

**Материалы для конкурса методического мастерства.
Четыре электронные лекции «Вхождение в программирование на
Ассемблере»**

Организация – Финансовый Университет при правительстве
Российской Федерации, Колледж информатики и программирования,
г.Москва

Автор – преподаватель Сибирев Иван Валерьевич

Лекции предназначены для обучающихся Колледжа информатики и программирования

– специальности 10.02.05 Обеспечение информационной безопасности автоматизированных систем, относятся к профессиональному модулю ПМ.02 "Защита информации в автоматизированных системах программными и программно-аппаратными средствами", междисциплинарному курсу МДК.02.03 "Машинно-ориентированное программирование для решения задач защиты информации".

Форма проведения занятий: проблемные лекции.

Форма представления методического материала на конкурс:
электронные лекции.

Тема лекций. Вхождение в программирование на Ассемблере

Цели лекций

Образовательные цели: ознакомить учащихся с языком машинно-ориентированного программирования; выработать у учащихся умение применять Ассемблер на практике для критически важных узких мест программных кодов, развитие профессиональных навыков программиста.

Развивающие цели: обеспечить условия для развития логики, алгоритмического машинно-ориентированного мышления; для формирования умения производить поиск информации, необходимой программисту для профессиональной работы, анализировать ее, наблюдать, сравнивать, применять эту информацию для решения профессиональных задач, в том числе в вопросах защиты информации и оптимизации программных кодов.

Воспитательные цели: обеспечить условия для профориентации обучающихся, их вхождения в профессию программиста, воспитания дисциплинированности, аккуратности, добросовестности и для формирования коммуникативной культуры и бережного отношения к своему и чужому программному коду.

Прикладные цели:

Согласно МДК.03.01 Техническая защита информации, специальность: 10.02.05 Обеспечение информационной безопасности автоматизированных систем. Преследуется цель сформировать умения: ” применять технические средства для криптографической защиты информации конфиденциального

характера;”. Вопрос актуален по тому что для применения криптографических средств требуются достаточные аппаратные мощности и высокая вычислительная производительность. Что выливается в оптимизацию производительности программных кодов на этапах разработки и поддержки. Именно машинное ориентированное программирование позволяет “приблизить к железу” реализацию криптографических алгоритмов, что позволяет вести оптимизацию с учетом специфики специализированных аппаратных криптографических чипов. Как не странно, подобные вопросы вполне решаемы в рамках обычных программных кодов современных универсальных микропроцессоров. Для приобретения подобного программистского умения просто необходимо уметь писать код на ассемблере. Даже если знаний ассемблера недостаточно, обладание ими все равно приводит к лучшему эффекту оптимизации программных кодов на языках высокого уровня.

Задачи

- познакомить учащихся с введением в машинно-ориентированное программирование на языках высокого уровня C++;
- познакомить учащихся с введением в машинно-ориентированное программирование на примере группы языков аппаратного уровня, BF, LamPanel, FlatAssembler;
- разобраться с шаблонами и логикой написания программ управления оперативной памятью на вершине стека вызова функций на языке BF на примере задачи вывода текста в консоль (подходы к решению вопроса у каждого ученика могут в корне отличаться);
- познакомиться с командами работы с внешними портами на примере программы LamPanel в игровом задании нарисовать рисунок в лампочках портов и сделать для него анимацию;
- отработать написание программ на основе указателей, массивов и команд пересылки данных в учебных заданиях на языке C++ с использованием дизассемблера;
- отработать написание программ на основе указателей, массивов, функций и команд пересылки данных на учебных заданиях на языке FlatAssembler под DosBox и Windows10x64.

Образовательные результаты

Личностные: развитие интереса к программированию, к решению трудных программистских задач; осознание важности изучения данной темы для расширения инструментария и возможностей программиста, для развития вариативности мышления; готовность к общению и сотрудничеству с преподавателем и сверстниками, уважительное отношение к ответу одноклассников и их мнению.

Метапредметные: умение производить поиск и выбор материала на заданную тему, умение критически его переосмысливать; выбирать способы достижения результата, оценивать правильность выполнения учебной задачи; умение применять полученные знания при

решении задач; умение четко формулировать собственное мнение, слушать и понимать товарищей и преподавателя.

Предметные: знание основных принципов и команд машинно-ориентированного программирования, умение оперировать понятиями регистров микропроцессора, условных и безусловных переходов, арифметических команд, указателей, процедур, функций, макросов и *.bat файлами; умение применять их на практике при программировании на Ассемблере.

Совершенствуются следующие универсальные учебные действия: критическое мышление; технологическая и информационная грамотность; навыки сотрудничества.

Краткая характеристика методических материалов

Актуальность содержания материала лекций. Ассемблер – машинно-ориентированный язык программирования, предназначенный для управления битами в регистрах и оперативной памяти. Освоение Ассемблера позволит обучающимся:

- понимать причины и следствия поведения программных кодов как высокого, так и машинного уровня;
- познакомиться с понятием стековой машины и с принципами организации исполняемых кодов компьютерных программ;
- получить инструмент оптимизации программных и аппаратных кодов;
- отлаживать или защищать элементы компьютерных систем на самом низком уровне, в том числе, за счет ассемблерных программных кодов.

В узкоспециализированных аппаратных задачах и задачах защиты информации Ассемблер не имеет себе равных. Языки программирования можно разделить на профессиональные и любительские по принципу возможности написания ассемблерных вставок. Именно они и определяют качество и глубину оптимизации написанных программных кодов.

Это свидетельствует об актуальности программирования на Ассемблере. Именно он является «первозыком» для всего остального программистского мира.

Возникновение Ассемблера датируется 1947 годом, язык содержит много диалектов, которые являются частными случаями реализации идеи машинно-ориентированного языка. Примеры из литературы часто не работают или требуют адаптации под современные реалии. В Интернет лишь каждый 10-20 раз будут попадаться примеры работающих программных кодов. Периодически встречаются посты с заведомо ложной информацией, которую новичку не под силу распознать. Программные коды, взятые из

литературы, обладают высокой скоростью роста сложности и поэтому не пригодны для начинающих. Все это создает высокий входной порог сложности материала данной темы и множество проблем для начинающего программиста.

Проблемное изложение. При чтении лекций традиционным является повествовательно-лекционный стиль изложения, при котором обучающимся результаты программистского поиска, а также способы преодоления ряда программистских проблем и решения задач предлагаются как устоявшиеся догмы. Большая часть обучающихся, привыкнув принимать предлагаемый материал как постулаты, не пытаются самостоятельно мыслить, творчески подходить к решению программистских задач.

Ассемблер же требует противоположного – безудержного желания и возможности изобрести заново весь материал, пройденный в рамках основ алгоритмизации. Невозможность сделать это по неработающим примерам из Интернета и запыленной бумажной литературе демотивирует учащегося.

Нами предлагаются учебные задачи и примеры работающих программных кодов, на которых можно себя почувствовать первооткрывателем программистского мира, как это было в эпоху изобретения и разработки первых языков программирования. Все задачи решаемы, все примеры работают, что мотивирует к обучению.

Данный материал относится к **проблемным лекциям**. Исторический путь развития программирования представляет дидактическую ценность. Этот путь хорошо можно проиллюстрировать на примере развития языков ассемблерной группы. **Проблемы развития данного направления**, а также **проблемы, возникающие у начинающего программиста** при вхождении в Ассемблер, предлагаемые обучающимся в проблемной постановке, будут способствовать развитию мышления, логики, позволят почувствовать радость открытий.

В стиле изложения, подразумевающим диалог с аудиторией, изложены проблемы вхождения в Ассемблер. В диалоге намечаются пути решения этих проблем, названы книги и Интернет-источники, которые могут служить «дорожной картой» для самостоятельного решения названных проблем учащимися. Автор лекций ведет спор с авторами некоторых Интернет-источников и с аудиторией, провоцируя слушателей к диалогу.

