

Array dinamici e puntatori

Motivazioni

- Gli oggetti considerati finora devono essere completamente definiti (tipo, nome e dimensione) a tempo di scrittura del programma
- Ciò non costituisce un problema per gli oggetti di tipo primitivo, ma può essere limitativo per oggetti di tipo strutturato quali array e stringhe
 - talvolta 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 **“chiedere al sistema (altra) memoria quando ce ne è bisogno durante l'esecuzione del programma”**
- Questo è possibile grazie al meccanismo di **allocazione dinamica della memoria**

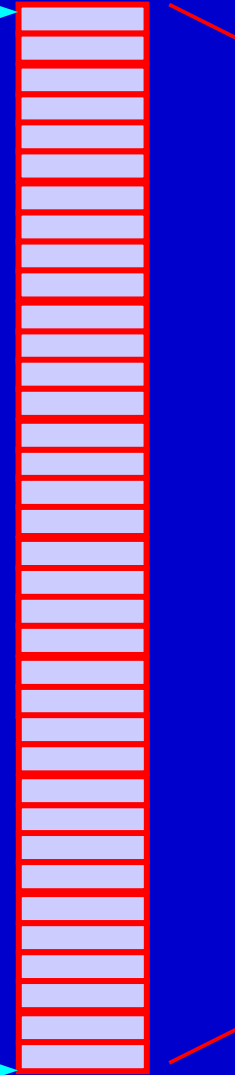
Memoria libera

- **Prima dell'inizio dell'esecuzione di un processo, il sistema operativo riserva ad esso uno spazio di memoria di dimensioni predefinite**
 - **locazioni consecutive di memoria (tipicamente da 1 byte l'una)**
- **Questo spazio di memoria è a sua volta organizzato in segmenti distinti**
- **Uno di questi segmenti è chiamato**
 - **memoria libera, memoria dinamica oppure heap**
- **E' possibile allocare oggetti all'interno della memoria dinamica in momenti arbitrari durante l'esecuzione del programma**
 - **ovviamente finché lo spazio non si esaurisce**

Memoria libera

Indirizzo prima
locazione, ad es.:
0xFFFF0000

Indirizzo ultima
locazione,
mobile (vedremo
in seguito)



Memoria
libera

Allocazione dinamica

- **Gli oggetti allocati in memoria dinamica sono detti dinamici**
- **Per i nostri scopi considereremo solo array dinamici**
 - **Array allocati in memoria dinamica durante l'esecuzione del programma**
 - **Il numero di elementi non è più vincolato ad essere definito a tempo di scrittura del programma**
- **Come si alloca un array dinamico?**
 - **Mediante l'operatore new**
- **Come si accede ad un oggetto in memoria dinamica?**
 - **Come ogni oggetto è caratterizzato da un indirizzo nello spazio di memoria del processo**
 - **L'operatore new ritorna l'indirizzo dell'oggetto allocato in memoria dinamica**

Operatore new

- Vediamo l'uso dell'operatore new solo per allocare array

```
new <nome_tipo> [<num_elementi>]
```

Alloca un array di <num_elementi> oggetti di tipo <nome_tipo>, non inizializzati

- Ad esempio

```
new int[10] ; // alloca un array dinamico
              // di 10 elementi
```

- Ma possiamo anche scrivere

```
int a ;
cin>>a;
new int[a] ; // alloca un array dinamico
              // di a elementi
```

