

Основы моделирования детекторных систем и их применение в ядерно- и космофизических экспериментах с использованием пакета программ GEANT4

Введение

При подготовке современных экспериментов в области ядерной физики и космофизики требуется выполнить огромный объём работы в процессе разработки комплекса программных средств и приложений. Особо важным является постоянно растущая необходимость в масштабном, точном и исчерпывающем моделировании регистрации частиц.

Современные реалии ведут к увеличению размера, сложности и чувствительности детекторов, что вызывает необходимость использовать мощные компьютерные системы, позволяющие проводить объёмное и сложное моделирование.

Для реализации задач, возникающих при разработке сложных детекторных систем, в настоящее время широко используется объектно-ориентированный инструмент моделирования Geant4.

История развития специализированных программ моделирования

- ❑ **1950е - середина 60х** – первые программы расчета взаимодействия частиц в веществе
- ❑ **середина 60х – начало 70х** – программы моделирования э/м и адронных каскадов
EGS – **E**lectron **G**amma **S**hower; GHEISHA – адронные ливни;
FLUKA - (**FL**Uktuierende **KA**skade, флуктуирующий каскад)
- ❑ **1974** – GEANT (**GE**ometry **and T**ransport)
- ❑ **1982** – GEANT3 (Fortran)
- ❑ **1993** - GEANT4 (C++)

Другие программы для расчёта прохождения частиц через вещество

- **MCNP** (**M**onte-**C**arlo **N**-**P**article Transport Code) – написана в Los Alamos National Laboratory, первоначально применялась для расчета реакторов
- **MARS** – расчет ливней
- **CALOR95** – расчет ливней и источников нейтронов
- **CORSIKA, AIRES** – расчет широких атмосферных ливней
- и другие

FLUKA (Fortran)

- Первоначально разрабатывалась для расчета защиты в проекте Super Proton Synchrotron в ЦЕРН (1962-1978)
- В настоящее время универсальная программа расчета взаимодействия частиц с веществом
- Хорошее моделирование адронных ливней
- Сложность описания геометрии
- Лицензионные ограничения

MCNP (Fortran)

- В настоящее время универсальная программа расчета взаимодействия частиц с веществом
- Хорошее моделирование нейтронных процессов и процессов при низких энергиях
- Оригинальный подход к описанию геометрии
- Лицензионные и экспортные ограничения
- После выхода версии 4 разделилась на две ветви: MCNP5 и MCNPX (MCNP+LANL)

GEANT3 (Fortran)

- Первая версия появилась в 1974 году в ЦЕРН
- Описание физических процессов основано на программах EGS (э/м ливни) и GHEISHA (адронные ливни)
- Пакет GEANT3 появился в 1982 году и был использован для моделирования детекторов в экспериментах Large Electron-Positron Collider
- Основной инструмент моделирования в физике частиц на протяжении 30 лет

Пакет программ GEANT4 (C++)

- Объектно-ориентированная программа с функциональностью GEANT3
- Первая версия пакета появилась в 1995 году
- Первое применение – эксперимент BaBar
Эксперимент в области физики элементарных частиц.
Проводился в Стэнфордской лаборатории SLAC в Калифорнии, США.
Цель: изучение нарушений CP-симметрии при распаде B-мезонов.
- С 2004 года – основная программа моделирования в экспериментах на LHC (Large Hadron Collider)
- Широкое применение в физике частиц, космонавтике (European Space Agency), радиационной медицине.

История разработки GEANT4

Разработка GEANT4 началась в результате двух независимых исследований, проводимых CERN и KEK (The High Energy Accelerator Research Organization, Japan) в 1993 году.

Обе группы занимались изучением того как современные компьютерные технологии могут быть применены для улучшения уже существующих программ GEANT3, которые были эталоном.

Эти исследования были объединены и представлены в CERN Detector Research and Development Committee (DRDC) с целью создания системы моделирования на основе объектно-ориентированных технологий.

Так был создан проект RD44, ставший результатом сотрудничества ученых и инженеров 10-ков экспериментов, проводимых в Европе, Канаде, России, США и Японии.

В процессе разработки проекта RD44 был реализован дизайн, адаптирующий объектно-ориентированную методологию языка C++.

В декабре 1998 год состоялся релиз первой версии продукта.

В январе 1999 года было создано сообщество, целью которого стала дальнейшая разработка и совершенствование продукта.

Сам же продукт получил название GEANT4.

Организация коллаборации

Memorandum of Understanding (MoU) подписан всеми участвующими сторонами и регулирует официальное сотрудничество.

MoU подлежит обновлению каждые два года и устанавливает структуру коллаборации, состоящую из совета по сотрудничеству (CB), Технического руководящего совета (TSB), и нескольких рабочих групп.

В меморандуме также определяется каким образом средства для совместной работы (деньги, рабочая сила, эксперименты и ключевые роли) измеряются в «единицах взносов» (Contribution Units (CU)).

В число участвующих групп входят отдельные кооперации, лаборатории и национальные институты.

В функции CB входит обновление MoU, управления ресурсами и распределение обязанностей между аффилированными лицами.

TSB представляет собой форум, на котором обсуждаются и решаются технические вопросы такие как: детали разработки программного обеспечения и вопросы реализации физических моделей, где приоритеты отдаются запросам пользователей.

Основными задачами TSB является надзор за производственным обслуживанием и поддержкой пользователей, а также контроль за дальнейшей разработкой проекта.

TSB возглавляет «представитель коллаборации», который назначается и подчиняется CB. Представитель избирается каждые два года.

12th International Geant4 Tutorial in Korea 2025

Feb 3 – 7, 2025
Pohang Accelerator Laboratory
Asia/Seoul timezone



geant4.web.cern.ch/collaboration/members



[Home](#) > [Collaboration](#) > **Geant4 members and contributors**

Geant4 members and contributors

157 members and 29 contributors

geant4.web.cern.ch/download/11.3.0.html

[Home](#) > [Download](#) > **Download Geant4-11.3.0**

Download Geant4-11.3.0

First released 06 Dec 2024

[Old releases](#)

The Geant4 developer team (Makoto Asai, Soon Yung Jun, Luis Sarmiento Pico)

Организация общей структуры и поддержка пользователей

Geant4 обладает модульной структурой.

Каждый домен программного обеспечения GEANT4 соответствует отдельному компоненту (библиотеке) и индивидуально управляется рабочей группой экспертов.

Для каждой из задач, таких как: тестирование, обеспечение качества, управление программным обеспечением и управление документацией, существует своя рабочая группа.

Координатор, выбираемый TSB, возглавляет каждую отдельную группу. Существует также общий координатор выпуска конкретной версии.

Каждая группа может работать параллельно.

Такая декомпозиция задач делает возможной разработку распределенного программного обеспечения во всем мире.

На официальном сайте GEANT4 доступна web-система отчетов и список часто задаваемых вопросов (FAQ), представлен форум пользователей с под форумами по соответствующими областями, представляющими особый интерес.

Оказывается помощь в решении проблем, связанных с кодом, проводятся консультации по использованию пакета, а также даются ответы на запросы об улучшении.

Оказывается помощь в исследовании аномальных результатов.

Способ моделирования процессов в детекторе

Методы Монте-Карло

Методы Монте-Карло – это численные методы решения прикладных математических задач при помощи моделирования случайных величин.

Идея родилась при работе над “Манхэттенским проектом”

Проект «Манхэттен» — кодовое название программы США по разработке ядерного оружия, осуществление которой началось 13 августа 1942 года. Перед этим исследования велись в «Урановом комитете» (*S-1 Uranium Committee*, с 1939 года). В проекте принимали участие учёные из Соединённых штатов Америки, Великобритании, Германии и Канады.

В рамках проекта были созданы три атомные бомбы: плутониевая «Штучка» (*Gadget*) (взорвана при первом ядерном испытании), урановый «Малыш» (*Little Boy*) (сброшена на Хиросиму 6 августа 1945 года) и плутониевый «Толстяк» (*Fat Man*) (сброшена на Нагасаки 9 августа 1945 года).

S.M. Ulam, J. von Neumann, “On combination of stochastic and deterministic processes”, Bull. Amer. Math. Soc. 53 1120 (1947)

S.M. Ulam, N. Metropolis, “The Monte-Carlo method”, J. Amer. Statist. Assoc. 1949 , 44 Vol 247, 335-341

Сущность метода Монте-Карло

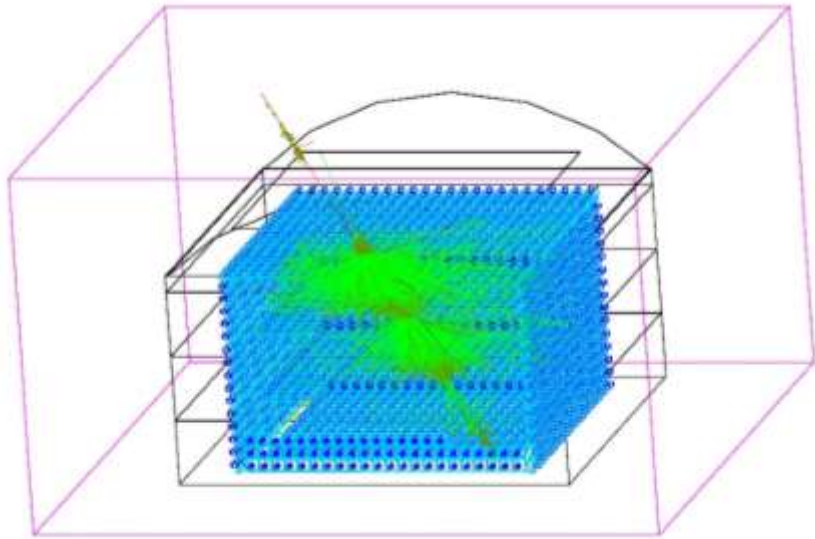
- Многие явления в природе являются случайными.
- Случайность означает многовариантность и непредсказуемость результата при повторных испытаниях в одних и тех же условиях.
- Анализ различных случайных явлений показал, что среди них есть такие, для которых наблюдается статистическая устойчивость, т.е. через хаос случайностей просвечивается нечто повторяющееся, закономерное, существует статистически устойчивая часть явления.
- Определение статистически устойчивой части случайной последовательности результатов является целью исследований случайных явлений.
- Последовательности случайных чисел, статистически эквивалентные тем, что получаются экспериментально, можно генерировать на компьютере.
- Соединяя генерацию случайных чисел и известные статистически устойчивые части случайного явления (распределения плотности вероятности), производят статистическое моделирование явления.
- Такой способ моделирования явлений получил название метода Монте-Карло.

Применение метода Монте-Карло в физике частиц

Эксперимент в физике частиц: измерение отклика (сигналов) детектора, возникающих при взаимодействии частиц с веществом детектора.

Взаимодействие частиц с материалом детектора – совокупность случайных процессов (ионизационные потери, радиационные потери, многократное Кулоновское рассеяние, ядерное рассеяние, ядерные реакции, фотоэффект,...) .

Моделирование случайных процессов сводится к моделированию дискретных случайных величин, которые определяют вероятность реализации этих случайных процессов.



Прохождение мюона с энергией 10 ГэВ через подземный сцинтилляционный телескоп Баксанской нейтринной обсерватории.

Метод Монте-Карло обеспечивает удобный способ математического моделирования процессов взаимодействия частиц с веществом детектора.

Каждый процесс описывается математической моделью с использованием **генератора случайных величин**.

ПРИМЕР. Плотность вероятности для процесса поглощения в слое вещества :

$$\rho(x) = \sigma n \times e^{-\sigma n x}$$

Численное моделирование распределений случайных величин для каждой модели позволяет определить наиболее вероятный процесс взаимодействия частиц с веществом детектора.

Общее описание GEANT4

GEANT4 представляет собой набор программ для моделирования прохождения частиц через вещество на основе методов Монте-Карло.

Основу GEANT4 составляют: «гибкое» описание геометрии, визуализация, интерфейс пользователя и набор физических моделей, содержащий информацию о взаимодействии частиц с материалами в широком диапазоне энергий.

