Silesian University of Technology

Project Report

Implementation of a custom single
linked list class

Author: Youssef AL BALI.

Teacher: Dr. Inż. Agnieszka Danek

Date:12 January 2022

Analysis of the problem:

The aim of the microproject is to implement a custom single linked list in the form of a class with a convenient programming interface.

The class should be implemented as a template, and an iterator that will allow the use of STL should be implemented as well.

Requirements:

• usage of smart pointers,

• default constructor,

• copy / move constructors,

• destructor,

• assignment / move operators,

• adding container elements,
• searching for an element,

• sorting content (different criteria),

• serialization / deserialization (reading and writing from byte stream).

External Specification:

The class can be declared and initialized with different data types as the following:

```
SingleLinkedList<int> s_int{1,2,5,4,80,100};
SingleLinkedList<std::string> s_names{"Youssef", "Viktor", "Andrii"};
SingleLinkedList<float> s_float{0.02, 12.1, 100.001, 25.5};
SingleLinkedList<char> s_chars{'a', 'b', 'C', 'F', 'g'};
```

The class supports the following methods:

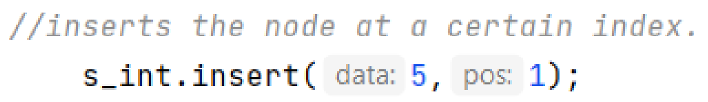
the push_back and push_front functions.

```
    ///Adding a new node at the the end of the list.
        s_int.push_back( data: 3);
        s_int.push_back( data: 5);

    ///Adding a new node at the the back of the list.
        s_int.push_front( data: 100);
        s_int.push_front( data: 258);
```

As well as the pop_back and pop_front methods:

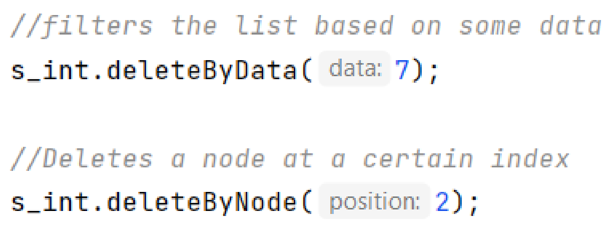
```
  //Deleting the node at the  beginning of the list.
  s_names.pop_front();


  //Deleting the node at the end of the list.
  s_names.pop_back();
```

Insert function:



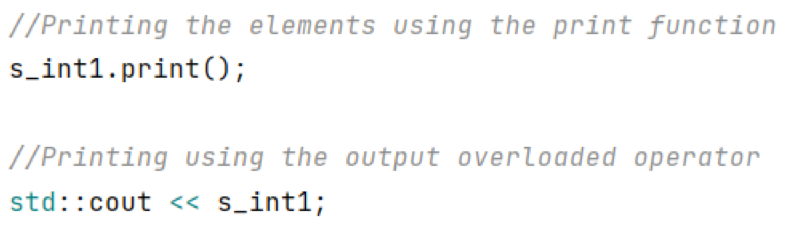
```
//inserts the node at a certain index.
    s_int.insert( data: 5, pos: 1);
```

Two delete functions (First one is deleting node based on Data so it's more like filtering and the second takes the position of the node as a parameter):



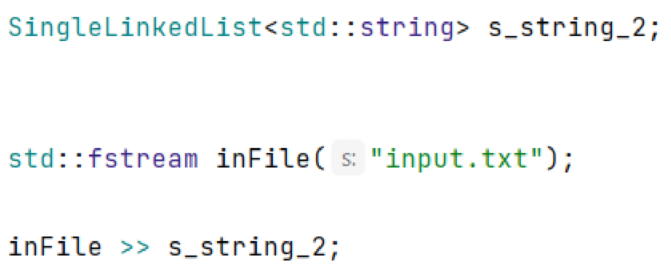
```
//filters the list based on some data
s_int.deleteByData( data: 7);

//Deletes a node at a certain index
s_int.deleteByNode( position: 2);
```

Function that prints all the elements od the list and the overloaded output operator:



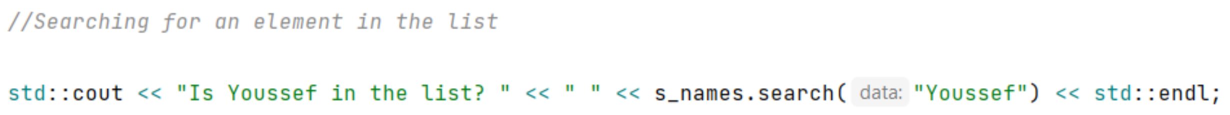
```
//Printing the elements using the print function
s_int1.print();

//Printing using the output overloaded operator
std::cout << s_int1;
```

