



Министерство науки и высшего образования
Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение
высшего образования
Национальный исследовательский ядерный университет
«МИФИ»

Институт интеллектуальных кибернетических систем

Кафедра №22 «Кибернетика»

ОТЧЁТ

*К РАБОТЕ ПО ДИСЦИПЛИНЕ
Технология промышленной разработки ПО*

*НА ТЕМУ:
«Лабораторная работа №1»*

Студент М23–524
(Группа)

(Подпись, дата)

Леонов В. В.
(И. О. Фамилия)

Преподаватель

(Подпись, дата)

Гагарин А. П.
(И. О. Фамилия)

Москва, 2024 г.

СОДЕРЖАНИЕ

1	Задание №1	3
2	Задание №2	6
3	Задание №3	10

1 Задание №1

Условие: создать основные типы проектов (для языков C++ и C#, для CLR, то есть .NET и без него). Запустить приложения, заложенные в проектах по умолчанию, убедиться в их работоспособности. Просмотреть, как они выглядят в файловой системе и через окна среды. В частности, посмотреть, как компилируется и связывается приложение, какие параметры устанавливаются и из каких модулей оно komponуется.

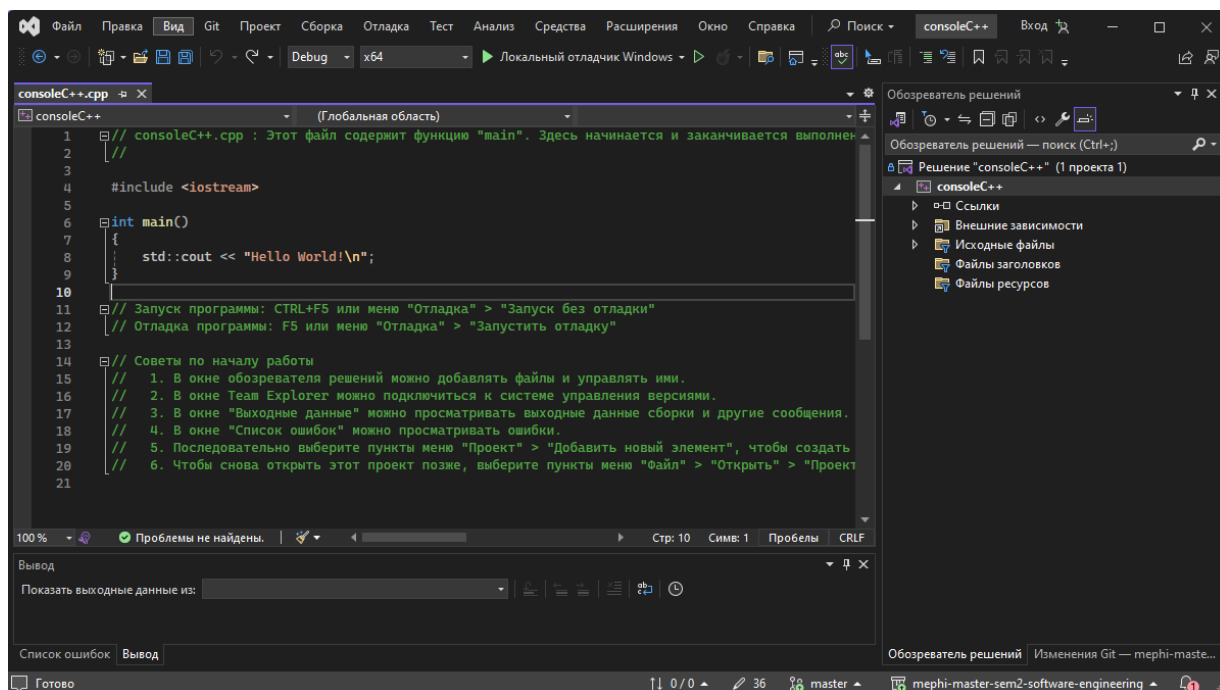


Рисунок 1.1 – Проект C++ без .Net

Для создания проектов на языке C++ без использования .NET была выбрана шаблонная консольная программа. Этот тип проекта создается с минимальными настройками и включает основной файл с расширением .cpp, в котором размещен код основной функции main(). Такой проект компилируется с использованием компилятора cl.exe и связывается с использованием компоновщика link.exe. Вся структура проекта, включая файлы настроек, исходный код и бинарные файлы, отображается в обозревателе решений (Solution Explorer) в Visual Studio. После успешной компиляции и запуска программы вывод в консоль подтверждает ее работоспособность.

