Mechatronic Engineering

Object Oriented Programing and Software Engineering
Laboratory instruction 1
C++ introduction

AGH Krakw, 2020

Materials created for educational purposes.
Dedicated for students attending Software Engineering course.
Author would apreaciate any feedback regarding errors of any kind found in the instruction script.
Please report those to the following email address: danielt@agh.edu.pl

Contents

1 First steps into C++ programming **4**
1.1 Creating *.cpp files and compiling 4
1.2 Source code basics 4

2 Variables **6**

3 Control instructions **7**
3.1 Conditional instruction if. 7
3.2 While loop. 8
3.3 Do while loop. 9
3.4 For loop. 9
3.5 Switch instruction. 10

4 Functions **11**

1 First steps into C++ programming

1.1 Creating *.cpp files and compiling

In order to write the first program in C ++ we will use the unix console and the Midnight Commander text editor. To start Midnight Commander Editor, enter the following in the console:

```
mcedit -c nazwapliku.cpp
```

A * .cpp file will be created on the hdd (in current directory), which is the source file of the program written in C ++.

In order to compile the source code into an executable file, we will use the g ++ compiler.

```
g++ nazwa_pliku.cpp o nazwa_pliku
```

To run the program, enter (being in the program directory):

```
./nazwa_pliku
```

1.2 Source code basics

To create a program in C++ it is necessary to use a library loading directive **#include** (similar as in C language).

#include <name> - is used to inform the preprocessor that the included library file is located in the standard folder with header files of the compiler. **#include "name"** is used to inform the preprocessor that the included library file is located in the same folder as the file containing the #include directive.

In C++ the standard input and output library is called **iostream**.

The next important part of the C++ source file is the **main()** function. All the major instructions for the program should be stored in it. The format of main function declaration goes as follows:

Example:

```
#include <iostream>

int main() {
   std::cout << "Hello world! \n";
   std::cout <<"Again" << "Hello"
         << "world!" << "\n";
return 0;
}
```

The above example is a very simple, working program written in C ++. Its function is to display a string on the screen.

In the above example one could see some new instructions. In C, for the purpose of displaying text on the screen **printf()** was used. In C++ the function responsible for that is **std::cout**.

```
std::cout << "Hello World! \n";
```

In C, **scanf()** was used to input the necessary data into any variables. C++ uses **std::cin**.

```
std::cin >> name_of_the_variable;
```

Example:

```
#include <iostream>

int main(){
   int age;
        std::cout << "Enter your age: ";
         std::cin >> age;
         std::cout << "\n";
         std::cout << "You are " << age << " years old" <<
             std::endl;
return 0;
}
```

2 Variables

The data in C++ is stored (in a similar way as in case of C language) in variables. Variables can bedeclared as in C, by providing the type and the variable name. .

In c++ the variable types can be divided in following categories:

- fundamental types;
- derived types;
- built-in types;
- user defined types.

Fundamental types:

Types used for real numbers:
short int; int; long int; long long int;

Types used for alphanumerical characters:
char;

Types used for floating point numbers:
float; double; long double;

In C++ it is possible to declare and define variables on the go.

Example:

```
#include <iostream>
int main() {
    std::cout << "Obliczmy pole kwadratu\n"
      << "W tym celu bedziemy potrzebowac dlugosci boku a\n"
      << "niestety program nie posiada jeszcze zadnej zmiennej.\n"
      << "Podaj dlugosc boku kwadratu: "; int dlugosc;
    std::cin >> dlugosc;
    std::cout << "\n";
    int p;
    p = dlugosc * dlugosc;
    std::cout << "Pole kwadratu wynosi: " << p << "\n";
return 0;
}
```

3 Control instructions

3.1 Conditional instruction if.

As the name suggests, the if instruction is used to check if a provided condition is full field. Two forms of it exist:

```
if(expression) instruction1;
```

and

```
if(expression) instruction1;
else         instruction2;
```

In the first case, the value of the expression is checked. If the value is not zero, the instruction 1 is executed. If the value is zero or negative, then the instruction1 is omitted.

In the second case, the value of the expression is also checked. If the value is not zero, the instruction 1 is executed. If the value is zero or negative, program runs instruction2.

