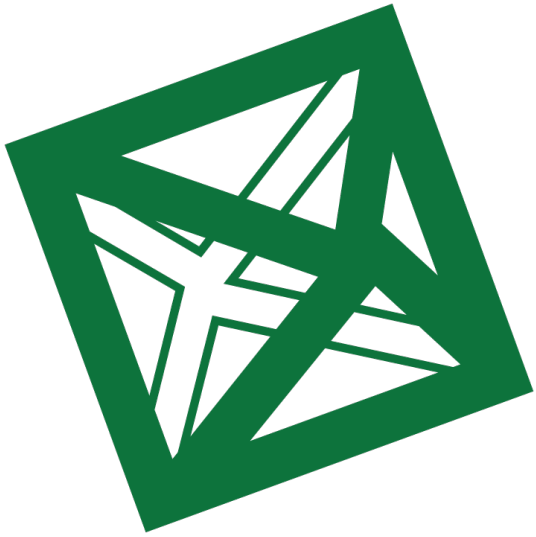
Università degli Studi di Milano Bicocca

Esame 17/09/2024



Relazione Progetto C++ e Qt

Nome:
Yehan Sanjula Edirisinghe Mudiyanselage

N°.matricola:
881175

Anno Accademico 2023–2024

Indice

1 Progetto C++ - Matrice Sparsa

Questa è la sezione inerente il progetto C++ che richiede la creazione di una classe per rappresentare matrici sparse. La libreria è inclusa ad un file **main.cpp** e una libreria **TestLib.hpp** per testare le varie funzionalità disponibili. Per la compilazione è sufficiente far uso del Makefile incluso nella cartella tramite comando **make**. A seguito della compilazione i file oggetto e l'eseguibile verranno prodotti dentro la cartella build. Per eseguire il programma basterà eseguire il comando **make run**.

1.1 Segnatura della classe

Siccome la Matrice Sparsa deve poter accettare elementi generici è stata ideata la classe come classe templata. Dunque la segnatura della classe risulta come in seguito:

```
template <typename T>
class  SparseMatrix{ ... };
```

1.2 Tipi Pubblici

Per favorire maggiore compatibilità e chiarezza sono presenti due typedef:

```
typedef unsigned long int  size\_type;\\[5mm]
typedef T                  value\_type;
```

1.3 Struttura dati

Siccome è richiesto nella consegna di non allocare memoria per dati non inseriti, è stato fatto uso di una lista dinamica dove ogni nodo contiene una variabile di tipo value_type. La struttura è composta da una testa che nel codice è:

node* _head;

contenuta in maniera privata nella classe. A questa si collegano il resto dei nodi.
IMMAGINE

1.3.1 Struttura Nodi

Ogni Nodo è composto dalla stessa struttura descritta da un oggetto della tipologia:

Listing 1: Struttura nodi

```
struct node{
    /// @brief x (or i) coordinate
    size_type _x;
    /// @brief y (or j) coordinate
    size_type _y;
    /// @brief value stored in the node
    T _value;
    /// @brief pointer to the next node
    node* _next;

    /**
     * @brief Construct a new node object
     *
     * @param x x coordinate
     * @param y y coordinate
```

```
     * @param value value to store
     */
    explicit node(size_type x, size_type y, T value): _x(x), _y(y), _value(value), _next(
        LOG("node(size_type x, size_type y, T value)");
    };

    /**
     * @brief Destroy the node object (empty by design)
     *
     */
    ~node(){
        LOG("~node()");
    };
```

nel nodo sono contenute le variabili per la posizione **_x** ed **_y**, il valore da salvare **_value** ed un puntatore al prossimo nodo **_next**.

Per un uso minimale della memoria è stato preferito non usare la struttura **element** dichiarata successivamente.

