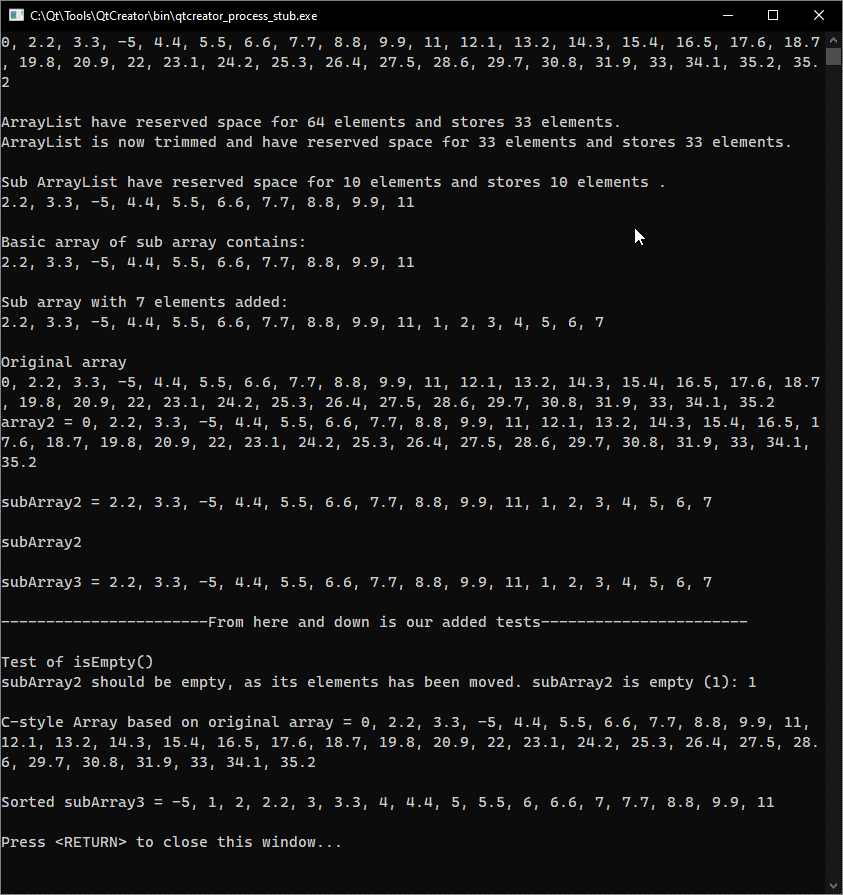
Template ArrayList

# Output



# Source code

## ”ArrayList.h”

#ifndef ARRAYLIST\_H

#define ARRAYLIST\_H

#include <stdexcept>

*template* <*typename* T>

*class* **ArrayList** {

*public*:

***/\*\**** ***Initializes*** ***an*** ***empty*** ***ArrayList.***

***\****

***\**** ***No*** ***initial*** ***size*** ***is*** ***given.***

***\*/***

**ArrayList**() {

*//* *If* *the* *user* *does* *not* *specify* *a* *size,*

*//* *it* *will* *be* *initialized* *as* *null*

\_size = 0;

\_reserved = 0;

\_elems = *nullptr*;

}

*//* *Copy* *constructor*

***/\*\**** ***Copies*** ***an*** ***Arraylist*** ***to*** ***create*** ***a*** ***new*** ***ArrayList.***

***\****

***\**** ***Takes*** ***attributes*** ***from*** ***ArrayList*** ***c***

***\**** ***and*** ***sets*** ***attributes*** ***of*** ***new*** ***object***

***\**** ***to*** ***the*** ***same.*** ***Reserves*** ***space*** ***in***

***\**** ***\_elems,*** ***if*** ***need*** ***be.***

***\****

***\**** @param ***c*** ***ArrayList*** ***to*** ***be*** ***copied***

***\*/***

**ArrayList**(*const* ArrayList<T>& c) {

\_size = c.\_size;

\_reserved = c.\_reserved;

*if* (\_size > 0) {

\_elems = *new* T[\_reserved];

}

std::copy(c.\_elems, c.\_elems + c.\_size, \_elems);

