API - Lab #1 30 / 4 /2020

Obiettivi

- Setup ambiente di sviluppo C++
- Organizzazione codice e scrittura CMakefile
- Familiarizzazione con classi C++ e definizione di API attraverso l'interfaccia di una classe
- Gestione della memoria: utilizzo dei vari tipi di costruttori e operatori di assegnazione
- Ottimizzazione dell'uso della memoria
- Verifica che il codice sia "safe" dal punto di vista rilascio corretto della memoria

Preparazione ambiente di sviluppo

Durante queste esercitazioni consigliamo di utilizzare come IDE CLion, free per gli studenti universitari.
In questo link trovate la guida per l'installazione nelle varie piattaforme:
https://www.jetbrains.com/help/clion/clion-quick-start-guide.html

CLion può essere utilizzato con vari compilatori (clang, gcc, microsoft visual C)
Inoltre è integrato con CMake che vi permette di semplificare il build di progetti con più file di codice.

CMake è uno strumento che consente di costruire un Makefile attraverso semplici direttive incluse in file chiamato **CMakeLists.txt** da inserire nella radice del progetto.
CLion gestisce per voi questo file quando aggiungete nuove classi al progetto, ma è possibile anche la gestione manuale.
Ecco un esempio minimale che compila due file di codice main.cpp e class1.cpp e li collega (link) nell'eseguibile myprogram

```
cmake_minimum_required(VERSION 3.0.0)
project(myproject VERSION 0.1.0)
set(CMAKE_CXX_STANDARD 17)
add_executable(myprogram main.cpp class1.cpp)
```

Per ottenere il Makefile, lanciare "cmake ." dalla cartella, che genererà il Makefile corrispondente.
Potete poi ottenere il vostro programma invocando "make"

Organizzazione del codice

Per una migliore manutenzione dei progetti è prassi suddividere il codice in vari file utilizzando questo criterio:

- main.cpp → contenente il main e la logica principale del programma
- classname.h → un file di definizione per ogni classe
- classname.cpp → l'implementazione delle singole classi

Qui trovate un esempio dettagliato:
https://www.learncpp.com/cpp-tutorial/89-class-code-and-header-files/

Problema

Un sistema di ricezione riceve dei messaggi di lunghezza variabile, contenti un id, un buffer di dati e la rispettiva lunghezza.
Realizzare una classe che consenta di gestire i messaggi in arrivo e una seconda classe che permetta di memorizzarli.

Prima parte: gestione messaggi

La classe Message è così definita:

```
class Message {
   long id;
   char *data;
   int size
public:
      …
   //implementare qui quanto necessario per il suo funzionamento
}
```

Per simulare il messaggio in arrivo usare questa funzione che crea un buffer lungo n è lo riempie con delle sillabe consonante/vocale. Aggiungerla come metodo privato della classe
NB: questo metodo alloca nuova memoria, è responsabilità della classe stessa assicurarsi che venga rilasciata in modo corretto!

```
#include <string>

char* mkMessage(int n){
    std::string vowels = "aeiou";
```

```
    std::string consonants = "bcdfghlmnpqrstvz";
    char* m = new char[n+1];
    for(int i=0; i<n; i++){
        m[i]= i%2 ? vowels[rand()%vowels.size()] :
                    consonants[rand()%consonants.size()];
    }
    m[n] = 0 ;
    return m;
}
```

Creare un costruttore che data una lunghezza “n” inizializzi un nuovo oggetto di tipo Message con un messaggio di lunghezza corrispondente; ogni nuovo messaggio deve avere un id progressivo univoco (hint: usare un attributo privato statico, inizializzato a 0 e incrementarlo ogni volta che viene costruito un nuovo oggetto)

```
Message m1(10);
Message m2(20);
```

Per poter stampare il contenuto di un Message, occorre **dichiarare** (cioè fare sapere al compilatore che esiste - tipicamente in un file “.h”) e **definire** (ovvero indicare come è fatta, tipicamente in un file “.cpp”) la seguente funzione chiamata “operator<<(...)”. Tale funzione deve essere esterna alla classe

```
std::ostream& operator<<(std::ostream& out, const Message& m){
      // accedere alle parti contenute nell’oggetto Message e
      // stamparle utilizzando    out << … ;
      // Questa funzione DEVE restituire l’oggetto out
      return out;
}
```

Esempio di uso:

```
Message m1(10);
Message m2(20);
std::cout << m1 << std::endl << m2 << std::endl;

//stampa:
//[id:0][size:10][data:ceracasice]
//[id:1][size:20][data:dimuzesedefoludavoca]
```

Implementare a questo punto i seguenti metodi di Message:

- costruttore di default (crea un messaggio vuoto con id=-1)
- costruttore di copia
- costruttore di movimento
- operatore di assegnazione
- operatore di assegnazione per movimento
- distruttore
- metodi di accesso in sola lettura ai dati contenuti nel messaggio (tali metodi devono essere dichiarati “const”: perché?)

Per ogni metodo, aggiungere all’inizio anche un linea che stampi il suo nome per debug, in modo da capire quando viene invocato. Es:

```
if constexpr (debug) cout<<”Message::copy constructor”<<endl;
```

A questo punto scrivere degli use case che simulino l’invocazione di tutti i metodi implementati e verificare che siano corretti
Verificare sempre anche che il distruttore venga sembra invocato (usare la delete dove necessario)

1. Che differenza c’è tra queste due costruzioni di un array di Message?
2. Quale costruttore viene invocato?
3. E’ necessario fare la delete dei buffer?

```
Message buff1[10];
Message *buff2 = new Message[10];
```

4. Cosa capita quando assegno un nuovo messaggio ad un elemento dell’array?
5. Cosa capita se commento il costruttore di default?
6. Se lo rendo privato?

```
buff1[0] = Message(100);
```

Provare l’esempio precedente commentando gli operatori di assegnazione.

