|  |  |  |
| --- | --- | --- |
| **SWO 3x** | **Übung zu Softwareentwicklung mit**  **klassischen Sprachen u. Bibliotheken 3** | **WS 2021/22, Übung 06** |

**Abgabe elektronisch bis Sa 8 Uhr in der KW 49**

* ~~Gr. 1,~~ Winkler, BSc Msc **Name: Roman Kofler-Hofer Aufwand in h: 15**
* **Gr. 1,** Dr. Pitzer

**Punkte: Kurzzeichen Tutor / Übungsleiter /**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Beispiel | L Lösungsidee | I Implementierung | T Tests | S = L+I+T | Multiplikator | S\*M |
| 1a | 3 | 4 | 3 | 10 | 6 | 60 |
| 1b | 3 | 4 | 3 | 10 | 4 | 40 |
|  |  |  |  |  | **Summe** | **100** |

1. **DES Bibliothek (Teilaufgabe a)**
   1. **Lösungsidee:**

Es gibt zwei zentrale Klassen. 1) Die Simulation und 2) Events, welche einen Zeitpunkt haben, an dem sie auftreten. Ein Objekt der Simulationsklasse beinhält eine priority\_queue, die die Events auf Basis des Eintrittszeitpunkts verwaltet.

Events benötigen Zugriff auf die Datenkomponenten, der Simulation zu der sie gehören. Daher enthält jedes Event auch eine Datenkomponente mit Zeiger auf die Simulation, die das Event verwaltet.

Ansonsten entspricht meine Herangehensweise der zweiten skizzierten Lösung. Die Klasse Event ist abstrakt und muss dann von konkreten Events überschrieben werden. Die Queue verwaltet daher Zeiger auf Events (für Polymorphismus). Aber nicht nur die Klasse "Event" ist abstrakt, sondern auch die Klasse Simulation. Es gibt nämlich die abstrakte Methode "hasEnded", über die man die Information erhält, ob das Endkriterium für eine Simulation schon eingetroffen ist oder nicht. Somit muss auch diese von einer konkreten Simulations-Klasse überschrieben werden (in meinem Fall heißt die abgeleitete Klasse "MarketSimulation".

Was bei mir nicht funktioniert hat, war der angegebene Vorschlag zur Deklaration der Queue bzw. der Vergleichsfunktion (den Vergleichsoperator für ein Event habe ich auch implementiert).

std**::**priority\_queue**<**event**\*,** std**::**vector**<**event**\*>,** std**::**function**<**bool**(**event**\*,** event**\*)>>**

m\_events **{**

**[](**event **\***a**,** event **\***b**)** **{** **return** **\***a **>** **\***b**;** **}**

**};**

In dieser Implementierung habe ich beim Aufruf von "addEvent" ständig einen Error "bad\_function\_call" erhalten, den ich nicht auflösen konnte.

Nach einiger Recherche bin ich auf folgende Lösung mit einem Funktor gekommen.

class Simulation

**{**

struct CustomComparator **{**

bool **operator()(**Event**\*** e1**,** Event**\*** e2**)** **{**

**return** e1**->**getEventTime**()** **>** e2**->**getEventTime**();**

**}**

**};**

protected**:**

std**::**priority\_queue**<**Event**\*,** std**::**vector**<**Event**\*>,** CustomComparator**>** m\_events**;**

int m\_currentTime**{**0**};**

**};**

Zu den geforderten Funktionalitäten der Simulations- & Event-Klasse:

**addEvent:**

einfaches Push eines Event-Pointers, wobei natürlich eine Funktion für den Vergleich der Events bereitgestellt werden musste.

**Event kann neues Event anlegen und pushen:**

Implementiert ist das dann erst in den konkreten, abgeleiteten Event-Klassen. Dort muss einfach ein neues Event mit "new" angelegt werden. Über den Zeiger, der die Besitzersimulation des Events referenziert, kann dann für das neue Event "addEvent" aufgerufen werden.

Auszug aus dem abgeleiteten Event "consumeEvent":

consumeEvent**\*** delayedEvent **=** **new** consumeEvent**(**m\_time **+** mySim**->**getWaitingTime**(),** m\_ownerSimulation**,** remainingProducts**);**

m\_ownerSimulation**->**addEvent**(**delayedEvent**);**

**step:**

Mit top hole ich das aktuellste Event und rufe für dafür "executeEvent" auf. Durch Polymorphismus wird die richtige executeEvent-Methode für das jeweilige Event aufgerufen. Mit pop entferne ich das Event aus der Queue (bei pop wird meines Wissens nach auch der Destruktor für das Objekt aufgerufen).