Данные для физических моделей получены из огромного количества источников по всему миру, и в этом отношении GEANT4 представляет собой беспрецедентное хранилище информации, включающее в себя значительную часть всего, что известно о взаимодействиях частиц.

Основные используемые модели:

- **Электромагнитные процессы**
- **Адронные процессы**
- **Фотон-адронные и лептон-адронные процессы**
- **Процессы с участием оптических фотонов**
- **Моделирование распадов**
- **Параметризация ливней**
- **Методики использования статистических весов**

Используемые в GEANT4 физические модели продолжают дорабатываться и развиваться.

Установка Geant4

Установка Geant4 Release 11.1.2 (19 Jun 2023) под ОС Linux Ubuntu 22.04.3 LTS (Jammy Jellyfish)

1. Установка Ubuntu (официальный сайт <https://releases.ubuntu.com/22.04>)

Шаг 1. Загрузка образа

Шаг 2. Запись образа на флешку (https://losst.pro/zapis-obraza-linux-na-fleshku#2_Запись_образа_Etcher)

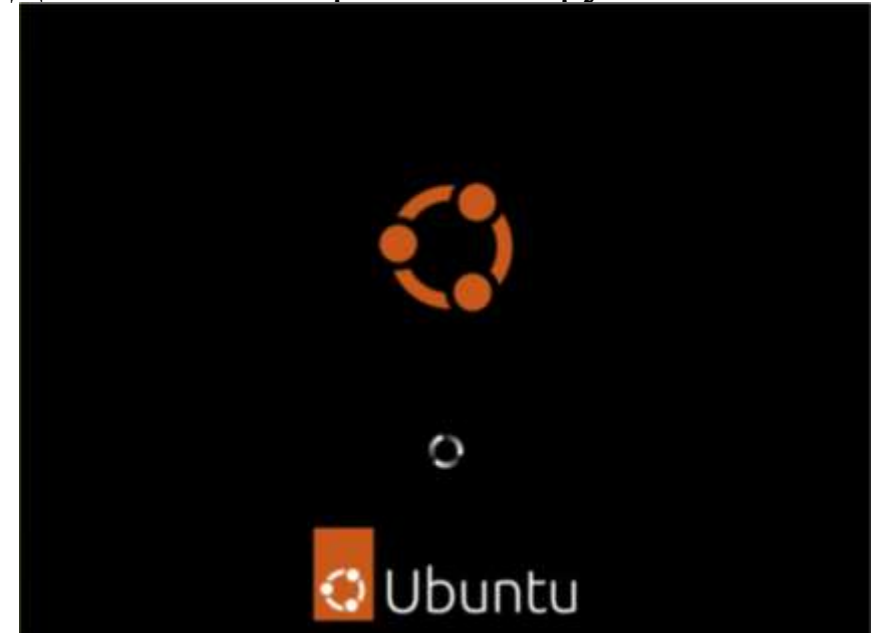
Шаг 3. Загрузка.

После завершения записи на флешку следует перезагрузить компьютер, открыть настройки BIOS и выбрать в качестве основного загрузочного устройства вашу флешку.

После загрузки с флешки отобразится меню Grub, в котором нужно выбрать первый пункт **Try or install Ubuntu**:



Дождитесь завершения загрузки системы:



Шаг 4. Запуск установки

Когда система загрузится, в открывшемся окне необходимо выбрать язык системы, а затем нажать **Install**

Ubuntu или **Установить Ubuntu**, если вы выбрали русский язык.

Шаг 7. Способ разметки диска

Шаг 8. Таблица разделов

Шаг 8.1 Разделы для загрузчика

Шаг 9. Корневой раздел

Шаг 10. Домашний раздел

Шаг 11. Завершение разметки

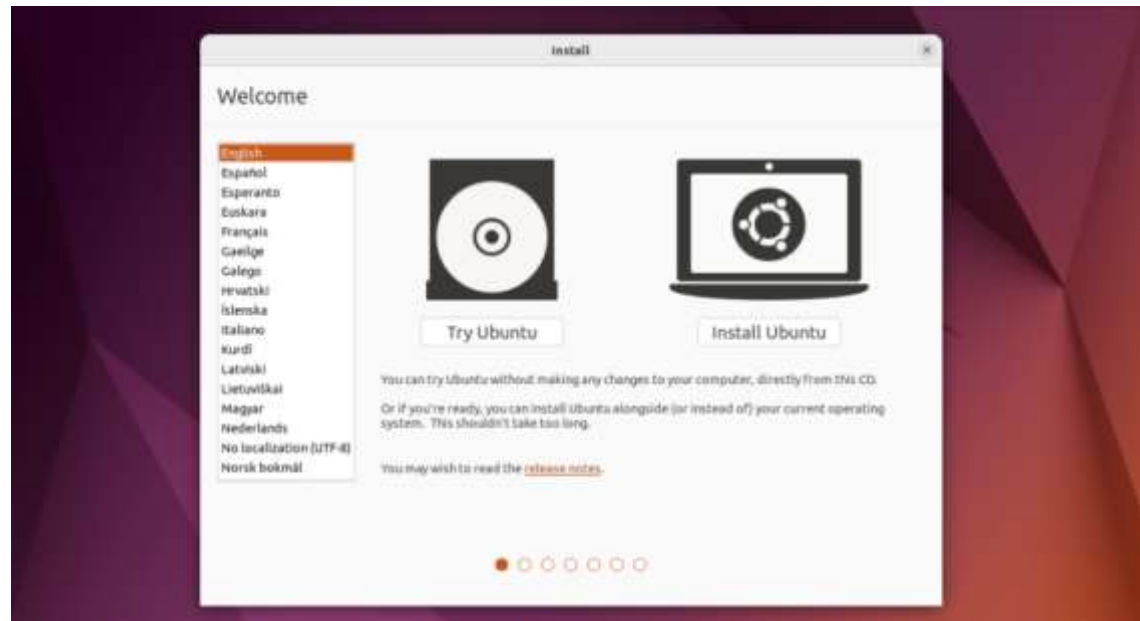
.....

Шаг 13. Создание пользователя

Шаг 14. Завершение установки

Шаг 15. Перезагрузка

.....



После установки **Ubuntu 22.04.3**

1) **sudo apt-get update** (обновляет список доступных пакетов программного обеспечения из официальных репозиториях)

2) **sudo apt-get install build-essential** (установка пакетов для компиляции программного обеспечения)

3) **sudo apt-get install expat** (установка потокоориентированной библиотеки парсинга XML)

4) **sudo apt install libhdf5-openmpi-dev** (установка пакета с файлами, обеспечивающими поддержку OpenMPI)

5) **sudo apt-get install qt5-qmake** (утилита для генерации make-файлов Qt5, кросс-платформенной инфраструктуры для написания приложений с пользовательским интерфейсом на C++, QT5 содержит набор графических элементов управления стандартного пользовательского интерфейса)

6) **sudo apt install qtbase5-dev** (пакет для построения приложений в Qt5)

7) **sudo apt-get install libx11-dev libxmu-dev** (пакеты, обеспечивающие интерфейс для основных функций оконной системы)

2. Установка Geant4 (официальный сайт: <https://geant4.web.cern.ch/>)

Процесс установки <https://dev.asifmoda.com/geant4/ustanovka>

1. Создать папку в домашней директории

`home/user/geant4/install_path/geant4-v11.1.2`

2. Скачать архив с дистрибутивом и распаковать его в выбранной директории

RELEASE NOTES

See:

[Main Release Notes](#) - [Patch-1](#) - [Patch-2](#) -

Source code

Source code is freely available from [CERN GitLab](#) or through [GitHub](#).

Source code can also be browsed through the [LXR source code browser](#).

Download zip

Download tar.gz

Download tar.bz2

Download tar

3. Создать директории для установки и компиляции

`home/user/geant4/install_path/geant4-v11.1.2-build`

`home/user/geant4/install_path/geant4-v11.1.2-install`

4. Настройка установки Geant4

В директории `home/user/geant4/install_path/geant4-v11.1.2-build` запустить `cmake`

```
cmake -DCMAKE_INSTALL_PREFIX=../geant4-v11.1.2-install ../geant4-v11.1.2 -DGEANT4_INSTALL_DATA=ON  
-DGEANT4_USE_QT=ON -DGEANT4_USE_OPENGL_X11=ON -DGEANT4_USE_RAYTRACER_X11=ON
```

5. Компиляция Geant4

Из той же директории
`make`

6. Установка Geant4

Из той же директории
`make install`

Создание проекта

Установка переменных окружения для работы с установленной версией

`source home/user/geant4/install_path/geant4-v11.1.2-install/bin/geant4.sh`

или включить в `.bashrc`

```
./home/user/geant4/install_path/geant4-v11.1.2-install/bin/geant4.sh
```

Сборка проекта с использованием cmake

Для работы `cmake` необходим файл `CMakeLists.txt`

Содержание `CMakeLists.txt`:

Расположение исходного кода и заголовочных файлов проекта

```
file(GLOB sources ${PROJECT_SOURCE_DIR}/src/*.cpp)
```

```
file(GLOB headers ${PROJECT_SOURCE_DIR}/include/*.hh)
```

Добавление исполняемого файла и его линковка с библиотеками Geant4

```
add_executable(example1 example1.cpp ${sources} ${headers})
```

```
target_link_libraries(example1 ${Geant4_LIBRARIES})
```

Компиляция проекта `make`

Очистить кэш компиляции `make clean`

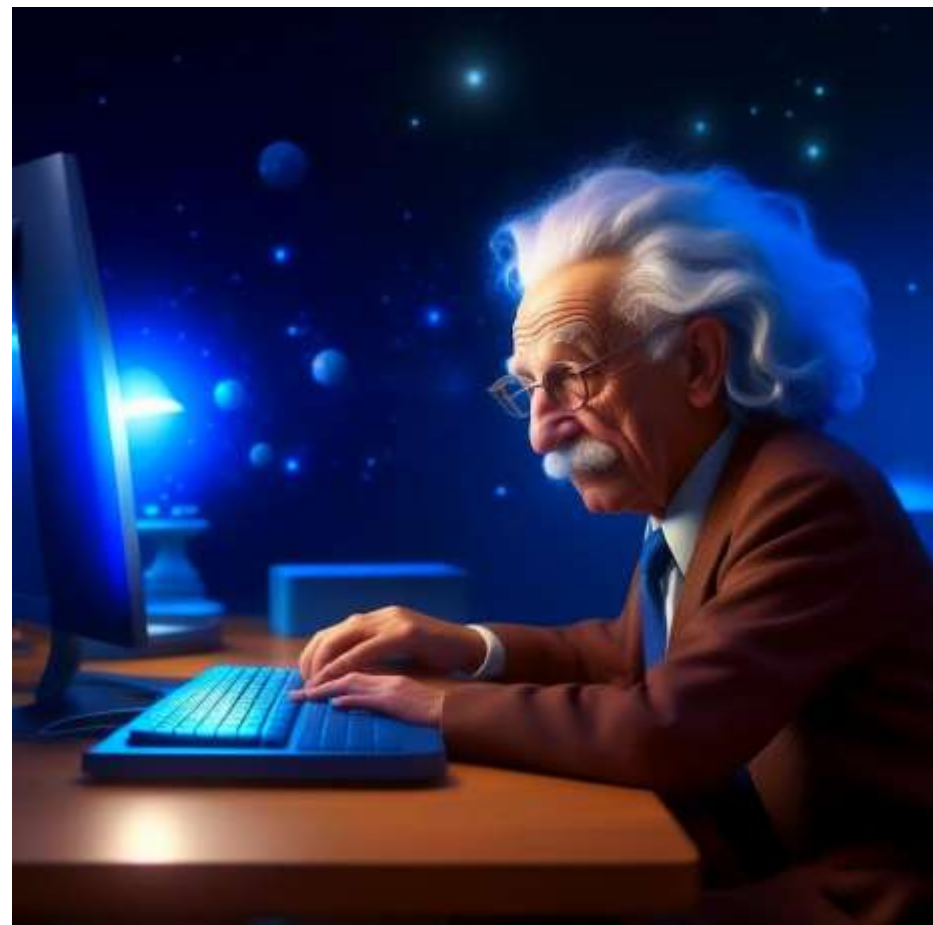
Geant4 User Guide

<https://geant4-userdoc.web.cern.ch/UsersGuides/ForApplicationDeveloper/fo/BookForApplicationDevelopers.pdf>

Дистрибутивы версий Geant4 включает в себя примеры, разделенные на три уровня:

1. Basic: базовые примеры для понимания возможностей.
2. Extended: примеры специализируются на специфических приложениях.
3. Advanced: программы созданные с помощью Geant4 в области исследований физики высоких энергий.