Input operator used to read data from a file:



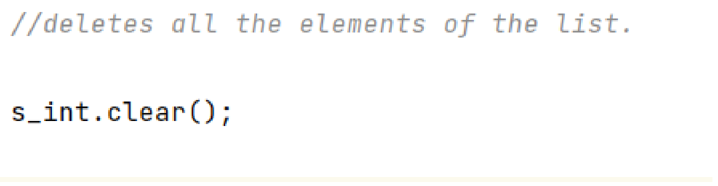
```
SingleLinkedList<std::string> s_string_2;


std::fstream inFile( s: "input.txt");

inFile >> s_string_2;
```

Search function:



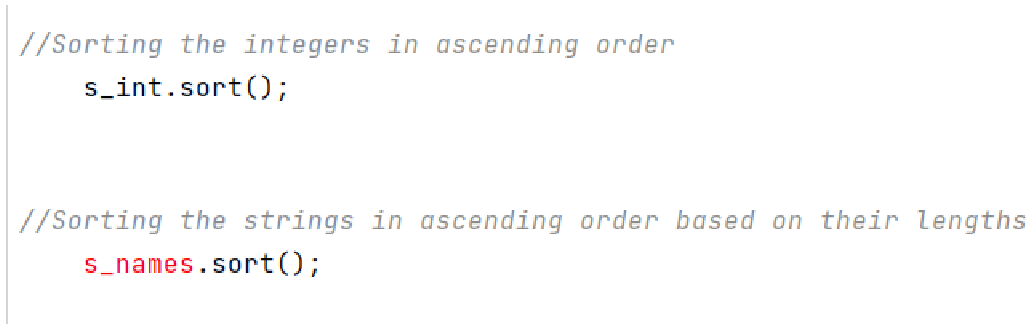
```
//Searching for an element in the list

std::cout << "Is Youssef in the list? " << " " << s_names.search("Youssef") << std::endl;
```

Clear function:



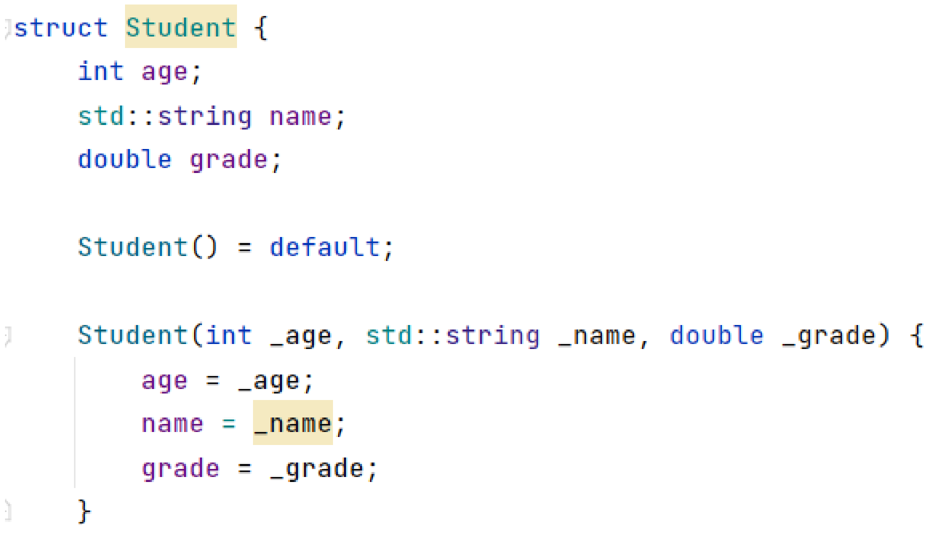
```
//deletes all the elements of the list.

s_int.clear();
```

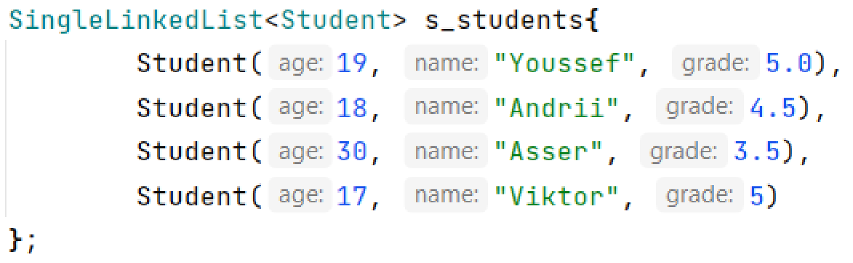
We can also sort the elements of the container in ascending order:

