Lezione 13

Array dinamici
Puntatori

Da oggetti statici a dinamici

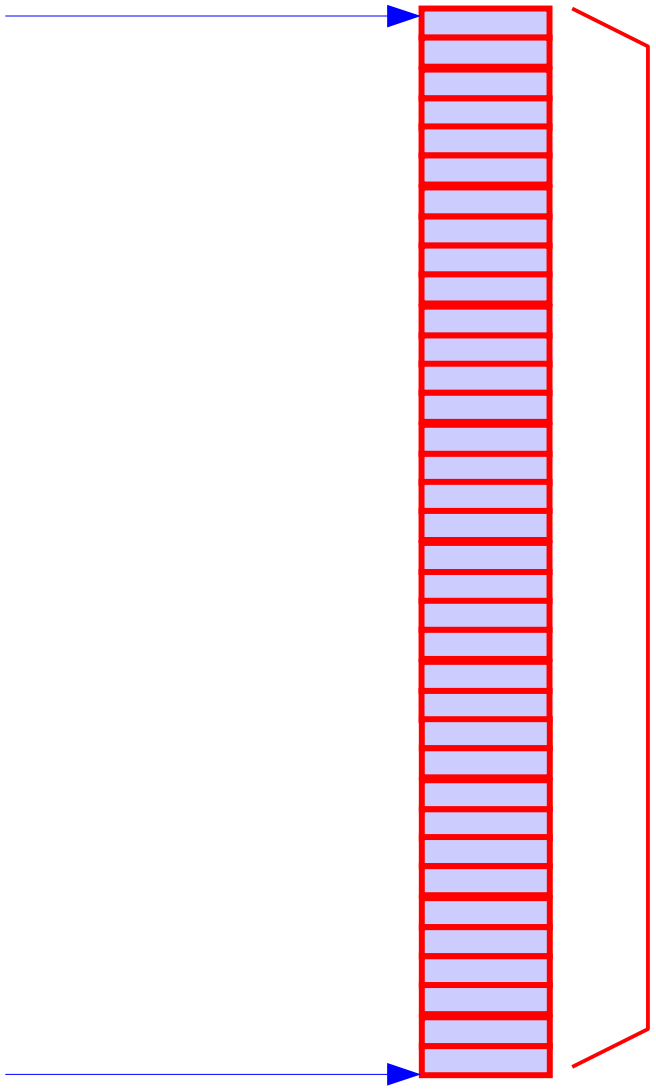
- La dimensione di tutti gli oggetti (concreti) considerati finora deve essere definita a tempo di scrittura del programma
- Talvolta però non è possibile sapere a priori la quantità di dati da memorizzare/gestire
- Per superare la rigidità della definizione statica delle dimensioni, occorre un modo per **“allocare in memoria oggetti le cui dimensioni sono determinate durante l'esecuzione del programma”**
- Questo è possibile grazie al 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 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**

