



UNIVERSITÀ DEGLI STUDI DI PADOVA

Liberare la memoria

Stefano Ghidoni



DIPARTIMENTO

DI INGEGNERIA

DELL'INFORMAZIONE



Agenda

- Memory leak e loro risoluzione
- Liberare la memoria
- Distruttore



Liberare la memoria

- È importante liberare la memoria quando non serve più
 - In C++ nessuno lo fa al posto nostro (nessun garbage collector!)
- La produzione di garbage è un **errore tecnico grave**
- La gestione corretta della memoria dinamica include la liberazione della memoria
- Forte differenza con Java (e altri linguaggi)



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Liberare la memoria

- Perché dobbiamo liberare la memoria manualmente?



Liberare la memoria

- Perché dobbiamo liberare la memoria manualmente?
 - Due punti importanti:
 - Efficienza: il garbage collector **impiega molte risorse**
 - Deve capire quale memoria è ancora raggiungibile
 - Controllo: decidiamo quando la memoria è rilasciata
 - perciò di nuovo disponibile per altro
- quindi ha una grande incombenza, che vuol dire utilizzare le risorse della macchina



Liberare la memoria

- Per liberare la memoria si usano istruzioni dedicate:
 - delete - se memoria allocata con new
 - delete[] - se memoria allocata con new[]
- Sia delete che delete[] si applicano ai puntatori



Caccia al memory leak

```
double* calc(int res_size, int max) {  
    double* p = new double[max];  
    double* res = new double[res_size];  
  
    return res;  
}  
  
// ...  
  
double* r = calc(100, 1000);  
  
// ... - ma nessuna deallocazione di r
```

- Dove sono i memory leak?



Caccia al memory leak

```
double* calc(int res_size, int max) {  
    double* p = new double[max];  
    double* res = new double[res_size];  
  
    return res;  
}  
// ...  
res viene distrutto, ma anche ritornato,  
quindi i dati non vengono persi
```

Necessario
liberare p

```
// ...  
double* r = calc(100, 1000);  
// ... - ma nessuna deallocazione di r
```

Necessario
liberare r

- Dove sono i memory leak?



Risolvere i memory leak

```
double* calc(int res_size, int max) {  
    double* p = new double[max];  
    double* res = new double[res_size];  
    delete[] p;  
  
    return res;  
}  
  
// ...  
  
double* r = calc(100, 1000);  
  
// ...  
delete[] r;
```



Passaggio di memoria tra funzioni

- Con l'allocazione dinamica della memoria possiamo creare un oggetto in uno scope (es., funzione) e passarlo a un altro scope (es., il chiamante)
 - Non esiste un meccanismo analogo a quello che libera la memoria di una variabile locale automatica
 - Ora è più chiaro perché si chiama *automatica*

Con le variabili locali automatiche è, appunto, automatica l'allocazione e la distruzione delle stesse una volta usciti dallo scope, mentre per un oggetto viene gestita da chi lo crea.

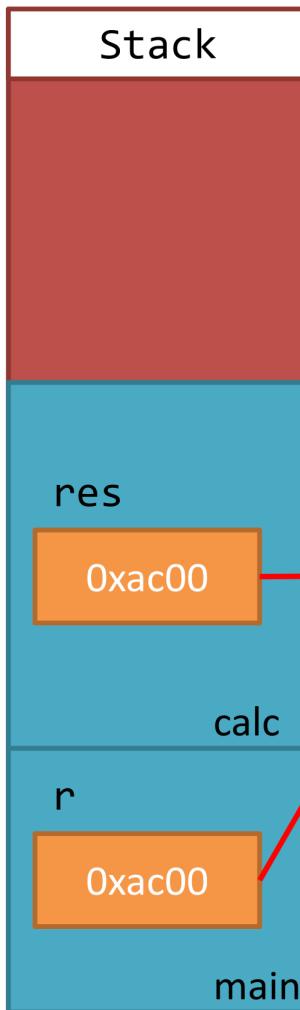


Passaggio di memoria tra funzioni

- Nel caso precedente: calc() **alloca res e lo ritorna al chiamante**
 - È copiato solo il puntatore, la memoria allocata rimane la stessa



Restituire un puntatore



```
double* calc(int res_size) {  
    double* res = new double[res_size];  
    return res;  
}  
  
int main()  
{  
    double* r = calc(1000);  
}
```



Restituire un puntatore

Stack



```
double* calc(int res_size) {  
    double* res = new double[res_size];  
    return res;  
}  
  
int main()  
{  
    double* r = calc(1000);  
  
    // ...  
    delete[] r;  
}
```



Deallocazione nel chiamante

- Un altro esempio

```
vector* f(int s) {
    vector* p = new vector(s);
    // fill p
    return p;
}

void pf() {
    vector* q = f(4);
    // use q
    delete q;
}
```

Solitamente è cattiva abitudine
allocare un vector con new



Errori frequenti

- Gestire la memoria tramite puntatori introduce varie sorgenti di errore
 - Doppia cancellazione
 - Dangling pointer (puntatore pendente/penzolante)