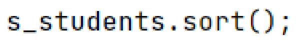


```
//Sorting the integers in ascending order
    s_int.sort();


//Sorting the strings in ascending order based on their lengths
    s_names.sort();
```

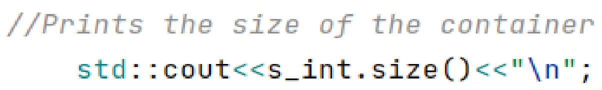
The container has also been tested using the “Student” class in order to sort this type by grade.


```
struct Student {
    int age;
    std::string name;
    double grade;

    Student() = default;

    Student(int _age, std::string _name, double _grade) {
        age = _age;
        name = _name;
        grade = _grade;
    }
```


```
SingleLinkedList<Student> s_students{
        Student( age: 19, name: "Youssef", grade: 5.0),
        Student( age: 18, name: "Andrii", grade: 4.5),
        Student( age: 30, name: "Asser", grade: 3.5),
        Student( age: 17, name: "Viktor", grade: 5)
};
```


```
s_students.sort();
```

And finally, a function that get the size of the list:


```
    //Prints the size of the container
        std::cout<<s_int.size()<<"\n";
```

Internal Specification:

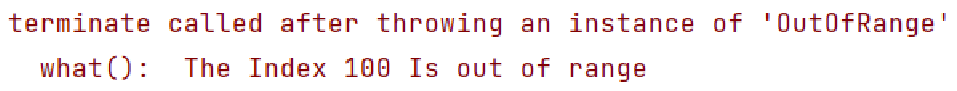
Everything concerning the internal Specification is included in the **Doxygen file** at the end of the report.

Testing:

The program was tested with many types of variables including Student class.

All the methods were tested and worked properly as expected.

There was a small issue concerning the insert function, when the index passed as parameter was out of range the program crashes so I decided to handle this by throwing an Exception that I made which derived from std::exception and it outputs the following message when the program throws the ***OutOfRange*** exception:



```
terminate called after throwing an instance of 'OutOfRange'
  what():  The Index 100 Is out of range
```

Single Linked List

Generated by Doxygen 1.9.3

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

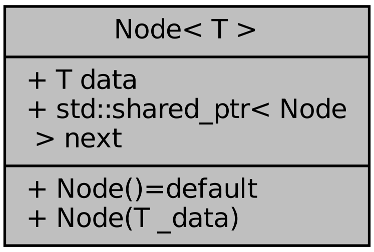
Class Documentation

3.1 Node< T > Struct Template Reference

struct Node.

`#include <SingleLinkedList.h>`

Collaboration diagram for Node< T >:



Public Member Functions

- **Node** ()=default

  default constructor
- Node (T _data)

  One Argument Constructor that assigns the Node data.

Public Attributes

- T **data**

  data of the Node.
- std::shared_ptr< Node > **next** {}

  pointer to the next node.

3.1.1 Detailed Description

template<class T>
struct Node< T >

struct Node.

Represent a node of the single linked list.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Node()

```
template<class T >
Node< T >::Node (
             T _data )  [inline]
```

One Argument Constructor that assigns the Node data.

Parameters

_data	Node data.

The documentation for this struct was generated from the following file:

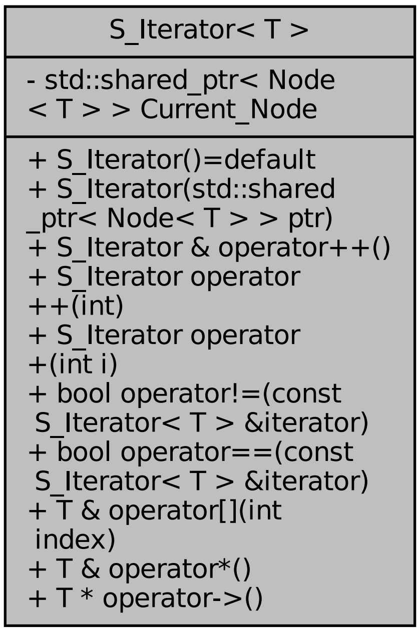
- SingleLinkedList.h

3.2 S_Iterator< T > Class Template Reference

Single linked list iterator.

```
#include <SingleLinkedList.h>
```

Collaboration diagram for S_Iterator< T >:


S_Iterator< T >
- std::shared_ptr< Node < T > > Current_Node
+ S_Iterator()=default
+ S_Iterator(std::shared_ptr< Node< T > > ptr)
+ S_Iterator & operator++()
+ S_Iterator operator++(int)
+ S_Iterator operator+(int i)
+ bool operator!=(const S_Iterator< T > &iterator)
+ bool operator==(const S_Iterator< T > &iterator)
+ T & operator[](int index)
+ T & operator*()
+ T * operator->()