Создание проекта на языке C# происходит аналогичным образом.

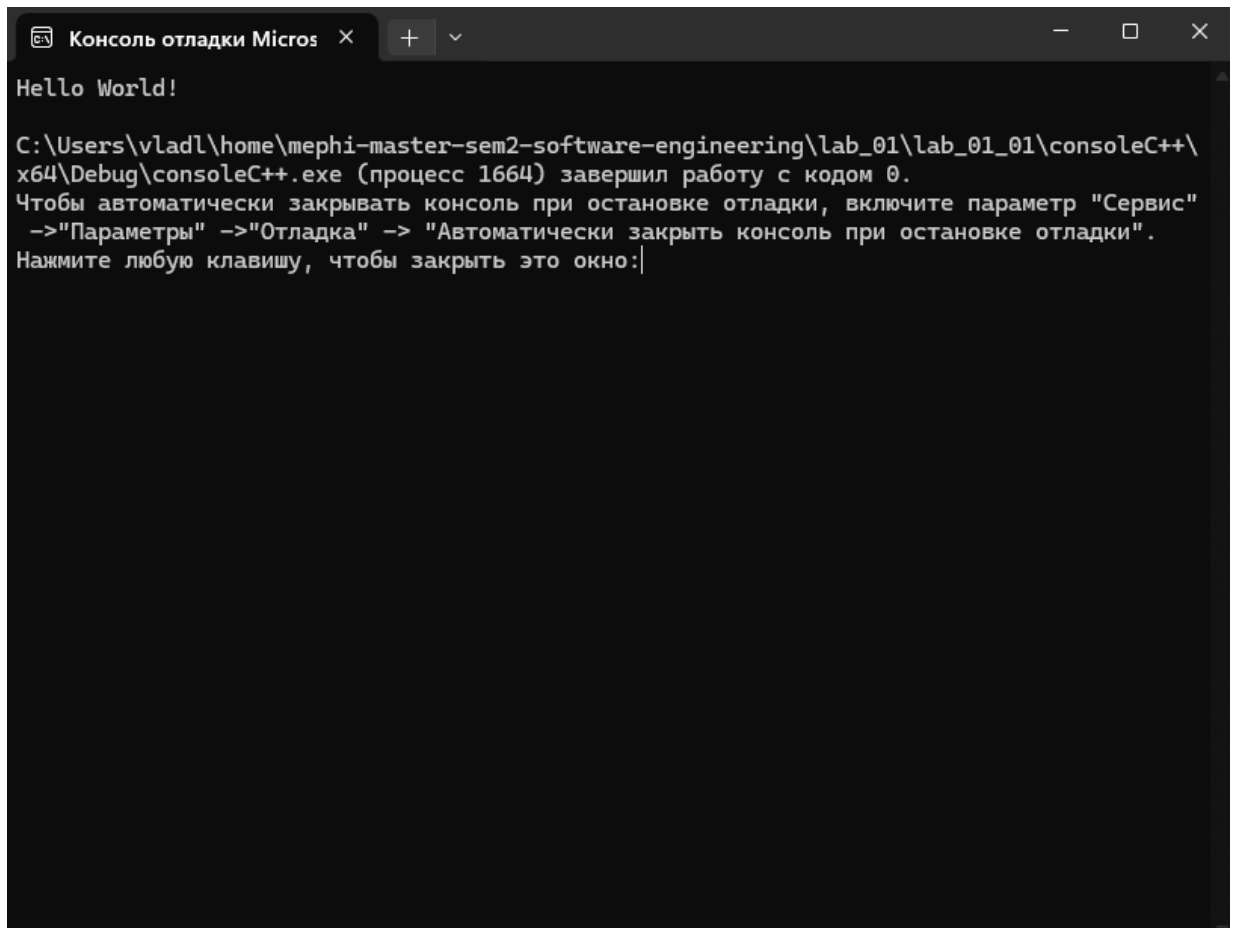


Рисунок 1.2 – Результат запуска проекта C++ без .Net

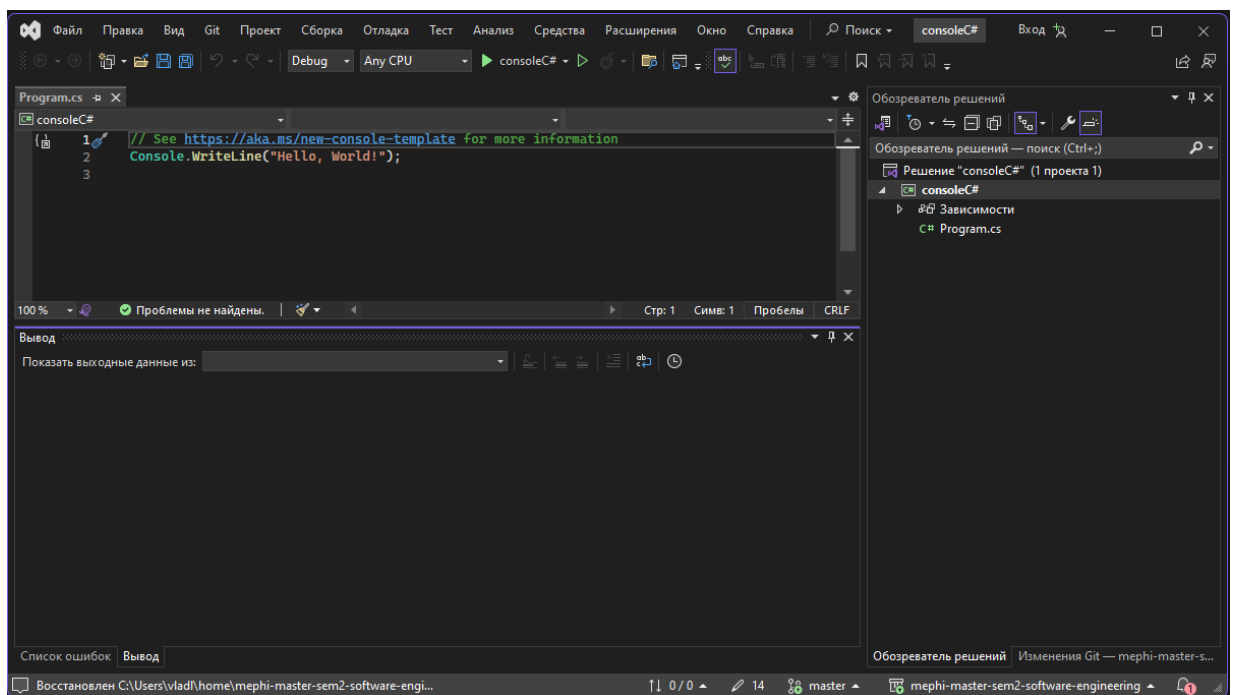


Рисунок 1.3 – Проект C# без .Net

Для создания проектов на языке C# был выбран .NET Framework и .NET Core. Проект на .NET Framework представляет собой традиционное консольное приложение, содержащее файл Program.cs с основной функцией Main(). Этот проект компилируется и выполняется с использованием инструментов, встроенных в .NET Framework, и отображает свою структуру в обозревателе решений. Проект на .NET Core также содержит файл Program.cs с основной функцией Main(), но компилируется и выполняется с использованием инструментов .NET Core, что делает его кроссплатформенным и более гибким в настройке.