**run:**

In einer while-Schleife werden solange Events abgearbeitet, bis die Endbedingung der Simulation eintrifft. Alternativ kann meine Simulation enden, wenn n Durchgängen zu keiner Lösung gekommen sind oder wenn alle Events bereits abgearbeitet wurden.

* 1. **Quellcode**

// file: utils.h

#pragma once

#include <random>

//taken from Stack Overflow

inline int randomNumWithin**(**int min**,** int max**)** **{**

std**::**random\_device rd**;** // obtain a random number from hardware

std**::**mt19937 gen**(**rd**());** // seed the generator

std**::**uniform\_int\_distribution**<>** distr**(**min**,** max**);** // define the range

**return** distr**(**gen**);** // generate numbers

**}**

//file: Simulation.h

#pragma once

#include <queue>

#include <vector>

#include <functional>

#include "Event.h"

class Event**;**

class Simulation

**{**

struct CustomComparator **{**

bool **operator()(**Event**\*** e1**,** Event**\*** e2**)** **{**

**return** e1**->**getEventTime**()** **>** e2**->**getEventTime**();**

**}**

**};**

protected**:**

std**::**priority\_queue**<**Event**\*,** std**::**vector**<**Event**\*>,** CustomComparator**>** m\_events**;**

int m\_currentTime**{**0**};**

int m\_maxIterations**;**

public**:**

Simulation**(**int maxIterations**=**1000**);**

Simulation**(**const Simulation**&** mySim**)** **=** **delete;**

Simulation**&** **operator=** **(**const Simulation**&** mySim**)** **=** **delete;**

virtual **~**Simulation**();**

int getCurrentTime**()** **{return** m\_currentTime**;}**

int setCurrenttime**(**int t**)** **{**m\_currentTime **=** t**;}**

virtual void runSimulation**();**

virtual bool hasEnded**()** **=** 0**;**

void step**();**

void addEvent **(**Event**\*** e**);**

virtual void printSimulation**();**

//for debugging / testing

void outputAndDestroyQueue**();**

**};**

//file Simulation.cpp

#pragma once

#include "Simulation.h"

Simulation**::**Simulation **(**int maxIterations**)** **:** m\_currentTime**{},** m\_events**{},** m\_maxIterations**(**maxIterations**)** **{}**

Simulation**::~**Simulation **()**

**{**

**while(!**m\_events**.**empty **())** **{**

m\_events**.**pop**();**

**}**

**}**

void Simulation**::**step **()**

**{**

**if(!**m\_events**.**empty **())** **{**

Event**\*** nextEvent **=** m\_events**.**top**();**

m\_events**.**pop**();**

m\_currentTime **=** nextEvent**->**getEventTime**();**

nextEvent**->**executeEvent**();**

**}**

**}**

void Simulation**::**runSimulation **()**

**{**

int i **=** 0**;**

**while(!**m\_events**.**empty**()** **&&** **!**hasEnded**()** **&&** i **<** m\_maxIterations**)** **{**

std**::**cout **<<** "Status of simulation at t=" **<<** m\_currentTime **<<** std**::**endl**;**

printSimulation**();**

Event**\*** currEvent **=** m\_events**.**top**();**

m\_events**.**pop**();**

m\_currentTime **=** currEvent**->**getEventTime**();**

currEvent**->**executeEvent**();**

i**++;**

**}**

std**::**cout **<<** "Result of simulation:" **<<** std**::**endl**;**

std**::**cout **<<** " - Reason for end of simulation: "**;**

**if(**m\_events**.**empty **())** **{**

std**::**cout **<<** "no more events in queue" **<<** std**::**endl**;**

**}**

**else** **if(**i **>=** m\_maxIterations**)** **{**

std**::**cout **<<** "no result in less than " **<<** m\_maxIterations **<<** " steps" **<<** std**::**endl**;**

