Allocazione dinamica della memoria

Anna Corazza

aa 2023/24

Dove studiare

- Str'13, sezione 11.2
- Str'14, sezioni 25.3, A.5.6

Str'13 Bjarne Stroustrup, The C++ Programming Language (4th edition), 2013 `https://www.stroustrup.com/4th.html`

Str'14 Bjarne Stroustrup, Programming: Principles and Practice using C++ (2nd edition), 2014 `https://www.stroustrup.com/programming.html`

Free store

- La vita di un oggetto con nome è determinata dallo scope in cui è stato definito.
- A volte, però, occorre creare un oggetto la cui esistenza sia indipendente dallo scope in cui è stato creato.
- Ad esempio, voglio creare un oggetto all'interno di una funzione e poi restituirlo in modo che possa continuare ad essere utilizzato.
- Per ottenere ciò si usano:

new	per creare un oggetto
delete	per distruggerlo
delete[ ]	se si tratta di un array

new

- `new` crea oggetti sullo **heap** (si dice anche in memoria dinamica o sullo free store)
- Nel caso la memoria sia esaurita, `new` lancia un'eccezione `bad_alloc`
- In caso di successo, `new` alloca almeno un byte e restituisce un **puntatore** alla memoria allocata.
- Esempi:

```
// alloca un int, senza inizializzarlo
int* p1=new int;
// alloca 100 interi senza inizializzarli
int* p2=new int[100]
...
// dealloca singoli oggetti
delete p1;
// dealloca un array
delete[] p2;
```

`new` con inizializzazione

- Oggetti built-in allocati con `new` non vengono inizializzati a meno che non sia fatto esplicitamente
- Esempi:

```
// alloca un int e lo inizializza a 7
int* p3=new int(7);
...
// dealloca singoli oggetti
delete p3;
```

- L'allocazione di oggetti di una classe con `new` viene fatta attraverso il costruttore della classe: il costruttore di default viene chiamato a meno che non sia specificata un'inizializzazione.

`delete`

- Se un oggetto è stato creato con `new` esso esiste fino a quando non viene esplicitamente distrutto con `delete`.
- Solo a quel punto lo spazio in memoria occupato dall'oggetto può venir riutilizzato.
- Un'implementazione C++ non garantisce la presenza di un **garbage collector**.
- L'operatore `delete` può venir applicato esclusivamente a puntatori restituiti da un operazione di `new` oppure a `nullptr`
  - L'applicazione di `delete` a `nullptr` non ha nessun effetto
- Quando il `delete` viene applicato ad una classe in cui è stato definito un distruttore, il distruttore viene eseguito prima che la memoria sia rilasciata.

`delete[]`

- Per liberare lo spazio allocato da una `new`, sia `delete` che `delete[]` devono conoscere la **dimensione** dell'oggetto allocato.
- A `delete` basta conoscere il **tipo** dell'oggetto puntato.
- Ma `delete[]` deve conoscere anche il **numero di oggetti** puntati.
- Quando viene allocata memoria sullo heap, la `new []` tiene traccia di quanta memoria/numero di elementi allocata/i,
- di solito salvando l'informazione in un segmento che precede immediatamente la memoria allocata.
- Ne consegue che un oggetto array allocato mediante l'implementazione standard di `new` occuperà uno spazio leggermente superiore del corrispondente statico.
- Quanto meno, infatti, lo spazio necessario a **contenere** la dimensione dell'oggetto.

Dimensione spazio da liberare

- Quindi: come fa `delete[]` a sapere quanto spazio deve liberare?
- Quando viene allocata memoria sullo heap, la `new []` tiene traccia di quanta memoria/numero di elementi allocata/i,
- **NB**: attenzione: l'operatore `sizeof` calcola la dimensione di un tipo di dato, non di una particolare istanza
- quindi il calcolo avviene durante la **compilazione**, non l'esecuzione.
- Questo vale solo per l'allocazione degli array: in tutti gli altri casi viene usato il tipo del puntatore.
- Ma quanta e quale informazione viene aggiunta **dipende dall'implementazione**! e non solo ...
- anche dal "mode" della compilazione: in debugging mode può venir allocata più memoria

Gestione della memoria

- I principali problemi che si possono incontrare nella gesione dello head sono:

  Oggetti abbandonati (leaked), risultato di un'allocazione con `new` che poi non è stata liberata don `delete`

