Programmazione di Sistema
Lab 1 - Introduzione al C++

Obbiettivi

- 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 del rilascio corretto della memoria

Panoramica ambiente di sviluppo

Durante queste esercitazioni consigliamo di utilizzare come IDE CLion, free per gli studenti universitari. Per la procedura su come installare sulle varie piattaforme e la guida sul come ottenere la licenza per studenti fare riferimento al materiale condiviso in precedenza. 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 un file chiamato **CMakeLists.txt**, da inserire nella cartella radice del progetto. CLion gestirà in automatico per voi questo file, aggiornandolo quando andrete a aggiungere ad esempio nuove classi al progetto, ma è possibile anche una 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 una compilazione manuale è possibile eseguire il comando `cmake .` nella cartella contenente il CMakeLists.txt, questo genererà il Makefile corrispondente; a questo punto invocando il comando `make` il programma verrà compilato e l'eseguibile generato.

Organizzazione del Codice

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

- **main.cpp**: contiene la funzione `main()` e la logica principale del programma
- **classname.h**: un file di definizione per ogni classe
- **classname.cpp**: un file con l'implementazione della data classe, uno per ogni classe

Esercizio 1

Un sistema di ricezione riceve dei messaggi di lunghezza variabile, contenenti un ID, un buffer di dati e la rispettiva lunghezza. Si scriva quindi:

- una classe che consenta di gestire i messaggi in arrivo
- una classe che permetta di memorizzarli

Gestione messaggi in arrivo

La classe 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 si può utilizzare la seguente funzione:

```
#include <string>

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

Per poter stampare il contenuto di un `Message` vi sono due possibili opzioni:

- Definire opportunamente i metodi getter per ottenere il valore delle variabili private della classe in modo da poterle stampare.

- Dal momento che `Message` non è un tipo base, per poter richiamare l'operatore `<<` direttamente sull'oggetto è necessario ridefinire questo operatore in modo da "istruirlo" su come comportarsi in caso debba agire su un oggetto di tipo `Message`

Per poter usare direttamente l'operatore `<<` è necessario **dichiarare** (tipicamente in un file .h) e **definire** la funzione `operator<<(...)` come riportato sotto. Questa funzione deve essere esterna alla classe.

```
std::ostream& operator<<(std::ostream &out, const Message &m){
    std::string s(m.getData());
    out <<"[id:"<<m.getId()<<"]"<<"[size:"<<m.getSize()
        <<"]"<<"[data:"<<s.substr(0, 20)<<"]";
    return out;
}
```

Un esempio di uso può essere il seguente:

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


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

Si implementino quindi i seguenti metodi per la classe `Message`:

- costruttore i default, deve creare un messaggio vuoto con id = -1
- costruttore di copia
- costruttore di movimento
- operatore di assegnazione
- operatore di assegnazione per movimento
- distruttori
- metodi di accesso in sola lettura ai dati contenuti nella classe `Message` (metodi getter). Tali metodi devono essere dichiarati `const` (perché?)

Si scriva a questo punto degli use case che simulino l'invocazione di tutti i metodi implementati e verificare che siano corretti.

Spunti di Riflessione

Dato il seguente pezzo di codice:

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

- Che differenza c'è tra queste due costruzioni di un array di `Message`?
- Quale costruttore viene invocato?
- È necessario fare la delete dei buffer?

Dato:

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

- Cosa capita quando assegno un nuovo messaggio ad un elemento dell'array?
- Cosa capita se commento il costruttore di default?
- Cosa capita se rendo privato il costruttore di default?
- Cosa succede se commento gli operatori di assegnazione implementati?

Si provi a confrontare il tempo impiegato dall'operatore di assegnazione per copia e l'operatore di assegnazione per movimento, si crei un buffer con 10 messaggi lunghi 1MB e lo si copi dentro un secondo buffer utilizzando i due operatori di assegnazione.
HINT: Per misurare l'esecuzione si può fare riferimento alle seguenti API, conformi allo standard POSIX che il C++ mette a disposizione:

- **clock**: il tempo trascorso dall'inizio del programma
- **time**: restituisce i secondo trascorsi dalle 00 del 111970 (epoch)

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 elementale), 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:

```cpp
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.

Memorizzazione dei 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 allocato nuovamente 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:

```cpp
class MessageStore{
    Message* messages;
    int dim; //dimensione corrente array
    int n; //incremento memoria
public:
    MessageStore(int n);
    ~MessageStore();

    void add(Message &m);
    //inserisce un nuovo messaggio
    //sovrascrive quello precedente
    //se e' presente un messaggio con lo stesso id

    std::optional<Message> get(long id);
    //restituisce il messaggio con il dato di, se esiste

    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
};
```

Si implementi opportunamente la classe e si provi con `n = 10`, inserimento di 100 messaggi da 1MB l'uno, successivamente se ne cancellino 50 e si esegua una operazione di compattamento.

Spunti di Riflessione

Come si può essere sicuri, alla fine del programma, che tutte le allocazione e deallocazioni di `Message` siano avvenute in modo corretto? Oltre all'utilizzo di smart pointer (che verranno introdotti e spiegati in seguito), ci sono varie tecniche di debug "manuale" che possono essere utilizzate. una molto semplice è quella di aggiungere un contatore statico dentro alla classe `Message`. Ogni volta che si chiama un costruttore viene incrementato di 1, ogni volta che viene invocato il distruttore viene decrementato. Alla fine dell'esecuzione deve essere zero se ogni `Message` è stato rilasciato in modo corretto. Si implementi 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.