**}**

**else** **{**

std**::**cout **<<** "end condition fulfilled" **<<** std**::**endl**;**

**}**

std**::**cout **<<** " - Simulation steps: " **<<** i **<<** std**::**endl**;**

**}**

void Simulation**::**addEvent **(**Event**\*** e**)**

**{**

m\_events**.**push**(**e**);**

**}**

void Simulation**::**printSimulation **()**

**{**

std**::**cout **<<** "Current time: " **<<** m\_currentTime **<<** std**::**endl**;**

std**::**cout **<<** "Events queued: " **<<** m\_events**.**size**()** **<<** std**::**endl**;**

std**::**cout **<<** "Next event: "**;**

**if(!**m\_events**.**empty**())** **{**

Event**\*** nextEvent **=** m\_events**.**top**();**

nextEvent**->**printEvent**();**

**}**

**else** **{**

std**::**cout **<<** "no more events queued" **<<** std**::**endl**;**

**}**

**}**

void Simulation**::**outputAndDestroyQueue **()**

**{**

**while(!**m\_events**.**empty **())** **{**

Event**\*** nextEvent **=** m\_events**.**top**();**

nextEvent**->**printEvent**();**

m\_events**.**pop**();**

**}**

**}**

//file: Event.h

#pragma once

#include <iostream>

#include "utils.h"

class Simulation**;**

class Event

**{**

protected**:**

int m\_time**;**

Simulation**\*** m\_ownerSimulation**;**

public**:**

explicit Event**(**int t**,** Simulation**\*** ownerSimulation**)** **:** m\_time**(**t**),** m\_ownerSimulation**(**ownerSimulation**)** **{};**

virtual void executeEvent**()** **=** 0**;**

int getEventTime**()** **{return** m\_time**;}**

virtual void printEvent**();**

**};**

//file: Event.cpp

#pragma once

#include "Event.h"

void Event**::**printEvent **()** **{**

std**::**cout **<<** "event at t= " **<<** m\_time**;**

**}**

* 1. **Tests:**

Nachdem Simulation und Event beides abstrakte Klassen sind, sieht man die richtige Funktionsweise der Queue-Verwaltung etc. erst in den Testfällen der abgeleiteten Simulation (Käufer / Produzent). Somit habe ich mir erlaubt, die Testfälle im 2. Punkt zusammenzufassen.

1. **Testszenario (Teilaufgabe b)**
   1. **Lösungsidee:**

In meiner Simulation gibt es zwei abgeleitete Events. Ein Event für das Produzieren von Waren und eines für den Verbrauch.

Von der Klasse Simulation wurde eine Klasse "MarketSimulation" abgeleitet in der ich zusätzlich Komponenten speichere:

* Puffer-Kapazität,
* Wartezeit, wenn Puffer leer / voll ist
* Endbedingung für Simulation (bestimmte Anzahl an konsumierten Stück)
* Bereits konsumierte Menge

Die Tabelle unterhalb soll die Zustände meiner Simulation skizzieren. Im Beispiel habe ich festgelegt, dass die **Puffer-Kapazität 50 Stück** beträgt und ein **Produzent/Konsument 10 Zeiteinheiten warten muss** wenn dieser voll bzw. leer ist (also ein neues Event in t+10 registriert wird).

**Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung**

Erklärung:

* t0: die Queue wurde mit 3 zufällig generierten Events initialisiert. Der Puffer beträgt zu dem Zeitpunkt natürlich noch 0. Als nächstes (erstes) Event sollen zum Zeitpunkt 5, 20 Stück produziert werden.
* t5: Zum Zeitpunkt 5 liegen nun also 20 Stück auf Lager. Das nächste anstehende Event ist eine Konsumation zum Zeitpunkt 10 mit 30 Stück.
* t10: Im Event hätten 30 Stück konsumiert werden sollen, es lagen aber nur 20 auf Lager. Ich habe es so implementiert, dass alle verfügbaren Stück genommen werden können. Somit hat der Puffer eine Größe von 0. Für die restlichen 10 Stück wurde aber ein neues Event in t+10 (also t=20) registriert (siehe rotes, neues Event).

Die MarketSimulation verwaltet also eine Priority\_Queue, die entweder Zeiger auf Consumer- oder Produce-Events enthält. Durch die Abarbeitung der jeweiligen Events wird deren Aktion ausgeführt, welche einen Einfluss auf die Parameter der "besitzenden" Simulation haben. Sobald Konsumenten eine bestimmte Anzahl an Produkten konsumiert haben ist die Simulation beendet.

