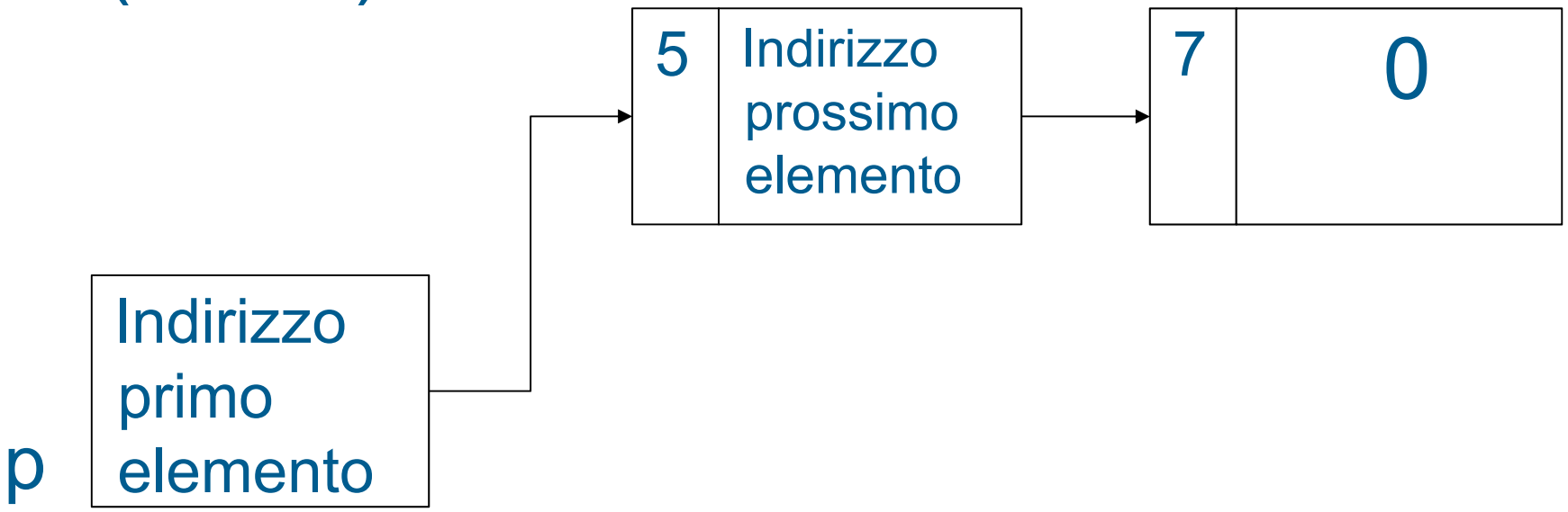
Variabile locale o globale: oggetto automatico o statico

Ultimo elemento 1/2

- L'elemento contenente il valore 7 è l'ultimo (ce ne sono solo due)
- Che valore possiamo assegnare al puntatore all'interno della struttura che lo rappresenta?
- Come facciamo a dire che non ci sono altri elementi dopo di lui?

Ultimo elemento 2/2

- Possiamo assegnargli il valore 0 (NULL)



- Abbiamo costruito un oggetto di tipo **lista concatenata**

Lista concatenata

- Struttura dati i cui oggetti/elementi sono disposti in ordine lineare
- Diversamente dall'array, in cui l'ordine è determinato dagli indici, l'ordine in una lista concatenata è determinato da un puntatore in ogni oggetto

Terminologia 1/2

- Diremo che ciascun elemento contiene un **campo informazione** ed un campo puntatore (oppure due, come stiamo per vedere)
- Il primo elemento di una lista è tipicamente chiamato **testa** (**head**) della lista
- L'ultimo elemento è tipicamente chiamato **coda** (**tail**) della lista

Terminologia 2/2

- Lista singolarmente concatenata o semplice: ciascun elemento contiene solo un puntatore al prossimo elemento

- Lista doppiamente concatenata o doppia: ciascun elemento contiene sia un puntatore al prossimo elemento che un puntatore all'elemento precedente

Lista semplice 1/2

- Ciascun elemento contiene solo un puntatore al prossimo elemento

Lista semplice 2/2

- Il puntatore al prossimo elemento della *coda* della lista contiene il valore 0 (NULL)
- Il puntatore alla testa della lista individua la lista stessa
  - E' perciò chiamato anche puntatore alla lista

Lista doppia

- Ciascun elemento contiene un puntatore al prossimo elemento e uno all'elemento precedente