Оригинальные методические решения при освоении Ассемблера воплощены в **авторские программистские задания**, выстроенные систематически, с последовательным введением нового программистского инструментария и нарастанием сложности. Они приближают слушателя шаг за шагом к программированию на Ассемблере. Задачи дифференцированы по уровню сложности (вплоть до олимпиадных) и сопровождаются комментариями.

Оформление электронных лекций произведено с использованием современного программного инструментария: в «Jupiter Notebook» (2015 г.), в качестве ядра используется язык Julia (2018 г. появления). Этот язык предназначен для постановки и проведения вычислительных экспериментов,

заточен под высокую производительность, имеет возможность для написания ассемблерных вставок. Julia – это язык высокого уровня, являющийся при этом «ассемблероблизким» языком. В дальнейшем это даст возможность перейти к изучению описанного технологического стека. Выбранный инструментарий позволяет запускать программные коды «прямо из лекций».

План лекций

Лекция 1.

1. Базовые инструменты командной разработки.
2. История появления системы контроля версий Git.
3. Возможности Git. Ускорение вхождения в разработку с использованием пользовательского графического интерфейса Git.
4. GitHub, как сервис для хранения и передачи Git репозитория.

Лекция 2.

1. Актуальность изучения Ассемблера. История языков ассемблерной группы. Проблемы вхождения в тему.
2. Эзотерические языки программирования. BrainF – младший брат Ассемблера (всего 8 команд, возможность работы со стеком вызова функции, прост в схемотехнической реализации, неожиданно упрощает понимание Ассемблера).
3. Освоение команд BrainF. Решение задач.

Лекция 3.

1. Обзор актуальной литературы, сайтов на тему «Программирование на Ассемблере».
2. Начинаем писать. Вывод на экран состояния регистров. LamPanel.
3. Рисунки с использованием LamPanel, анимация.

Лекция 4.

1. Команды пересылки данных в Ассемблере.
2. Ассемблерные команды.
3. Работа с массивами.

Лекция 5.

1. Изучение работы со строками. Изучение циклов.
2. Создание переменных и вывод на экран.
3. Механика условных переходов.
4. Создание циклов.
5. Сбор воедино моноблока ассемблерных кодов.

Методические указания для обучающихся по работе с лекционным материалом

Для успешного вхождения обучающихся в программирование на Ассемблере необходимо посещение лекций, ознакомление с основной и дополнительной литературой, предлагаемой в процессе лекций, поиск

актуальных Интернет-источников, разбор и анализ предлагаемых там программ, активная работа на лабораторных занятиях, выполнение учебных заданий, а также практика написания программных кодов.

Запись лекции – одна из форм активной самостоятельной работы обучающихся, требующая навыков и умения кратко, схематично, последовательно и логично фиксировать основные положения, выводы, обобщения, формулировки. В конце лекции преподаватель оставляет время (5 минут) для того, чтобы обучающиеся имели возможность задать уточняющие вопросы по изучаемому материалу.

Лекции прочитаны в очной форме, но учащимся предоставляется также электронный вариант лекций для дальнейшей работы с этим материалом. Для лучшего освоения материала и систематизации знаний по дисциплине, необходимо самостоятельно разбирать материалы лекций, т.е. прочитанная лекция – это материал для дальнейшей работы. В случае необходимости обращаться к преподавателю за консультацией.

Преподаватель некоторые вопросы выносит на самостоятельную работу студентов, рекомендуя ту или иную литературу. Лекции снабжены актуальными и современными ссылками на Интернет-источники. В лекциях производится анализ литературы и Интернет-источников, предлагается «дорожная карта», литература и задания для углубленного изучения предмета. В целях дифференциации обучения предложены различные траектории изучения темы: от поверхностного ознакомления с принципами работы Ассемблера и умения писать простейшие программы – до профессионального владения этим языком в будущем.

Электронные лекции снабжены работающим программным кодом, что делает работу с этим материалом – интерактивной. Наблюдая за работой программ, можно глубже понять принципы, на которых она построена, что улучшит освоение материала. Кроме того, работающий код можно использовать для решения других поставленных задач.

В тексте лекций приведено множество задач. Некоторые из них решены здесь же, как иллюстрирующий пример, в режиме диалога с аудиторией. Приведены основные теоретические положения, необходимые для решения, программный код и комментарии к решению задач. Здесь же предложены аналогичные задачи для самостоятельного решения (среди которых есть олимпиадные). Задачи дифференцированы по уровню сложности и сопровождаются комментариями.

Параллельно с лекциями в курсе предусмотрены лабораторные работы. В лекциях к краткой форме приводятся задания к лабораторным работам и комментарии к их выполнению. Некоторые задания рекомендуется выполнять парами в целях развития навыков профессиональной коммуникации.

Апробация методических материалов произведена при ведении курса лекций в 2019-20 и 2020-21 уч. г. в группах:
ЗОИБАС-618, ЗОИБАС-718, ЗОИБАС-818, 2ИСИП-118, 2ИСИП-218

ЗОИБАС-517,ЗОИБАС-617, ЗОИБАС-717,ЗОИБАС-1018.

Намечены перспективы для дальнейшей работы – написания электронного учебного пособия.

Материалы и программы есть, осталось описать:

- Вызов call и 3-4 способа передачи параметров внутрь функций.
- Макросы как средство повышения быстродействия (полный курс макросов).
- Стек вызова функций и размещение всех локальных переменных в нем.
- Компьютерная графика под DosBox.
- Подключение и вызов системных библиотек C++ Windows.
- Написание тестового стенда для изучения работы сдвигов под FASM Windows.
- Ловим ошибки разработчиков FASM.
- Компьютерная графика под WinOpenGL.

Ближайшие перспективы:

- Ассемблерные команды MMX - аппаратное сложение массивов, используется обычно в криптографии и компьютерной графике.
- Работа с fword 48 бит. Аппаратная работа с дробными числами.
- Перебор материала Мануал программера.flat assembler 1.71. Выясняем какие аппаратные процедуры работают под макроассемблер Windows FASM.

Практическая работа «Знакомство с Git/GitHub»

Время изучения темы – от 2 до 6 часов, из них 2 часа- лекция

Учебные цели занятия: знакомство с базовыми инструментами командной разработки в Git/GitHub.

Форма проведения занятия: практическая работа.

Учебные задания: зарегистрироваться на GitHub, создать репозиторий, создать файл отчета, проиндексировать, сделать коммит, залить изменения в Интернет на GitHub, написать отчет, залить изменения на GitHub.

Инструкции по проведению и ходу занятия.

В начале занятия обосновываются цель и задачи практического занятия, определяется проблема. Преподаватель излагает материал, затем следит за исполнением работы студентами. Допустима дискуссия о поставленной проблеме, ограниченная рамками технических вопросов.

Методические рекомендации:

- студент должен найти в методических указаниях инструкцию по выполнению указанных действий, выполнить их;
- изучить теоретический материал по теме занятия;
- после выполнения поставленной технической и теоретической задачи допустима дискуссия на тему история развития Git или выдача творческого задания повышенной сложности.

Вопросы для самоконтроля

- Записан ли в тетради мой логин/пароль?
- Создан ли мой репозиторий на GitHub?
- Выложен ли в репозиторий на GitHub результат практической работы?

Определения для самоконтроля:

репозиторий, система контроля версий, индексирование, коммит, push, pop, центральный репозиторий, персональный репозиторий, ветвление в Git, слияние ветвей.

Задание на лабораторную работу

Зарегистрироваться на GitHub, подтвердить аккаунт, создать репозиторий, клонировать репозиторий к себе на компьютер в “C:\\D”. По умолчанию, будем всегда работать из этой папки.

В полученной папке на компьютере создать файл Word. Написать в нем отчет со скриншотами. Выложить отчет на GitHub (проиндексировать, создать коммит, сделать push). Описать это в отчете, повторно выложить.

Сдать результаты работы преподавателю.

Временные затраты – от 30 минут до часа.

Творческое задание повышенной сложности

Выполнять только после основного задания. Предлагается онлайн JavaScript приложение GitLearn (https://learngitbranching.js.org/?locale=ru_RU). Там достаточно уроков и онлайн упражнений, чтобы затем чувствовать себя в консоли комфортно. После прохождения задания, оно остаётся отмеченным, как пройденное в текущем браузере до очистки Cookie файлов браузера. Преподавателю можно показать список пройденных упражнений и конспект материала (он обязательный).