<https://www.google.ru>



Создание модульного проекта

Geant4 предоставляет необходимые библиотеки и сервисы, включая визуализацию и возможность интерактивного режима работы

Задача пользователя — создание исполняемой программы моделирования установки.

Основными модулями моделирования прохождения частицы через вещество являются:

- ☐ геометрия и материалы
- ☐ взаимодействие частиц с веществом
- ☐ трекинг
- ☐ управление событиями и треками
- ☐ визуализация
- ☐ пользовательский интерфейс

Эти модули являются основой для категорий классов с последовательным интерфейсом.

Пользователь разрабатывает свою программу самостоятельно, выбирая необходимые компоненты из пакета программ Geant4.

В простейшем случае от пользователя требуется описание геометрии детектора, списка физических процессов, учитываемых в моделировании и генерация первичной вершины.

Простой проект: TrackStack

TrackStack.cc (основной файл)

DataWriter.cc (DataWriter.hh)

Loader.cc (DataWriter.hh)

Geometry.cc (Geometry.hh)

Action.cc (Action.hh)

PrimaryPart.cc (PrimaryPart.hh)

RunAct.cc (RunAct.hh)

EventAct.cc (EventAct.hh)

TrackAct.cc (TrackAct.hh)

StackAct.cc (StackAct.hh)

StepAct.cc (StepAct.hh)

github.com/AlexeyAnatolievichLeonov/Part1/

скачать TrackStack.zip

1. Сборка проекта из ~/build: cmake ..
2. Компиляция проекта из ~/build: make
3. Запуск проекта из ~/build:
 - ./TrackStack (интерактивный режим с графическим интерфейсом)
 - ./TrackStack input.in (пакетный режим, управляемый сценарием)

Запуск проекта Geant4, основы C++

Ubuntu, Midnight Commander

Редактирование файла:

F4 – открытие файла;

Alt+6 – копирование выделенного фрагмента текста в системный буфер обмена;

Ctrl+U – вставить текст из буфера;

Ctrl+K – удалить выделенный фрагмент текста;

Ctrl+X – закрыть файл, после подтверждения 'Y' или отказа 'N' от сохранения изменений.

github.com/AlexeyAnatolievichLeonov/Part1/

Проект: TrackStack


Копирование проекта в другую директорию TrackStack2.

После копирования в директории TrackStack2/build удалить: Директорию CmakeFiles, файл CmakeCache.txt

Переименовать файл TrackStack2/TrackStack.cc  TrackStack2/TrackStack2.cc

Исправить в файле TrackStack2/Cmakelists.txt:

set(Name TrackStack)  Set(Name TrackStack2)

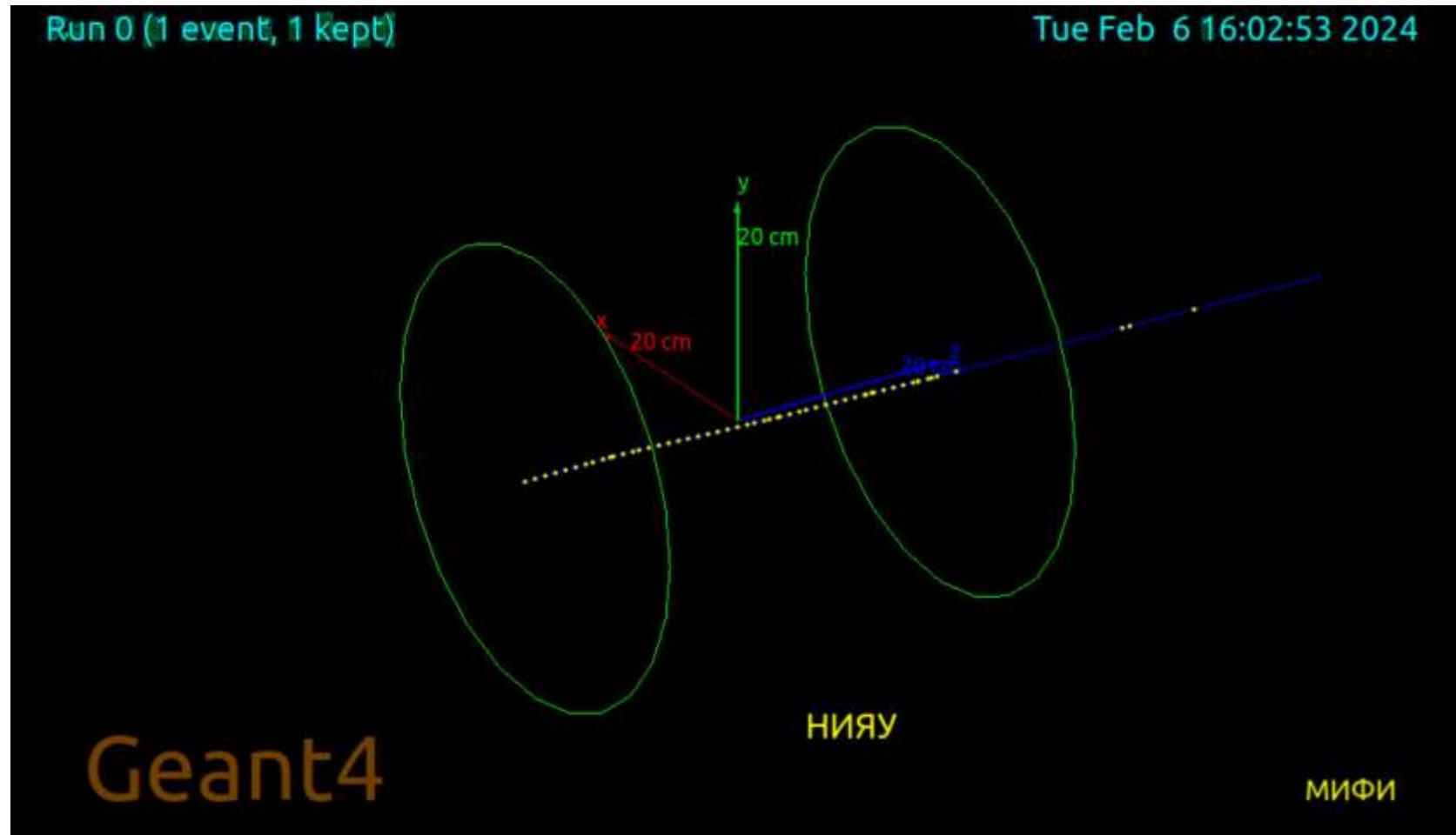
add_executable(\${NAME} TrackStack.cc \${sources} \${headers}) 
add_executable(\${NAME} TrackStack2.cc \${sources} \${headers})

Сборка проекта из ~/build: cmake .. Компиляция проекта из ~/build: make Запуск проекта из ~/build: ./TrackStack2

Простой проект: TrackStack

~/build/vis.mac – файл с настройками визуализации и запуска в интерактивном режиме с графическим интерфейсом

~/build/input.in – файл с настройками запуска в пакетном режиме, управляемом сценарием



Основные понятия Geant4

Сеанс (Run)

Событие (Event)

Трек (Track)

Шаг (Step)

Срабатывание (Hit)

Сеанс (Run)

- ✓ Период набора статистики, в котором не меняются условия проведения эксперимента (параметры пучка, конфигурация и параметры детектора, материал мишени и т.п.)
- ✓ В Geant4 – самый крупный элемент моделирования, состоящий из последовательности событий
- ✓ Во время сеанса описание геометрии и набор физических процессов остаются неизменными
- ✓ Представлен классом G4Run
- ✓ Управление осуществляется объектом класса G4RunManager

Событие (Event)

- ✓ Единичное независимое измерение физического явления детектором
- ✓ В Geant4 представлено классом G4Event
- ✓ G4Event содержит все входные и выходные характеристики (исходные частицы, срабатывания и т.д.) данного (текущего) события
- ✓ G4Event создается объектом класса G4RunManager и передается объекту класса G4EventManager, который осуществляет управление событием

Структура события

- Первичная вершина и первичная частица
- Траектории
- Коллекция срабатываний
- G4EventManager управляет объектами G4Track, соответствующими данному событию, взаимодействуя с объектами классов G4TrackManager и G4StackManager

Трек (Track) и Шаг (Step)

- Шаг представлен классом G4Step и описывает минимальное продвижение частицы через вещество с учетом различных физических процессов
- Треки представлены классом G4Track и содержат информацию о последнем шаге.
- Объект G4Track, таким образом, описывает полное продвижение частицы в веществе к моменту обращения к данному объекту

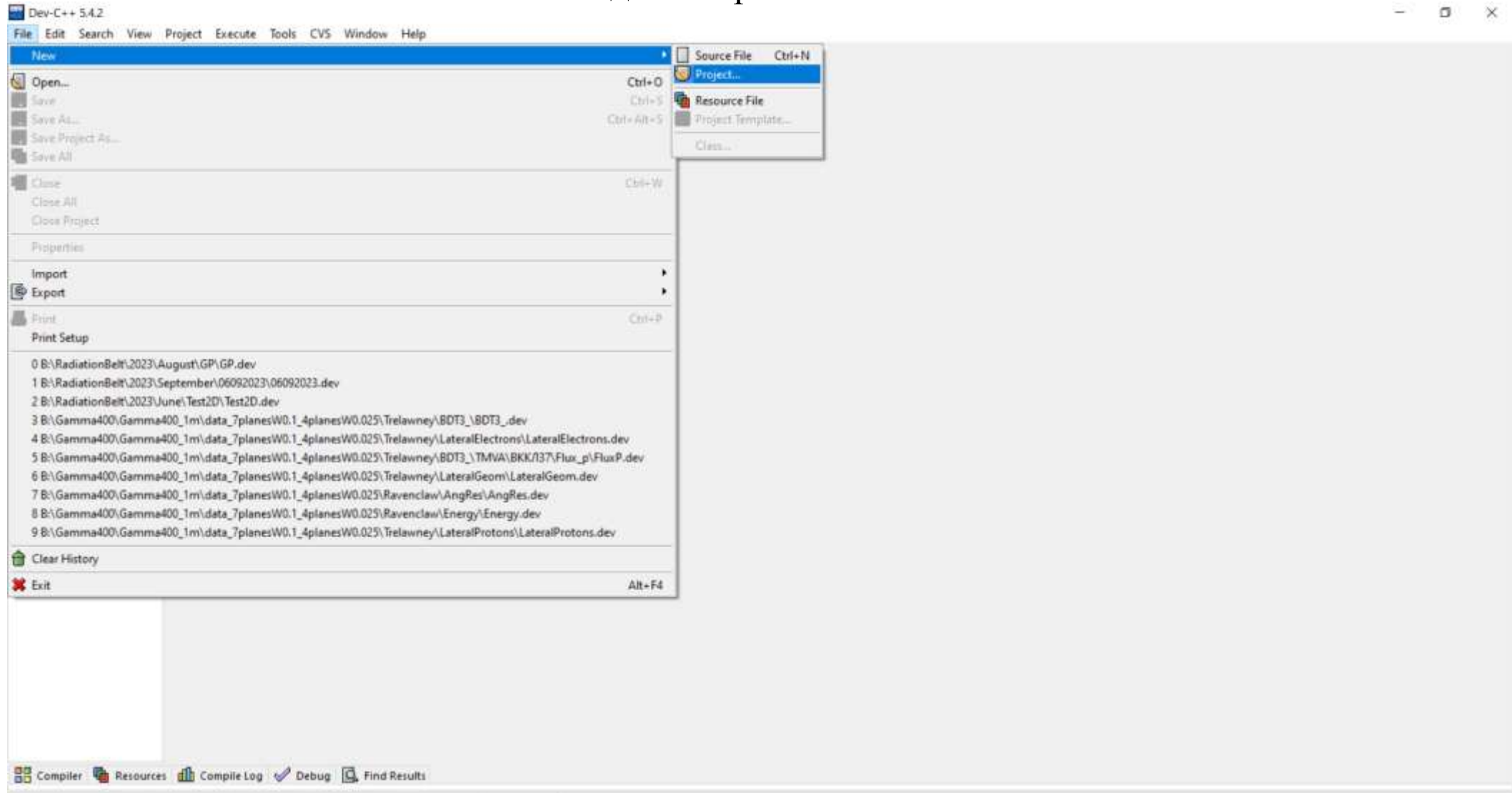
Срабатывание (Hit)