For the purpose of running more than one instructions, an instruction block is necessary. An instruction block is a set of instructions between { and } brackets.:

```
if(expression) {
    instruction1;
    instruction2;
}
```

It is possible to create a multi variant if instruction:

```
if(expression) instruction1;
else if(expression2) instruction2;
else if(expression3) instruction3;
```

Example:

```
#include <iostream>

int main(){
    int a=1, b=2, c=3;
```

```
    if (a) std::cout << "Zmienna a jest rozna od 0 a jej wartosc"
       <<" to: " << a << "\n";
    if (a > b)std::cout << "to sie nie wyswietli\n";
    else std::cout << a << " jest mniejsze od " << b << "\n";
    if (c < b) std::cout << "to sie nie wyswietli\n";
    else if (b = 2) {
         b++;
         std::cout<< "b = " << b << "\n";
    }
return 0;
}
```

3.2 While loop.

The way while loop works is very simple. The expression value is checked at the beginning of the loop. If its equal zero or less, nothing happens (instruction1 is not executed). If the value is greater than zero, the instructions in the loops instruction block are executed. Then the value of the expression is checked again. The loop runs while the expression has a positive value. The expression is always checked at the beginning of the loop cycle.

```
while (expression) instruction1;
```

Example:

```
#include <iostream>
using namespace std;

int main () {
    int wiek=0, lata=0;
    cout << "Ile masz lat? \n";
    cin >> wiek;
    cout << endl;
    while (wiek < 24) {
       wiek++;
       lata++;
    }

    if (wiek < 25) {
       cout << "Teoretycznie za " << lata;
       if (lata == 1) cout << " rok";
       else if (lata > 1 && lata < 5) cout << " lata";
```

```
        else if (lata >= 5 || lata==0) cout << " lat";
        cout << " skonczysz studia. \n";
    }
    else cout << "taki stary, a jeszcze studiuje... \n";
return 0;
}
```

3.3 Do while loop.

Do while loop works very similar to while loop. The main difference is the time when the expression value is checked. In the do while loop, the expression is checked at the end of the loop cycle. Because of that, a do while loop always executes at least once.

```
do instruction1 while(expression);
```

Example:

```
#include <iostream>
using namespace std;

int main() {
    int i=1;
    do {
        cout << "Wartosc i to: " << i << endl;
        i++;
    } while(i<=4 && i>=2);
return 0;
}
```

3.4 For loop.

```
for (initialization; expression; incrementation) instruction1;
```

Example:

```
for (i = 0; i < 10; i++)
{
cout << i;
}
```

Initialization – this instruction is executed before the first cycle of the loop.
Expression – the value of expression is calculated before every run of the loop. If the value is true, the loop executes.
Incrementation – incrementation that is executed at the end of every loop cycle.

Number of instructions provided in the initialization and incrementation may be greater than one. They have to be separated with the use of a colon. The incrementation, expression and initialization instructions can be left blank, however the semicolons that divide those, have to be provided. If the expression is left blank it is treated as always being true.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int i=0, ile=0;
    cout << "Wprowadz liczbe podsystemow: ";
    cin >> ile;
    cout << endl;
    cout << "Rozpoczynam skanowanie podsystemow" << endl;
    for (i = 1; i <= ile; i++) {
        cout << "Podsystem " << i << " przeskanowany" << endl;
    }
    cout << "Operacja zakonczona pomyslnie \n";
return 0;
}
```

3.5 Switch instruction.

The switch instruction is used for multi variant operations.

```
switch (wyrazenie)
{
     case war1:
     instr1; break;
     case war2:
     instr2; break;
     default:
     instr3; break;
```

```
}
```

When the expression value is obtained and it corresponds to the value of one of the provided cases, then the instructions in a specific case are executed until the brake instruction is found. If the value doesnt correspond to any case, then the default case is executed.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int i = 3;
    switch(i) {
         case 1: cout << "Wartosc jest rowna 1 \n"; break;
         case 2: cout << "Wartosc jest rowna 2 \n"; break;
         case 3: cout << "Wartosc jest rowna 3 \n"; break;
         case 4: cout << "wartosc jest rowna 4 \n"; break;
         default: cout << "wartosc jest inna niz 1,2,3,4 \n";
    }
return 0;
}
```

4 Functions

Functions in C++ may be called a subprograms. Function is used to define a set of user defined instructions, that can return values set by users to the main program. To invoke a function it is necessary to input function type, name and arguments used in it.