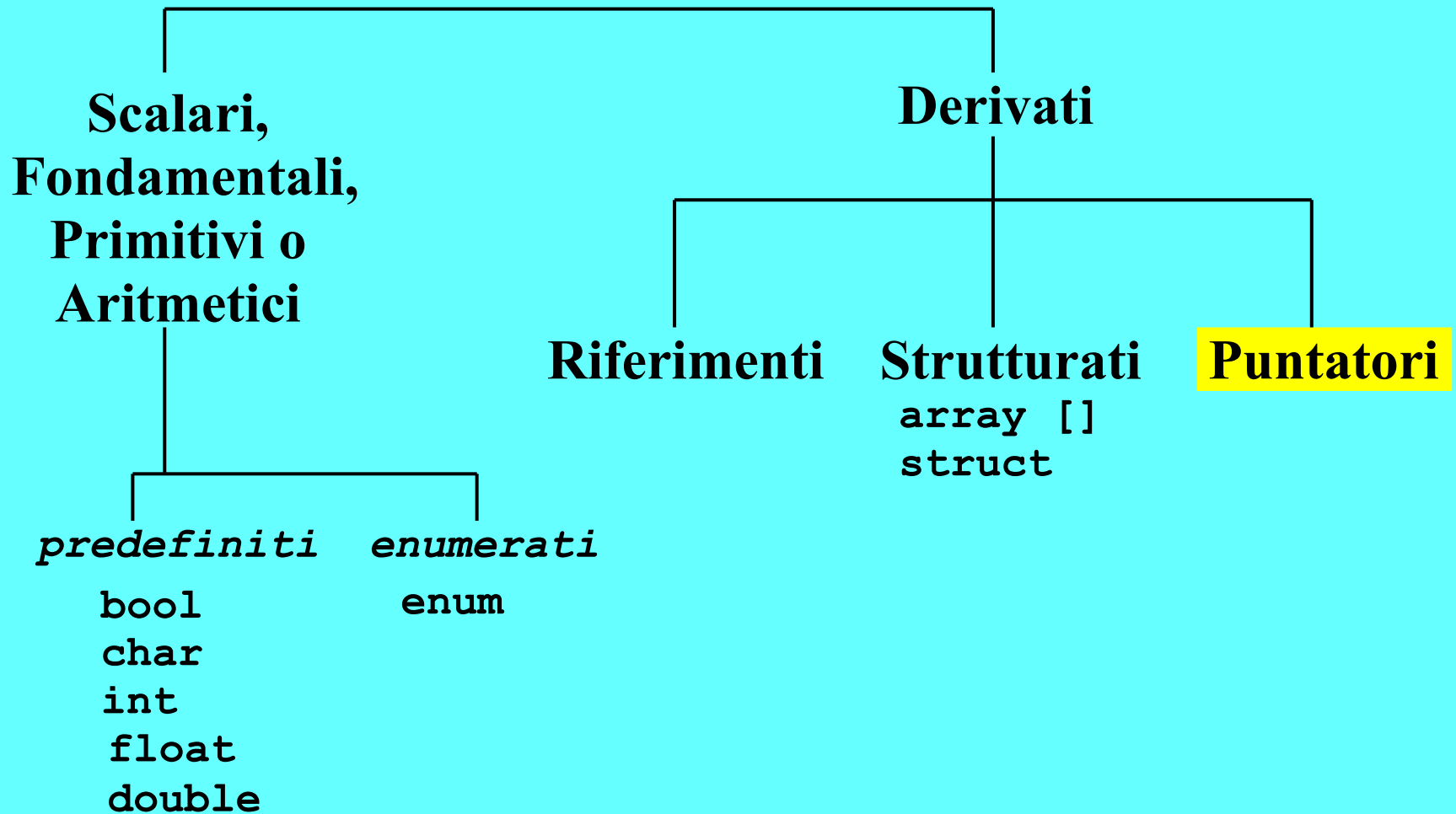
Array dinamico allocato nello heap



Valore di ritorno dell'operatore new

- 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?
 - L'operatore new ritorna l'indirizzo dell'oggetto allocato
- Possiamo accedere all'oggetto tramite tale indirizzo
- Ma dove lo memorizziamo?
 - Utilizziamo oggetti di tipo **puntatore**

Tipi di dato



Puntatori

- Un oggetto di tipo puntatore contiene un indirizzo di memoria
- 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:

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

Indirizzo array dinamico

0xFFFF0000

p

0xFFFF0008

p è inizializzato con il valore ritornato da new, ossia con l'indirizzo dell'array dinamico

Array dinamico

Memoria libera

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

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 ...**

Esempi di 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: ERRORE 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'ASSEGNAIMENTO, 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 quello di default) seguono le stesse regole di visibilità di tutti gli identificatori