- Описывает единичное взаимодействие частицы с веществом в детектирующем объеме
- Содержит информацию о координате и времени взаимодействия, энергии и импульсе частицы в этой точке, энерговыделении, геометрическую информацию (объем, в котором произошло взаимодействие и т.п.)
- Является “истинной” Монте-Карло информацией (Monte-Carlo truth)

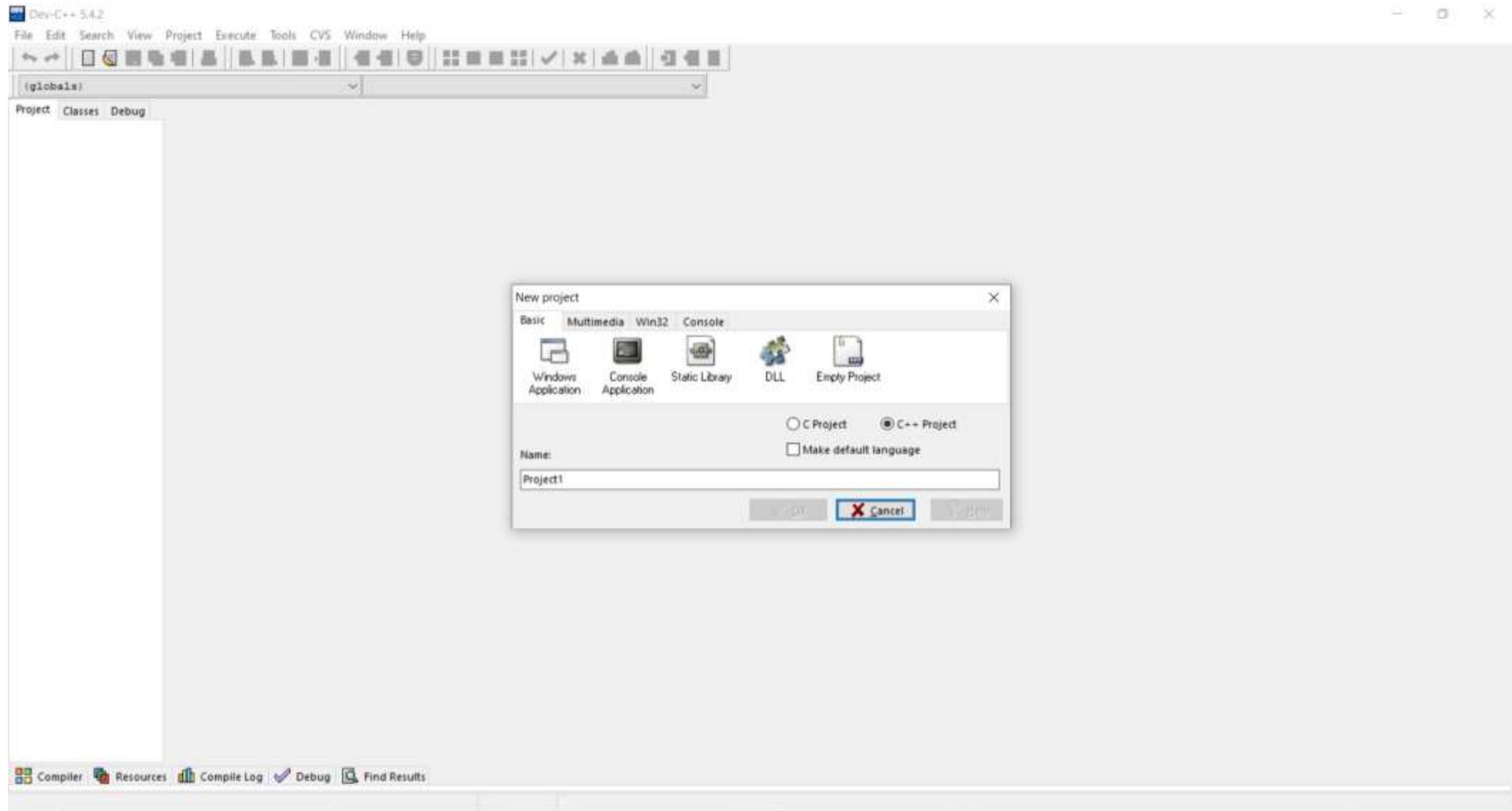
Основы C++

Dev-C++: интегрированная среда разработки приложений для языков программирования C/C++
(<https://bloodshed-dev-c.softonic.ru>)

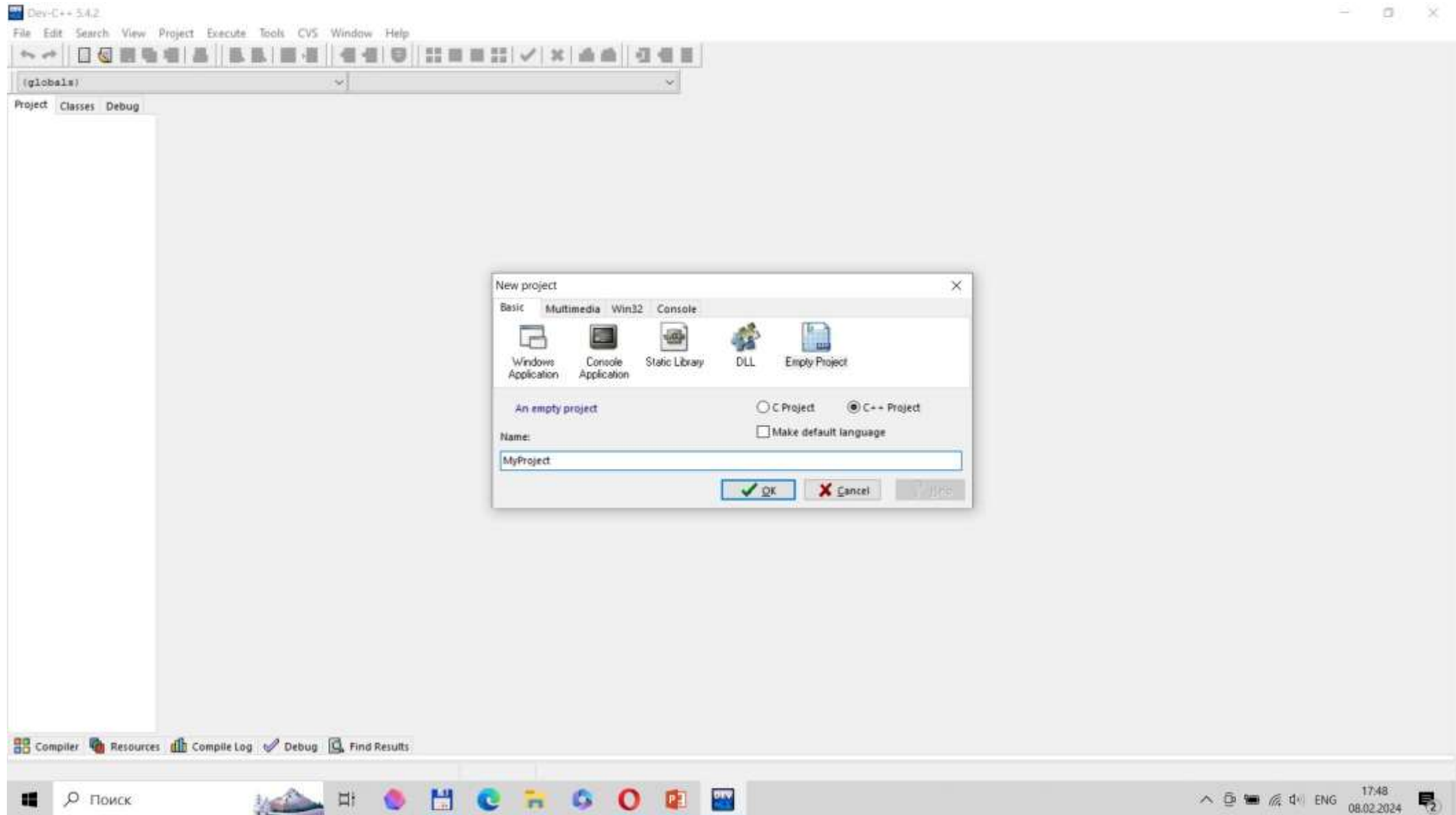
Создание проекта C++



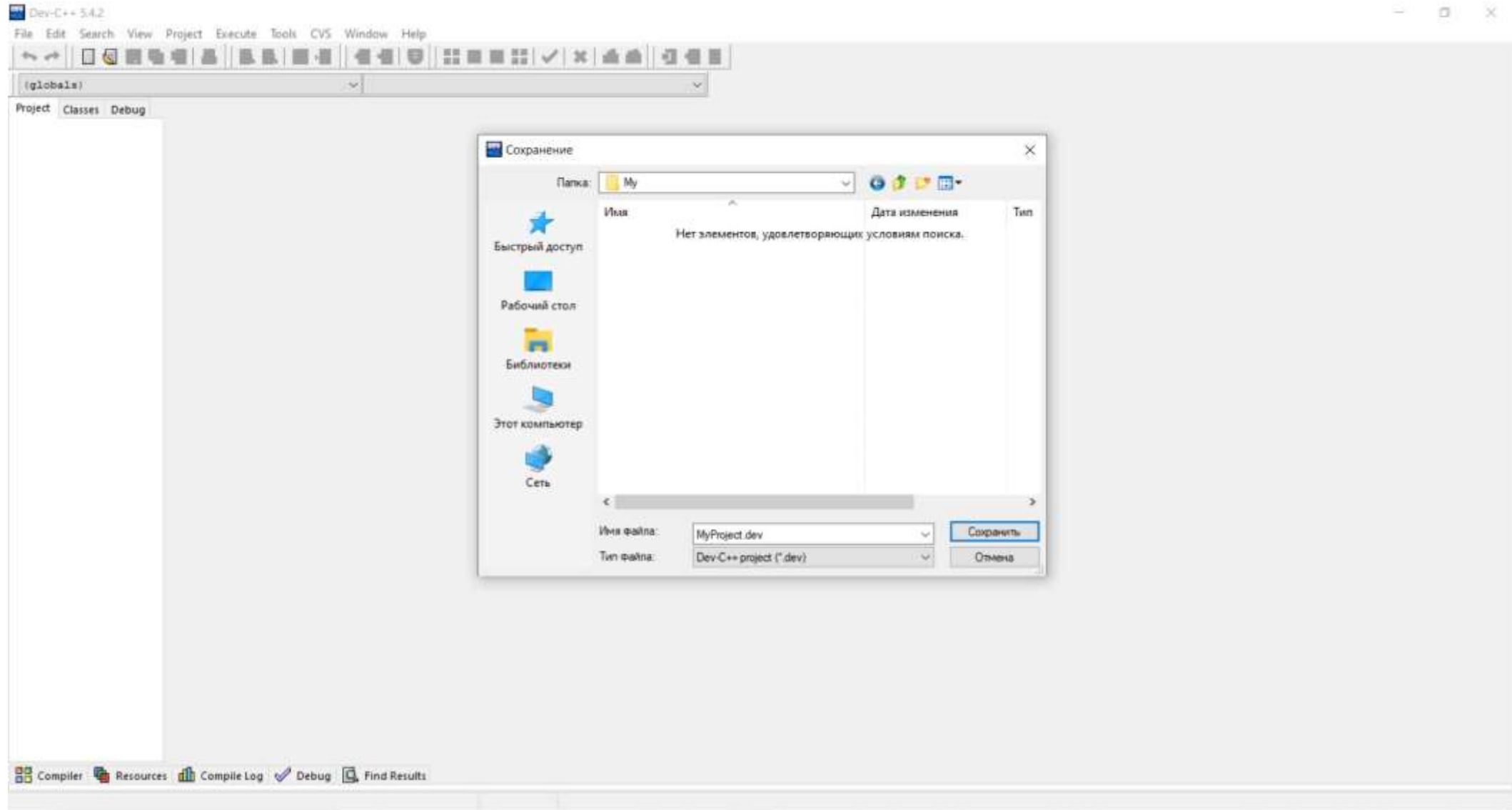
Выбрать 'Empty Project'