Public Member Functions

- **S_Iterator** ()=default

  Default constructor.
- S_Iterator (std::shared_ptr< Node< T > > ptr)

  Constructor that assigns the current Node.
- S_Iterator & operator++ ()

  Pre-increment operator.
- S_Iterator operator++ (int)

  Post-increment operator.
- S_Iterator operator+ (int i)

  operator +
- bool operator!= (const S_Iterator< T > &iterator)

  comparison operator.
- bool operator== (const S_Iterator< T > &iterator)

  comparison operator.
- T & operator[ ] (int index)

  Square bracket operator.
- T & operator∗ ()

  Dereference operator.
- T ∗ operator-> ()

  Pointer operator.

Private Attributes

- std::shared_ptr< Node< T > > **Current_Node**

  Pointer to the Current Node.

3.2.1 Detailed Description

template<class T>
class S_Iterator< T >

Single linked list iterator.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 S_Iterator()

```
template<class T >
S_Iterator< T >::S_Iterator (
             std::shared_ptr< Node< T > > ptr )  [inline]
```

Constructor that assigns the current Node.

Parameters

ptr	Node pointer.

3.2.3 Member Function Documentation

3.2.3.1 operator"!=()

```
template<class T >
bool S_Iterator< T >::operator!= (
             const S_Iterator< T > & iterator )  [inline]
```

comparison operator.

Parameters

iterator	the iterator to compare with.

Returns

True id the iterators are different, otherwise false

3.2.3.2 operator∗()

```
template<class T >
T & S_Iterator< T >::operator* ( )  [inline]
```

Dereference operator.

Returns

The data of the element at which the iterator is pointing to.

3.2.3.3 operator+()

```
template<class T >
S_Iterator S_Iterator< T >::operator+ (
             int i )  [inline]
```

operator +

Advances the operator and returns the reference of the result

Returns

reference to the result

3.2.3.4 operator++() `[1/2]`

```
template<class T >
S_Iterator & S_Iterator< T >::operator++ ( )  [inline]
```

Pre-increment operator.

Increments the operator and returns the reference to the result

Returns

reference to the result

3.2.3.5 operator++() [2/2]

```
template<class T >
S_Iterator S_Iterator< T >::operator++ (
             int  )  [inline]
```

Post-increment operator.

Increments the operator and returns the iterator before the incrementation

Returns

old iterator

3.2.3.6 operator->()

```
template<class T >
T * S_Iterator< T >::operator-> ( )  [inline]
```

Pointer operator.

Returns

Pointer to T

3.2.3.7 operator==()

```
template<class T >
bool S_Iterator< T >::operator== (
             const S_Iterator< T > & iterator )  [inline]
```

comparison operator.

Parameters

iterator	the iterator to compare with.

Returns

True id the iterators are different, otherwise false

3.2.3.8 operator[]()

```
template<class T >
T & S_Iterator< T >::operator[ ] (
             int index )  [inline]
```

Square bracket operator.

Parameters

index	index of the element starting from where the iterator is pointing to.

Returns

reference to the data

The documentation for this class was generated from the following file:

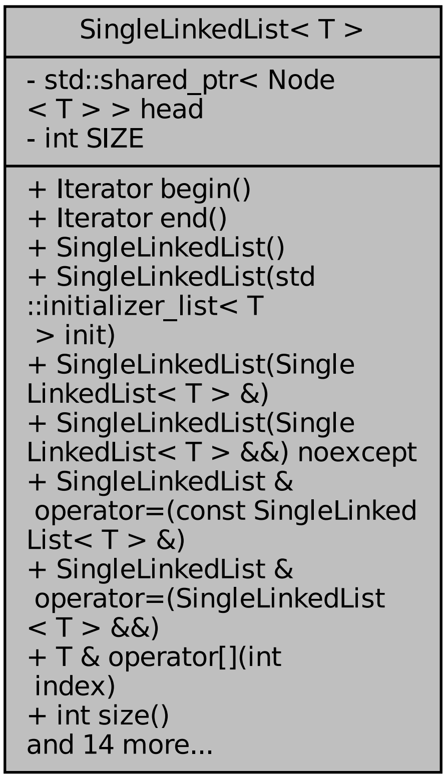
- SingleLinkedList.h

3.3 SingleLinkedList< T > Class Template Reference

Single Linked list class.

```
#include <SingleLinkedList.h>
```

Collaboration diagram for SingleLinkedList< T >:


SingleLinkedList< T >
- std::shared_ptr< Node < T > > head
- int SIZE
+ Iterator begin()
+ Iterator end()
+ SingleLinkedList()
+ SingleLinkedList(std ::initializer_list< T > init)
+ SingleLinkedList(Single LinkedList< T > &)
+ SingleLinkedList(Single LinkedList< T > &&) noexcept
+ SingleLinkedList & operator=(const SingleLinked List< T > &)
+ SingleLinkedList & operator=(SingleLinkedList < T > &&)
+ T & operator[](int index)
+ int size()
and 14 more...

