

## Overview

Coursework2 is actually the continuation of Coursework1. You may use the same Visual Studio project or make its copy.

In this coursework the *DataProvider* DLL creates a bunch of objects from class *Item*. The minimal number of items in a bunch is one and the maximal number is four. Number of items in a specific bunch is random.

The stream of bunches is not continuous: there are pauses between bunches. The longest pause is 10 seconds. The duration of a specific pause is random.

The Coursework2 multithreaded application must receive the bunches and store the corresponding to them entries into data structure implemented in Coursework1 (i.e. applying method *Data::InsertEntry()*).

Application Coursework2 is controlled by commands typed by the user.

## Initial data

The *DataProvider* DLL has two public functions (see also file *DataProducer.h*):

```
void Initialize();           // Reads initial data from text files and prepares
                             // the random number generators. In case of failure
                             // throws exception. Used also in Coursework1.

void Start(char ver, void *pCntr);
                             // Starts the generator of objects. In case of
                             // failure throws exception. Here the value of
                             // first parameter must be 'E' and the value of
                             // second parameter must pointer to an object
                             // from class Control_E.

class Control_E
{
public:
    stop_token stop;
    promise<queue<Item*>>*> pPr = nullptr;
    condition_variable_any cva;
    ~Control_E()
    {
        if (pPr)
        {
            delete pPr;
        }
    }
};
```

In the loop implemented in DLL (i.e. in the producer thread) the items are generated in the following way:

```
mutex mx;
queue<Item*>*> pBuf = nullptr;
while (!pCntr->stop.stop_requested())
{
    ..... // sleeping
    if (pCntr->stop.stop_requested())
```

```

{
    pCntr->pPr->set_value(nullptr);
    break;
}
if (pBuf)
{
    delete pBuf;
}
pBuf = new queue<Item *>;
..... // create a bunch and push into queue pBuf
cout << n << " items generated" << endl;
pCntr->pPr->set_value(pBuf);
unique_lock<mutex> lock(mx);
pCntr->cva.wait(lock, pCntr->stop,
               [&]() { return pBuf->empty(); });
}

```

Application Coursework2 in its consumer thread uses the same object from class *Control\_E*.

## General requirements

The Coursework2 application is controlled by the following commands:

- Start
- Stop
- Print
- Store
- Exit

The user types the commands into command prompt window. When the application is ready to accept a new command, it must print an appropriate message. Defective messages must be ignored.

The interaction with keyboard should be in a separate thread.

All the commands except *exit* must be repeatable.

## Start and stop

Command *start* launches the object generator as producer thread in DLL (see *Initial data*). It must also launch the consumer thread in application.

When the object generator is running, the keyboard must be blocked. The only allowed keystroke is *Esc* which stops the producer and consumer.

The generator may not react immediately, the waiting time before the first bunch may be up to 10 seconds.

## Print

Command *print* calls method *Data::PrintAll()*. If there are no entries, an appropriate message must appear.

## Store

Command *store* is to write all the entries in data structure into a disk file. To start the implementation, add into class *Entry* method

```
friend fstream& operator<<(fstream&, const Entry&);
```

and into class *Data* method

```
void StoreAll(fstream &);
```

Command *store* launches the following dialog:

- The application asks the user to type the complete path to directory into which the data will be stored.
- If this directory does not exist, the application asks permission to create it.
- If the directory exists, the application prints the complete list of files in it.
- Then the application asks the user to type the name of text file into which the data will be stored.
- If this file exists, the application asks whether to append the new data or to destroy the current contents.

If there are no entries, an appropriate message must appear.

If the user types incorrect answers, refuses to create a new directory or file handling operations fail, the application breaks off the execution of this command.

## Exit

Command *exit* clears everything and quits the application.

## Test cases

1. Launch application.
2. Command *print*.
3. Command *store*.
4. Command *start*, let it to create some bunches, then *Esc*.
5. Repeat step 4.
6. Command *print*.
7. Repeat step 4.
8. Repeat step 6.
9. Command *store*, instead of directory name type some nonsense.
10. Command *store*, specify a non-existing directory and do not allow to create a new.
11. Command *store*, specify a non-existing directory and allow to create a new. Specify a file to store. Check the results.
12. Command *store*, specify an existing directory. Specify an existing file to store with appending. Check the results.
13. A wrong command.
14. Exit the application.

## Evaluation

The student's work is accepted if the evaluation test runs correctly and produces all the supposed results.

The deadline is the end of the semester. However, it is strongly advised to present the results of coursework earlier. The students can do it after each lecture.

Presenting the final release is not necessary. It is OK to demonstrate the work of application in debug mode of the Visual Studio environment.

To get the assessment the students must attend personally. Electronically (e-mail, GitHub, etc.) sent courseworks are neither accepted nor reviewed. The students may be asked to explain their code or even write a small modification right on the spot.