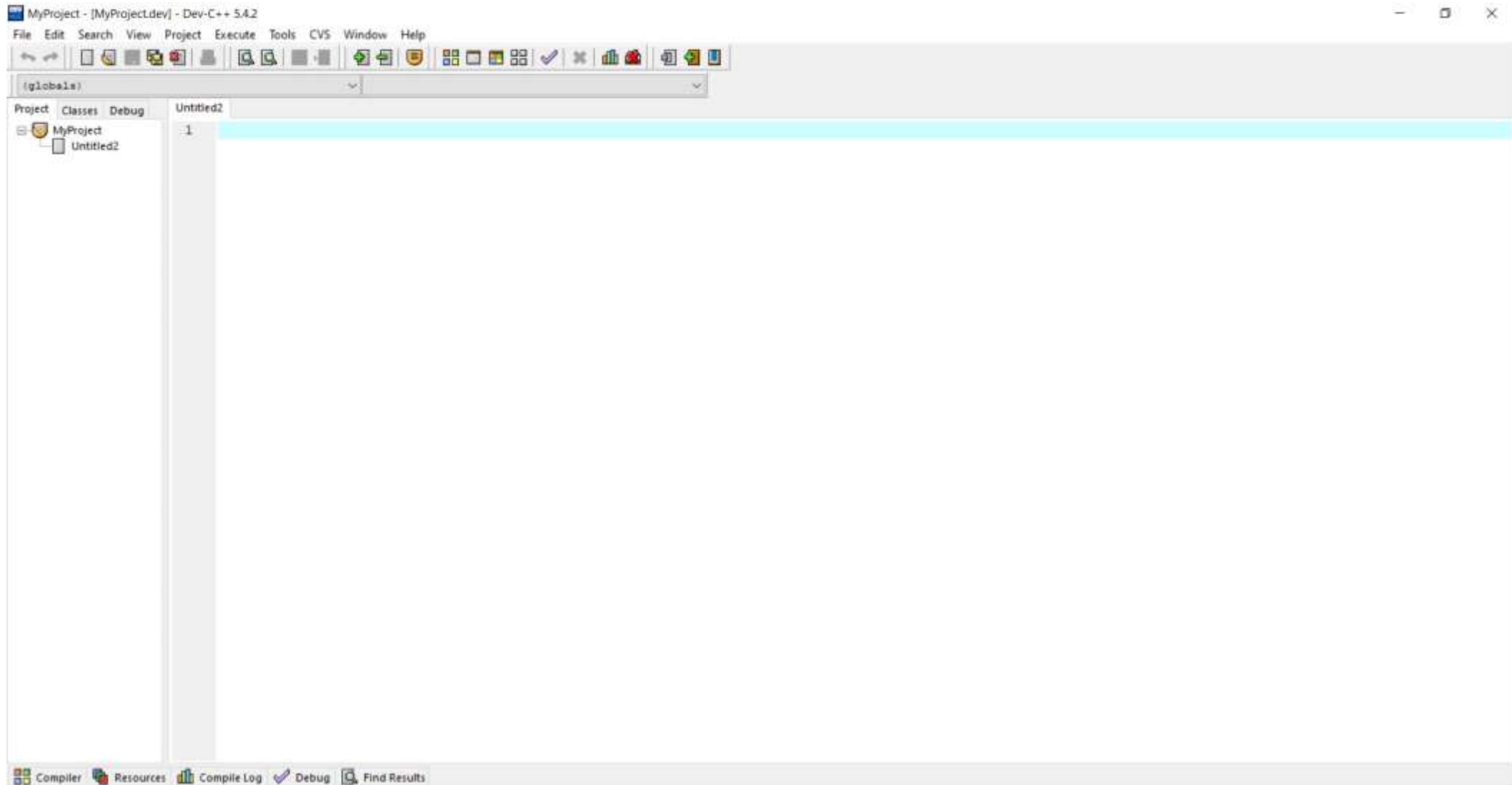
Ввести имя проекта (MyProject), выбрать Ok



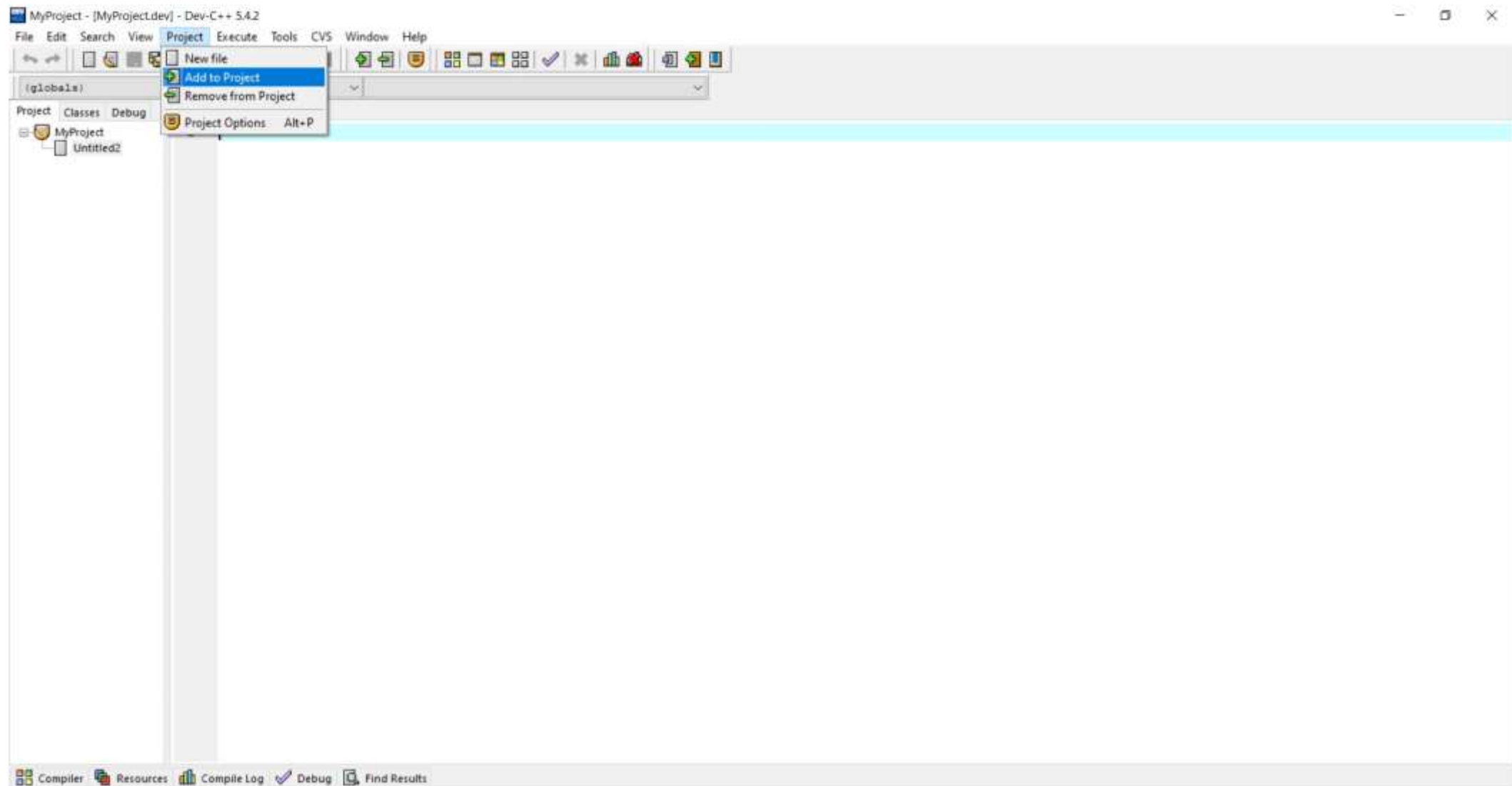
Выбрать директорию, где лежат файлы C++ для проекта и сохранить проект в этой директории



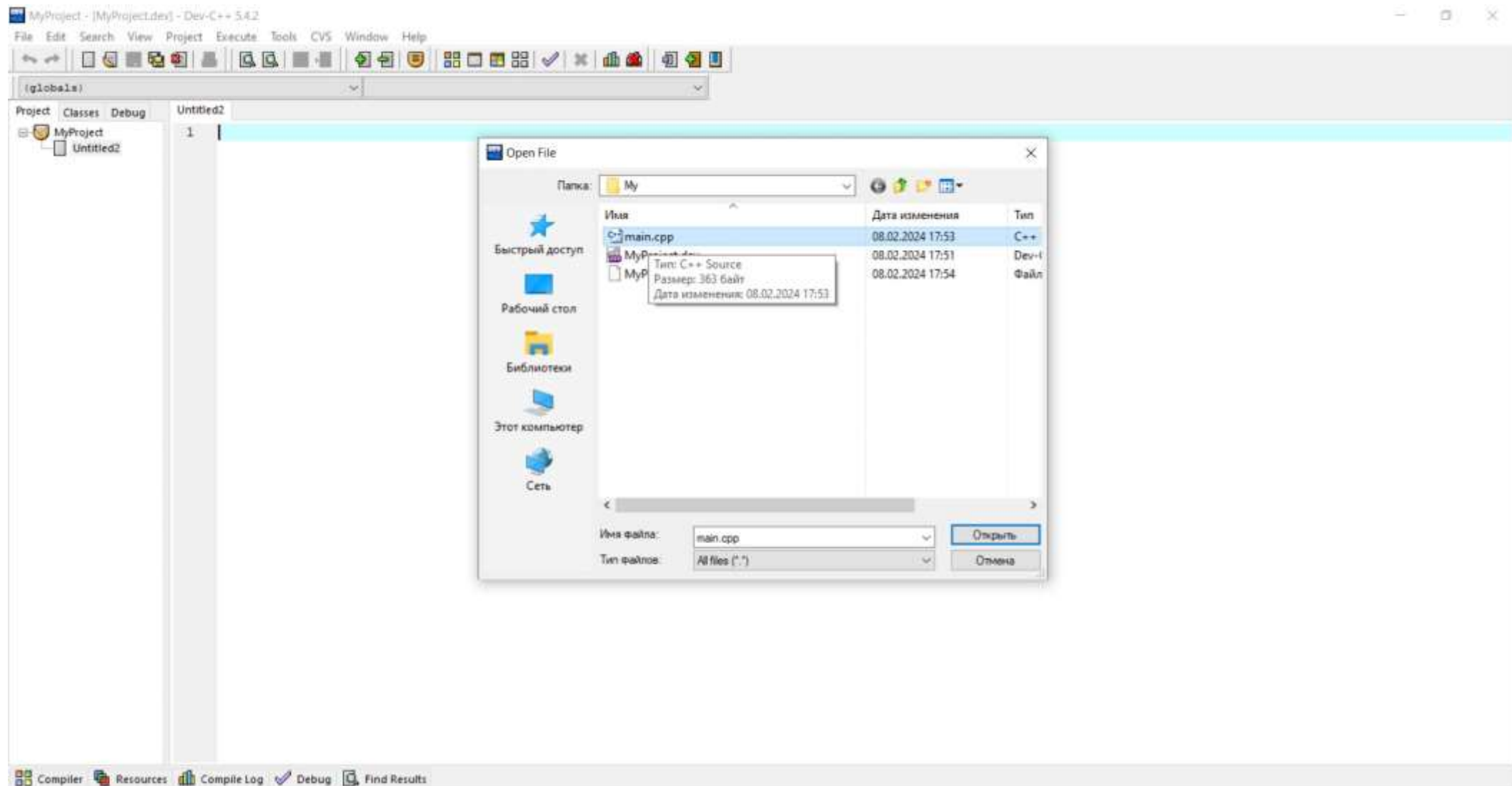
В созданном проекте находится пустой файл Untitled2