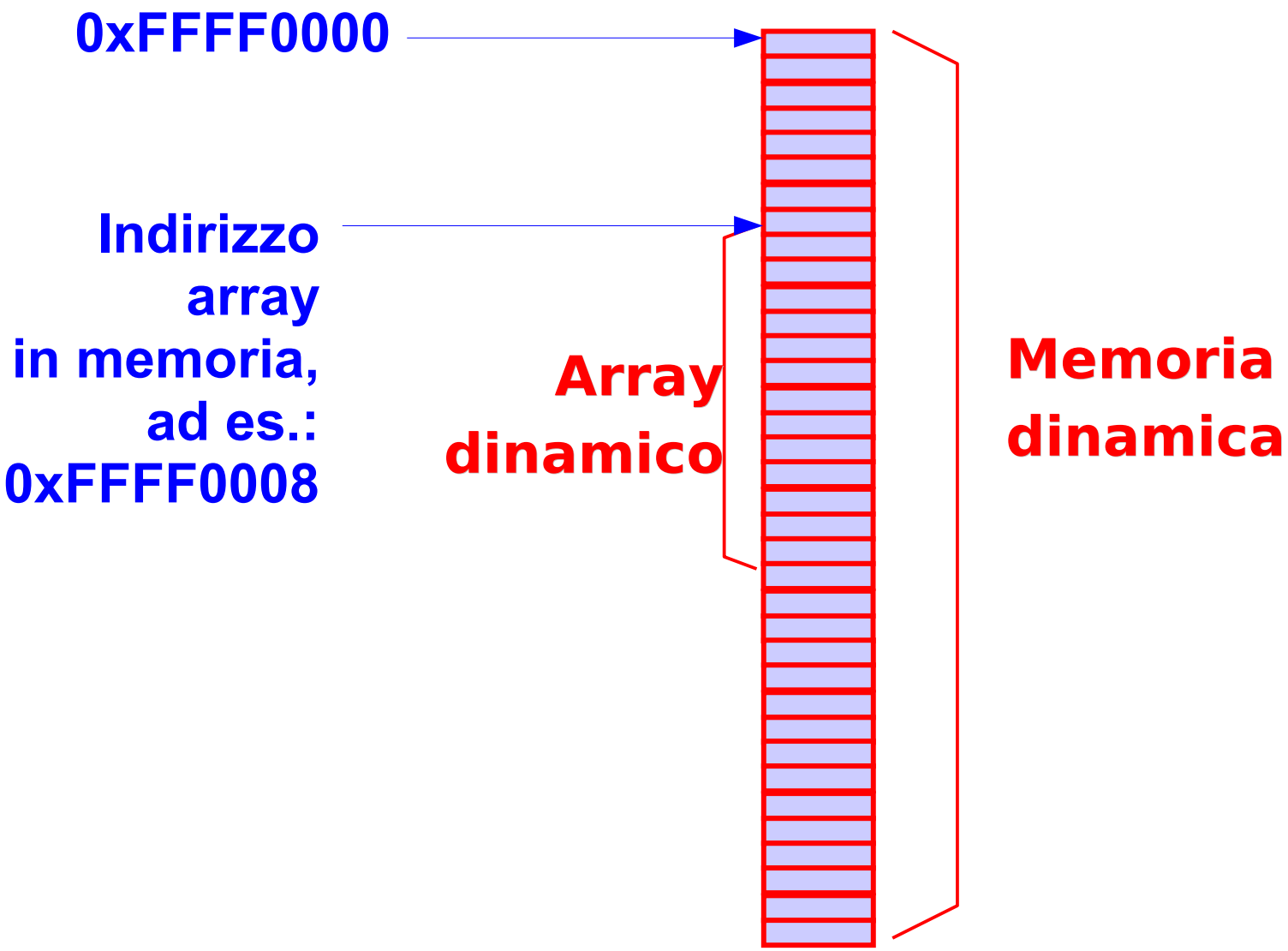
  new <*nome_tipo*> **[**<*num_elementi*>**]** **;**

  - 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 sinonimo) all'oggetto allocato
- Gli oggetti dinamici sono **oggetti senza nome**
- Come si fa per accedere a tali oggetti?
- In generale, quale informazione ci serve per accedere ad un oggetto?