Можно написать несколько *.bat файлов (это будет только в плюс).

Временные затраты не более 6 часов.

Оглавление

- Теоретический материал
 - Базовые инструменты командной разработки.
 - Как наладить работу GitExtensions-Portable.
 - Как создать репозиторий.
 - Как сделать Коммит.
 - Ветвление и клонирование.
 - Объединение веток, слияние.

- Разрешение конфликтов слияния.
- Выбор центрального хранилища, GitHub.
- А теперь попробуем при помощи GitExtensions-Portable получить репозиторий с GitHub.
- Задание на лабораторную работу.
- Творческое задание повышенной сложности.
- Разбор ошибок.

Знакомство с Git/GitHub

Проблема в том, что при трудоустройстве про Git не спрашивают, считается, что Вы его уже знаете.

Базовые инструменты командной разработки

История происхождения Git: (<https://techrocks.ru/2019/02/19/git-origin-story/>).

- **CVS** - с 1980 года, одна из первых систем контроля версий, имеющая по сравнению с сегодняшними огромное количество недостатков.
- **BitKeeper/BitMover** - примерно 2000-ные годы, распределенная система контроля версий. Были проблемы при разработке и выкладывании Linux. После судебных исков, был написан Git.
- **Git** –2005 год, система контроля версий, которая превзошла остальные и стала в последующие годы бесспорным промышленным стандартом (<https://git-scm.com/download/win>).

Git ставится на компьютер. Работает как консоль. Можно писать от руки (с клавиатуры) команды. Можно писать *.bat файлы (пакетно-запускаемые списки команд). Можно подключаться программно через дочернюю консоль или через Temp.bat, временный файл, с удалением после исполнения через CMD. Можно работать с удаленными депозитариями. Подробнее об этом в официальной документации Git Documentation (<https://git-scm.com/doc>).

Натренироваться можно. Вот онлайн JavaScript приложение GitLearn (https://learngitbranching.js.org/?locale=ru_RU). Там достаточно уроков и онлайн упражнений, чтобы затем чувствовать себя в консоли комфортно.

Для быстрого вхождения в распределенную систему контроля версий можно использовать сторонние программы, что предоставляют англо/русскоязычный интерфейс. Вот некоторые из них: GitExtensions(<https://github.com/gitextensions/gitextensions/releases/tag/v3.4.1>); OctoGit (<https://github.com/Sabjeet/Octo-Git/tree/master/Octo>) - как видно из содержимого, это просто набор батников; GitKraken (<https://www.gitkraken.com/>); SmartGit (<https://www.syntevo.com/smartgit/>); TortoiseGit (<https://tortoisegit.org/>); TeamExplorer (<https://docs.microsoft.com/ru-ru/visualstudio/ide/reference/team-explorer-reference?view=vs-2019>)- это расширение от Visual Studio 2019.

Первые 5 из графических интерфейсов - самоделки или графические интерфейсы от тех или иных ИТ компаний. TeamExplorer - чуть более серьезное исполнение от Microsoft. Но TeamExplorer губит понимание понятийного аппарата Git, сводя все его возможности к двум кнопкам. Да, в нем можно найти все, но лучше попробовать сперва что-то другое.

Проблема при эксплуатации в том, что периодически выпускаются обновления Git, вслед за ними с разной степенью запаздывания обновляются все последующие интерфейсы. Можно что-либо случайно обновить и «уронить» все, что на нем держалось.

Рекомендуется иметь в запасе 2 или 3 способа работы с Git/GitHub/...

Как наладить работу GitExtensions-Portable

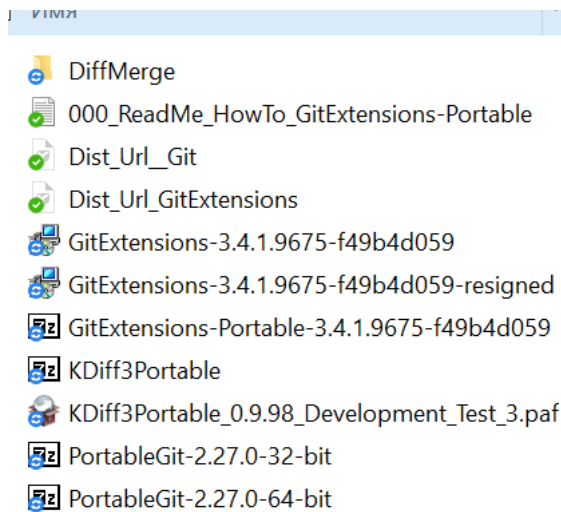


Рис. 1. Список архивов дистрибутивов

<https://git-scm.com/download/win>

<https://github.com/gitextensions/gitextensions/releases/tag/v3.4.1>

Шаг 1. Распаковываем GitExtensions-Portable-3.4.1.9675-f49b4d059

Шаг 2. Распаковываем PortableGit-2.27.0-64-bit

Шаг 3. Запускаем. GitExtensions-Portable-3.4.1.9675-f49b4d059/GitExtensions.exe

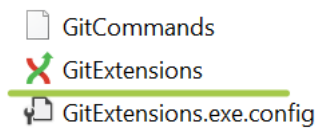


Рис.2. Что запускать

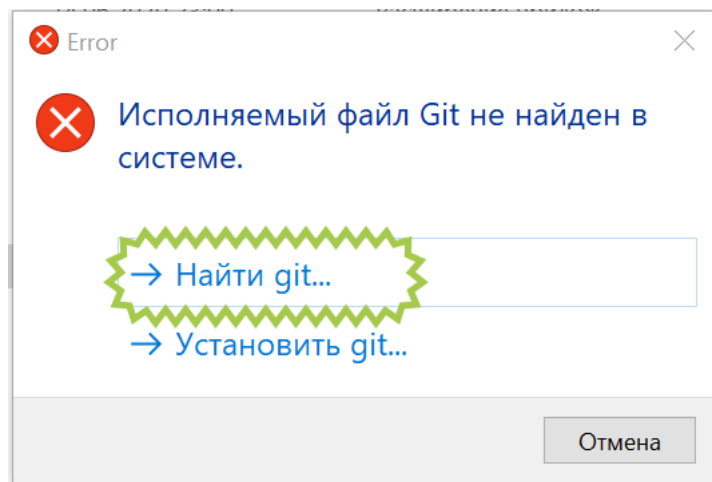


Рис.3. Если «будет ругаться», то указываем путь к Git
(PortableGit-2.27.0-64-bit)

Как создать репозиторий?

Шаг 1. Создать папку с названием проекта.

Шаг 2. Создать в этой папке репозиторий (как показано на рисунках ниже).