Public Types

- typedef S_Iterator< T > Iterator

  Size of the list.

Public Member Functions

- Iterator **begin** ()

  End Iterator.
- Iterator **end** ()
- SingleLinkedList ()

  Default constructor.
- **SingleLinkedList** (std::initializer_list< T > init)
- SingleLinkedList (SingleLinkedList< T > &)

  Copy constructor.
- SingleLinkedList (SingleLinkedList< T > &&) noexcept

  Move constructor.
- SingleLinkedList & operator= (const SingleLinkedList< T > &)

  Copy operator.

- SingleLinkedList & operator= (SingleLinkedList< T > &&)

    Move operator.
- T & operator[ ] (int index)

    Square Bracket operator.
- int size ()

    Function that return the size of a the list.
- void push_back (T data)

    Push_back.
- void push_front (T data)

    Push_front.
- void pop_front ()

    Pop_Front.
- void pop_back ()

    Pop_Front.
- void deleteByNode (int position)

    Function that deletes a node at a certain index.
- void insert (T data, int pos)

    Function that inserts a node at a certain index.
- void deleteByData (T _data)

    Function the filters the list by data.
- void **print** ()

    Function that prints the elements of the list.
- bool search (T _data)

    Search function.
- void sort ()

    Sort function.
- void **clear** ()

    Function that deletes all the elements of the list.
- void serialize (const std::string &filename)

    Serialize function.
- void deserialize (const std::string &filename)

    Deserialize function.
- void **sort** ()

Private Attributes

- std::shared_ptr< Node< T > > **head** {}

    Pointer to the head of the list.
- int **SIZE** {}

Friends

- template<class s_type >
  std::ostream & operator<< (std::ostream &os, const SingleLinkedList< s_type > &s)

    Output stream operator.
- template<class s_type >
  std::istream & operator>> (std::istream &input, SingleLinkedList< s_type > &s)

    Input stream operator.

3.3.1 Detailed Description

template<class T>
class SingleLinkedList< T >

Single Linked list class.

3.3.2 Member Typedef Documentation

3.3.2.1 Iterator

```
template<class T >
typedef S_Iterator<T> SingleLinkedList< T >::Iterator
```

Size of the list.

Begin Iterator.

3.3.3 Constructor & Destructor Documentation

3.3.3.1 SingleLinkedList() [1/3]

```
template<class T >
SingleLinkedList< T >::SingleLinkedList
```

Default constructor.

Setting the head to a Null-pointer When the list is created.

3.3.3.2 SingleLinkedList() [2/3]

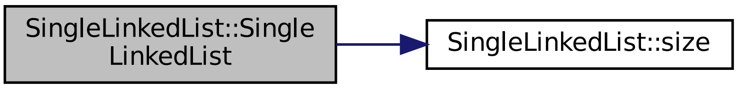
```
template<class T >
SingleLinkedList< T >::SingleLinkedList (
            SingleLinkedList< T > & s )
```

Copy constructor.

Parameters

s	A single linked list to copy from.

Here is the call graph for this function:



3.3.3.3 SingleLinkedList() [3/3]

```
template<class T >
SingleLinkedList< T >::SingleLinkedList (
            SingleLinkedList< T > && s )  [noexcept]
```

Move constructor.

Parameters

s	A single linked list to Move from.

3.3.4 Member Function Documentation

3.3.4.1 deleteByData()

```
template<class T >
void SingleLinkedList< T >::deleteByData (
            T _data )
```

Function the filters the list by data.

Parameters

_data	Element to be deleted.

3.3.4.2 deleteByNode()

```
template<class T >
```

```
void SingleLinkedList< T >::deleteByNode (
            int position )
```

Function that deletes a node at a certain index.

Parameters

position	Index od the element to be deleted.

3.3.4.3 deserialize()

```
template<class T >
void SingleLinkedList< T >::deserialize (
            const std::string & filename )
```

Deserialize function.

reads data in binary format

Parameters

filename	

3.3.4.4 insert()

```
template<class T >
void SingleLinkedList< T >::insert (
            T data,
            int pos )
```

Function that inserts a node at a certain index.

Parameters

position	The Position where to add the element.
data	The element to be added.