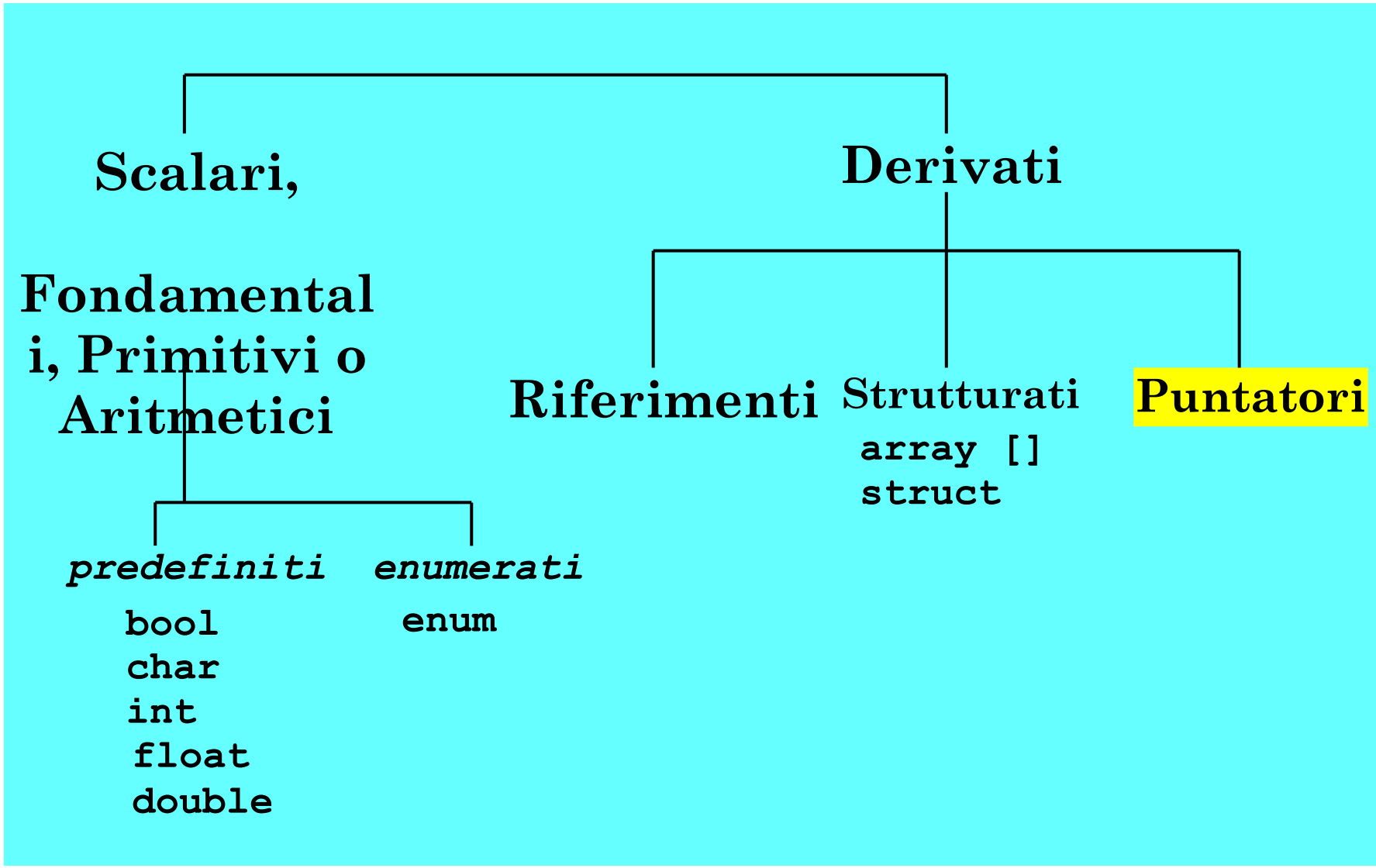
Risposta

- Il suo **indirizzo**

Ritorno operatore new

- L'operatore **`new`** ritorna proprio l'**indirizzo dell'oggetto allocato**
- Possiamo 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
predefiniti
bool
char
int
float
double
enumerati
enum
Derivati
Riferimenti
Strutturati
array []
struct
Puntatori

Puntatori

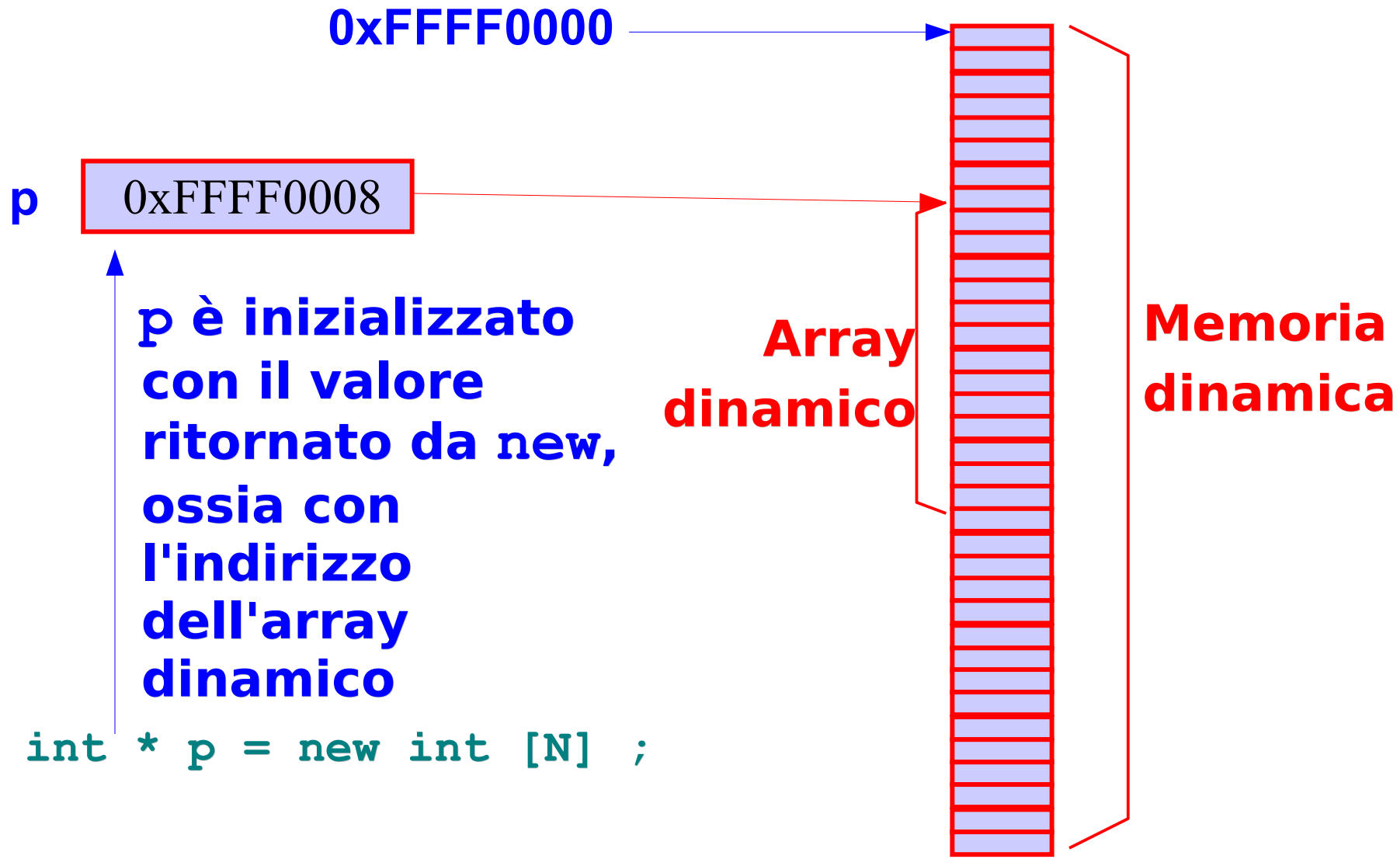
- Un oggetto di tipo puntatore ha per valore un indirizzo di memoria (che non è altro che un numero naturale)
- Le definizione di un oggetto puntatore ha la seguente forma