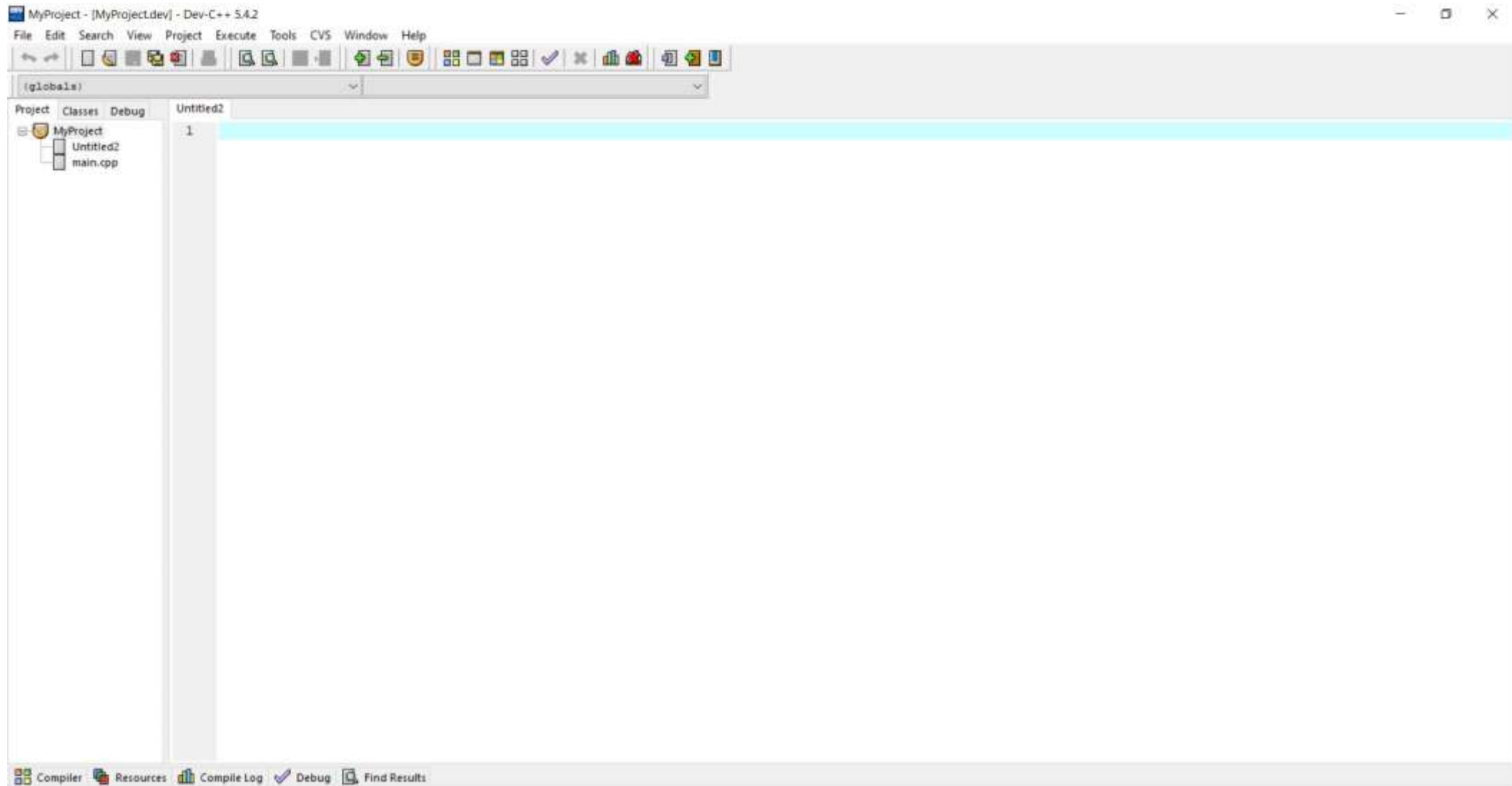
В меню 'Project', выбрать подменю 'Add to Project' для добавления в проект файлов



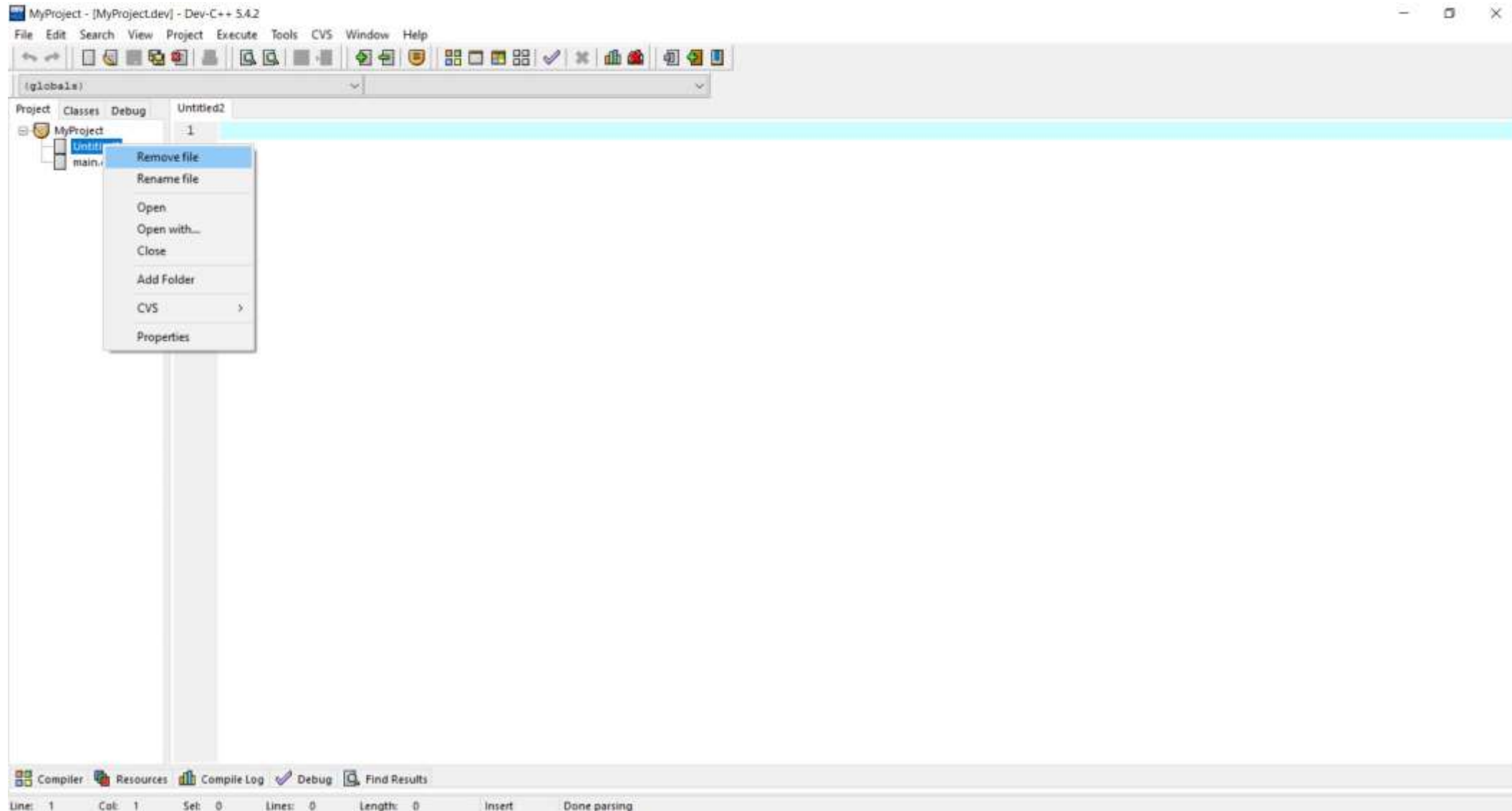
Выбрать файлы C++ (main.cpp) для добавления в проект, выбрать ‘Открыть’



