Tempo di vita (lifetime)

Abbiamo visto come il concetto di scope consenta di stabilire in quali punti del codice (*dove*) è visibile il nome introdotto da una dichiarazione. Oltre alla dimensione spaziale, è necessario prendere in considerazione anche la dimensione temporale, per stabilire in quali momenti (*quando*) è legittimo interagire con determinate entità.

Alcune entità (tipi di dato, funzioni, etichette) possono essere riferite in qualunque momento durante l'esecuzione del programma.
Un oggetto memorizzato in memoria, invece, è utilizzabile solo dopo che è stato creato e soltanto fino a quando viene distrutto, ovvero ogni accesso è valido solo durante il suo tempo di vita (lifetime).

Note iniziali

Anche se il codice eseguibile delle funzioni è memorizzato in memoria, tecnicamente le funzioni *NON* sono considerate oggetti in memoria e quindi non se ne considera il lifetime.

Il *tempo di vita* di un oggetto è influenzato dal modo in cui questo viene creato.
Gli oggetti in memoria sono creati:

- da una *definizione* (non basta una dichiarazione pura);
- da una *chiamata dell'espressione* `new` (oggetto nell'heap, senza nome);
- dalla *valutazione di una espressione* che crea implicitamente un nuovo oggetto (oggetto temporaneo, senza nome).

Il tempo di vita di un oggetto:

1. inizia quando termina la sua costruzione, che è composta da due fasi distinte:
   - allocazione della memoria "grezza";
   - inizializzazione della memoria (quando prevista);
2. termina quando inizia la sua distruzione, che è anch'essa composta da due fasi distinte:
   - invocazione del distruttore (quando previsto);
   - deallocazione della memoria "grezza".

   Si noti che un oggetto la cui costruzione NON termina con successo, NON avendo iniziato il suo tempo di vita, NON dovrà terminarlo, ovvero per quell'oggetto NON verrà eseguita la distruzione.

   Si noti anche che DURANTE le fasi di creazione e di distruzione di un oggetto si è fuori dal suo tempo di vita, per cui le operazioni che è possibile effettuare sull'oggetto sono molto limitate (le regole del linguaggio sono complicate e comprendono numerosi casi particolari, che per il momento non è opportuno approfondire).

Torna all'indice

Storage duration ("Allocazione")

Vi sono diversi tipi di *storage duration* (aka allocazione) per gli oggetti in memoria:

1. Allocazione statica
2. Allocazione thread local
3. Allocazione automatica
4. Allocazione automatica di temporanei
5. Allocazione dinamica

Torna all'indice

Allocazione statica

La memoria di un oggetto ad allocazione statica dura per tutta l'esecuzione del programma.

Sono dotate di memoria ad allocazione statica:

1. Le variabili definite a namespace scope (dette globali). Queste sono create e inizializzate *prima* di iniziare l'esecuzione della funzione main, nell'ordine in cui compaiono nell'unità di traduzione in cui sono definite.

   Nel caso di variabili globali definite in diverse unità di traduzione, l'ordine di inizializzazione *NON* è specificato.

```
// Il seguente programma stampa due stringhe,
// anche se il corpo della funzione main è vuoto

#include

struct S {
 S() { std::cout << "costruzione" << std::endl; }
 ~S() { std::cout << "distruzione" << std::endl; }
};

S s; // allocazione globale

int main() {}
```

2. I dati membro di classi dichiarati usando la parola 'static'. Questi sono creati come le variabili globali del punto precedente.

3. Le variabili locali dichiarate usando la parole chiave 'static'. Queste sono allocate prima di iniziare l'esecuzione della funzione `main`, ma sono inizializzate (solo) la prima volta in cui il controllo di esecuzione incontra la corrispondente definizione (nelle esecuzioni successive l'inizializzazione non viene eseguita).

```
// Il seguente programma stampa una stringha,
// indicando il numero totale di volte in cui
// la funzione `foo()` è stata chiamata:

#include

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();
 }
}
```

Torna all'indice

Allocazione thread local

Un oggetto thread local è simile ad un oggetto globale, ma il suo ciclo di vita non è collegato al programma, bensi ad ogni singolo thread di esecuzione creato dal programma (esiste una istanza distinta della variabile per ogni thread creato). Il supporto per il multithreading è stato introdotto con lo standard C++ 2011.

Torna all'indice

Allocazione automatica

Una variabile locale ad un blocco di funzione è dotata di allocazione automatica: l'oggetto viene creato dinamicamente (sullo *stack*) ogni volta che il controllo entra nel blocco in cui si trova la dichiarazione e viene automaticamente distrutto (rimuovendolo dallo stack) ogni volta che il controllo esce da quel blocco.

```
void foo() {
 int a = 5;
 {
     int b = 7;
     std::cout << a + b;
 } // 'b' viene distrutta automaticamente all'uscita da questo blocco
 std::cout << a;
} // 'a' viene distrutta automaticamente all'uscita da questo blocco
```

> Nel caso di funzioni ricorsive, sullo stack possono esistere contemporaneamente più istanze distinte della stessa variabile locale.

Torna all'indice

Allocazione automatica di temporanei

L'allocazione automatica di temporanei avviene quando un oggetto viene creato per memorizzare il valore calcolato da una sottoespressione che compare all'interno di una espressione.

```
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";
}
```

L'oggetto temporaneo viene *distrutto* quando termina la valutazione dell'espressione completa che contiene lessicalmente il punto di creazione. Nell'esempio precedente, il temporaneo è distrutto al termine dell'esecuzione di `foo`, ma prima della stampa di *"fine"*.

> Il lifetime di un oggetto temporaneo può essere esteso se l'oggetto viene utilizzato per inizializzare un riferimento; in tale caso, l'oggetto verrà distrutto nel momento in cui verrà distrutto il riferimento.
> Per esempio:

```
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"
}
```

Torna all'indice

Allocazione dinamica

Un oggetto (senza nome) può essere allocato dinamicamente nella memoria heap usando l'espressione `new` (che restituisce l'indirizzo dell'oggetto allocato, che va salvato in un opportuno puntatore).

```
int* pi = new int(42);
// pi contiene l'indirizzo di un oggetto int di valore 42.
```

La distruzione dell'oggetto NON è automatica, ma viene effettuata sotto la responsabilità del programmatore utilizzando l'istruzione `delete` (sul puntatore che contiene l'indirizzo dell'oggetto).

```
delete pi;
// l'oggetto puntato da pi è stato distrutto,
// ma pi continua a contenere il suo indirizzo, non più valido;
// pi è diventato un "dangling pointer" (puntatore penzolante)
```

L'allocazione dinamica è una sorgente inesauribile di errori di programmazione:

- Errore "use after free": Usare (per leggere o scrivere sull'oggetto puntato) un puntatore dangling. In pratica si usa un oggetto dopo che il suo lifetime è concluso.
- Errore "double free": Usare la delete due o più volte sullo stesso indirizzo, causando la distruzione di una porzione di memoria che non era più allocata (o era stata riutilizzata per altro).
- "Memory leak": Si distrugge l'unico puntatore che contiene l'indirizzo dell'oggetto allocato prima di avere effettuato la delete (quindi l'oggetto non verrà mai più distrutto, causando come minimo uno spreco di memoria).
- Accesso ad un "wild pointer": Variante della use after free; si segue un puntatore che indirizza memoria "causale", leggendo o scrivendo dove non c'è un oggetto (o non c'è l'oggetto inteso).
- Accesso al "null pointer": Si prova ad accedere ad un puntatore nullo.

Torna all'indice