  Cancellazione prematura viene liberato un oggetto, ma esiste un'altra copia di puntatore a quell'oggetto che poi verrà utilizzato

  Doppia cancellazione : l'oggetto viene cancellato due volte e se esiste un distruttore, esso viene invocato due volte sulla stessa area di memoria

Gestione della memoria

Cancellazione prematura: esempio

```
int* p1=new int{99};
int*p2=p1;
delete p1; // e quindi anche p2!
p1=nullptr; // ma dovrei farlo anche con p2!
char* p3=new char{'x'}; // p3 e p2 puntano alla
    stessa memoria
// ma con "interpretazioni" diverse!
*p2=999; // questo non va bene
cout<<*p3<<endl; // potrebbe non stampare 'x'!!
```

Gestione della memoria

Problemi: doppia cancellazione

- Il problema nasce dal fatto che tipicamente il gestore delle risorse non è in grado di tenere traccia di quale parte di codice è proprietaria di una risorsa.
- Quindi il risultato di un doppio `delete` non è prevedibile

```
void sloppy(){
  int* p=new int[1000];
  // usa *p ...
  delete[] p;
  // ... aspetta un po' ...
  delete[] p;
  // la funzione non possiede *p!
}
```

- Tra il primo e il secondo `delete[]` la memoria può essere stata riutilizzata

Come fare?

Due suggerimenti

- Evitiamo di mettere oggetti sullo free store a meno che non sia strettamente necessario: meglio usare uno scope il più ristretto possibile
- Quando un oggetto viene costruito sullo free store, meglio inserirlo all'interno di un **gestore di oggetti** (handle) che preveda un distruttore
- Come regola pratica: usiamo `new` dove c'è un costruttore e `delete` dove c'è un distruttore

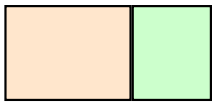
Prestazioni

Tempi non prevedibili

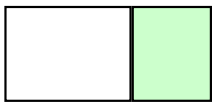
- Una gestione poco attenta dello heap (alternanza di `new` e `delete`) può portare alla sua **frammentazione**: la formazione di buchi nella memoria disponibile.
- Questi buchi sono troppo piccoli per poter essere utilizzati per allocare nuovi oggetti.
- L'effetto è che la memoria che resta disponibile è molto meno di quello che ci si aspettava.
- Ne consegue anche l'**aumento dei tempi** di esecuzioni di nuovi `new`: diventa sempre più difficile andare a cercare dove poter mettere il nuovo oggetto.
- Cerchiamo di capire come funziona questo meccanismo per cercare di prevenirlo

Esempio di frammentazione

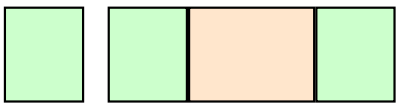
```
while(/* ... */){
  Msg *p = create();
  // ...
  Node *n1 = new Node;
  // ...
  delete p;
  Node *n2 = new Node;
  }
```



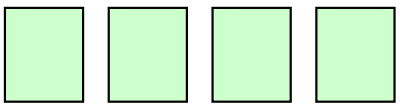
creato N1: 1 msg e 1 nodo



cancellato p: 1 “buco” e 1 nodo



1 buco



2 buchi

Possibili soluzioni

- Due alternative:
  - Un garbage collector per tappare i buchi
  - Il programmatore evita di crearli
- I puntatori rendono difficile creare un garbage collector
- Come possiamo evitare la formazione dei buchi?
- A volte basta riorganizzare la chiamata alle `new` e `delete`
- Ma non è una soluzione generale
- Prevenire: evitare usi del free store che provocano frammentazione
- Prima idea: evitiamo l'uso del `delete`, almeno non rallentiamo nuovi `new`, almeno nella maggior parte delle implementazioni, anche se non è garantito dallo standard.

Possibili soluzioni – continua

- Seconda idea: allochiamo tutta la memoria (statica e globale) all'inizio del programma e poi la usiamo.
  Svantaggi: struttura del programma non ottimale e uso di dati globali da evitare.
- Due strutture dati possono aiutarci in questo:

  Stack : visto che si allunga e si accorcia solo da una parte non può causare frammentazione

  Pool : una raccolta di oggetti della stessa dimensione. Essendo della stessa dimensione, non può esserci frammentazione.
- Con entrambe queste soluzioni, sia l'allocazione che la deallocazione hanno tempi prevedibili e veloci.