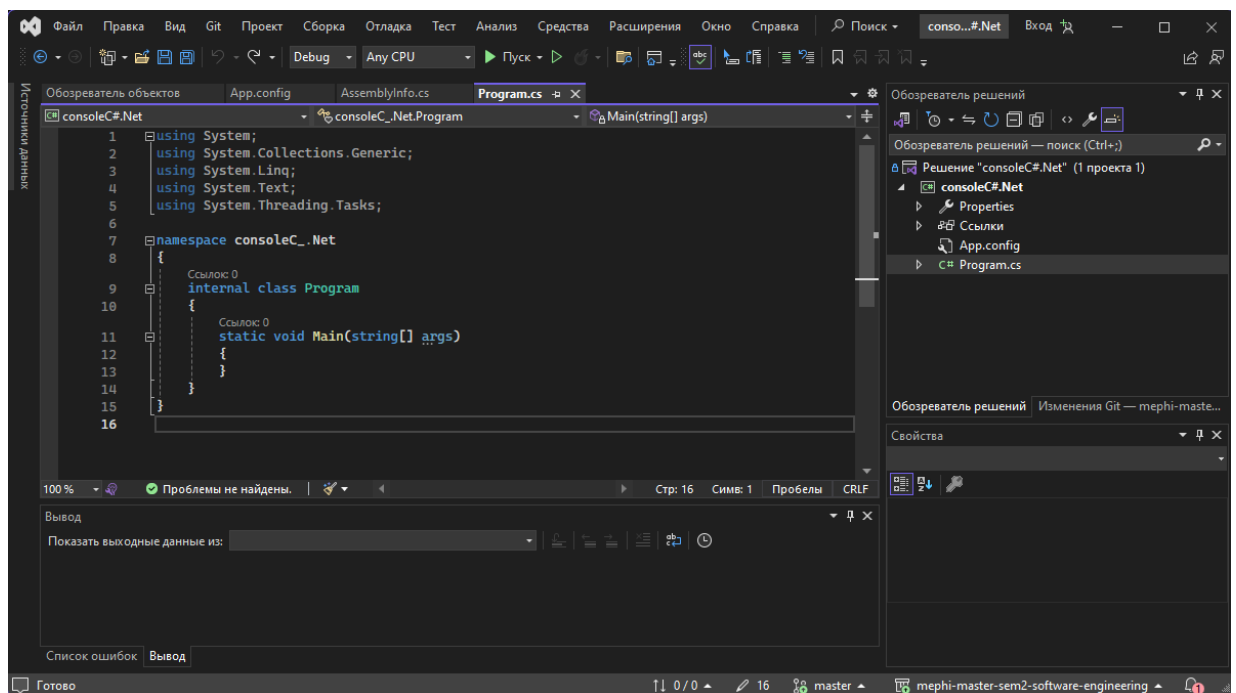


Рисунок 1.4 – Проект C# с .Net

Во всех созданных проектах была проверена их работоспособность путем компиляции и запуска. Компиляция выполнялась с использованием стандартных инструментов Visual Studio. Окно вывода предоставило информацию о процессе компиляции и связывания, включая используемые параметры и команды.

При исследовании параметров проекта через окна среды разработки были выявлены различные настройки компиляции, связывания и целевых платформ. Это позволило понять, из каких модулей компонуется приложение и какие параметры устанавливаются для успешной компиляции и выполнения.

2 Задание №2

Условие:

1. Создать в VS консольный проект по шаблону win32 (на языке C++) и запустить как приложение.
2. Произвести разбор проекта и составить краткий отчёт, включающий в себя описание назначения каждого модуля, модулей; перечень и назначение используемых функций API; основные последовательности вызова функций.
3. Предложить, опробовать и подтвердить скриншотами заметные изменения GUI (текст надписей, количество окон и т.п.).

В качестве консольного проекта предлагается использовать консольное приложение из Задания №1.

Редактор кода занимает центральную часть интерфейса и предназначен для написания и редактирования кода. Он поддерживает подсветку синтаксиса, автодополнение, сворачивание кода, а также множество других функций для облегчения написания и понимания кода.