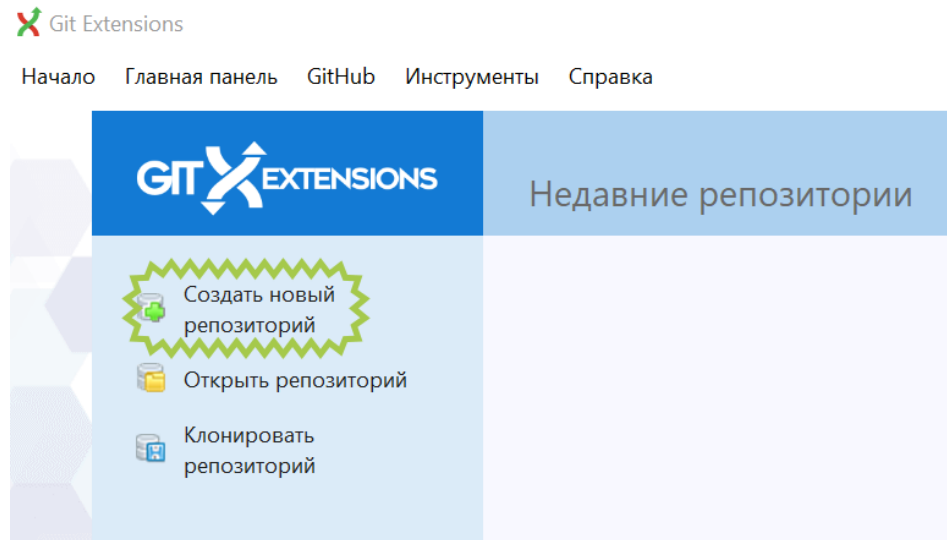


Рис. 4.Создание нового репозитория

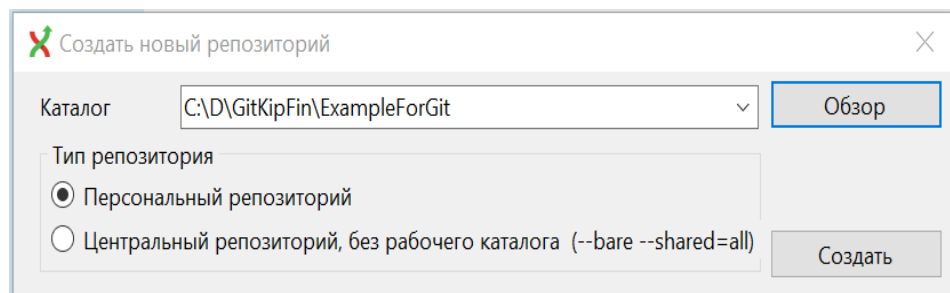


Рис. 5. Что выбирать: персональный или центральный репозиторий?

Центральный репозиторий состоит только из технических файлов Git, персональный – кладет все эти файлы в скрытую папку `”.git”`. Пример файлов и папок центрального репозитория прилагается (рис. 6).

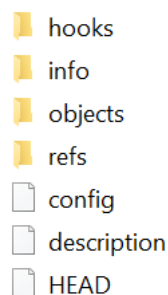


Рис. 6. Список файлов центрального репозитория

Как сделать Коммит?

Шаг 3. Как сделать наш первый Коммит.

В начале работы, при создании первого репозитория, рекомендуется положить в папку персонального репозитория какие-нибудь файлы с текстом. Затем нажимаем на кнопку “Коммит” (рис. 7).

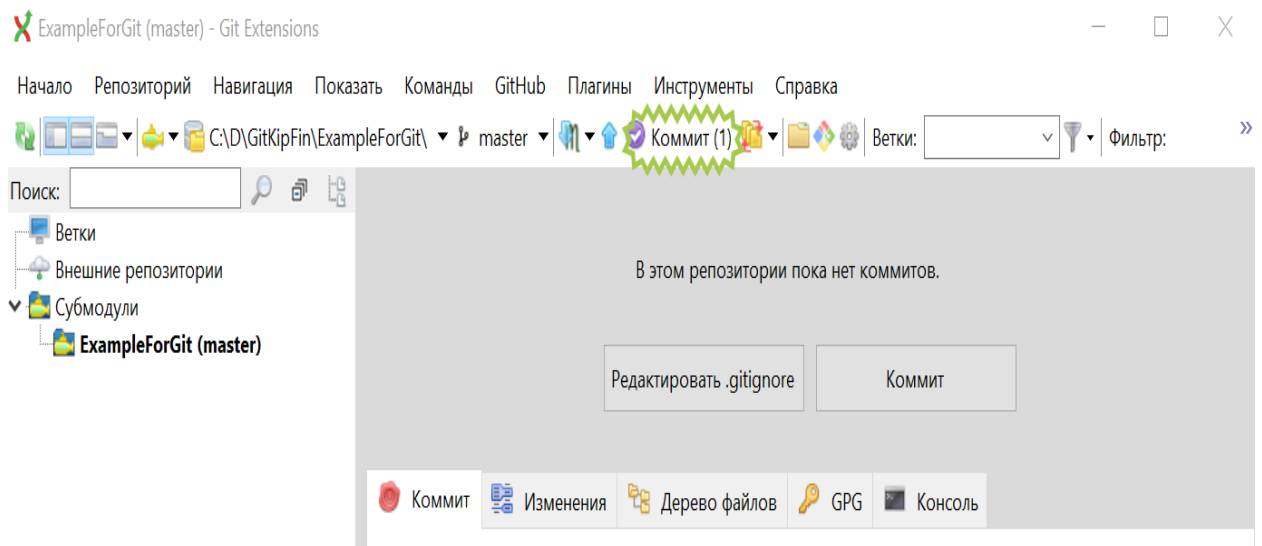


Рис. 7. Меню репозитория

Перед Вами откроется меню создания коммита (рис.8). Сверху слева – список непроиндексированных изменений файлов. Снизу слева – список проиндексированных файлов. В коммит попадают только проиндексированные изменения файлов. Сверху справа – программный код выбранного файла. Снизу справа – окно для ввода комментария к коммиту. Добавлять файлы в индекс (индексировать) можно с использованием стрелочек – вверх и вниз.

Затем можно создать коммит нажатием на кнопку “Зафиксировать”. Если нажать “Зафиксировать и отправить”, то коммит отправится ещё и в родительский репозиторий, если таковой существует (репозиторий источник клонирования).

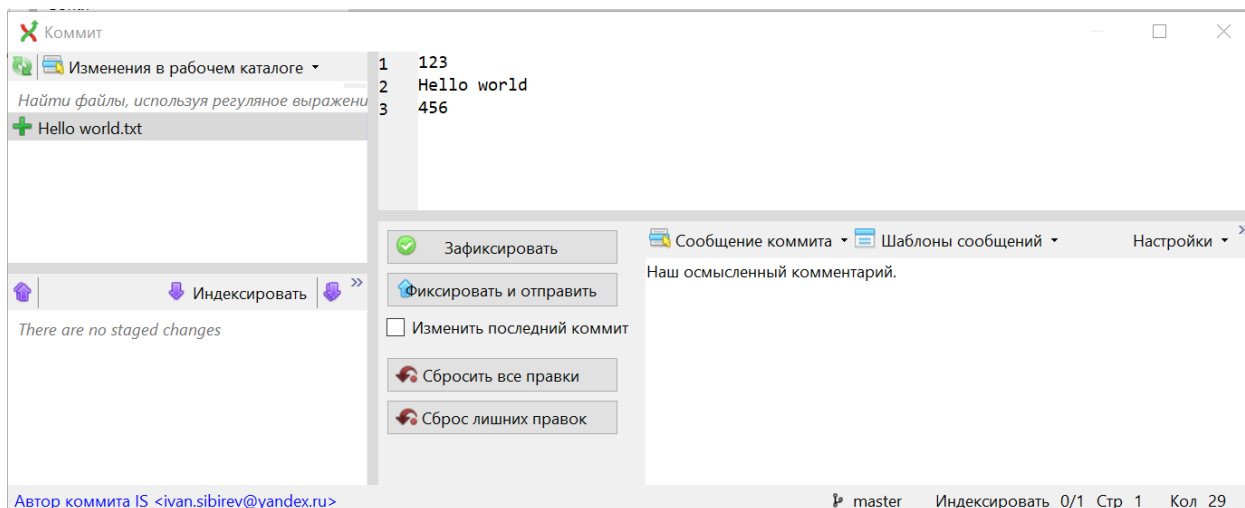


Рис. 8. Меню создания коммита

В качестве упражнения рекомендуется добавить текст в файл и сделать несколько коммитов (рис. 9).

