04_Lifetime

Table of contents

Lifetime

Potrebbe capitare che a livello di tempo d'esecuzione del programma, un certo nome non sia stato ancora creato oppure sia già stato distrutto. Alcune entità hanno tempo di vita, o **lifetime** diverso:

- entità come tipi di dato, funzioni, etichette, possono essere riferite in qualunque momento durante l'esecuzione del programma;
- un oggetto memorizzato in memoria è utilizzabile solo dopo che è stato creato.

Oggetti che **nascono** e **muoiono**, sono creati in memoria:

- da una definizione (ricordando che, dichiarazione pura $\neq$ creazione);
- da una chiamata dall'espressione `new` (inserito in heap, senza nome e quindi senza campo d'azione);
- dalla valutazione di una espressione che crea implicitamente un nuovo oggetto (acnh'esso temporaneo, senza nome)

Il tempo di vita quando inizia e quando finisce?

- Quando *termina* la sua *costruzione* con successo, inizia.
  Un'analogia: costruire l'enciclopedia "Treccani" non è immediato, ci sono centinaia di volumi e per crearli ci vuole del tempo; solo una volta costruiti, esistono effettivamente.
  1. viene allocata memoria "grezza", abbiamo la memoria ma dentro non c'è niente di rilevante;
  2. inizializzazione della memoria (quando prevista)
- Quando *inizia* la *distruzione* con successo, finisce.
  Le 2 fasi di ordinamento sono diverse rispetto la costruzione. Un'analogia: demolizione di un edificio e rimozione delle macerie (distruzione), lasciando soltanto il terreno (memoria "grezza").
  1. viene invocato il distruttore (quando previsto);
  2. distrutta la memoria "grezza"

Altre note sempre in contesto:

- Un oggetto a cui accade qualcosa di brutto durante la sua costruzione, non subirà distruzione, siccome non c'è niente da distruggere siccome la costruzione non è andata a buon fine. Questo è molto importante da tenere in mente, per un uso corretto di risorse nei programmi.
- Una distruzione non andata a buon fine possiamo a volte gestirla meglio siccome possiamo ignorare una porzione di codice che nel caso fallisca, "lascia perdere". Se lo stato del sistema viene compromesso, non possiamo risolvere e il sistema va in undefined behaviour.
- Se siamo durante la fase di costruzione/distruzione, siamo in una specie di limbo: qualcosa c'è durante le fasi, ma l'oggetto non esiste effettivamente.

Allocazione statica

L'allocazione avviene sotto il controllo del compilatore, la memoria permane per tutto il tempo d'esecuzione del programma:

- le variabili a `namespace` scope (globali) sono create e inizializzate *prima della funzione* `main()` nell'ordine in cui si trovano (definite) se nello stesso file, nell'ordine non stabilito se presenti in unità di traduzione diversa

```
#include <iostream>
struct S {
  S() { std::cout << "costruzione" << std::endl; }
  ~S() { std::cout << "distruzione" << std::endl; }
};
S s; // allocazione globale (unspecified behaviour per 'std' e simili)
int main() {}
```

- i dati membro delle classi dichiarate usando la parola chiave `static`, prendono vita prima, come le variabili globali;

- le variabili locali, se dichiarate con `static`, sono allocate staticamente e quindi l'allocazione della memoria "grezza" avviene prima dell'esecuzione ma non l'inizializzazione, che avviene la prima volta in cui noi entriamo nella funzione

```
// stampa il numero di volte che la funzione 'foo()' viene chiamata
#include <iostream>

struct S {
  int counter;
  S() : counter(0) { }
  ~S() { std::cout << "counter = " << counter << std::endl; }
};

void foo() {
  static S s; // allocazione locale statica
  ++s.counter;
}

int main() {
  for (int i = 0; i < 10; ++i) {
    foo();
  }
}
```

Allocazione thread local

Vorrei creare una variabile, visibile per solo un preciso thread.
La variabile nasce quando il thread viene creato, muore quando finisce l'esecuzione del thread.

Allocazione automatica

Non è l'allocazione a essere esattamente automatica, il termine è fuorviante: è automatica la de-allocazione, ovvero quando finiamo. L'oggetto viene creato a tempo d'esecuzione sullo stack di sistema ogni volta che si entra nel blocco, l'oggetto viene distrutto e lo spazio liberato, ogni volta che usciamo dal blocco.

```
void foo() {
  int a = 5;
  {
```

```
    int b = 7;
    std::cout << a + b;
  } // b viene distrutta automaticamente all'uscita da questoi blocco
  std::cout << a;
} // a viene distrutta automaticamente all'uscita da questo blocco
```

</> Esempio di calcolo fattoriale

Ogni volta che entriamo nella funzione `fact(int n)`, creiamo un nuovo *record d'attivazione* per questa: viene tenuta traccia della variabile `n`.

```
int fact(int n) {
  if (n == 0)
     return 1;
  else
      return n * fact(n-1);
}
int main() {
  return fact(5);
}
```

Allocazione automatica dei temporanei

Qual è il tempo d'esecuzione degli oggetti temporanei?
Viene allocato un temporaneo, automaticamente, per il solo scopo di eseguire una funzione. Al termine verrà fatta la distruzione automatica.

```
struct S {
  S(int);
  S(const S&);
  ~S() { std::cout << "distruzione"; }
};

void foo(S s);

void bar() {
  foo(S(42)); // allocazione di un temporaneo per S(42)
  std::cout << "fine";
}
```

Il lifetime può essere esteso, quando per esempio viene utilizzato per inizializzare un *riferimento*.

```
void bar2() {
  // il temporaneo S(42) è usato per inizializzare il riferimento s
  const S& s = S(42);
  std::cout << "fine";
  // il temporaneo è distrutto quando si esce dal blocco,
  // dopo avere stampato "fine"
}
```

Allocazione dinamica

In un programma dobbiamo allocare memoria, senza sapere quanto ce ne serva effettivamente. Viene fatta nell'*heap* (memoria dinamica), dove tutto è frammentato.
Usando l'espressione `new` creiamo un oggetto nell'heap.

```
// crea prima l'intero '42' e vi associa la variabile puntatore 'pi', che punta all'intero
int* pi = new int(42);
```

La de-allocazione del puntatore avviene automaticamente, l'oggetto puntato non ha scope ma ha ciclo di vita: inizia alla fine della costruzione, finisce quando lo esplicitiamo noi, con l'espressione `delete`.

```
// distrugge l'oggetto puntato da 'pi', diventa un dangling pointer
delete pi;
```

L'allocazione dinamica è sorgente di errori numerosi:

- "use after free"
  dopo avere eliminato la memoria, nulla viene puntato dal nostro puntatore
- "double free"
  usare la `delete` più volte del necessario sulla stessa risorsa che non c'è più, causa eliminazione di memoria che magari non centra nulla
- "memory leak"
  il puntatore viene eliminato prima che la memoria puntata sia distrutta, l'oggetto non verrà mai più distrutto, causando come minimo uno spreco di memoria heap
- "wild pointer"
  assomiglia molto alla "use after free", la differenza è che siccome ci viene fornita *aritmetica sui puntatori*, potremmo "sforare" la memoria che viene puntata e puntare a qualcos'altro
- "null pointer"
  tentativo di accesso a puntatore nullo, che non possiamo deferenziare

---

metodologie > CODE > Allocation > BrokenFactorial.cpp
metodologie > CODE > Allocation > DestructorLeak.cpp

01/03/2023