3.3.4.5 operator=() [1/2]

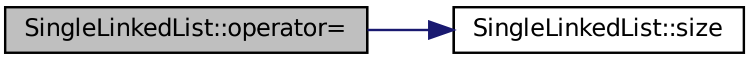
```
template<class T >
SingleLinkedList< T > & SingleLinkedList< T >::operator= (
            const SingleLinkedList< T > & other )
```

Copy operator.

Parameters

other	A single linked list to copy from.

Here is the call graph for this function:



3.3.4.6 operator=() `[2/2]`

```
template<class T >
SingleLinkedList< T > & SingleLinkedList< T >::operator= (
            SingleLinkedList< T > && other )
```

Move operator.

Parameters

other	A single linked list to Move from.

3.3.4.7 operator[]()

```
template<class T >
T & SingleLinkedList< T >::operator[ ] (
            int index )  [inline]
```

Square Bracket operator.

Parameters

index	The index of the element.

3.3.4.8 pop_back()

```
template<class T >
void SingleLinkedList< T >::pop_back
```

Pop_Front.

Deleting the node at the the end of the list.

3.3.4.9 pop_front()

```
template<class T >
void SingleLinkedList< T >::pop_front
```

Pop_Front.

Deleting the node at the beginning of the list.

3.3.4.10 push_back()

```
template<class T >
void SingleLinkedList< T >::push_back (
             T data )
```

Push_back.

Adding a new node at the the end of the list.

Parameters

data	The element to add.

3.3.4.11 push_front()

```
template<class T >
void SingleLinkedList< T >::push_front (
             T data )
```

Push_front.

Adding a new node at the the beginning of the list.

Parameters

data	The element to add.

3.3.4.12 search()

```
template<class T >
bool SingleLinkedList< T >::search (
             T _data )
```

Search function.

Parameters

_data	The element to search.

Returns

Returns true if the element exists, otherwise false.

3.3.4.13 serialize()

```
template<class T >
void SingleLinkedList< T >::serialize (
             const std::string & filename )
```

Serialize function.

stores data in binary format

Parameters

filename	

3.3.4.14 size()

```
template<class T >
int SingleLinkedList< T >::size ( )  [inline]
```

Function that return the size of a the list.

Returns

The size of the list.

3.3.4.15 sort()

```
template<class T >
void SingleLinkedList< T >::sort
```

Sort function.

Sorts the elements of the list using Bubble sort algorithm.

3.3.5 Friends And Related Function Documentation

3.3.5.1 operator<<

```
template<class T >
template<class s_type >
std::ostream & operator<< (
            std::ostream & os,
            const SingleLinkedList< s_type > & s )  [friend]
```

Output stream operator.

The operator is used to output the elements of the either to Console or to an output file

Parameters

os	std::ostream
s	Single linked list Object

Returns

std::ostream

3.3.5.2 operator>>

```
template<class T >
template<class s_type >
std::istream & operator>> (
            std::istream & input,
            SingleLinkedList< s_type > & s )  [friend]
```

Input stream operator.

The operator is used to read the elements of the list either to Console or to an output file.

Parameters

is	std::istream.
s	Single linked list Object.

Returns

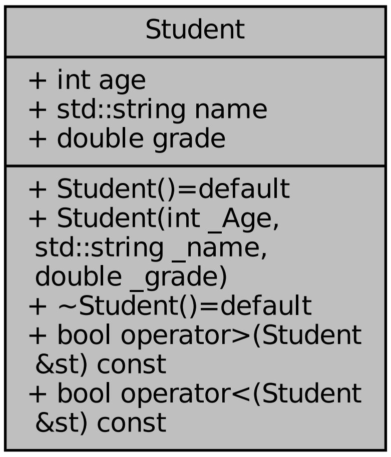
std::istream.

The documentation for this class was generated from the following file:

- SingleLinkedList.h

3.4 Student Struct Reference

Collaboration diagram for Student:


Student
+ int age
+ std::string name
+ double grade
+ Student()=default
+ Student(int _Age, std::string _name, double _grade)
+ ~Student()=default
+ bool operator>(Student &st) const
+ bool operator<(Student &st) const

Public Member Functions

- **Student** (int _Age, std::string _name, double _grade)
- bool **operator**> (Student &st) const
- bool **operator**< (Student &st) const

Public Attributes

- int **age**
- std::string **name**
- double **grade**

The documentation for this struct was generated from the following file:

