Отладка в VS Code

Код поисковой системы продолжает расти и расширяться. Уже сейчас он содержит более ста строк. С увеличением количества строк растёт и число возможных ошибок.

Совершать ошибки — это нормально. Без ошибок не было бы и работы. Процесс нахождения и удаления ошибок в коде называется «отладка». В простом случае для этого можно просто просмотреть код. Для более сложных — использовать программные средства. Одно из них — дебаггер, его ещё называют «отладчик». Дебаггер есть в любой IDE, исключением не стал и Visual Studio Code.

Вернёмся к той версии поисковой системы, где вы учили её рассчитывать релевантность документов. Предположим, что в некоторый момент у вас получился такой код:

▶ **Код поисковой системы**

Создайте новый проект в VS Code и скопируйте в него этот код. Скомпилируйте и запустите программу. Программа работает, но не так, как ожидается.

При входных данных:

MARKDOWN

```
a an on the in is has been are with for from have be was
4
a small curly guinea pig with grey hair has been found
a young 50 year old crocodile wants to make friends
a strange brown creature was seen in the box of oranges
a strange animal with big ears is building a house for its friends
cheburashka with big ears likes oranges
```

Результат работы программы такой:

```
{ document_id = 3, relevance = 2 }
```

Вместо ожидаемого:

```
{ document_id = 2, relevance = 1 }
{ document_id = 3, relevance = 2 }
```

В программе ошибка. Давайте найдем её в режиме отладки и исправим.

Breakpoints — точки останова

Прежде, чем запускать режим отладки, нужно расставить точки останова — отметки, на которых программа будет прерываться. Дойдя до строки с точкой останова, программа как бы ставится на паузу — все операции прекращаются.

В этот момент вы можете в деталях рассмотреть, в каком состоянии находится программа: какие значения видимых переменных, какие функции находятся в стеке вызовов. Стек вызовов — это список функций, которые привели к текущей позиции в коде программы. Вверху стека всегда функция, выполняемая прямо сейчас. Внизу стека обычно функция `main`. Между ними незавершённые на данный момент функции в том порядке, в котором они были вызваны. Элементы стека называются фреймами. При отладке вы можете подняться по стеку на несколько фреймов выше. В многопоточных приложениях одновременно может существовать несколько стеков вызовов.

Можно даже внести некоторые изменения: поменять значения переменных или вручную вызвать какую-либо функцию. Когда вы сделаете необходимые изменения, программу можно возобновить. Она продолжит выполнение с того же места, как ни в чём не бывало. Точки останова актуальны только в режиме отладки. При простом запуске они игнорируются: программа выполняется без остановок.

Поставьте точку останова на 115 строку `const string query = ReadLine();`. Для этого нажмите на поле перед цифрой 115. Должна появиться красная точка. На этом этапе программа будет ожидать ввода запроса `query`.