Example:

```
#include <iostream>
using namespace std;

float polekwadratu (float bok1, float bok2);

int main() {
    float a, b, pole;
    cout << "Podaj dlugosc pierwszego boku: ";
```

```
    cin >> a;
    cout << endl << "Podaj dlugosc drugiego boku: ";
    cin >> b;
    cout << endl;
    cout << "Twoj prostokat posiada boki o dlugosci: a = " << a;
    cout << ", b = " << b;
    cout << endl;
    pole = polekwadratu(a,b);
    cout << "Pole Twojego prostokata wynosi: " << pole << endl;
return 0;
}

float polekwadratu (float bok1, float bok2) {
    float wynik;
    wynik = bok1 * bok2;
    cout << "Obliczam pole...\n\n";
return wynik;
}
```

Usually at the function end a return instruction is used. It is used to return a value from function to the place in the main code where it was executed. In the example program, the function calculating the area of a rectangle was linked to a variable field. It means, that the variable value will be equal to the variable returned by the function. The variable and function that are linked together must have the same data type. Void function is an exception to this rule. Void functions cant return any data.

Stack
Stack is a kind of cache memory, that stores variables defined in the function.

Function arguments
Arguments sent to the function (actual arguments) are only copies of used variables or constants. They are saved on the stock. Any operations performed on the arguments doesnt affect the original variables. There is a way of sending the variables directly to the function. This method is called sending arguments by references.

Example:

```
#include <iostream>
using namespace std;
```

```
void zer (int wart, int &ref);
void nzer (int wart, int ref);

int main () {
    int a = 44, b = 77;
    cout << "Przed wywolaniem funkcji nzer:\n";
    cout << "a = " << a << ", b = " << b << endl;
    nzer (a,b);
    cout << "Po powrocie z funkcji nzer:\n";
    cout << "a = " << a << ", b = " << b << endl;
    cout << "Przed wywolaniem funkcji zer:\n";
    cout << "a = " << a << ", b = " << b << endl;
    zer (a,b);
    cout << "Po powrocie z funkcji zer:\n";
    cout << "a = " << a << ", b = " << b << endl;
return 0;
}
void zer (int wart, int &ref) {
    cout << "\tW funkcji zer przed zerowaniem: \n";
    cout << "\twart = " << wart << ", ref = "
    << ref << endl;
    wart = 0;
    ref = 0;
    cout << "\tW funkcji zer po zerowaniu: \n";
    cout << "\twart = " << wart << ", ref = "
    << ref << endl;
}
void nzer (int wart, int ref) {
    cout << "\tW funkcji nzer przed zerowaniem: \n";
    cout << "\twart = " << wart << ", ref = "
    << ref << endl;
    wart = 0;
    ref = 0;
    cout << "\tW funkcji nzer po zerowaniu: \n";
    cout << "\twart = " << wart << ", ref = "
    << ref << endl;
}
```

Task

In base of provided topics in the following instruction, please create a simple user interface for a vending machine:

Program requirements:
1. Software displays the list of provided products (at least five) with the amount of the stock. The user should be able to pick a product.
2. After the product has been picked, the user should be able to input quantity of the purchase. If the quantity exceeds the stock amount, the user is asked to input a proper value into the program.
3. After the transaction is finished, program returns to displaying the list of products with updated stock value.
4. Program must be equipped with a secret code unknown to user. After inputing the code, the program should stop running.
5. For the purpose of this program at least two functions must be used (additional to main() function).

Mechatronic Engineering

Object Oriented Programing and Software Engineering
Laboratory instruction 2
C++ introduction

AGH Krakw, 2020

Materials created for educational purposes.
Dedicated for students attending Software Engineering course.
Author would apreaciate any feedback regarding errors of any kind found in the instruction script.
Please report those to the following email address: danielt@agh.edu.pl

Contents

1 Arrays. 4
- 1.1 Elements of a single dimentional array. 4
- 1.2 Sending array into function. 5
- 1.3 Multidimensional arrays. 6
- 1.4 Sending multidimensional arrays into functions. 7

2 Strings. 8

3 File support 11
- 3.1 Fstream variable type. 11
- 3.2 Reading data from a file. 12
  - 3.2.1 Read data by stream. 12
  - 3.2.2 Read data with lines. 13
  - 3.2.3 Read data with blocks. 13
- 3.3 Save data to file 13
  - 3.3.1 Save data using stream. 13
  - 3.3.2 Save using data blocks 13

1 Arrays.

An array is a collection of objects of the same type that are stored continuously in memory. We define the arrays in a very simple way, as in the case of a variable we have to give it a type and a name, the main difference is that for the array it is necessary to provide a number of cells in it:

```
float tab[5];
```

In this way, the defined array places space for five float objects in memory. The size of this variable defined must be a constant declared before compilation.

Array can be created from:
- fundametal types (except void);
- enumeration types;
- pointers;
- other arrays;
- classes.

1.1 Elements of a single dimentional array.

The above defined array has five elements. It can be written as:

```
tab[0], tab[1], tab[2], tab[3], tab[4].
```

The numbering of elements in an array always starts with 0. The array can be filled with a simple assignment operation. Then, as with variables, you must assign a specific value to the particular element of the array. Here are two ways to assign values.

Example:

```
#include <iostream>
using namespace std;

main() {
    float marks[5], lp[5];
    float meanval=0;
    int i;
    marks[0] =3;
    marks[1] =5;
```

```
    marks[2] = 4;
    marks[3] =3;
    marks[4] =2;
    for (i = 0; i<5; i++) meanval += marks[i];
    meanval /=5;
    for (i = 0; i<5; i++) lp[i] = i + 1;
    cout << "List of marks: " << endl;
    for (i =0; i<5; i++) {
      cout << "\t Mark nr. "<< lp[i] << ". ";
      cout << marks[i] << endl;
    }
    cout << "Mean value of the marks: " << meanval << endl;

return 0;
}
```

As with variables, the objects in the array can be placed by initialization:

```
int tab[4] = {2, 13, 69, 2};
```

With this method, you can quickly fill an array in one line when defining it.

1.2 Sending array into function.

The array is sent to the function giving only the start address of the array. **The name of the array is also the address of its zero element**.

```
void function(double tab[ ]); <--- function definition
double array[4];            <--- array definition
function (array);            <--- array usage in function.
```

Example:

```
#include <iostream>
using namespace std;

float earings (float pln, int hours[ ]);

main() {
    int week[7];
    int i;
    float money, earing;
    cout << "\tWeekly earings calculator.\n"
```

```
    << "How much you earn in an hour: ";
    cin >> money;
    cout << endl;

    for (i=0; i < 7; i++) {
        cout << "How many hours you plan to work on";
        if (i == 0) cout << " Monday
        if (i == 1) cout << " Tuesday: ";
        if (i == 2) cout << " Wednesday: ";
        if (i == 3) cout << " Thursday: ";
        if (i == 4) cout << " Friday: ";
        if (i == 5) cout << " Saturday: ";
        if (i == 6) cout << " Sunday: ";
        cin >> week[i];
        cout << endl;
    }
    cout << endl;
    earing = earings(money, week);
    cout << "You are going to earn: " << earing
    << " PLN\n";

return 0;
}

float earings (float pln, int hours[ ]) {
    int lhr=0, a;
    for (a=0; a<7; a++)  lhr += hours[a];
return (lhr*pln);
}
```

1.3 Multidimensional arrays.

The multidimensional array is defined in this way:

```
int tab[3][2];
```

The compiler sees it as an array with the number of rows (the first value in brackets) and the columns (second value in brackets):

[0][0]	[0][1]
[1][0]	[1][1]
[2][0]	[2][1]

Example:

```
#include <iostream>
using namespace std;

main() {
    int board [10][10];
    int i, j;
    for (i=0; i<10; i++)
    for (j=0; j<10;j++)
    board[i][j] = 0;
    board[3][6] = 1; board[8][3] = 1; board[2][5] = 1;
    for (i=0; i<10; i++)
    for(j=0; j<10; j++)
    if (board[i][j]) cout << "Ships are on following fields: ["
    << i << "]" << "[" << j << "]" << endl;
return 0;
}
```

1.4 Sending multidimensional arrays into functions.

The array is sent to the function giving only the start address of the array. It is very similar, if you are sending a multidimensional array. For a function to correctly search for certain element in a multidimensional array, you must specify the number of columns in that array.

```
void function(double tab[ ][3546]); <--- function definition
double array[4][3546];              <--- array definition
function (array);                   <--- array usage in function.
```