- main.cpp

Chapter 4

File Documentation

4.1 SingleLinkedList.h

```


#ifndef PROJECT_SINGLELINKEDLIST_H
#define PROJECT_SINGLELINKEDLIST_H

#include <iostream>
#include <memory>
#include <fstream>
#include "Exceptions.h"
#include <cstring>



template<class T>
struct Node {
    T data;

    std::shared_ptr<Node> next{};

    Node() = default;


    Node(T _data) {
        data = _data;
    }

};


template<class T>
class S_Iterator {
public:
    S_Iterator() = default;



    S_Iterator(std::shared_ptr<Node<T>> ptr) : Current_Node(ptr) {};


    S_Iterator &operator++() {
        this->Current_Node = Current_Node->next;
        return *this;
    }


    S_Iterator operator++(int) {
        return ++(*this);
    }


    S_Iterator operator+(int i) {
        while (i != 0) { ;
            (*this)++;
            i--;
        }
        return *this;
    }


    bool operator!=(const S_Iterator<T> &iterator) {
        return Current_Node != iterator.Current_Node;
    }


    bool operator==(const S_Iterator<T> &iterator) {
        return Current_Node == iterator.Current_Node;
    }


    T &operator[](int index) {
         auto tmp = Current_Node;
         while (index != 0) {
             tmp = tmp->m_nextNode;
             index--;
         }
         return *tmp->m_data;
     }


     T &operator*() { return Current_Node->data; }


     T *operator->() { return &Current_Node->data; } // returns the address to this data

private:
    std::shared_ptr<Node<T>> Current_Node;
};

template<class T>
class SingleLinkedList {


    std::shared_ptr<Node<T>> head{};
    int SIZE{};
public:

    typedef S_Iterator<T> Iterator;

    Iterator begin() { return Iterator(head); }

    Iterator end() { return Iterator(nullptr); }


public:


    template<class s_type>
    friend std::ostream &operator<<(std::ostream &os, const SingleLinkedList<s_type> &s);


    template<class s_type>
    friend std::istream &operator>>(std::istream &input, SingleLinkedList<s_type> &s);


    SingleLinkedList();

    SingleLinkedList(std::initializer_list<T> init) {
        for (auto x: init)
            push_back(x);
    };


    SingleLinkedList(SingleLinkedList<T> &);




    SingleLinkedList(SingleLinkedList<T> &&) noexcept;




    SingleLinkedList &operator=(const SingleLinkedList<T> &);




    SingleLinkedList &operator=(SingleLinkedList<T> &&);


    T &operator[](int index) {
        auto tmp = head;
        while (index) {
            tmp = tmp->next;
            index--;
        }
        return tmp->data;
    }


    int size() { return SIZE; }


    void push_back(T data);


    void push_front(T data);



    void pop_front();


    void pop_back();


    void deleteByNode(int position);


    void insert(T data, int pos);


    void deleteByData(T _data);

    void print();


    bool search(T _data);


    void sort();

    void clear();




    void serialize(const std::string &filename);


    void deserialize(const std::string &filename);

};


template<class T>
SingleLinkedList<T>::SingleLinkedList() {
    std::cout « "Single linked list is created\n";
    head = nullptr;
}

template<class T>
SingleLinkedList<T>::SingleLinkedList(SingleLinkedList<T> &s) {
    head = s.head;
    SIZE = s.size();
}

template<class T>
SingleLinkedList<T>::SingleLinkedList(SingleLinkedList<T> &&s) noexcept {
    head = std::move(s.head);
    SIZE = std::move(s.SIZE);
    s.SIZE = 0;
}


template<class T>
SingleLinkedList<T> &SingleLinkedList<T>::operator=(const SingleLinkedList<T> &other) {

    if (this != &other) {
        head = other.head;
        SIZE = other.size();
    }
    return *this;
}

template<class T>

SingleLinkedList<T> &SingleLinkedList<T>::operator=(SingleLinkedList<T> &&other) {
    if (this != &other) {
        head = std::move(other.head);
        SIZE = std::move(other.SIZE);
        other.SIZE = 0;
    }
    return *this;
}
```

```


template<class T>
void SingleLinkedList<T>::push_back(T data) {

    auto n = std::make_shared<Node<T»(data);

    if (head == NULL) {
        head = n;
        SIZE++;
    } else {
        std::shared_ptr<Node<T» tmp_ptr = head;
        while (tmp_ptr->next != NULL) {
            tmp_ptr = tmp_ptr->next;
        }
        tmp_ptr->next = n;
        SIZE++;
    }

}

template<class T>
void SingleLinkedList<T>::push_front(T data) {

    auto n = std::make_shared<Node<T»(data);

    if (head == NULL) {
        head = n;
        SIZE++;
    } else {
        n->next = head;
        head = n;
        SIZE++;

    }


}


template<class T>