7. Compila?
8. E’ tutto corretto?
9. Disegnate su un foglio i blocchi di memoria allocati e simulare graficamente la copia

Ora riempire buff1 con 10 messaggi lunghi 1MB e poi copiarli dentro buff2.
Verificare la differenza dei tempi di copia tra l’operatore di assegnazione e l’operatore di assegnazione per movimento.
Per misurare i tempi, il C++ ha varie API conformi allo standard POSIX; per questo esercizio cercare la documentazione di:

- clock: il tempo trascorso dall’inizio del programma
- time: dà i secondi trascorsi dalle 00 del 1/1/1970 (epoch)

10. C’è garanzia che due chiamate successive di queste funzioni diano valori monotoni crescenti?
11. Perché?

Scegliere quello che meglio si presta alla situazione

Per evitare di inquinare la propria base di codice con variabili di varia natura, è possibile sfruttare il fatto che, se in un blocco di codice viene dichiarata una variabile UDT (User Defined Type, ovvero di tipo non elementare), ne verrà invocato il costruttore quando il programma raggiunge la riga in cui essa è definita e il distruttore, quando verrà raggiunta la fine del blocco. A questo scopo, si crei la classe DurationLogger, con la seguente struttura:

```
class DurationLogger {
  // mettere qui le necessarie variabili istanza
public:
    DurationLogger(const std::string& name) { //da implementare
    }
    ~DurationLogger() { //da implementare
    }
};
```

Il costruttore deve memorizzare l’argomento ricevuto e l’istante in cui viene invocato e stampare
“ >>>>>>>>>>>>>> starting [name]”
Il distruttore deve stampare “ <<<<<<<<<<<<<< ending [name]: [duration] sec”
Dove [name] è il valore della stringa passata al costruttore e [duration] è la differenza di tempo trascorsa tra l’invocazione del distruttore e quella del costruttore.
Si usi la classe così definita per dichiarare variabili locali da inserire all’inizio dei blocchi di cui si intende calcolare la durata.
Ad esempio, per determinare quanto tempo impiega la funzione main(...) ad essere eseguita, si può scrivere

```
int main() {
  DurationLogger("main()");
  //scrivere qui tutto il codice di cui si intende misurare la durata
}
```

Questa tecnica, chiamata RAII (Resource Acquisition Is Initialization) è un tipico idioma del C++ e trova numerosi esempi in vari campi, dal logging alla gestione della sincronizzazione.

Seconda parte: memorizzazione messaggi

Realizzare una classe MessageStore che consenta di memorizzare i messaggi in arrivo. La classe contiene un array di Message inizialmente dimensionato per contenere n messaggi.

- Man mano che arrivano i messaggi, se hanno del contenuto (message.id !=-1) vengono aggiunti nella prima posizione libera (riconoscibile perché ha id == -1)
- Se l'array è pieno, viene riallocato con dimensione pari dim + n, spostando il contenuto esistente nella nuova posizione
- Quando si cancella un messaggio, quello eliminato viene sostituito da un messaggio vuoto

Interfaccia della classe:

```
class MessageStore {
      Message *messages;
      int dim; // dimensione corrente array
      int n; // incremento memoria
public:
      Message(int n);
      ~Message();
      void add(Message &m);
      // inserisce un nuovo messaggio o sovrascrive quello precedente
      //se ce n'è uno presente con lo stesso id

      std::optional<Message> get(long id);
      // restituisce un messaggio se presente

      bool remove(long id);
      // cancella un messaggio se presente

      std::tuple<int, int> stats();
      // restituisce il numero di messaggi validi e di elementi vuoti
      // ancora disponibili

      void compact();
      // compatta l'array (elimina le celle vuole e riporta l'array
      // alla dimensione multiplo di n minima in grado di contenere
      // tutte le celle
};
```

Realizzata la classe e provarla con n = 10, inserire 100 messaggi da 1MB, cancellarne 50 e compattare.

12. Come ottimizzare la riallocazione dell'array che contiene i messaggi man mano che cresce?

Verificare quale operatore di Message viene utilizzato durante la copia tra il vecchio e il nuovo array, misurare i tempi (testare sia operatore di assegnazione che di movimento).

Come si può essere sicuri alla fine del programma che tutte le allocazioni e deallocazione di Message siano avvenute in modo corretto?
Oltre all'utilizzo di smart pointer (che verranno spiegati in seguito), ci sono varie tecniche di debug "manuale" che possono venire usate. Una molto semplice è aggiungere un contatore statico dentro la classe Message. Ogni volta che si chiama un costruttore viene incrementato di 1. Ogni volta che viene invocato il distruttore viene decrementato (**attenzione alle assegnazioni!**). Alla fine dell'esecuzione deve essere zero se ogni Message è stato rilasciato in modo corretto.
Implementare questa tecnica dentro Message e verificare che il contatore sia zero alla fine dell'esecuzione dell'esempio precedente.

Il metodo get(...) potrebbe presentarsi in vari modi alternativi. Ad esempio:

```
Message* get(long id);
Message get(long id);
Message& get(long id);
std::optional<Message> get(long id);

void get(int id, Message message);
void get(int id, Message *message);
void get(int id, Message **message);
void get(int id, Message &message);
void get(int id, const Message &message);
```

Discutere, per ciascuno di essi, pregi e difetti, indicando quali possono essere le varie problematiche che esso introduce.