Doppia cancellazione

- È un **errore grave** deallocare due volte la memoria

```
int* p = new int{5};  
  
delete p;  
  
// ... - ma nessun uso di p  
  
delete p;
```

- p non è più a nostra disposizione – il proprietario è il free store manager
 - Potrebbe esserci un altro oggetto

Ad esempio, se nello stesso posto venisse scritto un dato da un altro thread, poi verrebbe cancellata memoria a cui non avevamo più accesso



Dangling pointer

- Dopo delete, il puntatore mantiene lo stesso valore
- Tale valore non è più valido ed è un errore utilizzarlo
- Questa situazione prende il nome di dangling pointer (*puntatore pendente/penzolante*)



Dangling pointer

- Per evitare il dangling pointer è importante settare il puntatore a `nullptr`

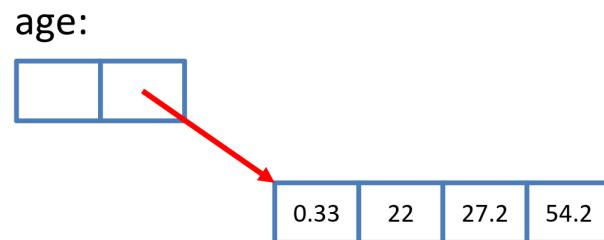
```
int* p = new int{5};  
//...  
delete p;  
p = nullptr;
```

Questo risolve anche il problema della doppia deallocazione, infatti la seconda "delete p" sarebbe completamente lecita e non creerebbe problemi logici.



Liberare la memoria con UDT

- Torniamo al nostro esempio di `vector<double>`



- A fine vita, è necessario deallocare la memoria
- Potrei usare una funzione `clean_up()`
 - Se l'utente si dimentica di chiamarla?
- Come possiamo fare per essere *sicuri* che ciò avvenga in maniera *automatica*?



Distruttore

- Un **distruttore** è una funzione che il compilatore chiama quando un oggetto deve essere distrutto
 - Esce dallo scope
 - Deallocazione di un oggetto
- Il distruttore è chiamato **implicitamente**



Distruttore di vector

```
class vector{
    int sz;
    double* elem;
public:
    vector(int s)
        : sz{ s }, elem{ new double[s] } {
    {
        for (int i = 0; i < s; ++i) elem[i] = 0;
    }
    ~vector()
    { delete[] elem; } // ...
};
```

la sintassi per il distruttore (che è ***uno*** e ***non*** ha argomenti)

Distruttore



Chiamata implicita del distruttore

```
void f3(int n){  
    double* p = new double[n];  
    vector v(n);  
    // ... uso p e v ...  
    delete[] p;  
}
```

v è liberato
automaticamente!

- vector fa automaticamente ciò che con p
dobbiamo fare manualmente



Chiamata implicita del distruttore

- I distruttori sono molto utili quando dobbiamo rilasciare risorse acquisite
 - File
 - Thread
 - Lock
 - Stream
 - ...



Distruttori creati dal compilatore

- Quando il distruttore non è implementato esplicitamente, è generato automaticamente dal compilatore (**compiler-generated destructor**)
 - Non è "intelligente": non dealloca al posto nostro
 - Chiama i distruttori di eventuali oggetti membro



Distruttori creati dal compilatore

```
struct Customer {  
    std::string name;  
    std::vector<string> address;  
    // ...  
};  
  
void some_fct() {  
    Customer Fred;  
    // init Fred  
    // use Fred  
}
```

Qui Fred è distrutto:

- Chiamata a distruttore di string
- Chiamata a distruttore di vector<string>



Pattern acquisizione risorse

- Un oggetto acquisisce risorse nel costruttore
- Durante la sua vita, l'oggetto può:
 - Acquisire altre risorse
 - Liberare alcune risorse
- A fine vita, l'oggetto rilascia tutte le risorse



Azioni di new e delete

- Operatore new
 - Alloca memoria
 - Invoca costruttore
- Operatore delete
 - Invoca distruttore
 - Libera memoria



- Allocare dinamicamente offre **vantaggi**:
 - Accedere a grandi quantitativi di memoria
 - Ottenere blocchi di memoria di dimensione non nota a tempo di compilazione
 - Controllare quando la memoria è allocata e deallocata



- Allocare dinamicamente offre **svantaggi**:
 - Gestione esplicita della deallocazione
 - Gestione della memoria tramite puntatori
 - Sorgente di molti errori subdoli
 - Maggior carico computazionale: allocazioni e deallocazioni sono gestite tramite **chiamate al sistema operativo**
 - Queste chiamate comportano un onere computazionale non trascurabile



- L'utilizzo dell'allocazione dinamica deve essere giustificata da un motivo
 - Dovete sapere perché è necessario allocare dinamicamente anziché usare altri strumenti
- **È un errore di design utilizzarla a sproposito**



Recap

- Deallocare la memoria
- Caccia al memory leak
- Passaggio di memoria dinamica tra funzioni e deallocazione nel chiamante
- Doppia cancellazione
- Distruttore



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Liberare la memoria

Stefano Ghidoni



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE