Выполнили:

студенты группы 19ВВ2

Гусев Д.О.

Кубасов И.М.

Приняли:

Синев М.П.

Дорошенко И.Н.

Пенза 2021

ОТЧЕТ

по лабораторной работе №8

по курсу «Технологии программирования»

на тему «Сериализация и восстановление»

Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

### Цель работы

Изучение механизмов сохранения и восстановления объектов данных. Получение навыков разработки самовосстанавливающихся структур данных.

### Общие сведения

Сериализация – это преобразование в последовательную форму. Применительно к программированию – это алгоритм сохранения объекта в линейном виде для последующего сохранения (в файле) или передачи по каналам связи.

Восстановление (или десериализация) – это преобразование из последовательной формы в параллельную. Применительно к программированию – это алгоритм восстановления объекта из последовательной формы. Т.е. алгоритм восстановления объекта из файла или данных полученных по каналам связи.

### Лабораторное задание

Лабораторное задание выполняется на базе лабораторной работы №7.

Требования к классам:

В класс **Interface** добавить две чистые виртуальные функции: Serialize и Deserialize.

В классах **Derived1**, **Derived2** и **Derived3** реализовать виртуальные функции Serialize и Deserialize.

* Serialize – функция сериализация данных. Функция должна сохранять все свои данные в буфер, указанный в параметрах функции.
* Deserialize – функция восстановления данных. Функция должна восстанавливать данные объекта из буфера, указанного в параметрах функции.

В класс **<Структура>** добавить два метода:

* Serialize – функция сериализация структуры данных. Функция должна сохранить все элементы структуры в файл. В своей работе функция должна вызывать соответствующую функцию Serialize для всех объектов.
* Deserialize – функция восстановления структуры данных. Функция должна восстанавливать все элементы структуры из файла. Во время работы функция должна вызывать соответствующую функцию Serialize для всех элементов. В процессе восстановления данных все элементы должны заново размещаться в динамической памяти. Т.е. все объекты классов **Derived1**, **Derived2** и **Derived3** должны заново создаваться, после чего размещенные объекты добавляются в структуру.

Требования к программе:

* В основной программе необходимо разместить объект класса **<Структура>** в динамической памяти.
* Выполнить добавление нескольких различных элементов в структуру (**Derived1**, **Derived2** и **Derived3**).
* Вывести на экран содержимое всей структуры.
* Сериализовать всю структуру в файл.
* Удалить объект класса **<Структура>** из динамической памяти. При этом все элементы структуры также должны быть удалены.
* Разместить новый объект класса **<Структура>** в динамической памяти.
* Восстановить сохраненную структуру данных из файла.
* Вывести на экран содержимое всей структуры. Повторная печать должна полностью повторить содержимое первой печати, доказав таким образом что данные восстановлены корректно.

### Листинг

Для экономии месте приведен только код относящийся к лб№8.

Пример сериализации на классе derived1

virtual void ser(char\* buffer) override

{

int i = 0;

char\* tmp\_year = new char[10];

char\* tmp\_TYPE = new char[10];

\_itoa(TYPE,tmp\_TYPE,10);

\_itoa(year\_m,tmp\_year,10);

strcat(buffer,tmp\_TYPE);

strcat(buffer,"&");

strcat(buffer,town\_m);

strcat(buffer, "&");

strcat(buffer,theme\_m);

strcat(buffer, "&");

strcat(buffer,author\_m);

strcat(buffer, "&");

strcat(buffer,tmp\_year);

strcat(buffer, "&");

strcat(buffer,izdat\_m);

strcat(buffer, "\*");

strcat(buffer, "\0");

int buff\_len = strlen(buffer);

ofstream fout;

fout.open(path);

if (!fout.is\_open())

{

cout << "ERR";

}

else

{

fout.write(buffer, buff\_len);

}

fout.close();

delete[] tmp\_year;

delete[] tmp\_TYPE;

}

class stack

{

public:

stack(int size)

{

MAX\_SIZE = size;

FIRST = -1;

LAST = -1;

flag = 0;

Stack = new stack\_element[MAX\_SIZE];

}

~stack()

{

delete[] Stack;

}

bool full()

{

return (FIRST == 0 and LAST == MAX\_SIZE - 1) or (FIRST == LAST + 1);

}

bool empty()

{

return FIRST == -1;

}

void push(Interface\* obj)

{

if (full())

{

cout << "STACK IS FULL" << endl;

}

else

{

if (FIRST == -1)

{

FIRST = 0;

}

LAST = (LAST + 1) % MAX\_SIZE;

Stack[LAST].element = obj;

flag++;

}

}

Interface\* pop()

{

if (empty())

{

cout << "STACK IS EMPTY";

}

else

{

delete Stack[LAST].element;

if (LAST == FIRST)

{

FIRST = -1;

LAST = -1;

}

else

{

LAST = (LAST - 1) % MAX\_SIZE;

flag--;

}

return Stack[LAST].element;

}

}

Interface\* GetNext(int index)

{

return Stack[(index + 1) % MAX\_SIZE].element;

}

Interface\* GetFirst()

{

return Stack[FIRST].element;

}

Interface\* GetLast()

{

return Stack[LAST].element;

}

void output\_stack()

{

int index = FIRST % MAX\_SIZE;

if (empty())

{

cout << "STACK IS EMPTY" << endl;

}

else

{

cout << "Elements in stack" << endl;

GetFirst()->output();

cout << endl;

while (index % MAX\_SIZE != LAST)

{

GetNext(index++)->output();

cout << endl;

}

}

}

void output\_element()

{

int index;

if (empty())

{

cout << "STACK IS EMPTY" << endl;

}

else

{

cout << "ENTER INDEX ";

cin >> index;

if ((index < 0) or (index > flag))

{

index = FIRST;

}

cout << "element with index " << index << endl;

GetNext(--index)->output();

cout << endl;

}

}

void ser\_st()

{

int index = FIRST % MAX\_SIZE;

if (empty())

{

cout << "STACK IS EMPTY" << endl;

}

else

{

GetFirst()->ser(BUFFER);

cout << endl;

while (index % MAX\_SIZE != LAST)

{

GetNext(index++)->ser(BUFFER);

}

}

BUFFER[0] = '\0';

}

void dser\_st()

{

int index = FIRST % MAX\_SIZE;

int id = 0;

if (FIRST >=0)

{

cout << "STACK IS NOT EMPTY" << endl;

}

else

{

char ch\_f = '\0';

int i = 0;

ifstream fin;

fin.open(path);

if (!fin.is\_open())

{

cout << "ERR";

}

else

{

while (fin.get(ch\_f))

{

BUFFER[i] = ch\_f;

i += 1;

}

BUFFER\_LEN = i;

}

fin.close();

Interface\* obj = nullptr;

id = BUFFER[0] - '0';

if (id == 1)

{

derived1 objd1;

objd1.dser(BUFFER, id);

obj = &objd1;

obj->output();

push(&objd1);

}

if (id == 2)

{

derived2 objd2;

objd2.dser(BUFFER, id);

obj = &objd2;

obj->output();

push(&objd2);

}

if (id == 3)

{

derived3 objd3;

objd3.dser(BUFFER, id);

obj = &objd3;

obj->output();

push(&objd3);

}

id = 0;

while(BUFFER[INDEX] != '\0')

{

id = BUFFER[INDEX] - '0';

switch (id)

{

case 1:

{

INDEX++;

derived1 objd1;

objd1.dser(BUFFER, INDEX);

obj = &objd1;

obj->output();

push(&objd1);

break;

}

case 2:

{

INDEX++;

derived2 objd2;

objd2.dser(BUFFER, INDEX);

obj = &objd2;

obj->output();

push(&objd2);

break;

}

case 3:

{

INDEX++;

derived3 objd3;

objd3.dser(BUFFER, INDEX);

obj = &objd3;

obj->output();

push(&objd3);

break;

}

default:

{

break;

}

}

}

}

}

class stack\_element

{

public:

Interface\* element;

};

private:

int MAX\_SIZE;

int FIRST, LAST;

int flag;

stack\_element\* Stack;

};

//-----------------------------------------------------------------

void func\_ser()

{

stack st(2);

derived1 A;

derived2 B;

Interface\* A1 = &A;

Interface\* B1 = &B;

A1->input();

B1->input();

st.push(A1);

st.push(B1);

st.ser\_st();

}

void func\_dser()

{

stack st2(2);

st2.output\_stack();

st2.dser\_st();

cout << "---------------------------" << endl;

cout << " стек после десериализации " << endl;

cout << "---------------------------" << endl;

}

int main()

{

setlocale(LC\_ALL, "Russian");