  [**const**] <*tipo_oggetto_puntato*>

  * [**const**] <*identificatore*> [= <*indirizzo*>] ;
- Il primo qualificatore **const** è presente se si punta ad un oggetto non modificabile
- Il secondo **const** è presente se il valore del puntatore, una volta inizializzato, non può 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
  - 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] = 0 ; // inizializzazione

  cout<<p[0]<<endl ;

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

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, NON VEDREMO ALTRE OPERAZIONI CON I PUNTATORI
  - Segue le stesse regole di tempo di vita di un qualsiasi altro tipo di oggetto
- I riferimenti ad un oggetto di tipo puntatore (compreso il riferimento di default, ossia il nome dell'oggetto) seguono le **stesse regole di visibilità di tutti gli identificatori**

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 potervi poi accedere in un secondo momento
- Una volta allocato, un array dinamico esiste fino alla fine del programma (o fino a che non viene deallocato, come stiamo per vedere)
  - Anche se non esistesse 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
              // puntato da 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

- Dalla decima esercitazione:
  - *crea_riempi_distruggi_array.cc*

Passaggio alle funzioni

- Passaggio di un array dinamico ad una funzione:
  - Possibile in modo identico agli array statici
    - Oltre che

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

      il parametro formale può essere dichiarato

      [**const**] <*nome_tipo*> ***** <*identificatore*>
  - 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

Riferimenti e puntatori 2/2

- A livello di linguaggio non si tratta quindi di un passaggio per riferimento, ma di un passaggio per valore
  - Il parametro formale contiene infatti 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

Ritorno da parte di funzioni

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

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

Esercizio

- Scrivere un programma che utilizzi una funzione per leggere da *stdin* un certo numero di valori di tipo **`int`** e li inserisca in un array allocato dinamicamente dalla funzione stessa
- La funzione deve restituire al **`main()`** il puntatore all'array dinamico creato. Stampare poi l'array nel **`main()`**

Algoritmo e struttura dati

- Algoritmo
  - Si chiede il numero di valori che si vogliono inserire
  - Si alloca un array dinamico della dimensione richiesta
  - Si scandisce tutto l'array, inserendo i valori elemento per elemento
- 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

Proposta programma 1/2

```
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 ;
}
```

Proposta programma 2/2

```
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 ;
}
```

Non si sa 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

- Estendere opportunamente il programma e scrivere anche il codice di stampa del contenuto dell'array

Programma

```
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 propriamente 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

- 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 ;
}
```

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
}
```

Flessibilità e problemi seri

- 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
- Infine più di un puntatore può puntare allo stesso oggetto
  - Quindi possono esservi effetti collaterali
- Ma anche di peggio ...

Problema: dangling reference

- Dangling reference (pending pointer)
  - Si ha quanto 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 e non possiamo riassegnarli ad altri array o magari a puntatori pendenti

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 termina
- 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

Problema serio: memory leak

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

Esempio

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

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

Tipo 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 dimensione definita 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
  - Le dimensioni degli array componenti devono essere 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 voglia omettere 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 usa per una matrice statica

Esercizi

- Svolgere tutti gli esercizi della decima esercitazione fino all'I/O non formattato escluso