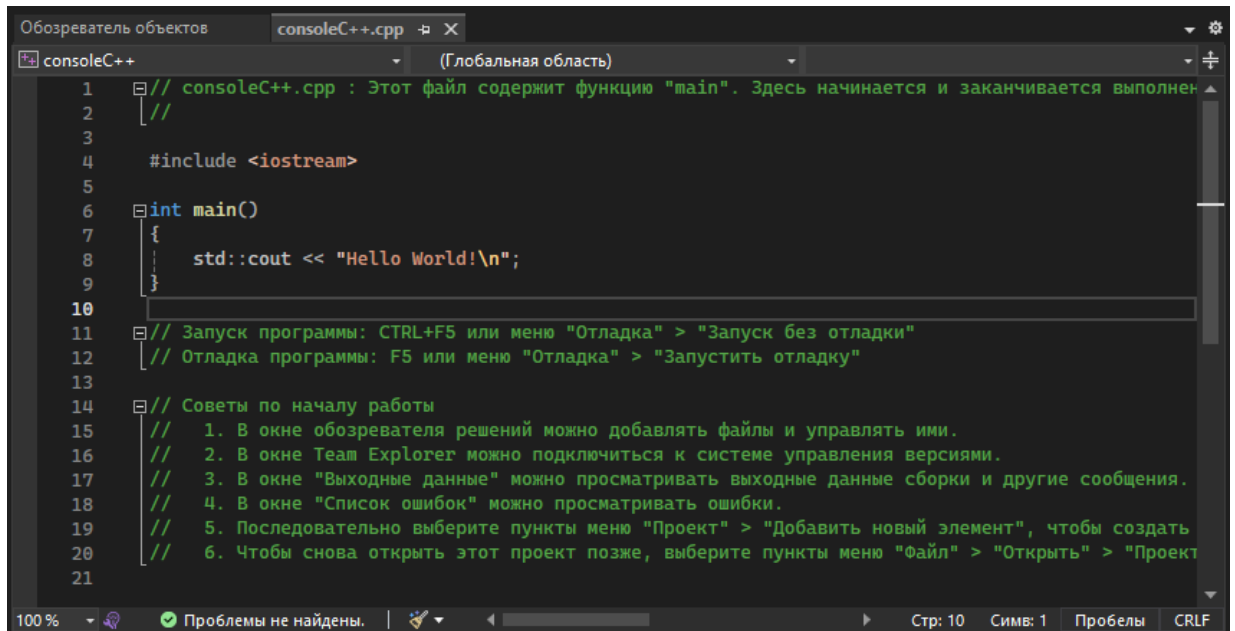


Рисунок 2.1 – Окно редактора кода

Обозреватель решений — это один из ключевых компонентов интерфейса Visual Studio, предоставляющий иерархическое представление всех файлов и

ресурсов, входящих в состав текущего решения. Он играет важную роль в управлении проектами и навигации по ним.

Обозреватель решений позволяет видеть структуру решения на верхнем уровне, где отображается само решение, содержащее один или несколько проектов. Каждый проект, в свою очередь, может содержать файлы исходного кода, ресурсы, конфигурационные файлы и другие артефакты, такие как библиотеки и ссылки на внешние ресурсы.

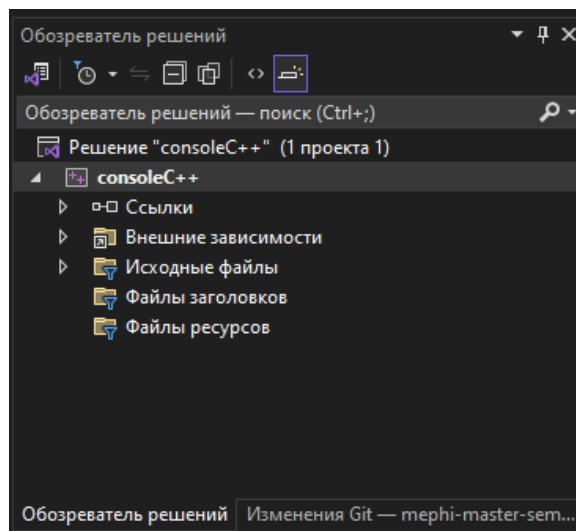


Рисунок 2.2 – Окно обозревателя решений

Панель ошибок находится в нижней части окна и показывает ошибки, предупреждения и информационные сообщения, связанные с кодом, что помогает разработчику быстро идентифицировать и исправлять ошибки в коде.