}

*//* *Move* *constructor*

***/\*\**** ***Moves*** ***an*** ***Arraylist*** ***to*** ***create*** ***a*** ***new*** ***ArrayList.***

***\****

***\**** ***Takes*** ***a*** ***temporary*** ***(moved)*** ***ArrayList***

***\**** ***and*** ***sets*** ***attributes*** ***of*** ***new*** ***object***

***\**** ***to*** ***the*** ***same.*** ***Then*** ***cleans*** ***up*** ***the*** ***temporary***

***\**** ***ArrayList.***

***\****

***\**** @param ***c*** ***ArrayList*** ***to*** ***be*** ***moved***

***\*/***

**ArrayList**(ArrayList<T>&& c) {

\_elems = c.\_elems;

\_size = c.\_size;

\_reserved = c.\_reserved;

c.\_elems = *nullptr*;

c.\_size = 0;

c.\_reserved = 0;

}

*//* *Constructor* *with* *initialization* *of* *"initialized"* *elements*

***/\*\**** ***Initializes*** ***an*** ***ArrayList*** ***with*** ***given*** ***reserved*** ***space.***

***\****

***\**** ***Initializes*** ***a*** ***new*** ***ArrayList.***

***\****

***\**** @param ***initialized*** ***desired*** ***reserve*** ***size***

***\*/***

**ArrayList**(int initialized) {

*if* (initialized <= 0) {

*throw* std::invalid\_argument("Index has to be larger than 0");

}

\_size = 0;

\_reserved = initialized;

\_elems = *new* T[\_reserved];

}

***/\*\**** ***Deconstructs*** ***object.***

***\****

***\**** ***Deconstructs*** ***object*** ***by*** ***deleting***

***\**** ***allocated*** ***space*** ***on*** ***heap.***

***\****

***\*/***

*virtual* ~***ArrayList***() {

*if* (\_reserved > 0) {

*delete*[] \_elems;

}

}

*//* *Copy* *assignment* *operator*

***/\*\**** ***Moves*** ***an*** ***Arraylist*** ***to*** ***overwride*** ***this.***

***\****

***\**** ***Takes*** ***a*** ***temporary*** ***(moved)*** ***ArrayList***

***\**** ***and*** ***sets*** ***attributes*** ***of*** ***new*** ***object***

***\**** ***to*** ***the*** ***same.*** ***Then*** ***cleans*** ***up*** ***the*** ***temporary***

***\**** ***ArrayList.***

***\****

***\**** @param ***a*** ***ArrayList*** ***to*** ***be*** ***moved***

***\*/***

ArrayList<T >& *operator*=(*const* ArrayList<T>& a) {

*if* (\_reserved > 0) {

*delete*[] \_elems;

}

\_size = a.\_size;

\_reserved = a.\_reserved;

*if* (\_size > 0) {

\_elems = *new* T[\_reserved];

}

std::copy(a.\_elems, a.\_elems + a.\_size, \_elems);

*return* \**this*;

}

*//* *Move* *assignment* *operator*

***/\*\**** ***Moves*** ***an*** ***Arraylist*** ***to*** ***overwride*** ***this.***

***\****

***\**** ***Takes*** ***a*** ***temporary*** ***(moved)*** ***ArrayList***

***\**** ***and*** ***sets*** ***attributes*** ***of*** ***new*** ***object***

***\**** ***to*** ***the*** ***same.*** ***Then*** ***cleans*** ***up*** ***the*** ***temporary***

***\**** ***ArrayList.***

***\****

***\**** @param ***a*** ***ArrayList*** ***to*** ***be*** ***moved***

***\*/***

ArrayList <T>& *operator*=(ArrayList <T>&& a) {

*if* (\_reserved > 0) {

*delete*[] \_elems;

}

\_elems = a.\_elems;

\_size = a.\_size;

\_reserved = a.\_reserved;

a.\_elems = *nullptr*;

a.\_size = 0;

a.\_reserved = 0;

*return* \**this*;