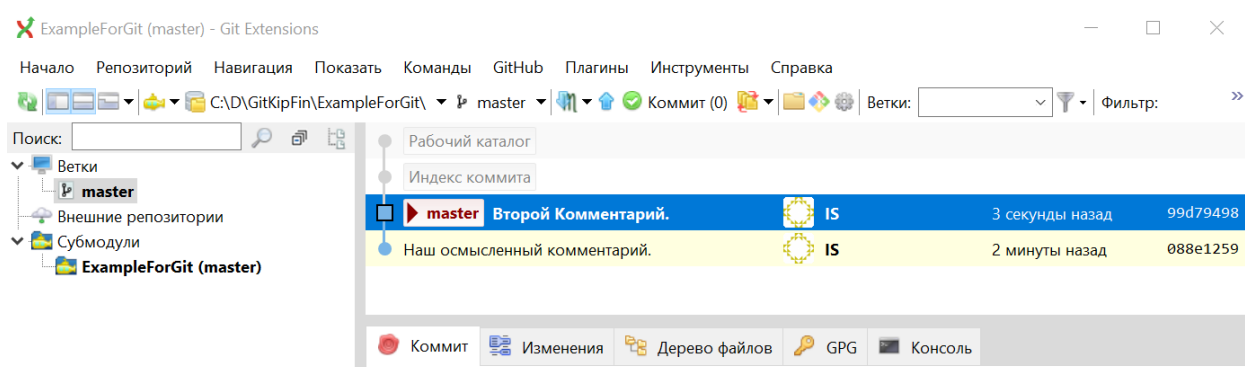


Рис. 9. Меню создания коммита

Ветвление и клонирование?

Шаг 4. Создать ветку.

По умолчанию существует ветка “master”. На практике ветки создаются под отдельную задачу, под отдельного программиста или под релизную версию, которую нужно иметь в рабочем состоянии на случай неожиданной презентации.

Как следствие этого, работа в команде начинается с клонирования репозитория “К себе” и создания новой ветки. Давайте попробуем это сделать.

Дано: центральный (родительский) репозиторий в папке с названием "ExampleForGit".

Что сделать: получить из центрального репозитория свой локальный репозиторий в папку с названием "ExampleForGit_Local", создать свою ветку, сделать в ней коммит и залить на центральный репозиторий «без происшествий» (это важно).

Давайте сделаем это.

Шаг 4.1. Клонирование репозитория.

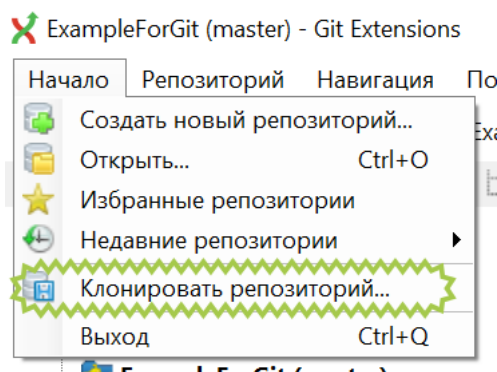


Рис. 10. Клонирование репозитория

Затем требуется указать все настройки клонирования.

- *Внешний репозиторий* – адрес центрального или родительского репозитория, он не обязательно должен быть центральным или локальным. Настройка Внешний репозиторий принимает в себя не только путь к локальной папке, но и ссылку на репозитори GitHub.

- *Назначение* – адрес локальной папки, в которой будет создана новая папка и размещен репозиторий. Подкаталог для создания – название новой папки. Также можно указать ветку, что сократит трафик. Весь репозиторий может занимать несколько гигабайт мелких и потому тяжело копируемых файлов. Скачивание только одной ветки ограждает нас от мучительных ожиданий при копировании.

Клонировать

Внешний репозиторий: C:\D\GitKipFin\ExampleForGit Обзор

Назначение: C:\D\GitKipFin\ Обзор

Подкаталог для создания: ExampleForGit_Local

Ветка: master

Репозиторий будет клонирован в новый каталог, расположенный здесь:
C:\D\GitKipFin\ExampleForGit_Local (Новая папка)

Тип репозитория

☒ Персональный репозиторий

☐ Публичный репозиторий, без рабочего каталога (--bare)

☒ Инициализировать все submodule ☒ Загрузить всю историю

Загрузить SSH ключ

Клонировать

Рис. 11. Клонирование репозитория, настройки

– Затем нужно создать *новую ветку*. Ветка это последовательность коммитов от самого корня до крайнего коммита. В теории, коммиты могут существовать, будучи не привязанными к конкретной ветке. На практике нам не предстоит с этим столкнуться. Это удел администраторов крупных репозиториях.

– *Коммит* – это перечень индексированных изменений файлов. Выбираем интересующий нас коммит, обычно это последний коммит ветки “master”. Правой кнопкой мышки вызываем контекстное меню, нажимаем кнопку «создать новую ветку здесь».

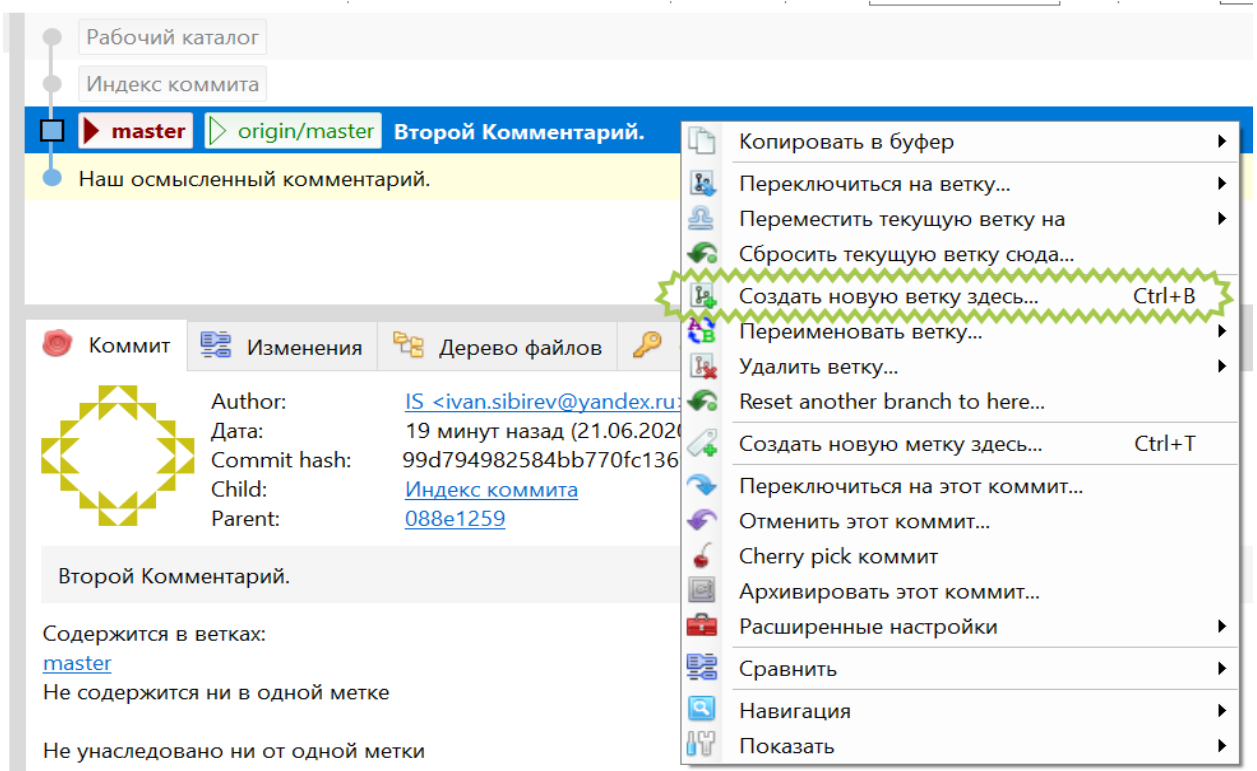


Рис. 12. Создание ветки

Затем следует указать название ветки и другие настройки. Обычно этот вопрос регламентируется документацией предприятия. Например: название ветки совпадает с названием задачи; название ветки совпадает с текстом описания задачи; название ветки содержит название релизной версии проекта; название ветки содержит ключи, например ключ DX – Delphi10; название ветки содержит произвольный текст и т. д.