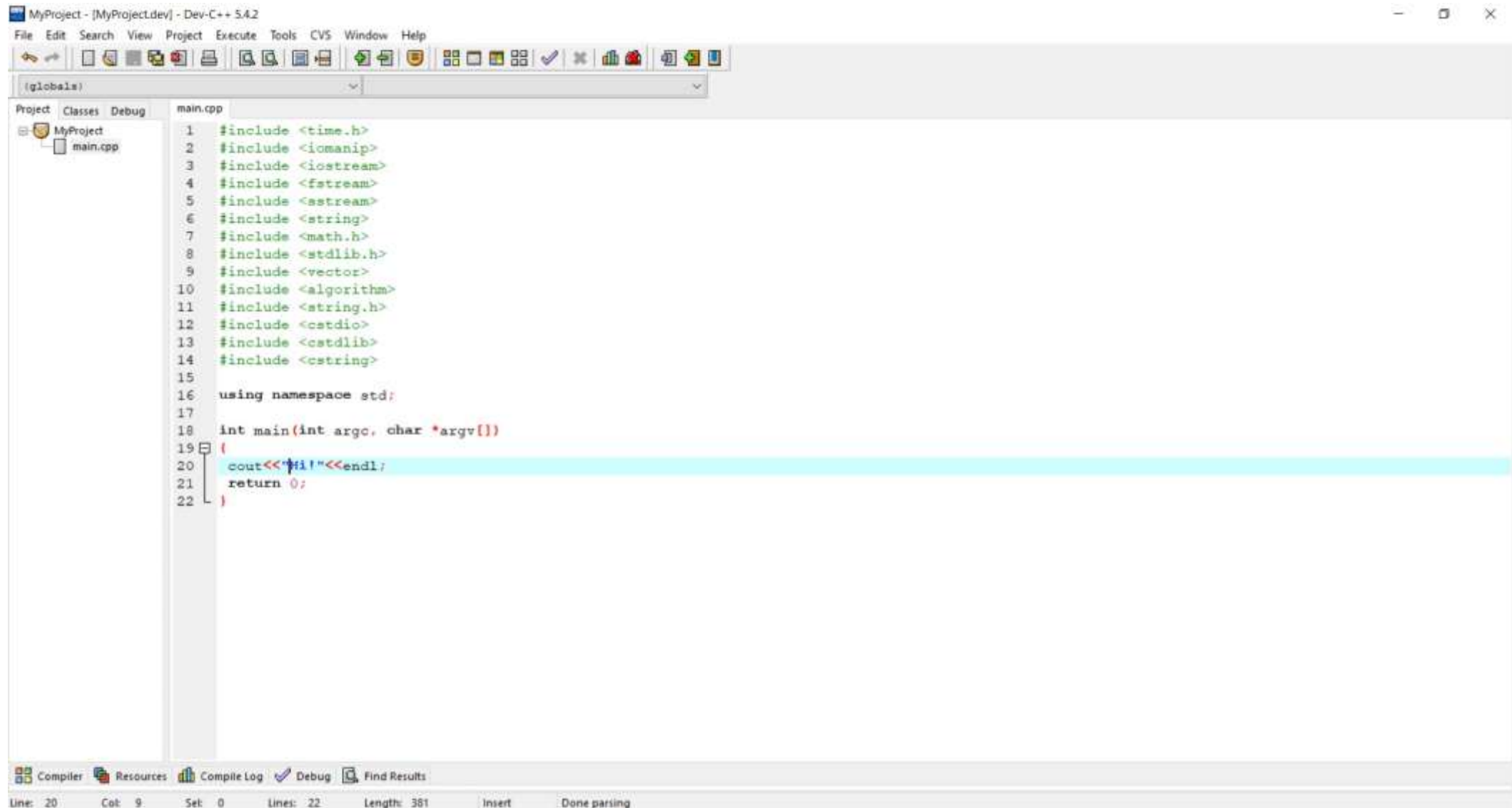
Файл main.cpp добавлен в проект



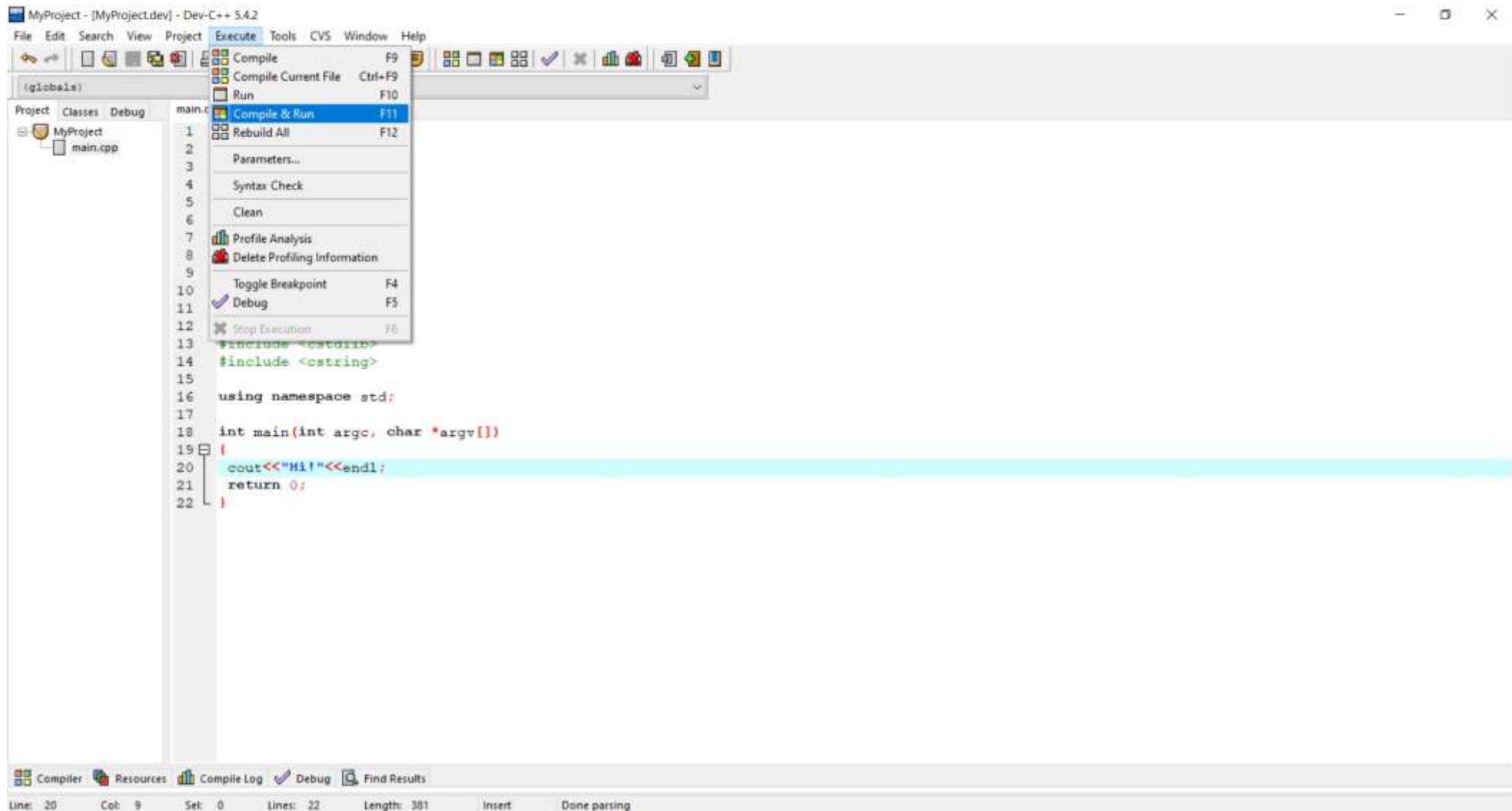
Правой кнопкой мыши выбрать файл Untitled для удаления, выбрать 'Remove file'



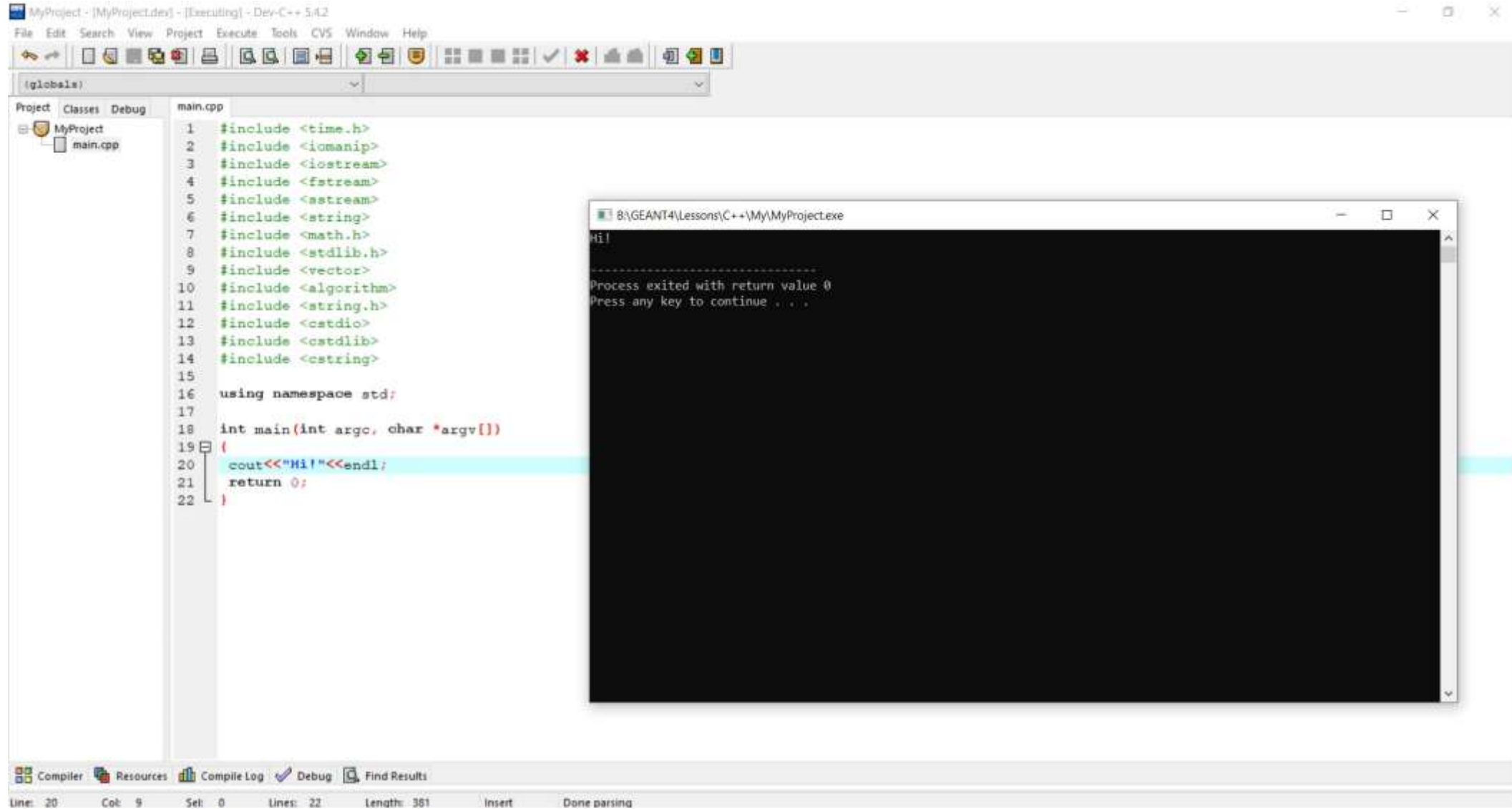
Левой кнопкой мыши выбрать файл main.cpp для просмотра содержимого



В меню 'Execute', выбрать подменю 'Compile & Run' для компиляции и запуска



Выполнение программы на C++



Копии переменных

Для начала давайте рассмотрим такой фрагмент кода:

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
5      std::string s1 = "Elementary, my dear Watson!";
6      std::string s2 = s1;
7
8      s1.clear(); // s2 никак не изменится
9
10     std::cout << s1 << "\n"; // пустая строка
11     std::cout << s2 << "\n"; // Elementary, my dear Watson!
12 }
```

Важно понимать, что здесь `s2` будет совершенно новой строкой, которая проинициализирована значением `s1`, но более никак с `s1` не связана. Это отличает C++ от некоторых других языков программирования — например, языка Python. В них после аналогичного присваивания строка осталась бы той же самой.

Создание новой строки `s2` требует ресурсов: нужно выделить новый блок памяти и скопировать туда старую строку.

Ссылки

Впрочем, в C++ есть возможность обращаться к уже существующему в памяти объекту под другим именем. Рассмотрим это на примере целых чисел:

```
1  #include <iostream>
2
3  int main() {
4      int x = 42;
5      int& ref = x;  // ссылка на x
6
7      ++x;
8      std::cout << ref << "\n";  // 43
9  }
```

Здесь `ref` — псевдоним для `x`. Это не самостоятельная переменная, а просто ссылка на объект, уже живущий в памяти. Формально типом `ref` является `int&` — ссылка на `int`.

Аналогично для строк:

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
5      std::string s1 = "Elementary, my dear Watson!";
6      std::string& s2 = s1;  // тут ссылка!
7
8      s1.clear();
9
10     std::cout << s2.size() << "\n";  // напечатает 0
11 }
```


Ссылка должна быть проинициализирована сразу в момент объявления. Например, так написать нельзя:

```
1 | int main() {  
2 |     int my_variable = 42;  
3 |     int& ref; // ошибка!  
4 |     // ...  
5 |     ref = my_variable;  
6 | }
```

Ссылка привязана к одному и тому же объекту со своего рождения. Переназначить её нельзя:

```
1 | int main() {  
2 |     int x = 42, y = 13;  
3 |     int& ref = x; // OK  
4 |     ref = y; // ссылка останется привязанной к x, значение x поменяется  
5 | }
```

Ссылки удобны там, где исходное имя слишком громоздко (например, является вложенным полем какой-либо структуры).

Указатели

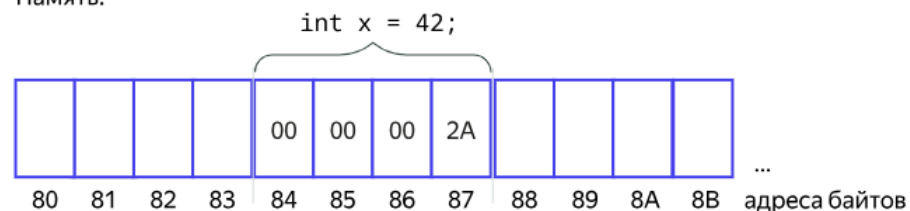
Другой (более базовый) способ сослаться на что-то уже существующее в памяти — указатели. Это специальные типы данных, которые могут хранить адрес какой-либо другой переменной в памяти. Здесь мы можем представлять себе память как длинную ленту с пронумерованными ячейками (байтами). Сам адрес переменной можно получить с помощью унарного оператора `&`:

```
1 | int main() {  
2 |     int x = 42;  
3 |     int* ptr = &x; // сохраняем адрес в памяти переменной x в указатель ptr  
4 |  
5 |     ++x; // увеличим x на единицу  
6 |     std::cout << *ptr << "\n"; // 43  
7 | }
```

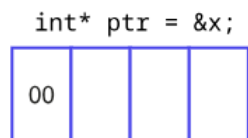
Пример вывода:

```
0x7ffdfce3188c  
0x7ffdfce31888  
0x7ffdfce31884
```

Память:



Указатель:



Формально указатель — это не номер ячейки памяти, а отдельный тип. Но обычно он может быть преобразован к целому числу. Вот такой код напечатает адреса переменных в шестнадцатеричном виде:

```
1 | #include <iostream>  
2 |  
3 | int main() {  
4 |     int x = 1, y = 2, z = 3;  
5 |     std::cout << &x << "\n";  
6 |     std::cout << &y << "\n";  
7 |     std::cout << &z << "\n";  
8 | }
```

Кроме адреса ячейки памяти переменная-указатель обладает ещё и типом данных, значение которого в этой ячейке лежит. Это позволяет компилятору правильно интерпретировать обращение к памяти по этому адресу. Поэтому мы используем не какой-либо абстрактный тип «указатель», а именно «указатель на `int`».