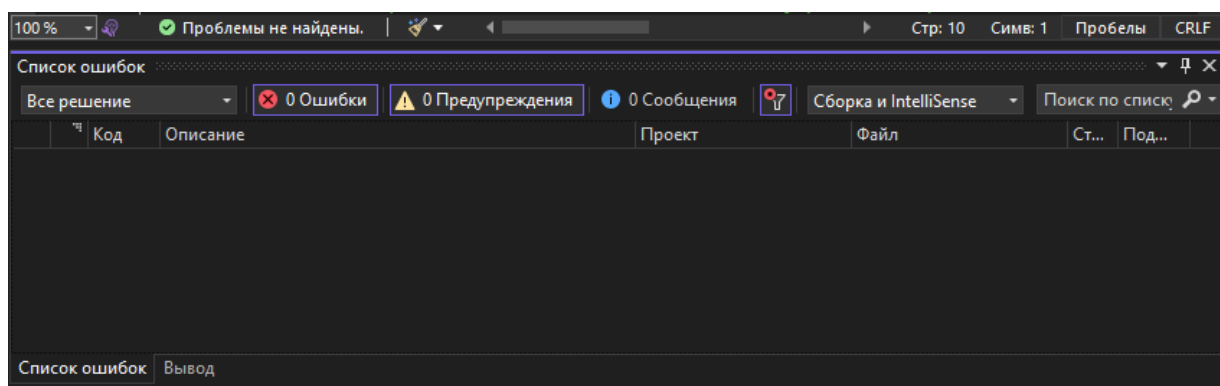


Рисунок 2.3 – Окно панели ошибок

Панель вывода также расположена внизу и отображает сообщения, связанные с процессом сборки, отладки и выполнения приложений. Здесь можно увидеть результаты компиляции, действия системы контроля версий, а также сообщения от различных расширений и инструментов.

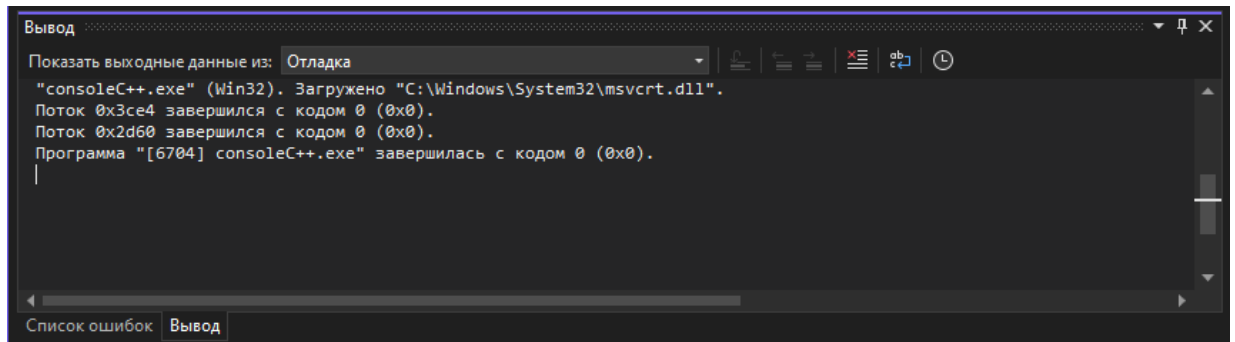


Рисунок 2.4 – Окно панели вывода

Обозреватель классов — это мощный инструмент, предоставляющий разработчикам структурированное иерархическое представление всех типов и их членов в проектах решения. Он предназначен для облегчения навигации по сложным проектам, где может быть большое количество классов, интерфейсов, методов и других элементов.

Обозреватель классов отображает все классы, структуры, интерфейсы, перечисления и делегаты, присутствующие в проекте. Эти элементы организованы в виде дерева, что позволяет легко просматривать их взаимосвязи и структуру. Каждое пространство имен отображается как узел, который можно развернуть, чтобы увидеть содержащиеся в нем типы и члены.