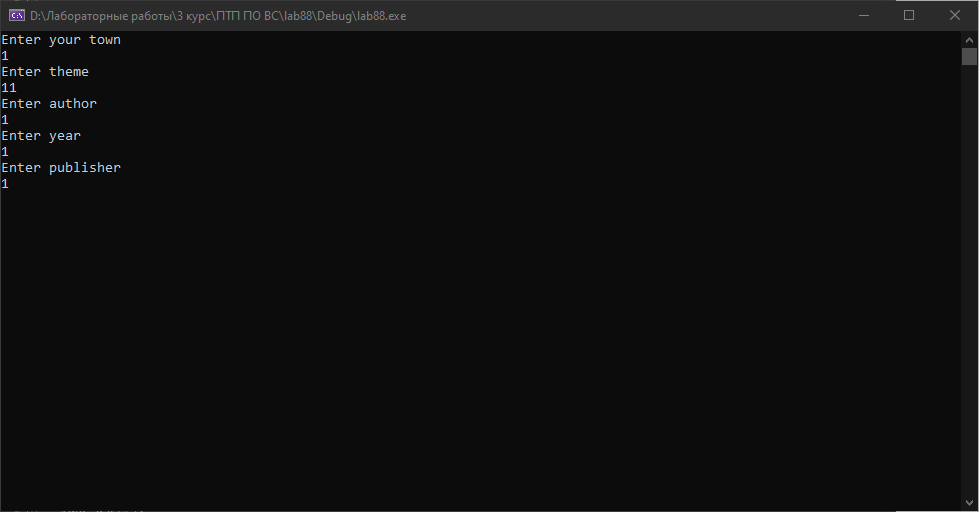
func\_ser();

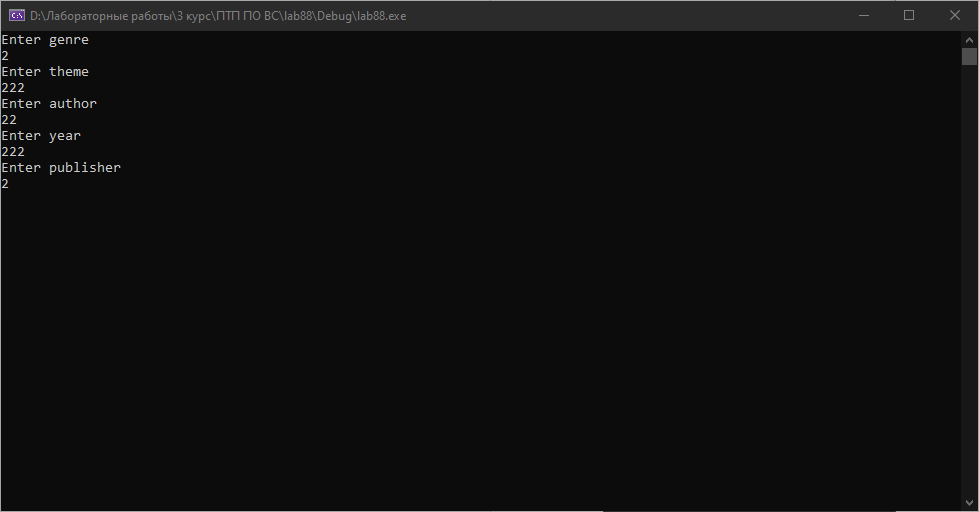
func\_dser();

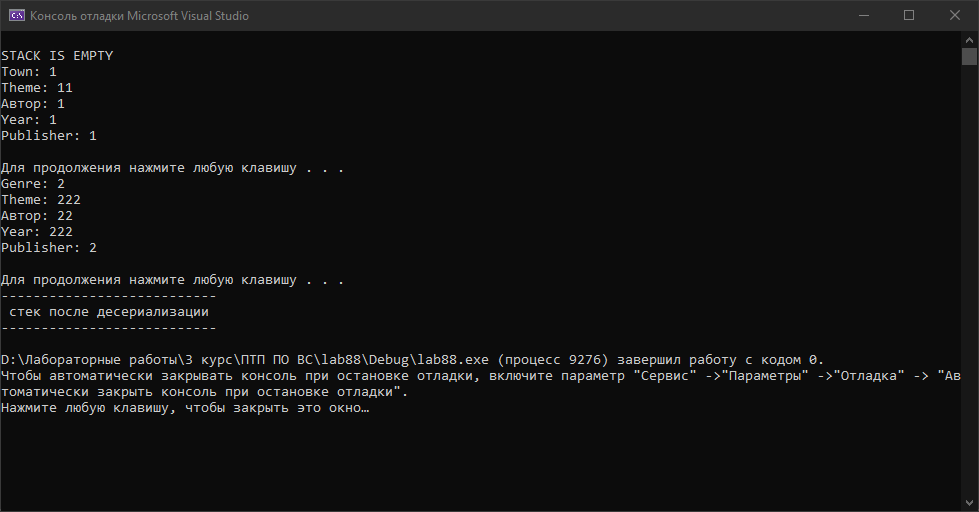
return 0;

}

### Результаты работы программы







### Выводы

Мы изучили механизмы сохранения и восстановления объектов данных. Получение навыков разработки самовосстанавливающихся структур данных.