Tempo di vita di un array dinamico

- Torniamo agli array dinamici
- **NON CONFONDETE UN PUNTATORE CON L'ARRAY A CUI PUNTA !!!**
- **Una volta allocato, un array dinamico esiste fino alla fine del programma**
 - Anche se non esistesse più il puntatore che contiene il suo indirizzo !!!
- **Si può però deallocare esplicitamente un array dinamico, ossia liberare lo spazio da esso occupato nella memoria dinamica, mediante l'operatore**

`delete []`

Operatore delete []

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

Es.:

```
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, 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?
}
```

Esercizi

- *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 sono sconosciute alla funzione chiamata**
 - **Il passaggio è per riferimento**
 - **Usare il qualificatore const se si vuole evitare modifiche**

Ritorno da parte delle funzioni

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

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

Esercizio 1 (*Specifica*)

- Scrivere un programma che utilizzi una funzione per leggere da input un certo numero di valori *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 l'array nel main

Esercizio 1 (*Algoritmo della funzione*)

- E' necessario chiedere in input il numero di valori che si vogliono inserire
- Si alloca un array dinamico della dimensione richiesta
- Si scandisce tutto il vettore, inserendo elemento per elemento

Esercizio 1 (*Rappresentazione informazioni*)

- 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 1 (*Programma*)

```
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();
    delete [] pv ;
}
```

Non si sa quanti elementi abbia l'array. Il main() e altre eventuali funzioni non possono utilizzare l'array senza sapere la dimensione.

Esercizio 1 (*Programma corretto*)

```
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

Esercizio 1 (*Programma alternativo*)

```
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 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.

Riferimento a puntatore

- Tipo derivato
- Però il tipo di partenza è un puntatore
- Definizione

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

Esercizio 2

- **Scrivere una funzione `vett_copy()` che prenda in ingresso l'indirizzo di un array di interi, ne crei un altro uguale, e ritorni (l'indirizzo del) secondo array per riferimento**

Esercizio 2

```
void vett_copy(const int* v1, int num,
               int*& pv2)
{
    pv2 = new int[num] ;
    for (int i=0; i<num; i++)
        pv2[i] = v1[i];
}

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

Qualificatore const

- **Un puntatore ad array dinamico costante non può essere utilizzato per modificare il valore degli elementi dell'array**
- **Una variabile di tipo puntatore può essere (ri)assegnata in ogni momento**
 - **Una costante di tipo puntatore invece può essere solo inizializzata**
 - **Questo può evitare gli errori pericolosi che vedremo tra qualche slide**

Esempio 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
}
```

Esempio puntatore costante

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

    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 serio: dangling reference

- **Dangling reference (pending pointer)**
 - **Si ha quanto un puntatore punta ad una locazione di memoria in cui non è stato allocato nessun oggetto**
 - **Tipicamente perché non è stato inizializzato, o perché l'oggetto è stato deallocato**

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 ; // ERRORE LOGICO
                       // E DI ACCESSO ALLA MEMORIA
}
```

Per ridurre i problemi

- Ovunque possibile, utilizzare perlomeno puntatori costanti

Es:

```
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**

Problema serio: memory leak

- **Esaurimento inaspettato della memoria causato da mancata deallocazione di oggetti non più utilizzati**
 - **Spesso correlato con la perdita di riferimenti all'oggetto stesso**

Esempio di memory leak

```
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
    // riferimento all'array, quindi, tra l'altro, non
    // si può più deallocarlo !
    ...
}
```


Stringhe dinamiche

- **E' possibile allocare array dinamici di oggetti di qualsiasi tipo**
 - *Come si alloca una stringa dinamica?*

Allocazione stringa dinamica

- **Stringa di 10 caratteri:**

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

- **Stringa di dimensione definita da stdin:**

```
int dim ;  
cin>>dim ;  
char * const str = new char[dim+1] ;
```

- **Esempio di definizione di puntatori costanti**

Allocazione di array di struct

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

persona * const t = new persona[10] ;
```

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 ;
```

Riepilogo

- Ingredienti fondamentali per la gestione degli array dinamici:
 - Operatore `new`
 - Operatore `delete[]`
 - Tipo di dato puntatore