}

*//* *Adds* *element* *to* *dynamic* *array*

***/\*\**** ***Adds*** ***element*** ***to*** ***dynamic*** ***array.***

***\****

***\**** ***Reserves*** ***more*** ***space*** ***in*** ***\_elems*** ***if*** ***need*** ***be***

***\**** ***and*** ***adds*** ***the*** ***element*** ***to*** ***the*** ***array.***

***\****

***\**** @param ***element*** ***desired*** ***element*** ***to*** ***add***

***\*/***

void **add**(*const* T& element) {

*if* (\_size == \_reserved)

extendStorage();

\_elems[\_size] = element;

++\_size;

}

*/\**

*\** *Inserts* *the* *element* *at* *placement* *"idx* *"* *in* *array* *and* *moves* *the* *remaining*

*\** *items* *by* *one* *place* *,* *restoring* *the* *old* *element* *at* *"idx* *".*

*\** *check* *whether* *it* *is* *needed* *to* *extend* *the* *storage* *.*

*\** *move* *all* *elements* *from* *\_size* *to* *idx* *(reverse)* *one* *element* *to* *the* *right* *in* *the* *array*

*\** *set* *\_elems* *[idx]* *equal* *to* *the* *element* *to* *be* *inserted*

*\*/*

***/\*\**** ***Inserts*** ***element*** ***to*** ***dynamic*** ***array*** ***at*** ***index.***

***\****

***\**** ***Inserts*** ***the*** ***element*** ***at*** ***placement*** ***"idx*** ***"*** ***in*** ***array*** ***and*** ***moves*** ***the*** ***remaining***

***\**** ***items*** ***by*** ***one*** ***place*** ***,*** ***restoring*** ***the*** ***old*** ***element*** ***at*** ***"idx*** ***".***

***\**** ***Checks*** ***whether*** ***it*** ***is*** ***needed*** ***to*** ***extend*** ***the*** ***storage*** ***.***

***\**** ***Moves*** ***all*** ***elements*** ***from*** ***\_size*** ***to*** ***idx*** ***(reverse)*** ***one*** ***element*** ***to*** ***the*** ***right*** ***in*** ***the*** ***array***

***\**** ***Sets*** ***\_elems*** ***[idx]*** ***equal*** ***to*** ***the*** ***element*** ***to*** ***be*** ***inserted***

***\****

***\**** @param ***idx*** ***desired*** ***index*** ***of*** ***insertion***

***\**** @param ***element*** ***desired*** ***element*** ***to*** ***add***

***\*/***

void **add**(int idx, *const* T& element) {

*if* (idx <= 0) {

*throw* std::invalid\_argument("Index has to be larger than 0");

}

*if* (\_size == \_reserved)

extendStorage();

*for* (int i = \_size + 1; i > idx; --i) {

\_elems[i] = \_elems[i - 1];

}

\_elems[idx] = element;

++\_size;

}

*//* *Get* *a* *const* *reference* *to* *the* *element* *at* *idx*

***/\*\**** ***Gets*** ***element*** ***at*** ***index.***

***\****

***\**** ***Get*** ***a*** ***const*** ***reference*** ***to*** ***the*** ***element*** ***at*** ***idx***

***\****

***\**** @param ***idx*** ***desired*** ***index*** ***of*** ***element***

***\*/***

*const* T& *operator*[](int idx) *const* {

*if* (idx < 0 || idx > \_size) {

*throw* std::invalid\_argument("Index out of range");

}

*return* \_elems[idx];

}

*//* *Get* *a* *reference* *to* *the* *element* *at* *idx*

***/\*\**** ***Gets*** ***element*** ***at*** ***index.***

***\****

***\**** ***Get*** ***a*** ***reference*** ***to*** ***the*** ***element*** ***at*** ***idx***

***\****

***\**** @param ***idx*** ***desired*** ***index*** ***of*** ***element***

***\*/***

T& *operator*[](int idx) {

*if* (idx < 0 || idx > \_size) {

*throw* std::invalid\_argument("Index out of range");

}

*return* \_elems[idx];

}

*/\**

*\** *Removes* *the* *element* *at* *placement* *"idx* *"* *by* *moving* *all* *the* *remaining* *elements*