Оператор разыменования (унарная звёздочка) противоположен оператору взятия адреса (унарному амперсанду). Сравните: `&x` — это адрес `x` в памяти, а `*ptr` — это значение, живущее по адресу, записанному в `ptr`.

Указатели, в отличие от ссылок, можно переназначать. Кроме того, есть выделенное значение никуда не ссылающегося указателя — `nullptr` («нулевой» указатель):

```
1  #include <iostream>
2
3  int main() {
4      int x = 42, y = 13;
5      int* ptr; // по умолчанию не инициализируется, тут лежит «случайный» адрес
6      ptr = nullptr; // «нулевой» указатель
7      ptr = &x; // теперь в ptr лежит адрес переменной x
8      std::cout << *ptr << "\n"; // 42
9      ptr = &y; // можно поменять адрес, записанный в ptr
10     std::cout << *ptr << "\n"; // 13
11 }
```

Указатель `nullptr` нельзя разыменовывать: это приведёт к неопределённому поведению.

Отдельно рассмотрим указатели на структуру. Для обращения к полям структуры через указатель есть отдельный оператор `->` :

```
1  #include <iostream>
2
3  struct Point {
4      double x, y, z;
5  };
6
7  int main() {
8      Point p = {3.0, 4.0, 5.0};
9
10     Point* ptr = &p;
11
12     std::cout << (*ptr).x << "\n"; // обращение через * и . требует скобок
13     std::cout << ptr->x << "\n";  // то же самое, но чуть короче
14 }
```

Константность

Константа — это переменная, предназначенная только для чтения. Её значение должно быть зафиксировано в момент присваивания. При этом оно не обязательно должно быть известно в момент компиляции:

```
1  #include <iostream>
2
3  int main() {
4      const int c1 = 42;  // эта константа известна в compile time
5
6      int x;
7      std::cin >> x;
8      const int c2 = 2 * x;  // значение становится известным только в runtime
9
10     c2 = 0;  // ошибка компиляции: константе нельзя присвоить новое значение
11 }
```

У константного вектора или строки нельзя будет вызвать функции, которые их будут изменять:

```
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5      const std::vector<int> v = {1, 3, 5};
6      std::cout << v.size() << "\n";  // OK, напечатает 3
7      v.clear();  // ошибка компиляции: константный вектор нельзя изменять
8      v[0] = 0;  // тоже ошибка компиляции
9  }
```

Ссылки и указатели можно комбинировать с константностью:

```
1  int main() {
2      int x = 42;
3
4      int& ref = x; // обычная ссылка
5      const int& cref = x; // константная ссылка
6      ++x; // OK
7      ++ref; // OK
8      ++cref; // ошибка компиляции: псевдоним cref предназначен только для чтения
9
10     int* ptr = &x; // обычный указатель
11     const int* cptr = &x; // указатель на константу
12     ++*ptr; // OK
13     ++*cptr; // ошибка компиляции: разыменованный cptr – константа!
14 }
```

Если исходная переменная уже была константной, то взять обычную ссылку или указатель на неё не получится. Другими словами, константность нельзя просто так отменить, её можно только добавить:

```
1  int main() {
2      const int cx = 42;
3
4      int& ref = cx; // ошибка компиляции: константность нельзя убрать
5      const int& cref = cx; // OK
6
7      int* ptr = &cx; // тоже ошибка компиляции
8      const int* cptr = &cx; // OK
9  }
```


Базовый тип и слово `const` можно менять местами. Так что `const T` и `T const` — это одно и то же. Но следует различать указатель на константу (`const T*`) и константу типа «указатель» (`T* const`):

```
1  int main() {
2      int x = 42;
3      const int cx = 13;
4
5      int* ptr = &x;  // обычный указатель
6      ptr = &cx;  // ошибка компиляции
7
8      const int* cptr = &x;  // ОК: через *cptr нельзя будет изменить x
9      cptr = &cx;  // ОК
10
11     int* const ptrc = &x;  // ОК: *ptrc можно менять, но сам ptrc менять нельзя
12     ptrc = nullptr;  // ошибка компиляции
13
14     const int* const cptrc = &x;  // ОК, для &cx тоже бы сработало
15 }
```

Пример в последней строке похож на константную ссылку: указатель `cptrc` не позволяет менять содержимое ячейки `&x` (первый `const`) и в него нельзя записать адрес другой переменной (второй `const`).

Добавление константности

«Висячие» ссылки и указатели

Может так оказаться, что переменная, на адрес которой ссылается указатель, уже вышла из своей области видимости. Похожая ситуация может произойти и со ссылками. В таком случае обращаться к памяти через ссылку или указатель нельзя — это приведёт к неопределённому поведению.

```
1  #include <iostream>
2
3  int main() {
4      int* ptr = nullptr;
5
6      {
7          int x = 42;
8          ptr = &x;
9      }
10
11     // обращаться к памяти, в которой жила переменная x, уже нельзя:
12     std::cout << *ptr << "\n"; // неопределённое поведение!
13 }
```

Аналогичная ситуация произойдёт при обращении к уже несуществующему элементу вектора:

```
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5      std::vector<std::string> words = {"one", "two", "three"};
6
7      std::string& ref = words[0]; // псевдоним для начального элемента вектора
8
9      words.clear();
10
11     // обращаться к ссылке ref уже нельзя!
12     std::cout << ref << "\n"; // неопределённое поведение!
13 }
```

Передача параметров в функцию

Аргументы, которые представляют переменные или константы, могут передаваться в функцию **по значению** (by value) и **по ссылке** (by reference).

Передача аргументов по значению

При передаче аргументов **по значению** функция получает копию значения переменных и констант. Например:

```
1 #include <iostream>
2
3 void square(int);    // прототип функции
4
5 int main()
6 {
7     int n {4};
8     std::cout << "Before square: n = " << n << std::endl;
9     square(n);
10    std::cout << "After square: n = " << n << std::endl;
11 }
12
13 void square(int m)
14 {
15     m = m * m;    // изменяем значение параметра
16     std::cout << "In square: m = " << m << std::endl;
17 }
```

Вывод на экран

```
Before square: n = 4
In square: m = 16
After square: n = 4
```

Функция `square` принимает число типа `int` и возводит его в квадрат. В функции `main` перед и после выполнения функции `square` происходит вывод на консоль значения переменной `n`, которая передается в `square` в качестве аргумента.

И при выполнении мы увидим, что изменение параметра `m` в функции `square` действуют только в рамках этой функции. Значение переменной `n`, которое передается в функцию, никак не изменяется:

Передача аргументов по ссылке

При передаче параметров **по ссылке** передается ссылка на объект, через которую мы можем манипулировать самим объектом, а не просто его значением. Так, перепишем предыдущий пример, используя передачу по ссылке:

При передаче параметров **по ссылке** передается ссылка на объект, через которую мы можем манипулировать самим объектом, а не просто его значением. Так, перепишем предыдущий пример, используя передачу по ссылке:

```
1 #include <iostream>
2
3 void square(int&);    // прототип функции
4
5 int main()
6 {
7     int n {4};
8     std::cout << "Before square: n = " << n << std::endl;
9     square(n);
10    std::cout << "After square: n = " << n << std::endl;
11 }
12 void square(int& m)
13 {
14     m = m * m;    // изменяем значение параметра
15     std::cout << "In square: m = " << m << std::endl;
16 }
```

Вывод на экран

```
Before square: n = 4
In square: m = 16
After square: n = 16
```

Теперь параметр `m` передается **по ссылке**. Ссылочный параметр связывается непосредственно с объектом, поэтому через ссылку можно менять сам объект. То есть здесь при вызове функции параметр `m` в функции `square` будет представлять тот же объект, что и переменная `n`

Передача по ссылке позволяет вернуть из функции сразу несколько значений. Также передача параметров по ссылке является более эффективной при передаче очень больших объектов. Поскольку в этом случае не происходит копирования значений, а функция использует сам объект, а не его значение.

От передачи аргументов по ссылке следует отличать передачу ссылок в качестве аргументов:

```
1 #include <iostream>
2
3 void square(int);    // прототип функции
4
5 int main()
6 {
7     int n = 4;
8     int &nRef = n;    // ссылка на переменную n
9     std::cout << "Before square: n = " << n << std::endl;
10    square(nRef);
11    std::cout << "After square: n = " << n << std::endl;
12 }
13 void square(int m)
14 {
15     m = m * m;    // изменяем значение параметра
16     std::cout << "In square: m = " << m << std::endl;
17 }
```

Вывод на экран

```
Before square: n = 4
In square: m = 16
After square: n = 4
```

Если функция принимает аргументы по значению, то изменение параметров внутри функции также никак не скажется на внешних объектах, даже если при вызове функции в нее передаются ссылки на объекты.

Указатели в параметрах функции

Параметры функции в C++ могут представлять указатели. Указатели передаются в функцию по значению, то есть функция получает копию указателя. В то же время копия указателя будет в качестве значения иметь тот же адрес, что оригинальный указатель. Поэтому используя в качестве параметров указатели, мы можем получить доступ к значению аргумента и изменить его.

```
1 #include <iostream>
2
3 void increment(int*);
4
5 int main()
6 {
7     int n {10};
8     increment(&n);
9     std::cout << "main function: " << n << std::endl;
10 }
11 void increment(int *x)
12 {
13     (*x)++; // получаем значение по адресу в x и увеличиваем его на 1
14     std::cout << "increment function: " << *x << std::endl;
15 }
```

Вывод на экран

```
increment function: 11
main function: 11
```

Для изменения значения параметра применяется операция разыменования с последующим инкрементом: `(*x)++`. Это изменяет значение, которое находится по адресу, хранимому в указателе `x`.

Поскольку теперь функция в качестве параметра принимает указатель, то при ее вызове необходимо передать адрес переменной: `increment(&n);`.

В итоге изменение параметра `x` также повлияет на переменную `n`, потому что оба они хранят адрес на один и тот же участок памяти:

В то же время поскольку аргумент передается в функцию по значению, то есть функция получает копию адреса, то если внутри функции будет изменен адрес указателя, то это не затронет внешний указатель, который передается в качестве аргумента:

```
1 #include <iostream>
2
3 void increment(int*);
4
5 int main()
6 {
7     int n {10};
8     int *ptr {&n};
9     increment(ptr);
10    std::cout << "main function: " << *ptr << std::endl;
11 }
12 void increment(int *x)
13 {
14     int z {6};
15     x = &z;    // переустанавливаем адрес указателя x
16     std::cout << "increment function: " << *x << std::endl;
17 }
```

Вывод на экран

```
increment function: 6
main function: 10
```

В функцию `increment` передается указатель `ptr`, который хранит адрес переменной `n`. При вызове функция `increment` получает копию этого указателя через параметр `x`. В функции изменяется адрес указателя `x` на адрес переменной `z`. Но это никак не затронет указатель `ptr`, так как он представляет другую копию. В итоге поле переустановки адреса указателя `x` и `ptr` будут хранить разные адреса.