Example:

```
#include <iostream>
using namespace std;

float earnings (float pln, int hours[ ][4]);

main() {
    int week[7][4];
```

```
    int i,j;
    float money, earning;
    cout << "\tMonthly earings calculator.\n"
     << "How much you earn in an hour: ";
    cin >> money;
    cout << endl;
    for (j=0; j<4; j++) {
         cout << "Week " << j+1 << endl;
         for (i=0; i < 7; i++) {
             cout << "How many hours you plan to work on";
             if (i == 0) cout << " Monday: ";
             if (i == 1) cout << " Tuesday: ";
             if (i == 2) cout << " Wednesday: ";
             if (i == 3) cout << " Thursday: ";
             if (i == 4) cout << " Friday: ";
             if (i == 5) cout << " Saturday: ";
             if (i == 6) cout << " Sunday: ";
             cin >> week[i][j];
             cout << endl;
         }
    }
    cout << endl;
    earning = earnings(money, week);
    cout << "You will earn: " << earning << " PLN" << " in four
       weeks\n";

return 0;
}

float earnings (float pln, int hours[ ][4]) {
    int lhr=0, a, b;
    for (b=0; b<4; b++) {
        for (a=0; a<7; a++)  lhr += hours[a][b];
    }
return (lhr*pln);
}
```

---

2 Strings.

Strings are used to store a group of char objects. In C ++, you can use them in two ways. The first is to create a char array:

```
char tekst[20];
```

In this case, it must be remembered that the array must end with a sign \0 (null) therefore, the size of the array should be one cell larger than the length of the text we want to fit in it. This method is also quite troublesome and tiring. Therefore, the second method will be discussed in greater detail.

In C ++ a special class has been created: **std::string** that simplifies work with strings. In fact, the resulting data type is still an array of characters, but working with it is simplified by the functions contained in the string header file. Use the appropriate directive to use it:

```
#include <string>
```

To work with strings, create a string object:

```
string name_of_the_variable;
```

Example:

```
#include <iostream>
#include <string>
using namespace std;

main() {
    string text;
    cout << "Enter a string " << endl;
    cin >> text;
    cout << "\nThis is your string:\t " << text
    << endl;
return 0;
}
```

In the above example, we do not need to use a loop to populate a text string variable. Unfortunately, if more than two words (separated by spaces) were included in the above program, only the first would be written to the variable. This is because the cin statement does not interpret white signs. The following examples show how to deal with this problem.
Example:

```
#include <iostream>
#include <string>
using namespace std;
```

```
main() {
    string text, text1, text2, text3;
    char space = ' ';
    cout << "Enter a string " << endl;
    cin >> text1 >> text2 >> text3;
    cout << "\nThis is your string:\t "
    << text1 <<" " << text2 << " " << text3 << endl;

    //Connecting strings:
    text = text1+text2+text3;

    cout << "\nThis is your string::\t " << text
    << endl;

    //adding spaces
    text = text1+" "+text2+" "+text3;

    cout << "\nThis is your string:\t " << text
    << endl;
    text = text1+space+text2+space+text3;
    cout << "\nThis is your string:\t " << text
    << endl;

    //to clear the content of a string
    text2.clear();

    cout << "\nThis is your string after clearing one element"
    <<":\t " << text1 <<" " << text2 << " "
    << text3 << endl;
return 0;
}
```

To simplify the space addition one can use **getline()** instruction.

```
getline(cin, name_of_the_variable);
```

Example:

```
#include <iostream>
#include <string>
using namespace std;

main() {
```

```
    string text;
    cout << "Write a sentence with spaces between words: " << endl;
    getline(cin, text);
    cout << "\nThis is your string:\t " << text
    << endl;
return 0;
}
```

3 File support

You will need a library to work with external files: **fstream**. Udostpnia ona programicie narzdzia do zapisu i odczytu plikw.

```
#include <fstream>
```

3.1 Fstream variable type.

Before you start working on the file it is necessary to create a variable that will allow us to perform operations on the selected file:

```
std::fstream file;
```

or with use of std name space:

```
fstream file;
```

The created variable does not point to any file. To assign a specific file to it, use the **open()** function from class **std::fstream**:

```
file.open( "file_name.txt", file_open_mode);
```