* 1. **Quellcode:**

//file: MarketSimulation.h

#pragma once

#include "Simulation.h"

#include "Event.h"

#include "produceEvent.h"

#include "consumeEvent.h"

#include <iostream>

class MarketSimulation**:** public Simulation

**{**

private**:**

int m\_bufferCapacity**;**

int m\_curBufferSize**{**0**};**

int m\_consumedProducts**{**0**};**

int m\_endSimulationSum**{**100**};**

int m\_waitingTime**;** //time consumer/producer has to wait when buffer is empty/full

public**:**

MarketSimulation**(**int bufferCapacity**,** int waitingTime**,** int endSimulationSum**=**100**,** int maxIterations**=**1000**);**

void randomInitSimulation**(**int nEvents**,** int maxEventTime**=**500**,** int maxEventNrPieces**=**50**);**

int getBufferCapacity**()** **{return** m\_bufferCapacity**;}**

int getCurBufferSize**()** **{return** m\_curBufferSize**;}**

int getConsumedProducts**()** **{return** m\_consumedProducts**;}**

int getWaitingTime**()** **{return** m\_waitingTime**;}**

void setBufferSize**(**int v**)** **{**m\_curBufferSize **=** v**;}**

void setConsumedProducts**(**int v**)** **{**m\_consumedProducts **=** v**;}**

virtual bool hasEnded**()** override**;**

virtual void printSimulation**()** override**;**

**};**

//file MarketSimulation.cpp

#pragma once

#include "MarketSimulation.h"

MarketSimulation**::**MarketSimulation**(**int bufferCapacity**,** int waitingTime**,** int endSimulationSum**,** int maxIterations**)** **:**

Simulation**(**maxIterations**),** m\_bufferCapacity**(**bufferCapacity**),**

m\_waitingTime**(**waitingTime**),** m\_endSimulationSum**(**endSimulationSum**)** **{}**

void MarketSimulation**::**randomInitSimulation **(**int nEvents**,** int maxEventTime**,** int maxEventNrPieces**)**

**{**

std**::**cout **<<** "~~~~Simulation randomly initialised with the following events~~~~" **<<** std**::**endl**;**

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

int randEvent **=** randomNumWithin**(**1**,**2**);**

int randTime **=** randomNumWithin**(**1**,** maxEventTime**);**

int randAmount **=** randomNumWithin**(**1**,** maxEventNrPieces**);**

**if(**randEvent **==** 1**)** **{**

addEvent**(new** produceEvent**(**randTime**,** **this,** randAmount**));**

std**::**cout **<<** "Production event at time: " **<<** randTime **<<** " | amount: " **<<** randAmount **<<** std**::**endl**;**

**}**

**else** **{**

addEvent**(new** consumeEvent**(**randTime**,** **this,** randAmount**));**

std**::**cout **<<** "Consumption event at time: " **<<** randTime **<<** " | amount: " **<<** randAmount **<<** std**::**endl**;**

**}**

**}**

**}**

bool MarketSimulation**::**hasEnded **()** **{**

**return** m\_consumedProducts **>=** m\_endSimulationSum**;**

**}**

void MarketSimulation**::**printSimulation **()**

**{**

std**::**cout **<<** "Current time: " **<<** m\_currentTime **<<** std**::**endl**;**

std**::**cout **<<** "Current buffer size: " **<<** m\_curBufferSize **<<** std**::**endl**;**

std**::**cout **<<** "Already consumed products: " **<<** m\_consumedProducts **<<** std**::**endl**;**

std**::**cout **<<** "Events queued: " **<<** m\_events**.**size**()** **<<** std**::**endl**;**

std**::**cout **<<** "Next event: "**;**

**if(!**m\_events**.**empty**())** **{**

Event**\*** nextEvent **=** m\_events**.**top**();**

nextEvent**->**printEvent**();**

**}**

**else** **{**

std**::**cout **<<** "no more events queued" **<<** std**::**endl**;**

**}**

**}**

//file: produceEvent.h

#pragma once

#include "Event.h"