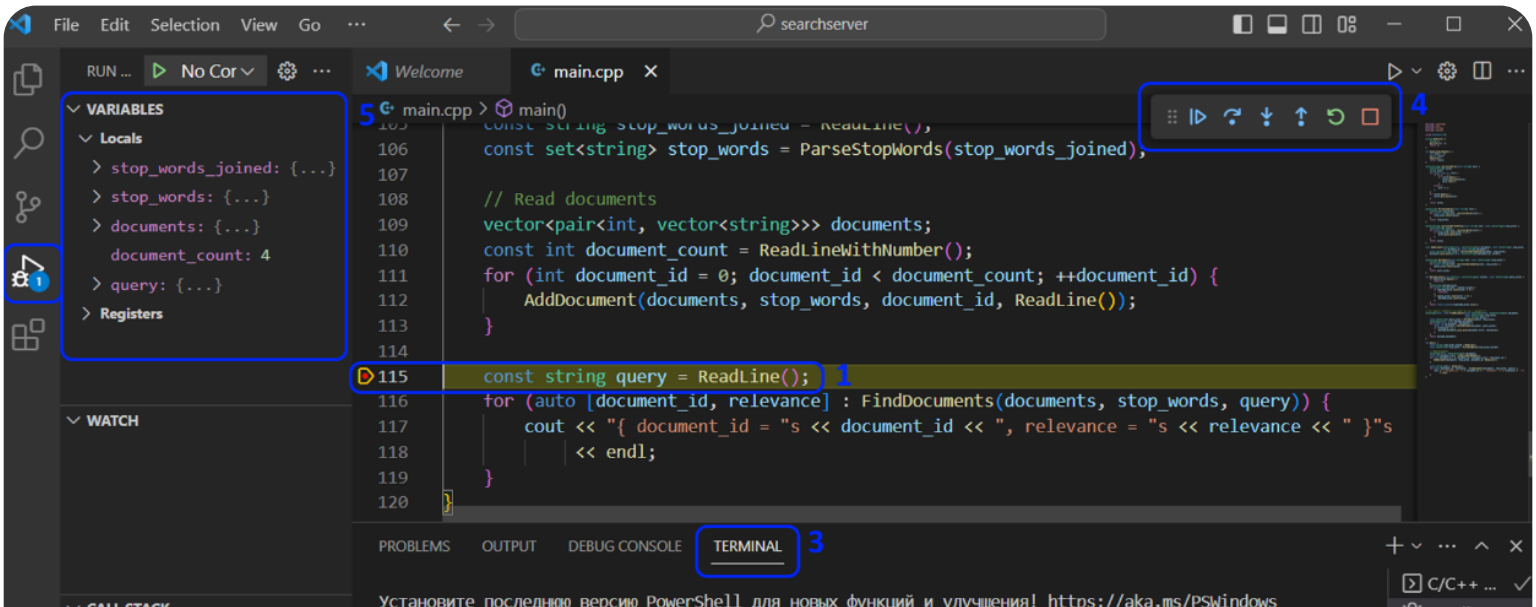
```cpp
}

int main() {
    const string stop_words_joined = ReadLine();
    const set<string> stop_words = ParseStopWords(stop_words_joined);

    // Read documents
    vector<pair<int, vector<string>>> documents;
    const int document_count = ReadLineWithNumber();
    for (int document_id = 0; document_id < document_count; ++document_id) {
        AddDocument(documents, stop_words, document_id, ReadLine());
    }

    const string query = ReadLine();