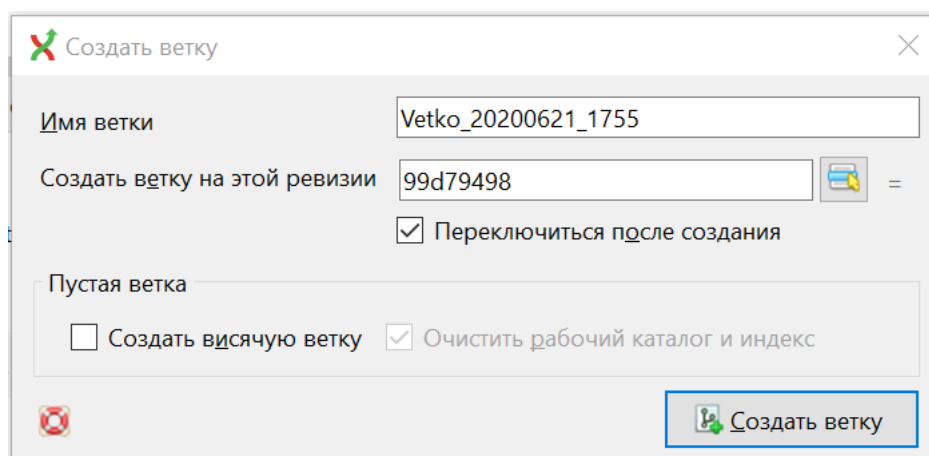


Рис. 13. Создание ветки, настройки

Затем работаем, индексируем изменения, создаем коммит.

Кнопка *Push* – отправить изменения в родительский репозиторий (рис. 13).

Кнопка *Pull* – забрать изменения этой ветки из родительского репозитория (рис. 13).

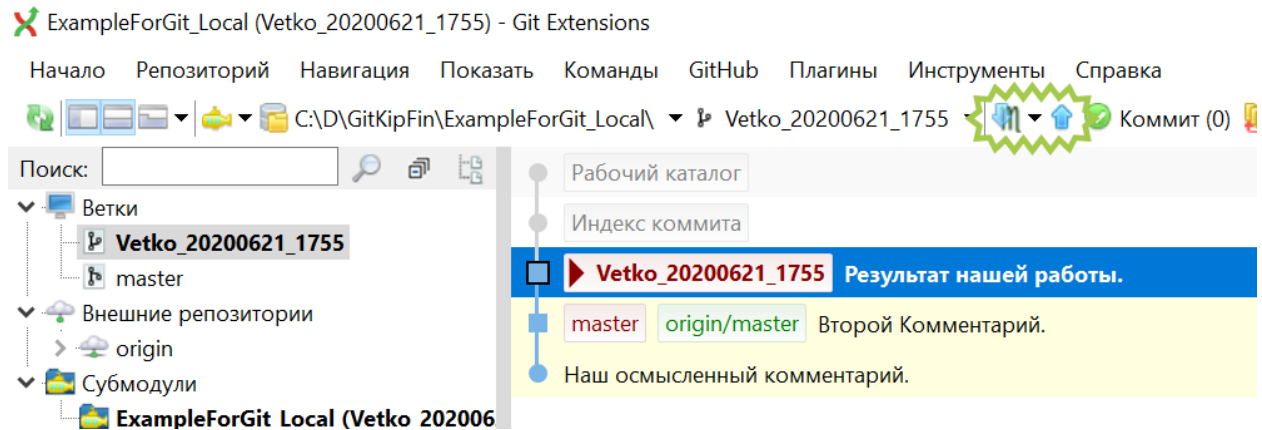


Рис. 14. Коммит, созданный нами в своей ветке, в локальном репозитории

Объединение веток или слияние

Шаг 4.2. Объединение веток на стороне разработчика и заливка их на сервер.

Шаг 4.2.1. Дописываем свой программный код, в своей ветке делаем конечный коммит.

Шаг 4.2.2. Переключаемся на мастер ветку.

Существуют две версии мастер ветки, чаще всего оно совпадают. “Master” – локальная мастер ветка. “origin/Master” – мастер ветка центрального (родительского) репозитория. Выбираем “Master” (рис. 15). При этом мы попадем в Head мастер ветки, то есть выбранный в мастер ветке коммит, чаще всего он самый последний в цепочке. При этом в локальной папке все файлы получают состояние на момент этого коммита.

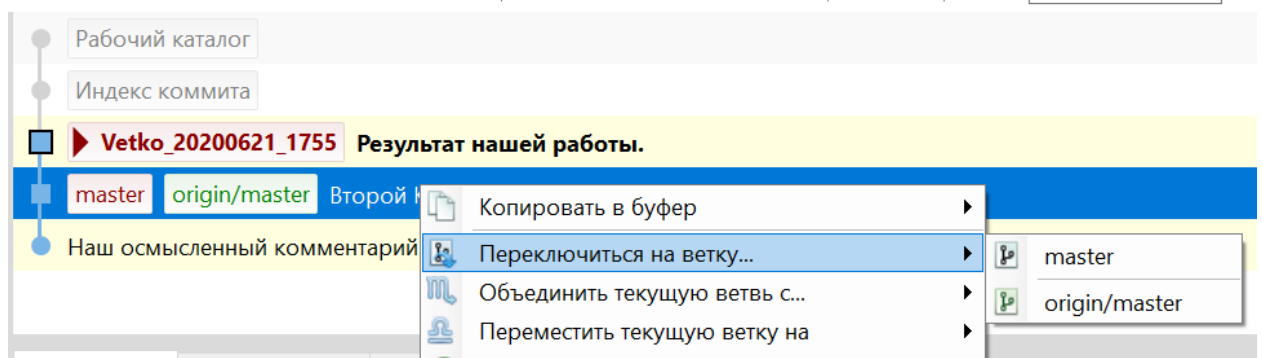


Рис. 15. Переключаемся на мастер ветку, способ первый

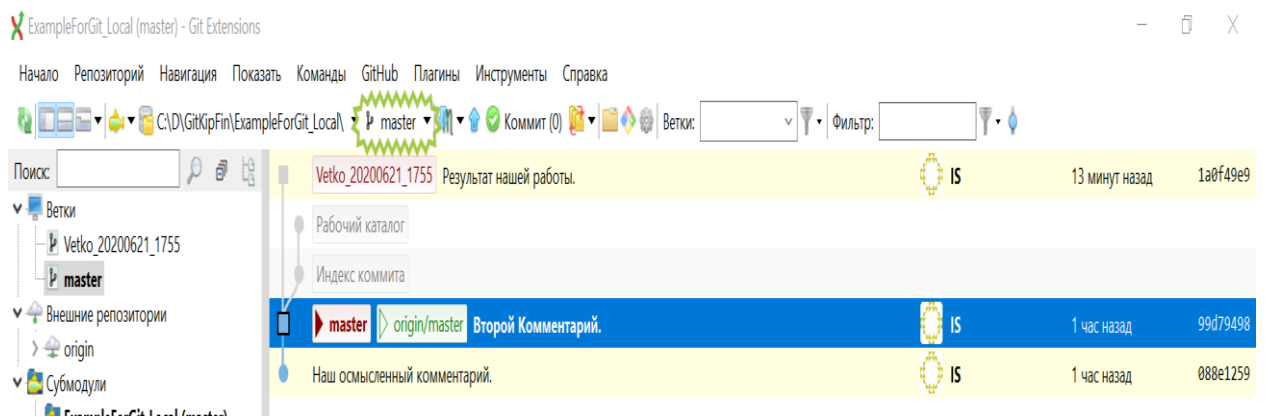


Рис. 16. Переключаемся на мастер ветку, способ второй

Затем нужно сделать Pull мастер ветки, получить изменения к себе. Если так не сделать, то можно «поймать» ошибку, при заливании другой версии мастер ветки на сервер. Это делается на случай, если кто-то успел что-то изменить и залить (рис. 17).

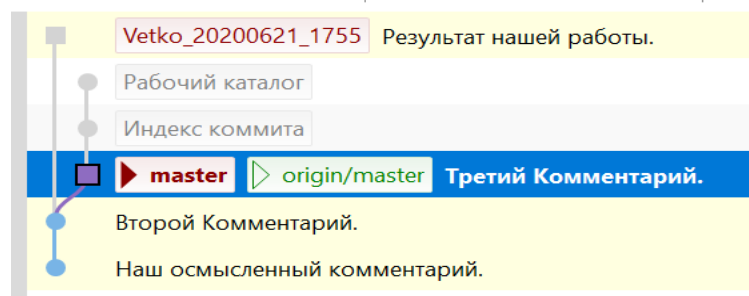


Рис. 17. Полученная мастер ветка с изменениями от другого пользователя

Теперь переходим в мастер ветку, выбираем последний коммит нашей локальной ветки, открываем контекстное меню, выбираем «объединить с текущую ветку с», выбираем нашу ветку.