#include "MarketSimulation.h"

class produceEvent **:** public Event

**{**

private**:**

int m\_numberProducts**;**

public**:**

produceEvent**(**int time**,** Simulation**\*** ownerSimulation**,** int numberProducts**);**

void executeEvent**()** override**;**

void printEvent**()** override**;**

**};**

//file: produceEvent.cpp

#include "produceEvent.h"

produceEvent**::**produceEvent**(**int time**,** Simulation**\*** ownerSimulation**,** int numberProducts**)** **:**

Event**(**time**,** ownerSimulation**),** m\_numberProducts**(**numberProducts**)** **{}**

void produceEvent**::**executeEvent **()**

**{**

MarketSimulation**\*** mySim **=** **dynamic\_cast<**MarketSimulation**\*>(**m\_ownerSimulation**);**

std**::**cout**<<** std**::**endl **<<** "## Processing next event ##" **<<** std**::**endl**;**

**if((**mySim**->**getCurBufferSize**()** **+** m\_numberProducts**)** **>** mySim**->**getBufferCapacity **())** **{**

//make buffer full

int remainingProducts **=** m\_numberProducts **-** **(**mySim**->**getBufferCapacity**()** **-** mySim**->**getCurBufferSize**());**

std**::**cout **<<** "Buffer too small. Only " **<<** mySim**->**getBufferCapacity**()-**mySim**->**getCurBufferSize**()**

**<<** " pieces will be added to buffer" **<<** std**::**endl**;**

mySim**->**setBufferSize**(**mySim**->**getBufferCapacity**());**

std**::**cout **<<** "For remaining " **<<** remainingProducts **<<** " pieces new production event in "

**<<** m\_time**+**mySim**->**getWaitingTime**()** **<<** " is queued" **<<** std**::**endl **<<** std**::**endl**;**

//new event for remaining production size

produceEvent**\*** delayedEvent **=** **new** produceEvent**(**m\_time **+** mySim**->**getWaitingTime**(),** m\_ownerSimulation**,** remainingProducts**);**

m\_ownerSimulation**->**addEvent**(**delayedEvent**);**

**}**

**else** **{**

std**::**cout **<<** "Everything ok" **<<** std**::**endl **<<** std**::**endl**;**

mySim**->**setBufferSize**(**m\_numberProducts**+**mySim**->**getCurBufferSize**());**

**}**

**}**

void produceEvent**::**printEvent **()**

**{**

std**::**cout **<<** "Production event at t=" **<<** m\_time **<<** " | producing: " **<<** m\_numberProducts **<<** " pieces" **<<** std**::**endl**;**

**}**

//file consumeEvent.h

#pragma once

#include "Event.h"

#include "MarketSimulation.h"

class consumeEvent**:** public Event

**{**

private**:**

int m\_numberProducts**;**

public**:**

consumeEvent**(**int time**,** Simulation**\*** ownerSimulation**,** int numberProducts**);**

void executeEvent**()** override**;**

void printEvent**()** override**;**

**};**

//file: consumeEvent.cpp

#include "consumeEvent.h"

consumeEvent**::**consumeEvent **(**int time**,** Simulation**\*** ownerSimulation**,** int numberProducts**)** **:**

Event**(**time**,** ownerSimulation**),** m\_numberProducts**(**numberProducts**)** **{}**

void consumeEvent**::**executeEvent **()**

**{**

MarketSimulation**\*** mySim **=** **dynamic\_cast<**MarketSimulation**\*>(**m\_ownerSimulation**);**

std**::**cout **<<** std**::**endl **<<** "## Processing next event ##" **<<** std**::**endl**;**

**if(**mySim**->**getCurBufferSize **()** **-** m\_numberProducts **>** 0**)** **{** //enough products in buffer to consume

std**::**cout **<<** "Everything ok" **<<** std**::**endl **<<** std**::**endl**;**

mySim**->**setBufferSize**(**mySim**->**getCurBufferSize**()** **-** m\_numberProducts**);**

mySim**->**setConsumedProducts**(**mySim**->**getConsumedProducts**()+**m\_numberProducts**);**

**}**

**else** **{** //wait, register consumption event for later again

//let consumer consum remaining products from buffer

mySim**->**setConsumedProducts**(**mySim**->**getConsumedProducts**()+**mySim**->**getCurBufferSize**());**