    for (auto [document_id, relevance] : FindDocuments(documents, stop_words, query)) {
        cout << "{ document_id = "s << document_id << ", relevance = "s << relevance << " }"s
             << endl;
    }
}
```

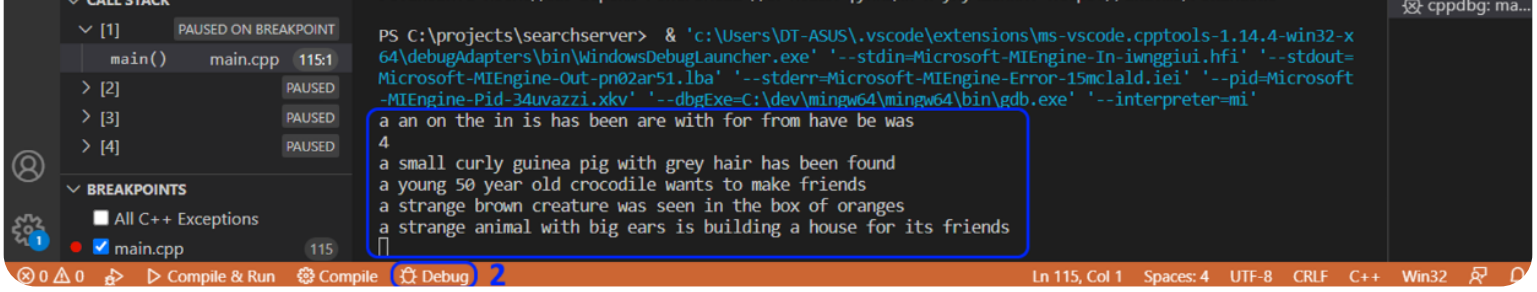
Установка точки останова

Режим отладки

Запустите программу в режиме отладки, нажав на кнопку Debug. В английском “bug“ — «ошибка» — также означает «жук». Поэтому кнопка запуска дебага во многих IDE выглядит соответствующе.