void SingleLinkedList<T>::print() {
    auto tmp = head;
    std::cout « "The elements of the list:\n";
    while (tmp != NULL) {
        std::cout « "Data: " « tmp->data « "\n";
        tmp = tmp->next;
    }
}

template<typename T>
std::ostream &operator«(std::ostream &os, const SingleLinkedList<T> &s) {
    {
        auto tmp = s.head;
        std::cout « "The elements of the list:\n";
        while (tmp != NULL) {
            os « "Data: " « tmp->data « "\n";
            tmp = tmp->next;
        }
        return os;
    }
}


template<class T>
void SingleLinkedList<T>::pop_front() {
    if (head == NULL) {
        std::cout « "The container is already empty\n";
    } else {
        head = head->next;
        SIZE--;
    }
}


template<class T>
void SingleLinkedList<T>::pop_back() {

    if (head == NULL) {
        std::cout « "The container is already empty\n";
    } else {
        auto tmp = head;
        auto prev_node = head;
        while (tmp->next != NULL) {
            prev_node = tmp;
            tmp = tmp->next;
        }
        prev_node->next = NULL;
        SIZE--;

    }
}


template<class T>
void SingleLinkedList<T>::deleteByNode(int position) {


    if (head == NULL) {
        std::cout « "The container is already empty (Nothing to delete)\n";
    } else {
        auto tmp = head;
        auto prev_node = head;
        while (position != 0) {
            prev_node = tmp;
            tmp = tmp->next;
            position--;
        }
        prev_node->next = tmp->next;
        SIZE--;
    }

}

template<class T>
void SingleLinkedList<T>::deleteByData(T _data) {

    auto tmp = head;
    auto prev_node = head;
    while (tmp != NULL) {
        if (tmp->data == _data) {
            prev_node->next = tmp->next;
            tmp = tmp->next;
            SIZE--;
        } else {
            prev_node = tmp;
            tmp = tmp->next;
        }

    }
}

template<class T>
void SingleLinkedList<T>::insert(T data, int pos) {
    auto tmp = head;
    auto n = std::make_shared<Node<T»(data);
    if (pos < 0 || pos > SIZE) {
        throw OutOfRange(pos);
    } else {
        while ((pos - 1) != 0) {
            tmp = tmp->next;
            pos--;
        }
        n->next = tmp->next;
        tmp->next = n;
        SIZE++;
    }
}

template<class T>
std::istream &operator»(std::istream &is, SingleLinkedList<T> &s) {
    T tmp;
    while (is » tmp) {
        s.push_back(tmp);
        s.SIZE++;
    }
    return is;
}

template<class T>
void SingleLinkedList<T>::clear() {
    head = NULL;
    SIZE = 0;
}

template<class T>
void SingleLinkedList<T>::sort() {
    if (head == NULL) {
        return;
    }
    auto tmp = head;
    bool swapped = true;
    while (swapped) {
        swapped = false;
        while (tmp->next != NULL) {

            if (tmp->data > tmp->next->data) {
                std::swap(tmp->data, tmp->next->data);
                swapped = true;
            }
            tmp = tmp->next;

        }
        tmp = head;
    }
}

template<>
inline void SingleLinkedList<std::string>::sort() {
    if (head == nullptr) {
        return;
    }
    auto tmp = head;
    bool swapped = true;
    while (swapped) {
        swapped = false;
        while (tmp->next != nullptr) {

            if (tmp->data.length() > tmp->next->data.length()) {
                std::swap(tmp->data, tmp->next->data);
                swapped = true;
            }
            tmp = tmp->next;

        }
        tmp = head;
    }
}

template<class T>
bool SingleLinkedList<T>::search(T _data) {
    auto tmp = head;
    while (tmp != nullptr) {
        if (tmp->data == _data) {
            return true;
        }
        tmp = tmp->next;
    }
    return false;

}

template<class T>
void SingleLinkedList<T>::serialize(const std::string &filename) {
    std::ofstream OutFile(filename,
                          std::ios::out | std::ios::binary);//flag that tells that you should write in a
   binary form
    if (OutFile) {
        for (const auto &e: *this) {
            OutFile.write((char *) &e, sizeof(e));//1 param : ptr to the first byte
        }
        OutFile.close();
    }

}

template<class T>
void SingleLinkedList<T>::deserialize(const std::string &filename) {
    std::ifstream InFile(filename, std::ios::in | std::ios::binary);
    if (InFile) {
        T tmp;
        while (InFile.read(const_cast<char *>((char *) &tmp), sizeof(tmp))) {
            push_back(tmp);
        }
        InFile.close();

    }
}


#endif //PROJECT_SINGLELINKEDLIST_H
```

Index