**חלק יבש**

שרה גריפית + ליאור בר יוסף

341312304 + 207022443

סעיף א':

לא ניתן להסתפק בלהגדיר את פעולות האירטור הרגיל כ-const משום שיש מתודות ופונקציות שברצוננו להשתמש בהם על אובייקט מטיפוס queue שמשנה אותו.

ה-ConstIterator הוא איטרטור אשר מצביע על אובייקט const. כלומר, הוא מצביע על אובייקטים אשר לא ניתן לשנות את ערכם או האיברים בהם.

כחלק מדרישת מימוש המחלקה תור, נרצה להוסיף איברים לתור (בעזרת המתודה pushBack()), להסיר איברים מהתור (בעזרת המתודה popFront()), לשנות את האיבר הראשון במערך (בעזרת המתודה front()) ואף לשנות את איברי התור (בעזרת הפונקציה transform()). כדי לבצע את הפעולות הללו, יש צורך במעבר על איברי התור עד ההגעה לאיבר הרצוי עבור הפעולה הדרושה. הדרך לעשות זאת (במיוחד עבור פונקציות חיצוניות למחלקה, כמו transform(), שאין עבורם גישה לשדות הפנימיים של התור) הוא בעזרת איטרטורים.

אם האיטרטור היה ConstIterator, אזי הוא היה מצביע על queue const, ואז לא היה ניתן לשנות את התור, בניגוד לצורך של המתודות והפונקציות הנ"ל! ולכן עבור המתודות והפונקציות הללו, נדרש איטרטור שמצביע על queue שלא const, ובכך ניתן להצביע את האיבר הרצוי בתור ולשנותו כנדרש.

סעיף ב':

קיימות לא מעט מתודות ופונקציות בממשק התור שעליהם יש הנחות על הטיפוס הטמפלייטי:

**מחלקת Queue:**

* ה-constructor – מתודת Queue()
  + עבור השורות:

m\_node = new Node;

Node \*nodeNew = new Node;

בשורות הללו, נוצרים שני node-ים חדשים על ידי הקצאת זיכרון בעזרת הפעולה new. כל node מכיל משתנה מהטיפוס הטמפלייטי, ולכן ביצירת ה-nodes, הconstructor של הnodes נקרא, ומשם נקראה הconstructor של הטיפוס הטמפלייטי. יש כאן הנחה שאם יש צורך **בהקצאת זיכרון** עבור יצירת אובייקט מהטיפוס הטמפלייטי, אז זהו נעשה בconstructor שלו.

* + עבור השורות:

delete m\_node;

delete nodeNew;

בשורות האלו, "מוחקים" את הזיכרון שהקצנו עבור המשתנים הללו במקרה שנזרק שגיאת bad\_alloc בפעולת new. אלו שני משתני nodes, אשר כל אחד אמור להכיל משתנה מהטיפוס הטמפלייטי. כאשר מוחקים את הnodes, הdestructor של הטיפוס הטמפלייטי נקרא גם הוא. יש כאן הנחה, שאם הוקצה זיכרון עבור יצירת הטיפוס הטמפלייטי, אזי **הזיכרון משוחרר** כאשר נקרא הdestructor שלו.

* הcopy-constructor – מתודת Queue(const Queue& original)
  + עבור השורות:

m\_node = new Node;

Node \*nodeNew = new Node;

בשורות הללו, נוצרים שני node-ים חדשים על ידי הקצאת זיכרון בעזרת הפעולה new. כל node מכיל משתנה מהטיפוס הטמפלייטי, ולכן ביצירת ה-nodes, הconstructor של הnodes נקרא, ומשם נקראה הconstructor של הטיפוס הטמפלייטי. יש כאן הנחה שאם יש צורך **בהקצאת זיכרון** עבור יצירת אובייקט מהטיפוס הטמפלייטי, אז זהו נעשה בconstructor שלו.

* + עבור השורות:

delete m\_node;

delete nodeNew;

בשורות האלו, "מוחקים" את הזיכרון שהקצנו עבור המשתנים הללו במקרה שנזרק שגיאת bad\_alloc בפעולת new. אלו שני משתני nodes, אשר כל אחד אמור להכיל משתנה מהטיפוס הטמפלייטי. כאשר מוחקים את הnodes, הdestructor של הטיפוס הטמפלייטי נקרא גם הוא. יש כאן הנחה, שאם הוקצה זיכרון עבור יצירת הטיפוס הטמפלייטי, אזי **הזיכרון משוחרר** כאשר נקרא הdestructor שלו.