В терминале появится окно ввода. Вставьте в него входные данные. Программа отработает до строчки 115 и остановится на ней. Строка выделится желтым цветом. Окно VS Code должно выглядеть так:





1 — Строка, на которой остановилась программа, 2 — Кнопка запуска дебаггера, 3 — Терминал для ввода данных, 4 — Панель с действиями при отладке, 5 — Окно переменных

Посмотрим, что изменилось. Панель статуса стала оранжевой, появилась панель с действиями 4 и открылась вкладка Run&Debug с окном переменных 5.

Рассмотрим кнопки панели 4 подробнее.

- **Continue/Pause** — продолжить: возобновить выполнение программы до следующей точки останова или до завершения программы. Пауза: остановка запущенной программы в месте выполнения. Пока вы не ввели данные, была активна пауза, потому что программа выполнялась, хоть и ждала вашего ввода.
- **Step Over** — шаг с обходом: выполнить до следующей строки, не заходя вовнутрь функций.
- **Step Into** — шаг с заходом: выполнить до следующей строки, но в случае вызова функции зайти вовнутрь.
- **Step Out** — шаг с выходом: выполнить всё до завершения текущей функции.
- **Restart** — перезапуск программы.
- **Stop** — остановка программы.

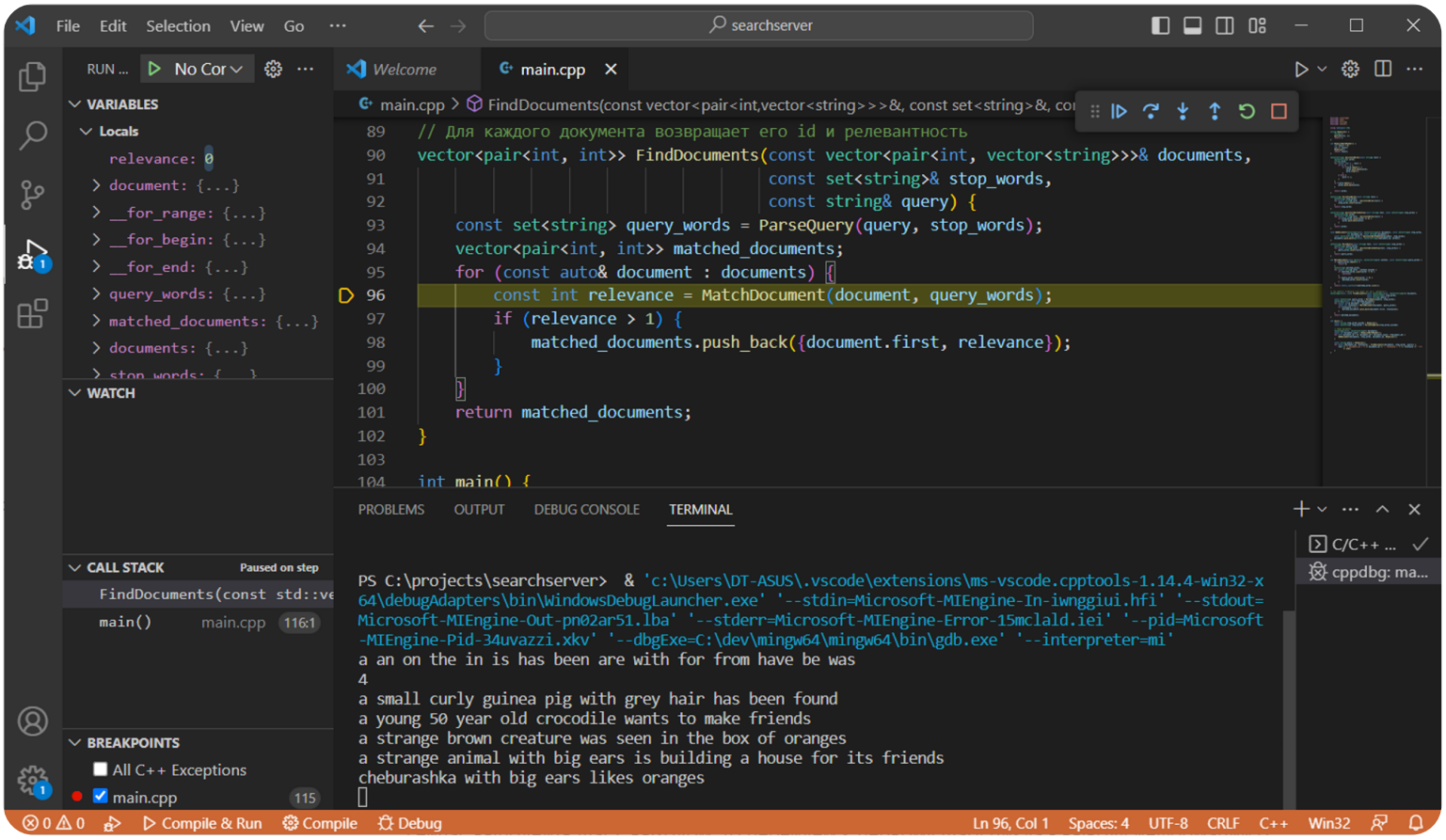
Если при выполнения шага программа проходит через точку останова, то происходит прерывание.