//create new event for remaining prodcuts from this event

int remainingProducts **=** m\_numberProducts **-** mySim**->**getCurBufferSize**();**

std**::**cout **<<** "Buffer too small. Only " **<<** mySim**->**getCurBufferSize**()** **<<** " pieces can be consumed" **<<** std**::**endl**;**

mySim**->**setBufferSize**(**0**);**

std**::**cout **<<** "For remaining " **<<** remainingProducts **<<** " pieces new consumption event in "

**<<** mySim**->**getWaitingTime**()** **+** mySim**->**getCurrentTime**()** **<<** " is queued" **<<** std**::**endl **<<** std**::**endl**;**

consumeEvent**\*** delayedEvent **=** **new** consumeEvent**(**m\_time **+** mySim**->**getWaitingTime**(),** m\_ownerSimulation**,** remainingProducts**);**

m\_ownerSimulation**->**addEvent**(**delayedEvent**);**

**}**

**}**

void consumeEvent**::**printEvent **()**

**{**

std**::**cout **<<** "Consume event at t=" **<<** m\_time **<<** " | consuming: " **<<** m\_numberProducts **<<** " pieces" **<<** std**::**endl**;**

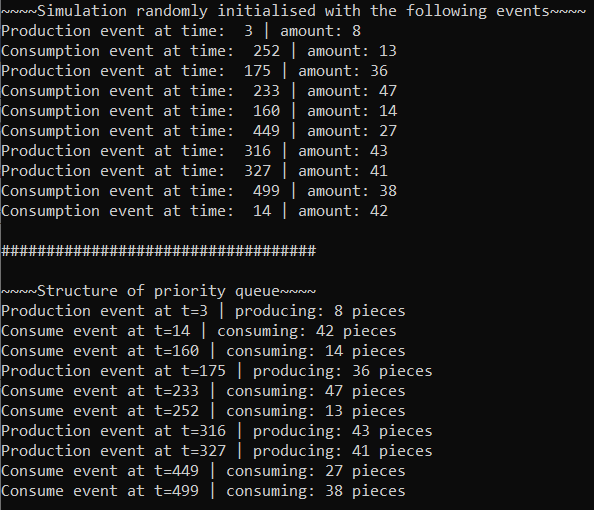
**}**

* 1. **Testfälle:**

**Initialisierung und push-Funktion:**

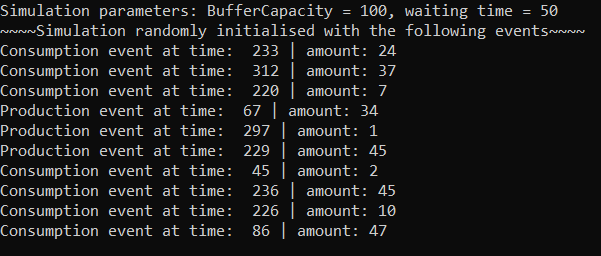
Soweit ich ergoogelt habe gibt es für eine priority\_queue keinen Iterator. Somit habe ich die Queue mit top und pop ausgegeben wobei die Queue jedoch zerstört wird.

Der obere Teil der Simulation zeigt die Events, die ich zufällig erstellt und gepusht habe. Der untere Teil zeigt, in welcher Reihenfolge die Events dann abgearbeitet wurden. Wie man sieht sind die Events aufsteigend nach der Zeit des Eintretens sortiert.

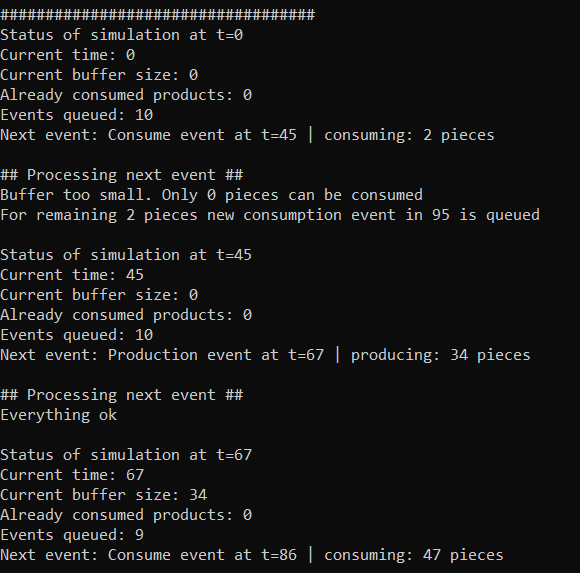


**Step-Methode**

1. Initialisierung der Simulation mit 10 Events. Hinweis, das ist nicht die Reihenfolge, in der die Events in der Queue gespeichert sind (ohne Iterator hätte ich die Queue bei der Ausgabe zerstört)