1.4 Metodi Pubblici

1.4.1 Costruttore e Costruttore-Copia

Per impedire errori di interpretazione automatica del compilatore è stato rimosso il costruttore di default. Infatti a riga 81-SparseMatrix.cpp si può osservare:

```
/// @brief ```default constructor not permitted```
SparseMatrix() = delete;
```

É invece permesso un costruttore secondaro e copy constructor in quanto necessari per altre operazioni fondamentali della classe:

Listing 2: Costruttore e CCopia

```
SparseMatrix(const size_type& n,const size_type& m, value_type D):
    _head(nullptr),_D(D){
    ...
};

SparseMatrix(const SparseMatrix& mat): _n(mat._n), _m(mat._m), _D(mat._D),
    _head(nullptr){
    ...
}
```

1.4.2 Metodi Secondari non richiesti

Per comodità e necessità sono stati aggiunti dei metodi di stream e di ritorno informazioni.

Il metodo **printInfo()** ed il metodo **printData()** sono ideati per mandare sullo stream di output informazioni rispettivamente su dimensione,valore di default e valori contenuti all'interno (diversi da default).

I metodi **get_row()**, **get_col()** e **get_D()** sono ideati per permettere l'accesso a dati privati della classe in maniera sicura.

1.4.3 set()

Questo metodo ha il compito di inseririre un valore di tipo **value_type** nella metrice sparsa.

```
/**
 * @brief Set the value of the matrix at the given coordinates
 *
 * @param value value to set
 * @param x x coordinate
 * @param y y coordinate
 * @return true if the value was set, false if an error occurred
 */
bool set(const value_type& value, const size_type& x,const size_type& y){
    //check if coordinates are in the matrix
    assert(x < _n);
    assert(y < _m);

    if(_head == nullptr){
        if(value == _D) return true;
        try{
            _head = new node(x,y,value);
            return true;
        }catch(...){
            _head = nullptr;
            return false;
        }
    }

    node* tmp = _head;
    node* prev = _head;

    while(tmp != nullptr){

        if( tmp->_x == x && tmp->_y == y){
            if(value == _D){  // if value is default value remove old value from list
                node* next = tmp->_next;

                if(tmp == _head) {

                    try{ //check delete exceptions
                        delete _head;
                        _head = nullptr;
                        _head = next;
                        return true;

                    }catch(...){
                        return false;
                    }
                }

                try{ //check delete exceptions
                    delete tmp;
                    prev->_next = next;
                    return true;

                }catch(...){
```

```
                    return false;
                }
            }
            //otherwise change the value
            tmp->_value = value;
            return true;
        }
        //go to next node
        prev = tmp;
        tmp = tmp->_next;
    }
    if( value == _D) return true;

    try{ // check new exceptions
        node* new_node = new node(x,y,value);
        prev->_next = new_node;
        return true;
    }catch(...){
        return false;
    }
}
```

La prima condizione che viene controllata è il caso particolare in cui la lista è vuota, ovvero _head == nullptr. In questo caso si controlla se il valore da allocare è diverso dal valore di **default** e in tal caso si alloca il nodo e si inserisce il valore desiderato.

Se invece il nodo iniziale non è nullo siamo nel caso di una lista non vuota e bisogna accertarsi che nella posizione richiesta non ci sia già un valore. Per questo si itera l'intera lista tramite un ciclo while. Se nella posizione desiderata è presente un valore diverso da **default** lo si modifica mentre se il valore da inserire è default si elimina il nodo facendo attenzione al caso della testa **_head**. Se invece non viene trovato un valore nella casella, se ne inserisce uno nuovo.

Il metodo ritorna un valore booleano che indica se l'allocazione è avvenuta con successo oppure no. In caso di mancato successo la lista finisce in uno stato coerente possibilmente uguale allo stato precedente alla chiamata di funzione.

1.4.4 operator()

Questo metodo ha lo scopo di riferire all'utente il valore contenuto nella posizione (i,j) della matrice.

Listing 3: operator()

```
/**
* @brief Get the value of the matrix at the given coordinates
*
* @param x x coordinate
* @param y y coordinate
* @return value_type value at the given coordinates
*/
value_type operator()(const size_type& x, const size_type& y) const{
    node* tmp = _head;

    while(tmp != nullptr){

        if( tmp->_x == x && tmp->_y == y){
            return tmp->_value;
```

```
            }
            tmp = tmp->_next;
        }

        return _D;
    }
```

Si iterano tutti gli elementi della lista cercando un elemento con posizione (i,j). Se non viene trovato viene restituito il valore di **default**.