*\** *by* *one* *place* *to* *the* *left* *in* *the* *array*

*\*/*

***/\*\**** ***Removes*** ***element*** ***at*** ***index.***

***\****

***\**** ***Removes*** ***the*** ***element*** ***at*** ***placement*** ***"idx*** ***"*** ***by*** ***moving*** ***all*** ***the*** ***remaining*** ***elements***

***\**** ***by*** ***one*** ***place*** ***to*** ***the*** ***left*** ***in*** ***the*** ***array***

***\****

***\**** @param ***idx*** ***desired*** ***index*** ***of*** ***element***

***\*/***

void **remove**(int idx) {

*if* (idx < 0 || idx > \_size) {

*throw* std::invalid\_argument("Index out of range");

}

*for* (int i = idx; i < \_size - 1; i++) {

\_elems[i] = \_elems[i + 1];

}

--\_size;

}

*//* *Returns* *the* *number* *of* *elements* *stored*

***/\*\**** ***Returns*** ***the*** ***number*** ***of*** ***elements*** ***stored.***

***\****

***\**** ***Returns*** ***the*** ***number*** ***of*** ***elements*** ***stored***

***\****

***\*/***

int **size**() *const* { *return* \_size; }

*//* *Returns* *the* *number* *of* *items* *currently* *reserved* *inmemory*

***/\*\**** ***Returns*** ***the*** ***number*** ***of*** ***items*** ***currently*** ***reserved*** ***inmemory.***

***\****

***\**** ***Returns*** ***the*** ***number*** ***of*** ***items*** ***currently*** ***reserved*** ***inmemory.***

***\****

***\*/***

int **reserved**() *const* { *return* \_reserved; }

*//* *Returns* *true* *if* *number* *of* *elements* *in* *array* *is* *zero*

***/\*\**** ***Returns*** ***true*** ***if*** ***number*** ***of*** ***elements*** ***in*** ***array*** ***is*** ***zero.***

***\****

***\**** ***Returns*** ***true*** ***if*** ***number*** ***of*** ***elements*** ***in*** ***array*** ***is*** ***zero.***

***\****

***\*/***

bool **isEmpty**() *const* { *return* (\_size == 0) ? *true* : *false*; }

*//* *Trims* *the* *storage* *array* *to* *the* *exact* *number* *of* *elements* *stored.*

***/\*\**** ***Trims*** ***the*** ***storage*** ***array*** ***to*** ***the*** ***exact*** ***number*** ***of*** ***elements*** ***stored.***

***\****

***\**** ***Trims*** ***the*** ***storage*** ***array*** ***to*** ***the*** ***exact*** ***number*** ***of*** ***elements*** ***stored.***

***\****

***\*/***

void **trimToSize**() {

*//* *Sets* *the* *reserved* *size* *to* *be* *the* *same*

*//* *as* *the* *number* *of* *elements* *in* *array*

\_reserved = \_size;

*//* *Reserves* *space* *in* *a* *temporary* *variable.*

T\* temp = *new* T[\_reserved];

*//* *Moves* *elements* *from* *active* *\_elems* *to* *temporary* *list.*

std::move(\_elems, \_elems + \_size, temp);

*//* *Deletes* *\_elems* *to* *make* *room* *for* *new* *data*

*//* *as* *only* *elements* *and* *not* *\_reserved* *in* *\_elems*

*//* *has* *been* *moved.*

*delete*[] \_elems;

*//* *Reserves* *space* *in* *\_elems*

\_elems = *new* T[\_reserved];

*//* *Moves* *the* *elements* *from* *temp* *array* *to* *elems.*

std::move(temp, temp + \_size, \_elems);

}

*/\**

*\** *Sorts* *the* *array* *using* *insertion* *sort* *(or* *another* *algorithm)*

*\** *You* *are* *not* *allowed* *to* *use* *standard* *algorithms* *from* *algorithm* *header.*

*\*/*

***/\*\**** ***Sorts*** ***the*** ***array*** ***using*** ***insertion*** ***sort.***

***\****

***\**** ***Sorts*** ***the*** ***array*** ***using*** ***insertion*** ***sort.***

***\****

***\*/***

void **sort**() {

T relative;

int i, j;