В окне 5 можно отследить переменные, которые доступны в текущем контексте выполнения программы. Документы уже введены, поэтому видим, что значение переменной `document_count` равно четырём.

Отладка

Выполните шаг с обходом. Программа встанет в режим паузы и будет ожидать ввода запроса. В терминале отобразится запрос. Остаётся только подтвердить ввод — нажать Enter. И мы перейдем на строку 116.

Выполните шаг с заходом, чтобы отследить выполнение функции `FindDocuments()`. Если сейчас выполним шаг с выходом, то перейдём к первому шагу цикла — выводу информации о документе. Мы хотим убедиться, что функция работает правильно, поэтому выполним шаг с обходом и дойдём до строки 96.

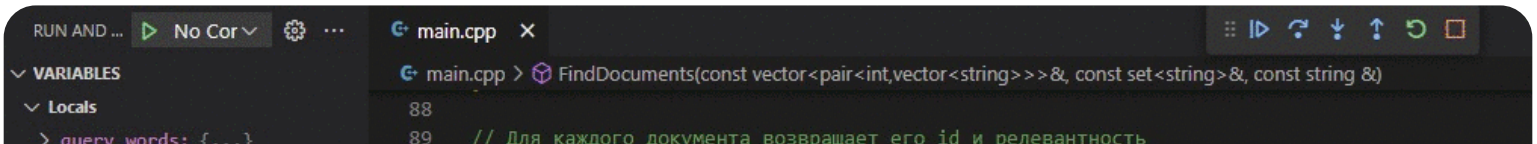


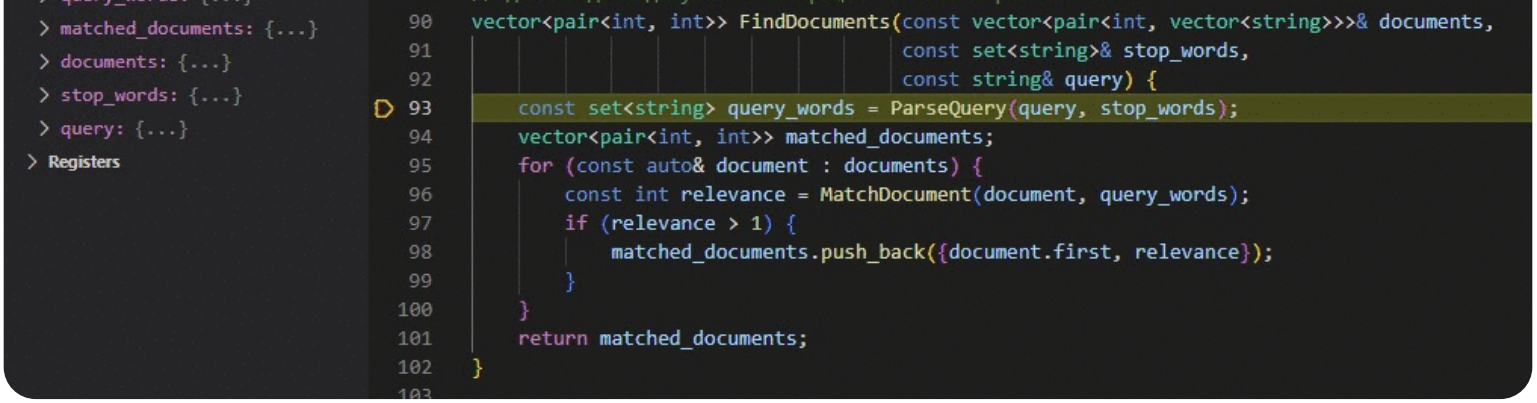
Остановка в функции FindDocuments

В окно с переменными добавилась переменная `relevance`. Её текущее значение 0 подсветилось. Это значит, что она или только добавилась, или поменяла своё значение с предыдущего шага.

В нашем случае действия по её расчету ещё не произошли, поэтому переходим на следующую строку, чтобы узнать релевантность первого документа.

Релевантность первых двух документов равна нулю, поэтому никаких изменений не происходит. Продолжайте выполнять шаги с обходом. Релевантность третьего документа равна единице, информация о нём должна возвращаться в результате выполнения функции. Но условие оператора `if` не истинно, потому что содержит ошибку: нужно добавлять документы с релевантностью больше нуля, а не единицы.





Отладка функции FindDocuments

На этом отладка программы завершена. Остановите её, исправьте ошибку и убедитесь, что программа работает правильно.

Отладка нештатных ситуаций

Рассмотрим ещё одну возможность дебаггера. Допустим, вы хотите упростить себе процесс обучения и для этого пишите программу-ассистента. Она рассчитывает количество задач, которые надо выполнять ежедневно, чтобы уложиться в дедлайн. Получился такой код:

CPP

```cpp
#include <iostream>

using namespace std;

int main() {
    int tasks, days;
    cout << "Number of tasks: "s;
    cin >> tasks;
    cout << "Amount of days: "s;
    cin >> days;
    double result =  tasks / days;
    cout << "You need to complete "s << result << " tasks daily"s << endl;
}
```

Для начала программа будет поддерживать только целочисленные значения. Проверим её работу:

MARKDOWN

```
Number of tasks: 8
Amount of days: 4
You need to complete 2 tasks daily
```

Программа верно рассчитала количество задач.