1.5 Classe Iteratori

Per l'uso degli iteratori è stato usato il puntatore ai nodi della lista dinamica. Ogni iteratore se dereferenziato tramite operatore * restituisce un oggetto di tipo element:

```
/**
 * @brief element object for the iterator
 *
 */
struct element{
    size_type _i, _j;
    value_type _value;

    element(const size_type& i, const size_t& j, const value_type value) : _i(i), _j(j),
    element(const element& e) : _i(e._i), _j(e._j), _value(e._value){}
};
```

dove **_i** e **_j** sono gli indici di posizione e **_value** il valore inserito.

Gli iteratori sono disegnati per essere monodirezionali: partono dalla testa e continuano fino alla coda iterando tra i valori inseriti diversi da dal valore di **default**.

Per ottenere l'inizio dei valori nella matrice basta chiamare il metodo **begin()** ed **end()** per la fine.

1.6 ~SparseMatrix()

in questa sezione è contenuto il codice atto alla distruzione di tutti i blocchi di memoria allocati a causa della matrice.

1.7 Funzione evaluate()

Questa funzione esterna alla classe SparseMatrix ha lo scopo di contare il numero di entrate della matrice che soddisfano un predicato. Siccome il tipo della matrice e il predicato non devono essere specificate a priori, la funzione è templata:

```
template <typename T, typename P>
typename SparseMatrix<T>::size_type evaluate(const SparseMatrix<T>& mat){
    ...
}
```

il metodo richiede dunque come parametro templato un oggetto che prevede un operatore **operator()** booleano della tipologia:

```
struct P_size{
    bool operator()(const int& value){
        if(value >= 10) return true;
        return false;
```

```
    }
};
```

2 Progetto Qt Creator in C++

In questa sezione si discute della creazione di una finestra di login con controllo delle iscrizioni e gestione admin tarmite il programma QtCreator.

2.1 Introduzione

La struttura utilizzata per la creazione della finestra è una QMainWindow. La disposizione degli elementi è stata fatta quanto possibile seguendo le direttive del tema d'esame. Non essendo specificato il tipo di adattamento degli oggetti alla riscalatura della finestra è stato lasciato libero il movimento nel riquadro superiore.

2.2 Organizzazione della classe

L'intera finestra di Login è contenuta nella classe **LoginDialog**. Al suo interno si trovano anche gli elementi necessari per la costruzione della finestra **admin**. Le variabili sono differenziate quanto possibile per chiarezza sull'appartenenza alla classe LoginDialog o alla finestra admin.

2.3 Logica di esecuzione

La finestra è dotata di due possibili azioni: login e iscrizione.

In entrambi i casi è stato scritto il codice per il controllo dell'inserimento corretto dei dati. Ovvero non sono accettati campi vuoti e spazi. Nel caso delle email/n.tel è stata aggiunta la richiesta di validità del numero di telefono con solo numeri tramite il metodo **is_valid_email()**.

Alla fine di ogni accesso o iscrizione l'interfaccia viene ripulita tramite il metodo **cleanUI()**.

2.4 Finestra Admin

Nel caso i dati di login combacino con quelli dell'admin (admin@pas.com, admin) si apre una nuova finestra. In questa sono presenti una lista degli iscritti con nome, cognome, email/n.tel ed età. Successivamente ci sono due grafici: uno sul sesso degli utenti e l'altro sulla loro età divisa in 5 fasce.

I grafici vengono aggiornati tramite due funzioni: **updateUIList()** e **updateUIPie()**. Questi sono rispettivamente per aggiornare gli elementi della lista e i due grafici a torta. Nel caso la finestra admin sia già aperta, dopo una nuova iscrizione la finsetra non rischiede di essere chiusa ma viene automaticamente aggiornata.

2.5 Struttura dati

I dati utente sono salvati nella classe tramite un **std::vector** contenente oggetti struct **login_info** ovvero:

```
/**
 * @brief Struct per contenere i dati di un singolo utente
 */
struct login_info{

    std::string _nome, _cognome;
    std::string _email_or_tel;
    std::string _password;
    int _giorno_nascita, _mese_nascita, _anno_nascita;
    int _et ;
    bool _genere; //m = 0, f = 1
```

```
    ...
};
```