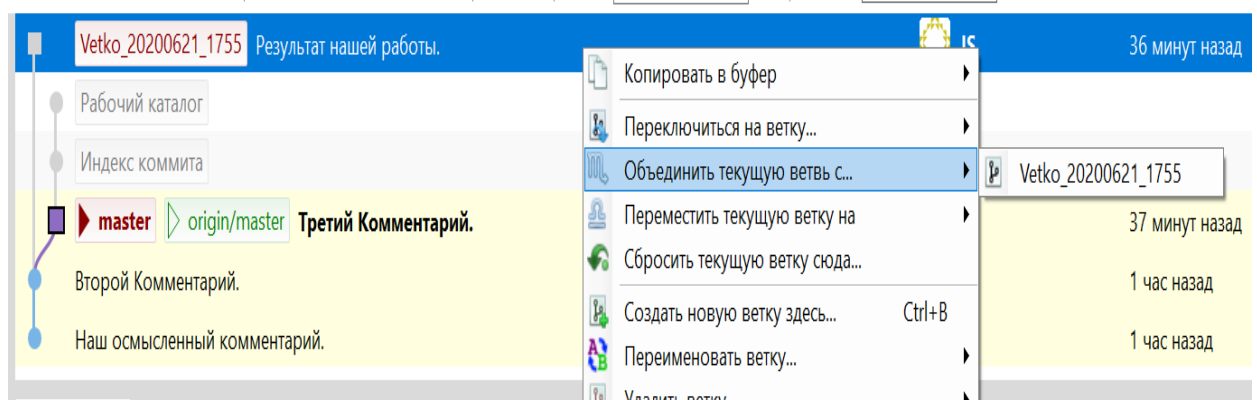


Рис. 18. Объединение веток

Есть два способа исправления ошибок слияния. Первый: самому, клонировать два репозитория, один для мастер ветки, другой для локальной ветки, и попарно просматривать файлы, внося изменения в мастер ветку, чтобы потом сделать коммит. Минусы этого способа: надолго останавливает доступ к мастер ветке, с ней никто другой работать в это время не может (у двух работающих будут конфликты слияний).

Чаще, пользуются инструментами слияния, предварительно настроенными в среде... Для этого используется diffuze или DiffMerge, KDiff3Portable и др. (в настройках можно посмотреть полный список).

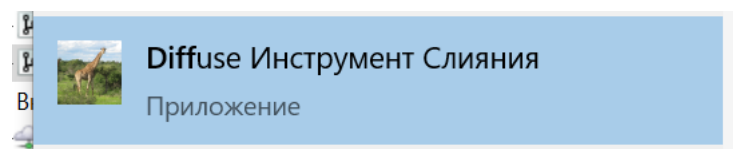


Рис.21. Инструмент слияния DiffMerge

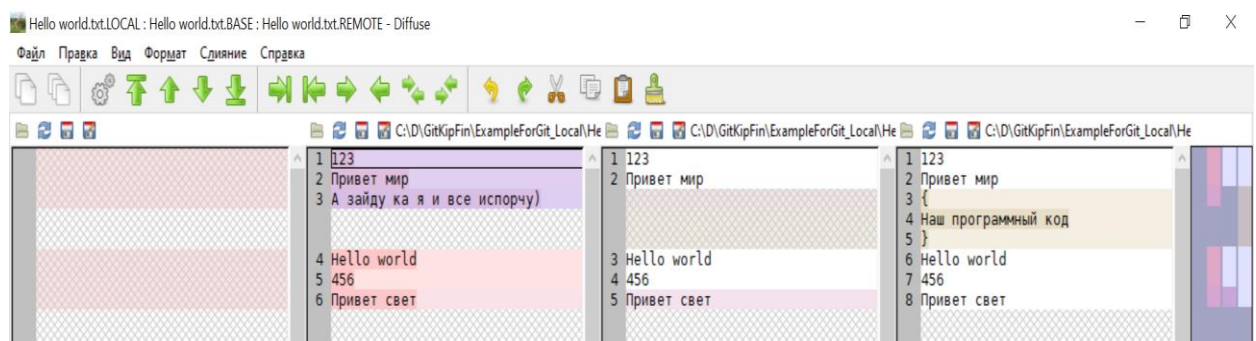


Рис. 22. Инструмент слияния DiffMerge

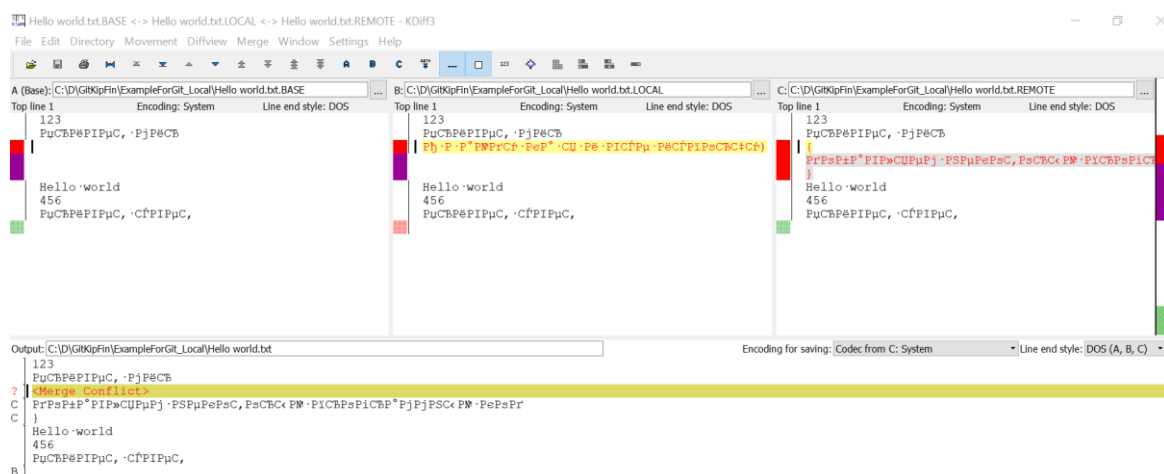


Рис. 23. Инструмент слияния KDiff3Portable

DiffMerge поддерживает русский язык.

Порядок столбиков определяется либо открытием файлов, либо операндами среды. Порядок по умолчанию: "\$MERGED" "\$LOCAL" "\$BASE" "\$REMOTE" – "Результат ", "Мастер ветка", "Общий предок", "Сливаемая ветка ", соответственно.

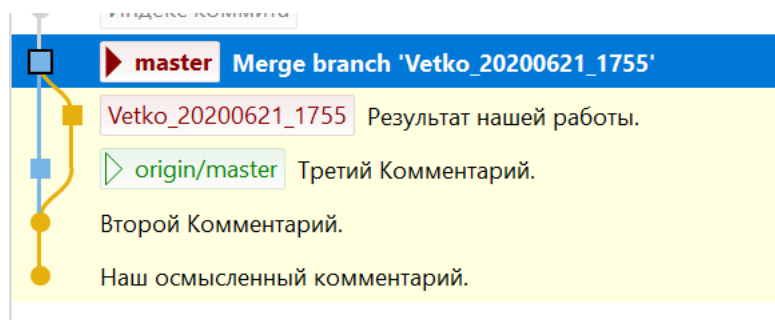


Рис. 24. Инструмент слияния KDiff3Portable.

Локальные ветки в центральный репозиторий попадают, только если их специально залить, что делает историю проекта чище, чем его история на компьютере разработчика.

Подводные камни или «Только клонирование!!!»

Git репозитории плохо переносят архивирование и копирование через проводник.