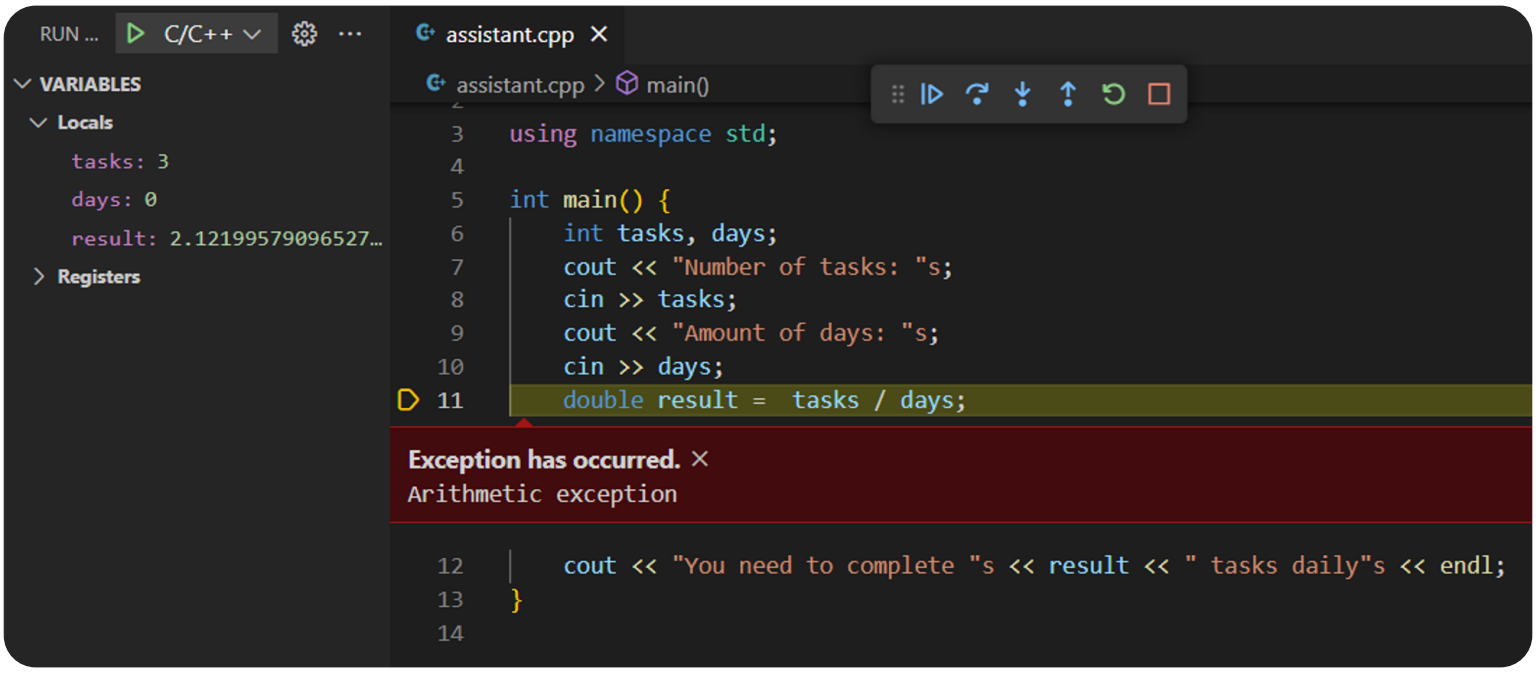
А теперь проверим работу программы в последний день дедлайна, когда остаётся 0 дней:

MARKDOWN

```
Number of tasks: 3
Amount of days: 0
```

Программа завершила работу, но не вывела последнюю строку. Точное поведение будет зависеть от операционной системы и компилятора. Программа могла вывести какое-то сообщение или молча завершиться. Но ясно одно: что-то пошло не так.

Чтобы разобраться в некорректном поведении, используем дебаггер. При нештатной ситуации точку останова можно не ставить, программа сама прервётся при попытке совершить недопустимую операцию. Запустим программу в режиме отладки и введем те же данные:



Дебаггер остановился на строке с исключением

В 11 строке происходит деление на ноль. Это недопустимая операция. Она приводит к неопределённому поведению. Как и отмечено выше, на разных

платформах неопределённое поведение проявляется по-разному. Как правило, дебаггер обнаруживает деление на ноль и останавливает выполнение программы. Следующий шаг отладки приведёт к её аварийному завершению.

Дебаггер помог определить место возникновения ошибки. Далее в курсе вы узнаете, как защититься от подобных ситуаций и сделать программу безопасной.

В этом уроке мы рассмотрели только самые основы отладки. Дебаггер имеет ещё много возможностей. Например, с его помощью можно:

- делать точки останова условными;
- работать с трейспоинтами — отметками, которые не останавливают выполнение, но выводят определённую информацию;
- работать с брейкпоинтами на функцию или даже на память;
- изучать выполняемые процессорные инструкции.

С помощью отладки мы по шагам отследили выполнение программ, посмотрели, что происходит на разных этапах их выполнения и определили места возникновения ошибок. В дальнейшем вам не раз придётся отлаживать свои программы, находить в них ошибки и исправлять их. С практикой отладка будет становиться проще и войдёт в привычку.