* מתודת pushback()
  + עבור השורה:

endIt.m\_currentNode->m\_data = data;

ניתן לראות שיש פעולת השמה (Assignment Operator) בין שני אובייקטים מהטיפוס הטמפלייטי. לכן ההנחה כאן היא **שקיימת** מתודה כזו עבור הטיפוס הטמפלייטי, ושהיא מבצעת **העתקה** של האובייקט (ולא למשל העברת פוינטרים בלבד) **ומקצה זיכרון** עבורו במקרה הצורך.

* מתודת front()
  + עבור השורות:

T& Queue<T>::front()

return m\_node->m\_next->m\_data;

כאן אנו מחזירים reference לאובייקט מהטיפוס הטמפלייטי. ההנחה כאן היא שהאובייקט הזה **איננו NULL**, משום שreference לא יכול להיות null. כלומר, בשום מקום בתכנית, האובייקט הזה לא הופך לNULL: ביצירה שלו, בשינוי שלו בפונקצית transform וכו'...

* מתודת front() const
  + עבור השורות:

const T& Queue<T>::front() const

return m\_node->m\_next->m\_data;

כאן אנו מחזירים const reference לאובייקט מהטיפוס הטמפלייטי. ההנחה כאן היא שהאובייקט הזה **איננו NULL**, משום שreference לא יכול להיות null. כלומר, ביצירת האובייקט, אנו מניחים שהוא איננו נוצר כNULL (משום שהוא CONST הוא לא יכל להשתנות אחרי יצירתו).

**מחלקת Queue, non-member functions:**

* פונקציית filter()
  + עבור השורה:

for (const T& elem : givenQueue) {

כאן אנו משתמשים בconst reference לאובייקט מהטיפוס הטמפלייטי. ההנחה כאן היא שהאובייקט הזה **איננו NULL**, משום שreference לא יכול להיות null. כלומר, בשום מקום בתכנית, האובייקט הזה לא הופך לNULL: ביצירה שלו, בשינוי שלו בפונקצית transform וכו'...

* + עבור השורה:

if (c(elem)) {

ניתן לראות שאובייקט const מהטיפוס הטמפלייטי נשלח לפונקציית c מטיפוס טמפלייטי שני אשר מתקבלת כקלט של הפונקציה. ההנחה כאן היא שהפונקציה c **מקבלת** **אובייקט מהטיפוס הטמפלייטי** הראשון, **איננה משנה** את האובייקט וכן **מחזירה ערך בולייאני**.

* + שדגכ
* פונקציית transform()
  + עבור השורה:

for (T& elem : givenQueue) {

כאן אנו משתמשים בreference לאובייקט מהטיפוס הטמפלייטי. ההנחה כאן היא שהאובייקט הזה **איננו NULL**, משום שreference לא יכול להיות null. כלומר, בשום מקום בתכנית, האובייקט הזה לא הופך לNULL: ביצירה שלו, בשינוי שלו בפונקצית transform וכו'...

* + עבור השורה:

c(elem);

ניתן לראות שאובייקט מהטיפוס הטמפלייטי נשלח לפונקציית c מטיפוס טמפלייטי שני אשר מתקבלת כקלט של הפונקציה. ההנחה כאן היא שהפונקציה c **מקבלת** **אובייקט מהטיפוס הטמפלייטי** הראשון.

**מחלקת Node:**

* המתודה destroyNode()
  + עבור השורות:

delete node;

בשורה הזו, "מוחקים" את הזיכרון שהקצנו עבור כל הnodes המשורשרים שהתקבלו כפרמטר למתודה. כל אחד מהnodes מכיל משתנה מהטיפוס הטמפלייטי. כאשר מוחקים את הnodes, הdestructor של הטיפוס הטמפלייטי נקרא גם הוא. יש כאן הנחה, שאם הוקצה זיכרון עבור יצירת הטיפוס הטמפלייטי, אזי **הזיכרון משוחרר** כמו שצריך כאשר נקרא הdestructor שלו.