*for* (i = 1; i < \_size; i++) {

relative = \_elems[i];

j = i - 1;

*//* *Move* *the* *elements* *of* *pos* *[0* *to* *i-1],*

*//* *that* *are* *greater* *than* *the* *"relative"* *one* *position*

*//* *ahead* *of* *their* *current* *pos*

*while* (j >= 0 && \_elems[j] > relative)

{

\_elems[j + 1] = \_elems[j];

--j;

}

\_elems[j + 1] = relative;

}

}

*//* *Returns* *a* *new* *ArrayList* *with* *elements* *from* *"fromIdx"* *index* *to* *"toIdx"*

***/\*\**** ***Returns*** ***a*** ***new*** ***ArrayList*** ***with*** ***elements*** ***from*** ***"fromIdx"*** ***index*** ***to*** ***"toIdx".***

***\****

***\**** ***Returns*** ***a*** ***new*** ***ArrayList*** ***with*** ***elements*** ***from*** ***"fromIdx"*** ***index*** ***to*** ***"toIdx".***

***\****

***\**** @param ***fromIdx*** ***desired*** ***first*** ***index***

***\**** @param ***toIdx*** ***desired*** ***second*** ***index***

***\*/***

ArrayList<T> **subArrayList**(int fromIdx, int toIdx) *const* {

*if* (fromIdx > toIdx) {

*throw* std::invalid\_argument("fromIdx is larger than toIdx");

} *else* *if* (fromIdx == toIdx) {

*throw* std::invalid\_argument("The two indexes are the same");

} *else* *if* (fromIdx < 0 || toIdx < 0) {

*throw* std::invalid\_argument("An index is less than 0");

} *else* *if* (fromIdx > \_size || toIdx > \_size) {

*throw* std::invalid\_argument("An index is greater than the size of the ArrayList");

}

ArrayList<T> array((toIdx - fromIdx) + 1);

*for* (int i = fromIdx; i <= toIdx; ++i) {

array.add(\_elems[i]);

}

*return* array;

}

*//* *Returns* *a* *new* *C* *style* *array* *(copy* *created* *with* *new)* *with* *all* *elements*

***/\*\**** ***Returns*** ***a*** ***new*** ***C*** ***style*** ***array*** ***(copy*** ***created*** ***with*** ***new)*** ***with*** ***all*** ***elements.***

***\****

***\**** ***Returns*** ***a*** ***new*** ***C*** ***style*** ***array*** ***(copy*** ***created*** ***with*** ***new)*** ***with*** ***all*** ***elements.***

***\****

***\*/***

T\* **toArray**() {

T\* cArray = *new* T[\_size];

*for* (int i = 0; i < \_size; i++)

{

cArray[i] = \_elems[i];

}

*return* cArray;

}

*private*:

*/\**

*\** *extendStorage():*

*\** *create* *new* *array* *with* *size* *2\** *\_reserved*

*\** *copy* *old* *data* *to* *the* *new* *array*

*\** *delete* *old* *array*

*\** *update* *pointer* *\_elems* *to* *point* *to* *the* *new* *array*

*\** *(Since* *this* *method* *is* *private,* *the* *method* *will* *only* *be* *used* *internally,*

*\** *but* *the* *functionality* *is* *needed).*

*\*/*

***/\*\**** ***Extends*** ***the*** ***reserved*** ***storage*** ***of*** ***\_elems.***

***\****

***\**** ***Extends*** ***the*** ***reserved*** ***storage*** ***of*** ***\_elems.***

***\****

***\*/***

void **extendStorage**() {

*//* *Reserves* *1* *spot,* *if* *there* *are* *non* *reserved* *else* *twice* *the* *size*

\_reserved = (\_reserved == 0) ? 1 : \_reserved \* 2;

*//* *Reserves* *just* *space* *for* *elements* *in* *a* *temporary* *variable.*

T\* temp = *new* T[\_size];

*//* *Moves* *elements* *from* *active* *\_elems* *to* *temporary* *list.*

std::move(\_elems, \_elems + \_size, temp);