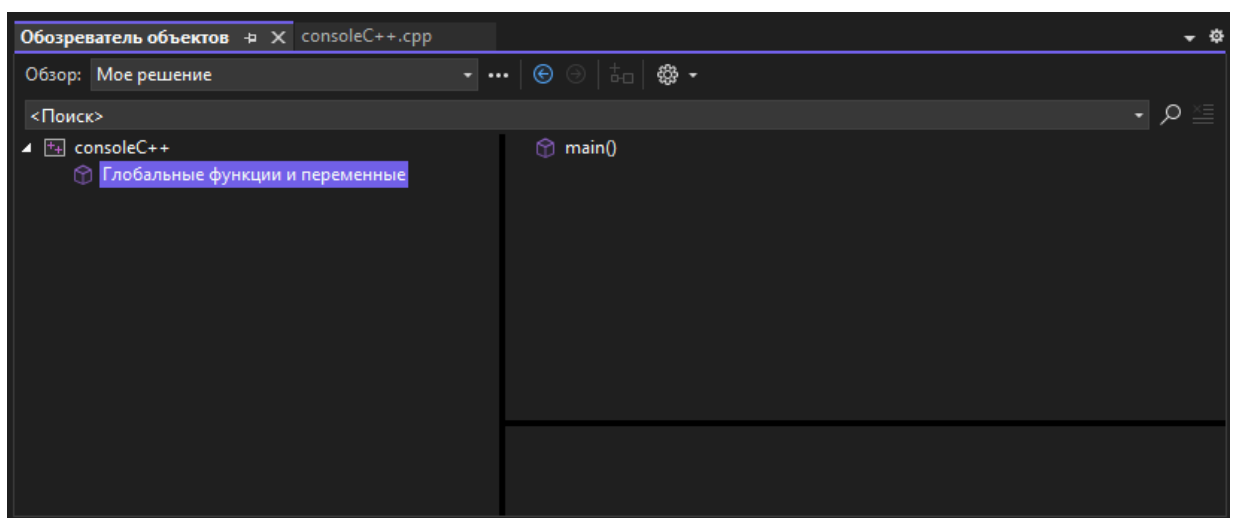


Рисунок 2.5 – Окно обозревателя классов

Для проектов, находящихся под управлением системы контроля версий (например, Git) *инструменты управления версиями* позволяют выполнять операции, связанные с контролем версий, такие как клонирование репозитория, коммит, слияние, разрешение конфликтов и другие.

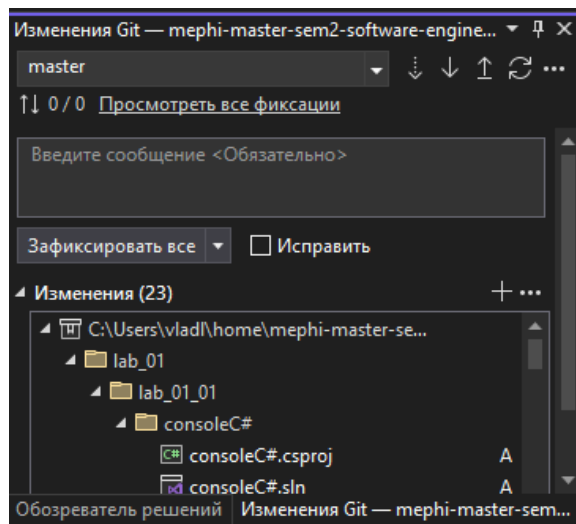


Рисунок 2.6 – Окно инструментов контроля версий

3 Задание №3

Условие:

1. Проверить работоспособность предлагаемой программы.
2. Изменить исходную программу, чтобы в функцию-счётчик можно было вводить как параметр начальное значение порождаемой последовательности.
3. Изменить функцию `Factory()` так, чтобы k -ый экземпляр генератора функций-счётчиков описывал последовательность целых с k .

```
1  #include <iostream>
2  auto Factory()
3  {
4      int count = 0;
5      auto ff = [count]() mutable
6      {
7          return count++;
8      };
9      return ff;
10 }
11 int main()
12 {
13     auto Counter1 = Factory();
14     std::cout << Counter1() << std::endl;
15     std::cout << Counter1() << std::endl;
16     auto Counter2 = Factory();
17     std::cout << Counter2() << std::endl;
18     std::cout << Counter2() << std::endl;
19     std::cout << Counter2() << std::endl;
20     system("pause");
21 }
```