**מחלקת Iterator:**

* המתודה operator\*
  + עבור השורות:

T& Queue<T>::Iterator::operator\*() const

return m\_currentNode->m\_data;

כאן אנו מחזירים reference לאובייקט מהטיפוס הטמפלייטי. ההנחה כאן היא שהאובייקט הזה **איננו NULL**, משום שreference לא יכול להיות null. כלומר, בשום מקום בתכנית, האובייקט הזה לא הופך לNULL: ביצירה שלו, בשינוי שלו בפונקצית transform וכו'...

**מחלקת ConstIterator:**

* המתודה operator\*
  + עבור השורות:

const T& Queue<T>::ConstIterator::operator\*() const

return m\_currentNode->m\_data;

כאן אנו מחזירים const reference לאובייקט מהטיפוס הטמפלייטי. ההנחה כאן היא שהאובייקט הזה **איננו NULL**, משום שreference לא יכול להיות null. כלומר, ביצירת האובייקט, אנו מניחים שהוא איננו נוצר כNULL (משום שהוא CONST הוא לא יכל להשתנות אחרי יצירתו).

סעיף ג':

Text

Description automatically generatedכאשר הסטודנט ינסה לקמפל את התרגיל כאשר מחלקת הQueue שלו ממומש בקבוץ c++, אזי נזקרת שגיאה של **"Undefined Reference"**.

השגיאת קומפילציה הזו מתרחשבת **בשלב הקישור** (linking process), והיא נזרקת כאשר יש פנייה לאבוייקט כלשהו (מחלקה, פונקציה, משתנה, וכו'...) והלינקר בשלב הקומפילציה לא מוצא את ההגדרה שלו.

במקרה שלנו:

עבור מחלקת Queue, השתמשנו ב**template**.

ההחלפה של הטיפוס הטמפלייטי קורה בזמן הקומפילציה. כלומר, הקומפיילר "יוצר עותק" של המחלקה לפי הטיפוס הטמפלייטי הנדרש בתכנית. לכן הקומפיילר צריך גישה ישירה לכל המחלה (כולל מימושה) כדי להתאים אותה לטיפוס הנדרש.

יש לקומפיילר גישה לקבוץ ה".h", ולכן בתוך קובץ זה צריך להיות כל המידע על המחלקה הטמפלייטי. **אם מימוש המחלקה ממוקם** **מחוץ לקובץ ה".h",** **אין לקומפיילר גישה אליו**. הקומפיילר לא יצליח לקשר בין המחלקה לבין המימוש שלה, וכתוצאה מכך, הקומפליציה נכשלת ונזרקת השגיאה המתאימה"Undefined Reference".

לכן עבור templates, צריך למקם את מימוש המחלקה בתוך קובץ ה".h" ולא בתוך קובץ ה".cpp".

סעיף ד':

הפונצקיית filter() במחלקת Queue מקבלת שני פרמטרים: const reference לאובייקט מטיפוס queue, וגם object function או מצביע לפונקציה. היא עוברת על כל האיברים בתור, ובודקת אם האיברים עונים על התנאי של הobject function/מצביע לפונקציה הניתן כפרמטר. אם כן, אז האיברים נכנסים לתור חדש, אשר הוא הערך המוחזר מהפונקציה filter().

הסטודנטית רוצה להשתמש בפונקציה הזו כדי לסנן תור של מספרים שלמים כך שישארו רק מספרים המתחלקים במספר כלשהו שאינו ידוע בזמן הקומפילציה (אלא רק בזמן ריצה).

היא יכולה לעשות זאת באמצעות השלבים הבאים:

* בניית function object:

הסטודנטית צריכה לבנות מחלקה חדשה, אשר מכילה member אחד אשר הוא המספר שבו היא רוצה לחלק את האיברים בתור.