1. In einer for loop rufe ich step 10x auf (Ausschnitt zeigt nur 3 Aufrufe)



**RunSimulation-Methode**

|  |  |
| --- | --- |
|  | **Kommentar** |
|  |  |
|  | Zu t=0 ist der Puffer noch leer. Es sind 10 Events in der Queue. Das erste anstehende Event ist eine Konsumation.  Nachdem der Puffer leer ist, wird das Event für t+50 (also 27+50 = 77) noch einmal eingeplant  Nach dem ersten Event ist der Puffer noch immer auf 0, weil ja nur versucht wurde zu konsumieren. Das nächste Event steht für t=28 an. Aber leider wieder Konsumation.  Puffer ist noch immer leer aber nun steht endlich eine Produktion an.  Das Event wird normal verarbeitet |
|  | Nachdem im letzten Event 31 Stück produziert wurden, liegen diese nun auf Lager.  Hier taucht nun das allererste Event wieder auf, das für t+50 neu eingeplant wurde. Dieses mal sind genug Stück auf Lager. Das Event wird erfolgreich ausgeführt. |
|  | Bisher bereits 23 Stück konsumiert. Im nächsten Event sollen wieder 31 folgen. Der Puffer ist groß genug.  Nachdem nun die Abbruchbedingung von 50 konsumierten Stück erreich ist wird die Simulation beendet. |

1. **Testprogramm (main)**

//file: main.cpp

#include <set>

#include <iterator>

#include <iostream>

#include "Simulation.h"

#include "produceEvent.h"

#include "consumeEvent.h"

**using** **namespace** std**;**

void test\_simulation\_creation **()** **{**

MarketSimulation**\*** mySimulation **=** **new** MarketSimulation**(**70**,**50**,**100**);** //parameter: bufferCap., waitingTime, endEcondition

mySimulation**->**randomInitSimulation**(**10**);**

cout **<<** endl**<<** "###################################" **<<** endl**;**

//output of queue

cout **<<** endl **<<** "~~~~Structure of priority queue~~~~" **<<** endl**;**

mySimulation**->**outputAndDestroyQueue**();**

**delete** mySimulation**;**

**}**

void testStep **()** **{**

MarketSimulation**\*** mySimulation **=** **new** MarketSimulation**(**100**,**50**,**100**);** //parameter: bufferCapacity, waitingTime, end condition

cout **<<** "Simulation parameters: BufferCapacity = 100, waiting time = 50" **<<** endl**;**

mySimulation**->**randomInitSimulation**(**10**);**

cout **<<** endl**<<** "###################################" **<<** endl**;**

**for(**int i **=** 0**;** i **<** 10**;** i**++)** **{**

cout **<<** "Status of simulation at t=" **<<** mySimulation**->**getCurrentTime**()** **<<** endl**;**

mySimulation**->**printSimulation**();** cout **<<** endl**;**

mySimulation**->**step**();**

**}**

**delete** mySimulation**;**

**}**

void testSimulation **()** **{**

int bufferCapacity **=** 100**;** //max 100 pieces can be in the buffer

int waitingTime **=** 50**;** //if an event cannot be completed it is rescheduled for t+50

int endCondition **=** 50**;** //once consumers got 50 pieces simulation is ended

cout **<<** "Simulation parameters:" **<<** endl**;**

cout **<<** " - Buffer capacity = " **<<** bufferCapacity **<<** endl**;**

cout **<<** " - Waiting time = " **<<** waitingTime **<<** endl**;**

cout **<<** " - End condition = " **<<** endCondition **<<** endl **<<** endl**;**

MarketSimulation**\*** mySimulation **=** **new** MarketSimulation**(**bufferCapacity**,** waitingTime**,** endCondition**);**

mySimulation**->**randomInitSimulation**(**10**);**

cout **<<** endl**<<** "###################################" **<<** endl**;**

mySimulation**->**runSimulation**();**

**delete** mySimulation**;**

**}**

int main **()** **{**

//test\_simulation\_creation();

//testStep();

testSimulation**();**

**return** 0**;**

**}**