Lezione3

Table of contents

- Lifetime
    1. Allocazione statica
    2. Allocazione thread local
    3. Allocazione automatica
    4. Allocazione automatica dei temporanei
    5. Allocazione dinamica
- Tipi
    1. Fondamentali
    2. Composti
    3. Qualificati
        1. `const`
        2. `volatile`
        3. Costanti letterali
            1. Letterali definiti da utente

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

Tipi

Fondamentali

- Tipi booleani
  `bool`
- Tipi carattere
  `char`, `signed char`, `unsigned char` → narrow character type
  `wchar_t`, `char16_t`, `char32_t` → wide character type
- Tipi interi standard con segno
  `signed char`, `short`, `int`, `long`, `long long`
- Tipi interi standard senza segno
  `unsigned char`, `unsigned short`, `unsigned int`
- Tipi floating point
  `float`, `double`, `long double`
- Tipo `void`
  ha insieme vuoto di valori, per indicare che una funzione non ritorna alcun valore o, usando un cast esplicito, che il valore di una espressione deve essere scartato

  ```
  (void) foo(3); // chiama foo(3) e scarta il risultato qualunque esso sia
  ```

- Tipo `std::nullptr_t`
  tipo puntatore convertibile implicitamente in qualunque altro tipo puntatore, l'unico valore che può assumere è `nullptr`, il puntatore nullo

Composti

- `T&` riferimento a lvalue, dove possiamo scrivere un valore
- `T&&` riferimento a rvalue, riferimento a cosa può stare soltanto a destro di assegnamento (come `a=5`, dove `5` è un valore che non può stare a sinistra)
- `T*` puntatore
- `T[n]` tipi array, dove `n` costante intera valutabile a tempo di compilazione
- `T(T1, ..., Tn)` tipi funzione
- enumerazioni e classi/struct

Qualificati

`const`

L'oggetto può essere accesso solo per lettura e mai scrittura.
Fornito tipo `T`, possiamo fornire la versione qualificata tramite `const T`, stando attenti quando `T` è composto:

```
struct S {
  int v;
  const int c;
  S(int cc) : c(cc) { v = 10; } // creiamo l'oggetto e inizializziamo
  // lista inizializzione classi basi e membro, 'v' viene creato ma non inizializzato
};

int main() {
  const S sc(5)
  sc.v = 20; // errore: sc è const e anche le sue componenti

  S s(5);
  s.v = 20; // legittimo: s non è const e S::v non è const
  s.c = 20; // errore: s non è const, ma S::c è const
}
```

Lo stesso oggetto può essere modificato a seconda del modo usato per accedervi:

```
struct S { int v; };
void foo() {
  S s;
  s.v = 10; // legittimo
  const S& sr = s; // riferimento a s, qualificato const
  sr.v = 10; // errore: non posso modificare s passando da sr.
}
```

I riferimenti all'oggetto `const` deve essere anche lui stesso un `const`, perché altrimenti si violerebbe la regola base.

`volatile`

Costanti letterali

Varie sono le disposizioni per definire valori costanti; al valore viene associato un tipo specifico che in alcuni casi dipende dall'implementazione.

- `bool`
  false, true
- `char`
  'a', '3', 'Z', '\n' (ordinary character literal)
  u8'a', u8'3' (UTF-8 character literal)
- `signed char`, `unsigned char`:
  (-)
- `char16_t`
  u'a', u'3' (prefisso case sensitive)
- `char32_t`
  U'a', U'3' (prefisso case sensitive)
- `wchar_t`:
  L'a', L'3' (prefisso case sensitive)
- `short`, `unsigned short`
  (-)
- `int`
  12345

In assenza di suffissi, viene attribuito il primo tipo tra `int`, `long` e `long long` che sia capace di rappresentare il valore. I suffissi sono intesi come:

- `U` per `unsigned` (può comparire insieme agli altri);
- `L` per `long` e `long long`;
- `LL` per `long long`

```
12345L
12345LL
12345U
12345ULL
```

Per i floating point, abbiamo la decisione tra decimale o "scientifica":

```
123.45F // float
1.2345e2F // float

123.45 // double
1.2345e2 // double

123.45L // long double
1.2345e2L // long double
```

Per `void` non c'è alcuna costante letterale.
Per `nullptr_t` sarebbe `nullptr`.
Per i letterali di stringa, come la stringa `"Hello"`, sarebbe un array di 6 caratteri (con il delimitatore, `char[6]`); per le stringhe "grezze", usiamo dei delimitatori per creare sequenze di caratteri per non poi preoccuparci di fare "escaping" dei caratteri.

```
// dentro a 'string' possiamo scrivere di tutto
R"DELIMITATORE(string)DELIMITATORE"
```

Letterali definiti da utente

Possibili letterali definiti da utente: il letterale da noi definito avrà lo scopo di invocare una funzione di conversione implicita, anche lei definita da noi.
Un esempio sono le stringhe stile C++ 2014 con il suffisso `s`, per indicare una conversione implicita da stringhe di tipi C, a stringhe tipo `std::string`.

```cpp
#include <iostream>
#include <string>

int main() {
  using namespace std::literals;
  std::cout << "Hello";  // stampa la stringa C   (tipo const char[6])
  std::cout << "Hello"s; // stampa la stringa C++ (tipo std::string)
}
```

28/02/2023