המחלקה תכיל מתודה אחת אשר דורסת את operator(), שבודקת אם הקלטים המתקבלים מתחלקים בmember של האובייקט עליו נקראת המתודה.

class DividesBy {

public:

//Constructor

DividesBy(int divisor) : m\_divisor(divisor) {}

//Default copy constructor, assignment operator and destructor

DividesBy(const DividesBy&) = default;

DividesBy & operator=(const DividesBy& other) = default;

~ DividesBy() = default;

//operator() overloading

bool operator()(int dividend) const

{

return (dividend % m\_divisor == 0);

}

private:

int m\_divisor;

}

נשים לב, שבבניית אובייקט מהטיפוס של המחלקה הזו נקראת הconstructor (לדוגמא *DividesBy firstDivision(5);*), והקלט (בדוגמא שלנו, המספר 5) יהיה המספר שבו רוצים לחלק את האיברים בתור (*m\_divisor = 5*). בקריאת אותו האובייקט שנבנה עם הסוגריים "()"

(*bool result = firstDivision(25);*), זהו בעצם קריאה למתודה operator() על הקלט (כלומר יבדק אם 25 מתחלק ב5, ולכן בresult יהיה true).

* בניית אובייקט תור מטיפוס int (כי היא דורשת שהתור יכיל רק מספרים שלמים), והכנסה לתור מספרים שלמים לפי צרכיה, כמו בדוגמא למטה:

Queue<int> queue1;

queue1.pushBack(564);

* השמה לתוך התור המקורי את תוצאת הפונקציה filter(), כאשר לפונקציה ניתן אותו התור כפרמטר ואת ה-function object שנבנה בשלב קודם.

הfunction object שישלח כפרמטר יקבל כערך התחלתי את המספר שצריך לחלק בו את כל האיברים בתור. הערך ההתחלתי יכול להיות משתנה שנקבע בתכנית של הסטודנטית בשלב קודם לכן, ובכך עונה על הדרישה כי המספר הזה נקבע רק בזמן ריצה. בדוגמא למטה, המשתנה הזה נקרא divisor.

ההשמה של התוצאה של filter() היא לתוך אותו התור המקורי, משום שהסטודנטית רוצה לסנן את התור המקורי ובעצם לשנות אותו. ההשמה תעתיק את התור החדש שנוצר בfilter() לתוך התור המקורי, ובכך הסטודנטית תישאר עם התור המקורי המעודכן.

queue1 = filter(queue1, DividesBy(divisor))

להלן התכנית לדוגמא:

**מחלקת DividesBy בקובץ "DividesBy.h":**

#ifndef QUEUE\_H\_DIVIDESBY\_H  
#define QUEUE\_H\_DIVIDESBY\_H  
  
/\*  
 \* DividesBy:  
 \* This class builds function objects that check if the input integer divides by a given number  
\*/  
class DividesBy {  
public:  
 //Constructor  
 DividesBy(int divisor) : m\_divisor(divisor) {}  
  
 //Default copy constructor, assignment operator and destructor  
 DividesBy(const DividesBy&) = default;  
 DividesBy & operator=(const DividesBy& other) = default;  
 ~ DividesBy() = default;  
  
 //operator() overloading  
 bool operator()(int dividend) const  
 {  
 return (dividend % m\_divisor == 0);  
 }  
private:  
 int m\_divisor;  
};  
  
#endif //QUEUE\_H\_DIVIDESBY\_H

**התכנית הראשית בקובץ "mainQuestionD.cpp":**

#include "Queue.h"  
#include "DividesBy.h"  
#include <iostream>

int main () {  
 Queue<int> queue1;  
 for (int i = 1; i <= 100; i++) {  
 queue1.pushBack(i);  
 }  
 int queueSize = queue1.size();  
 std::cout << "Before changes, the queue's size is: " << queueSize << std::endl;  
 std::cout << "The numbers in queue: " << std::endl;  
 for (int& elem : queue1) {  
 std::cout << elem << std::endl;  
 }  
  
 //Can be decided upon during the student's program  
 int divisor = 5;  
  
 queue1 = filter(queue1, DividesBy(divisor));  
  
 queueSize = queue1.size();  
 std::cout << "After changes, the queue's size is: " << queueSize << std::endl;  
 std::cout << "The numbers in queue: " << std::endl;  
 for (int& elem : queue1) {  
 std::cout << elem << std::endl;  
 }  
   
 //Continuation of student's program  
  
 return 0;  
}

**התוצאה של התוכנית:**

אכן התור סונן בהצלחה!

*Graphical user interface, text

Description automatically generated-(all the numbers between 1 and 100)-*

Text

Description automatically generated