Programmare a oggetti

- Classe
- Costruttori e distruttore
- Copia di oggetti
- Spostamento (move)
- Creazione di oggetti
- Metodi inline, const, statici
- Overloading degli operatori e funzioni friend
- Codice di esempio
  - https://github.com/egalli64/corso-cpp folder b5

Classe

- È il modello di riferimento usato dal compilatore per creare un oggetto
- Le keyword **struct** e **class** sono quasi equivalenti
- Si preferisce struct per le classi più semplici (es: DTO)
  - Raggruppamento di dati coerenti, per default pubblicamente acessibili
- Nell'uso normale si preferisce class
  - Tipicamente contiene dati e funzionalità, per default non accessibili esternamente
    - data member *(proprietà, campo, attributo, …)*, metodo *(functional member)*
- Dato un oggetto, per accedere ai suoi membri (dati o metodi che siano) si usa
  - L'operatore ".", a partire dall'oggetto o un suo reference
  - L'operatore "->", per un puntatore

Accesso ai class member

- Classi (e strutture) possono essere divise in aree con diversa visibilità d'accesso ai suoi membri
- **private**, ristretta alla classe corrente (è il default per *class*)
  - Preferita per i data member (data hiding)
- **public**, libera a tutti (è il default per *struct*)
  - Preferita per i metodi, interfaccia tra classe e mondo esterno
- **protected**, limitata alla classe corrente e derivate
  - Può essere usata all'interno di gerarchie di classi

Costruttore e distruttore

- Il metodo costruttore (**ctor**) inizializza un nuovo oggetto, ha lo stesso nome della classe, non ha un return type
  - Un ctor se **explicit** non è invocabile implicitamente, richiede l'uso esplicito (anche via static cast)
- Il **default ctor** ha la lista dei parametri vuota, implicitamente fornito dal compilatore se non ce ne sono altri
  - Può essere forzatamente rimosso via "**= delete**", o generato via "**= default**"
- La lista di inizializzazione si trova dopo la signature del ctor, tra ":" e la graffa aperta del body
  - Permette di inizializzare i campi dell'oggetto, approccio obbligatorio nel caso di campi costanti
    - Nel body del ctor si assegna un valore al data member, non è inizializzazione
  - *(e, nel caso di ereditarietà, invocare i ctor delle eventuali classi base)*
- Il nome del parametro è di solito simile o uguale al nome del campo corrispondente, ad esempio:
  - Se il parametro ha un nome (es: "x") il campo ha un underscore postifisso al nome (es: "x_")
  - Se hanno entrambi lo stesso nome, nel body, si usa il puntatore "this" sul campo per risolvere l'ambiguità
    - Nella lista di inizializzazione la sintassi x { x } non è ambigua
- Il distruttore (**dtor**) è il metodo speciale che viene invocato al termine della vita dell'oggetto
  - Ha lo stesso nome della classe, prefissato da tilde ~, non ha return type
  - Implicitamente fornito dal compilatore se non fornito esplicitamente

Copia di oggetti

- Shallow copy
  - Semplice, economica
  - Crea un nuovo accesso all'oggetto
  - Un reference a un oggetto è una sua shallow copy
- Deep copy
  - Più sicura ma di solito più complessa e costosa
  - Crea una copia indipendente
  - In ogni classe il compilatore genera di default
    - Il **copy ctor**, per l'inizializzazione di un oggetto per copia, es: A(const A& other)
    - L'**operatore di assegnamento**, es: A& operator=(const A& other)
  - Vanno ridefiniti entrambi (assieme al dtor) se l'oggetto gestisce internamente risorse

Spostamento (move)

- Passaggio della responsabilità di gestione di una risorsa ad un altro oggetto
- Il “move” permette di ottenere il risultato a costo molto ridotto
- Invece di fare una "deep copy", e poi eliminare il vecchio oggetto, si fa una "shallow copy"
  - La risorsa posseduta dal vecchio oggetto viene passata al nuovo
    - Spesso è memoria allocata sullo heap, il campo che richiede il “move” è un puntatore
  - Pratica comunemente usata negli aggregati, la vediamo nei container della libreria standard C++
- Uso di uno specifico elemento sintattico, **&&** *(right hand side reference)*
  - L'oggetto "other" deve essere alla destra di un assegnamento, e diventerà "vuoto"
  - **Move ctor** per la classe A: A(A&& other);
  - **Move assigment** operator: A& operator=(A&& other);
- Per forzarne l'uso: **std::move**(), invece di usare un reference, usa un rhs reference

Creazione di oggetti

- Gli oggetti basati su classi possono essere allocati sullo stack o sullo heap
  - Tutto ciò che è creato via operatore **new** va distrutto via **delete**
    - E lo stesso per new [] / delete []
- Le funzioni per la gestione della memoria del C sono disponibili anche in C++
  - malloc - calloc - realloc / free
  - Sono usate solo se necessario, di solito solo per blocchi di memoria "raw"
    - Non conoscono i meccanismi C++ di ctor e dtor
    - Se si deve usare questo approccio, va tenuto ben distinto da quello proprio del C++
- Gli smart pointer hanno lo scopo di semplificare l'uso di oggetti nello heap

Metodi inline, const, statici

- **inline**, si chiede al compilatore di rimuovere l'overhead della chiamata al metodo
  - Il codice associato viene espanso al posto dell'invocazione del metodo
    - Il compilatore decide se seguire l'indicazione, in base alle sue regole di ottimizzazione
  - Un metodo definito in una classe è implicitamente inline
  - *(Da C++17 le cose sono più complesse e anche le variabili possono essere dichiarate inline)*
- **const** dichiara che un metodo di istanza non modifica l'oggetto corrente
  - Su un oggetto costante, o acceduto in modo costante, possono essere invocati solo metodi costanti
- **static**, un membro statico non è legato a una particolare istanza della classe
  - Un data member statico rappresenta quindi lo stato della classe, non di una istanza
  - Un metodo statico non può accedere direttamente ai membri non statici della classe
  - Si accede a un membro statico via operatore di risoluzione di scopo ::
    - Modalità obbligatoria se non c'è a disposizione un oggetto del tipo richiesto
    - Altrimenti è possibile usare un operatore di dereferenziamento come per i metodi di istanza

Operator overloading + friend

- Dopo aver creato una classe, si possono ridefinire operatori su di essa
  - Non se ne può cambiare né la cardinalità né precedenza, non se ne possono creare di nuovi
  - Non tutti sono ridefinibili, ad esempio l'operatore ternario "?:" e quello di dereferenziazione "."
  - Dovrebbe fornire alla classe una funzionalità simile a quella dell'operatore standard, o almeno intuitiva
- Tra gli operatori overloaded comunemente utilizzati
  - Operatore **<<** (put-to), usato comunemente per gli stream di output definisce come un oggetto debba essere serializzato
  - Operatore **()**, permette ad un oggetto di operare come una funzione, e viene perciò chiamato **Function Object** (o Functor)
  - Operatori **++** e **--**, sia in modalità prefissa sia postfissa (usando un secondo parametro di tipo int)
  - Operatori "type", per specificare come l'oggetto possa essere convertito ad un altra classe
    - Reciproco di un ctor di conversione, utile se non abbiamo accesso alla classe target
- Funzioni **friend**
  - Una funzione non membro di una classe che ha accesso completo alla classe
  - Utile nel caso di operatori che operano su una certa classe ma non possono esserne membro
  - Esempio, "<<" da un oggetto a uno stream, se richiede proprietà normalmente non accessibili
  - "os << x" è interpretato come os.operator<<(x), quindi dovrebbe essere definito in ostream