When assigning a file to a variable, specify one or more file opening modes:
ios::app — Sets the internal pointer to the file at its end. File open in write-only mode. Data can only be saved at the end of the file.
ios::ate — Sets the internal pointer to the end of the file at the time the file opens.
ios::binary — Information for the compiler for data to be treated as a binary data stream.
ios::in — Allows reading data from a file.
ios::out — Allows writing data to a file.

ios::trunc — File content is cleared when opening.

```
fstream file;
file.open( "name_of_file.txt", ios::in | ios::out );
```

When the operation with the file ends, close the file. For this purpose, the function **close()** will be used:

```
file.close();
```

Example:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream file;
    file.open( "name_of_file.txt", ios::in | ios::out );
    if( file.good() == true ) {
       cout << "File access granted!" << endl;

        file.close();
    }
    else cout << "File access forbidden " << endl;

    return( 0 );
}
```

3.2 Reading data from a file.

There are several methods for reading data from a file.

3.2.1 Read data by stream.

Data reading via stream is analogous to **std::cin¿¿** operation. Data read in this way is always treated as text. Unfortunately, this feature will not allow you to read white-space information (as in Chapter 2).

```
name_of_file_variable >> variable_that_stores_data;
file >> data;
```

3.2.2 Read data with lines.

We will use the function known from Chapter 2 to read the data: **getline()**.

```
getline( file, data);
```

Data loaded with the getline () function are always treated as text.

3.2.3 Read data with blocks.

Function **read()** will be used to read the data blocks. Data loaded by it can be read as text or as binary data(by enabling **ios::bin**).
In the following example, we can read 1024 characters from the file into the temp buffer.

```
char temp[ 1024 ];
file.read( temp, 1024 );
```

3.3 Save data to file

Saving data to a file is as simple as reading it up from a file. However, keep in mind that the write process works specifically. It is possible to append data if we are at the end of a file, or overwrite data when we are in a different location than EOF (end of file).

3.3.1 Save data using stream.

This way of writing works analogically to **std::cout¡¡**, but the text is sent to the file rather than to the peripheral devices.

```
name_of_file_variable << variable_that_stores_data;
file << data;
```

3.3.2 Save using data blocks

Function **write()** will be used to write data to blocks. In the following example, the variable text (string type) acts as a buffer(&text[0] is a pointer to the first element of the variable) whereas text.length () tells the compiler to copy the amount of data equal to the length of the text buffer.

```
getline( cin, text );
```

```
file.write( & text[ 0 ], text.length() );
```

Example:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream file;
    file.open( "file.txt", ios::in | ios::out );
    if( file.good() == true ) {
       cout << "File access granted!" << endl;


        file << "This text will show in the file" << endl;
        string inscr, text;
        text="this is a string, that will be placed in the file\n";
        file << text;
        cout<<"Enter any string, it will be copied into "
        <<" the file"<<endl;
        getline(cin, inscr);
        file.write(&inscr[0],inscr.length());

        //reading from file

        //moving the cursor by 3 bytes towards the beginning of the
           file
        file.seekg( + 3, std::ios_base::beg);

        string inscr1, inscr2;
        file >> inscr1;
        cout << "First line:" << endl << inscr1 << endl;

        getline(file, inscr2);
        cout << "Next line from the file: " << endl << inscr2 <<
           endl;

        char temp [10];
        file.read(temp, 10);
        cout << "another line from file: " << endl;
        for(int i=0; i<10; i++) cout <<temp[i];
        cout << endl;
```

```
        file.close();
    }
    else cout << "File access forbidden " << endl;

    return( 0 );
}
```

For the above example, create a text file: file.txt in the directory with * .cpp file.

Task

Based on the informations provided in this manual, please improve the simple utility interface of the snack and beverage vending machine created on previous labratories.

Program requirements:
1. The program loads a list of "products" along with the number of available items, from an external text or binary file at startup and each time the list is refreshed.
2. Once purchase have been made, the number of products is refreshed and saved to an external file.
3. If any of the products is exhausted, a note about the need to replenish the product is added to the external file
4. At the start of the program (and only at start-up), if an annotation about the need to replenish a product is found, its quantity is increased to 10 units.