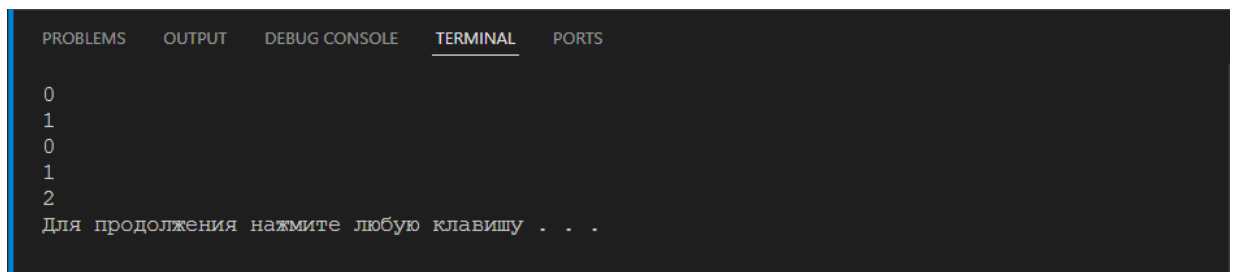


Рисунок 3.1 – Работа исходной версии программы

Чтобы изменить программу так, чтобы функция-счётчик принимала начальное значение последовательности в качестве параметра, нужно передать это значение в функцию.

```

1  #include <iostream>
2
3  auto Factory(int initialValue)
4  {
5      int count = initialValue;
6      auto ff = [count]() mutable
7      {
8          return count++;
9      };
10     return ff;
11 }
12
13 int main()
14 {
15     auto Counter1 = Factory(0);
16     std::cout << Counter1() << std::endl; // 0
17     std::cout << Counter1() << std::endl; // 1
18     auto Counter2 = Factory(10);
19     std::cout << Counter2() << std::endl; // 10
20     std::cout << Counter2() << std::endl; // 11
21     std::cout << Counter2() << std::endl; // 12
22     auto Counter3 = Factory(-5);
23     std::cout << Counter3() << std::endl; // -5
24     std::cout << Counter3() << std::endl; // -4
25     system("pause");
26 }

```

Функция `Factory` теперь принимает параметр `initialValue`, который используется для инициализации переменной `count`. Функция захватывает это значение по значению и увеличивает его при каждом вызове.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

0
1
10
11
12
-5
-4
Для продолжения нажмите любую клавишу . . .

```

Рисунок 3.2 – Работа первой модификации

Чтобы изменить функцию `Factory` так, чтобы k -ый экземпляр генератора функций-счётчиков порождал функцию-счётчик, генерирующую последовательность целых чисел, начинающуюся с k , нужно использовать глобальную или статическую переменную для отслеживания количества созданных экземпляров.

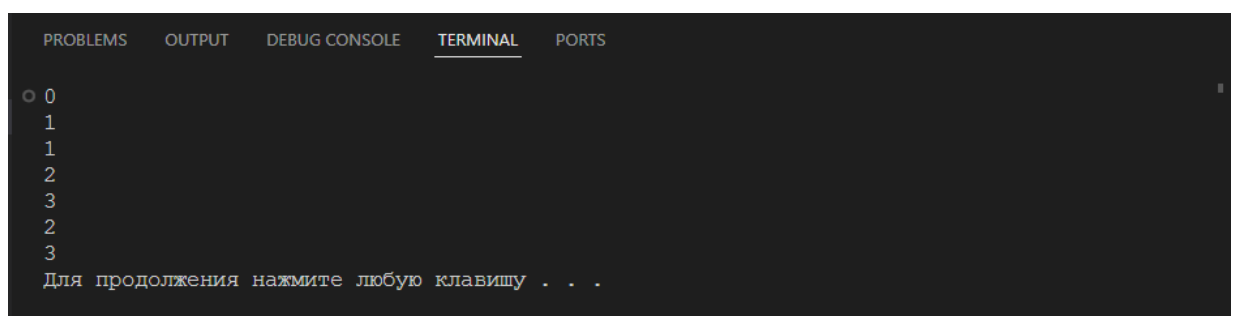
```

1  #include <iostream>
2
3  auto Factory()
4  {
5      static int instanceCount = 0;
6      int count = instanceCount;
7      instanceCount++;
8      auto ff = [count]() mutable
9      {
10         return count++;
11     };
12     return ff;
13 }
14
15 int main()
16 {
17     auto Counter1 = Factory();
18     std::cout << Counter1() << std::endl; // 0
19     std::cout << Counter1() << std::endl; // 1
20
21     auto Counter2 = Factory();
22     std::cout << Counter2() << std::endl; // 1
23     std::cout << Counter2() << std::endl; // 2
24     std::cout << Counter2() << std::endl; // 3
25
26     auto Counter3 = Factory();
27     std::cout << Counter3() << std::endl; // 2
28     std::cout << Counter3() << std::endl; // 3
29
30     system("pause");
31 }

```

Статическая переменная `instanceCount` сохраняет количество вызовов функции `Factory`. Она увеличивается на 1 при каждом вызове функции, что позволяет каждому новому экземпляру начинать с соответствующего значения.

Переменная `count` инициализируется значением `instanceCount`, которое затем увеличивается, чтобы обеспечить уникальность начального значения для следующего вызова `Factory`.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
0
1
1
2
3
2
3
Для продолжения нажмите любую клавишу . . .

```

Рисунок 3.3 – Работа второй модификации