*//* *Deletes* *\_elems* *to* *make* *room* *for* *new* *data*

*//* *as* *only* *elements* *and* *not* *\_reserved* *in* *\_elems*

*//* *has* *been* *moved.*

*delete*[] \_elems;

*//* *Reserves* *space* *in* *\_elems*

\_elems = *new* T[\_reserved];

*//* *Moves* *the* *elements* *from* *temp* *array* *to* *elems.*

std::move(temp, temp + \_size, \_elems);

}

*//* *Member* *variables*

int \_reserved; *//* *The* *current* *capacity* *of* *"* *\_elems* *"* *array*

int \_size; *//* *The* *number* *of* *elements* *stored*

T\* \_elems; *//* *Array* *for* *storing* *the* *elements*

};

# endif *//* *ARRAYLIST\_H*

## “Main.cpp”

#include <iostream>

#include "ArrayList.h"

int **main** () {

ArrayList<double> array;

*for* (int i = 0; i < 33; ++i) {

array.add(i \* 1.1);

}

array.add(4, -5);

array.remove(1);

*for* (int i = 0; i < array.size() + 1; ++i) {

std::cout << array[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

std::cout << "ArrayList have reserved space for " << array.reserved()

<< " elements and stores " << array.size () << " elements." << std::endl;

array.trimToSize();

std::cout << "ArrayList is now trimmed and have reserved space for " << array.reserved()

<< " elements and stores " << array.size() << " elements." << std::endl << std::endl;

ArrayList<double> subArray = array.subArrayList(1 ,10);

std::cout << "Sub ArrayList have reserved space for " << subArray.reserved ()

<< " elements and stores " << subArray.size() << " elements ." << std::endl;

*for* (int i = 0; i < subArray.size(); ++i) {

std::cout << subArray[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

double\* sArray = subArray.toArray();

*for* (int i = 0; i < 7; ++i) {

subArray.add(i+1);

}

*//* *The* *basic* *array* *prints* *after* *adding* *elements* *to* *subArray*

std::cout << "Basic array of sub array contains: " << std::endl;

*for* (int i = 0; i < subArray.size() - 7; ++i) {

std::cout << sArray[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

*//* *The* *sub* *array* *now* *has* *elements*

std::cout << "Sub array with 7 elements added: " << std::endl;

*for* (int i = 0; i < subArray.size(); ++i) {

std::cout << subArray[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

*//* *The* *original* *array* *has*

std::cout << "Original array " << std::endl;

*for* (int i = 0; i < array.size(); ++i) {

std::cout << array[i] << ", ";

}

std::cout << "\b\b " << std::endl;

*//* *Copy* *constructor*

*//* *Copy* *assignment*

ArrayList<double> array2 = array;

ArrayList<double> subArray2;

subArray2 = subArray;

std::cout << "array2 = ";

*for* (int i = 0; i < array2.size(); ++i) {

std::cout << array2[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

std::cout << "subArray2 = ";

*for* (int i = 0; i < subArray2.size(); ++i) {

std::cout << subArray2[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

*//* *Move* *assignment* *operator*

ArrayList<double> subArray3 = std::move(*subArray2*);

array2 = std::move(*subArray2*);

std::cout << "subArray2 = ";

*for* (int i = 0; i < subArray2.size(); ++i) {

std::cout << subArray2[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

std::cout << "subArray3 = ";

*for* (int i = 0; i < subArray3.size(); ++i) {

std::cout << subArray3[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

std::cout << "-----------------------From here and down is our added tests----------------------- \n\n";

std::cout << "Test of isEmpty() \nsubArray2 should be empty, as its elements has been moved. subArray2 is empty (1): ";

std::cout << subArray2.isEmpty();

std::cout << std::endl << std::endl;

double\* cArray = array.toArray();

std::cout << "C-style Array based on original array = ";

*for* (int i = 0; i < array.size(); ++i) {

std::cout << cArray[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

subArray3.sort();

std::cout << "Sorted subArray3 = ";

*for* (int i = 0; i < subArray3.size(); ++i) {

std::cout << subArray3[i] << ", ";

}

std::cout << "\b\b " << std::endl << std::endl;

*return* 0;

}