Следствие – баги на уровнях Git и IDE. Например, ошибки при заливании проекта на центральный репозиторий или ссылка IDE на старую версию файла, где-то в недрах жесткого диска или корзины даже после перезагрузки... Лечится только клонированием.

В GitExtensions изредка встречается такой баг: ошибка побитого файла глобальной конфигурации. Лечение – переименованием: в той же папке лежат 2-3 бекапа.

При "Merge" или слиянии веток, если изменен один и тот же файл, то агрегатор Git может не справиться. Тогда нужно использовать инструмент слияния. Команды вызова инструмента из среды от версии к версии могут отличаться, да и расположение в настройках – тоже (инструменты/настройки).

"C:/Program Files (x86)/Diffuse/diffusew.exe" "\$MERGED" "\$LOCAL"
"\$BASE" "\$REMOTE".

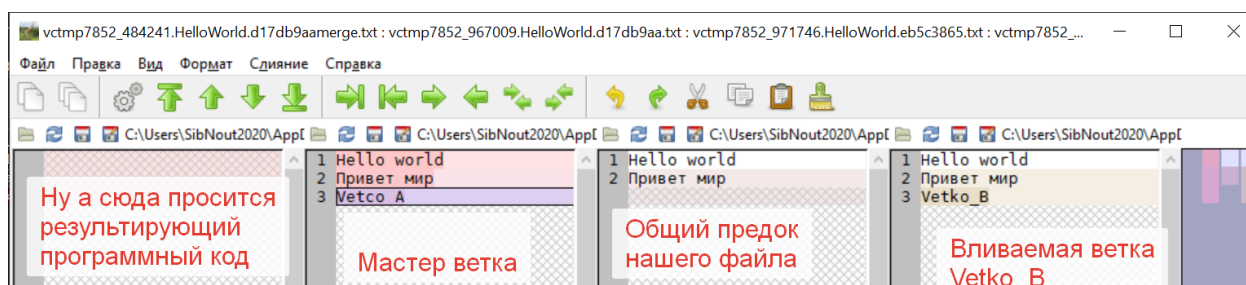


Рис. 25. Инструмент слияния, DiffMerge

Редактировать нужно и можно только "\$MERGED".
Как следствие, почти каждая такая операция считается «радиоактивной», и объединения веток ложится в базу данных «костылем», требующим редактирования администратора.

Выбор центрального хранилища, GitHub

Предлагаются следующие варианты.

- 1) Это может быть специально настроенный, удаленный, защищенный сервер.
- 2) Это может быть сетевая папка с центральным репозиторием.
- 3) Это может быть какое-либо облачное хранилище, например, OneDrive или ЯндексДиск (минус – низкая степень безопасности).
- 4) GitHub – специальный сервис, позволяющий создать удаленный центральный репозиторий, с ограничением не более 25 мегабайт на файл. Нам удавалось залить на GitHub виртуальную машину Windows 10 размером в 37 гигабайт.

GitHub позволяет создавать публичные и приватные репозитории, позволяет ими делиться с коллегами по ссылке. GitHub – это сосредоточение разработки свободного программного обеспечения. И тем не менее, GitHub обладает низкой степенью безопасности, но она по крайней мере выше, чем у ЯндексДиск. Для наших целей вполне хватит.

Требуется зарегистрироваться на GitHub (<https://github.com>), подтвердить регистрацию... Затем можно справа сверху вызвать контекстное меню вашего профиля и выбрать «Открыть ваши репозитории».

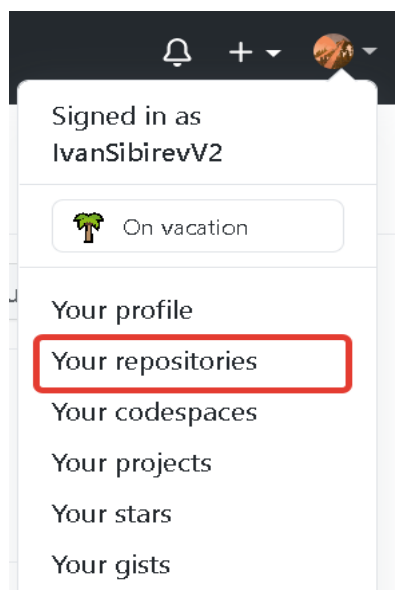


Рис. 26.Список репозиториев

Так выглядит список репозиториев. При нажатии на кнопку New можно создать новый репозиторий.

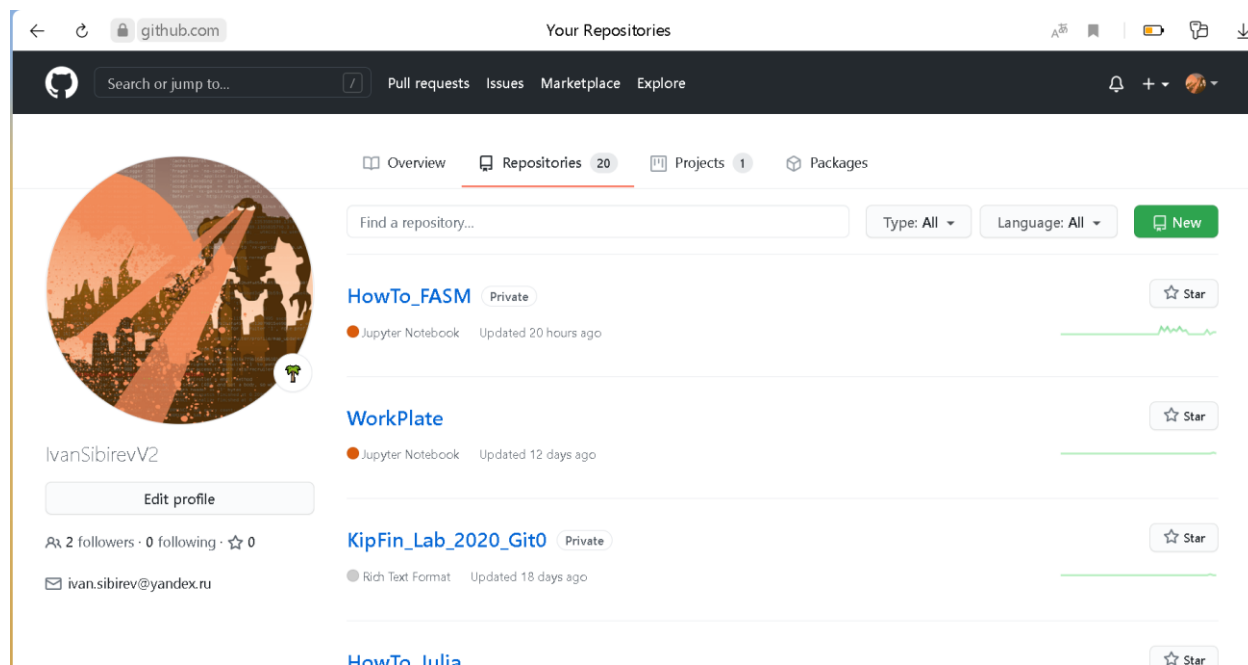


Рис. 27. Меню создания репозитория

Откроется меню создания репозитория. Требуется ввести имя репозитория. Во избежание мучительных казусов сторонних графических интерфейсов с

ключами пакетно-запускаемых файлов, передачей по сети Интернет и русским текстом, рекомендуется поначалу использовать только англоязычные символы без пробелов и цифр. Ошибочно созданный репозиторий можно удалить через настройки GitHub.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?

[Import a repository.](#)

Owner *



IvanSibirevV2 ▾

Repository name *



Great repository names are short and memorable. Need inspiration? How about **sturdy-octo-guide**?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**

Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**

A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

Рис. 28. Вкладки репозитория

Перейдя в сам репозиторий на сайте GitHub во вкладке «код» можно получить ссылку для скачивания репозитория, скачать всю папку архивом, посмотреть и скачать отдельный файл. Вкладка AddFile позволяет добавлять файлы в репозиторий. Это удобно на случай если настройки сети не позволяют графическим интерфейсам работать с портами.

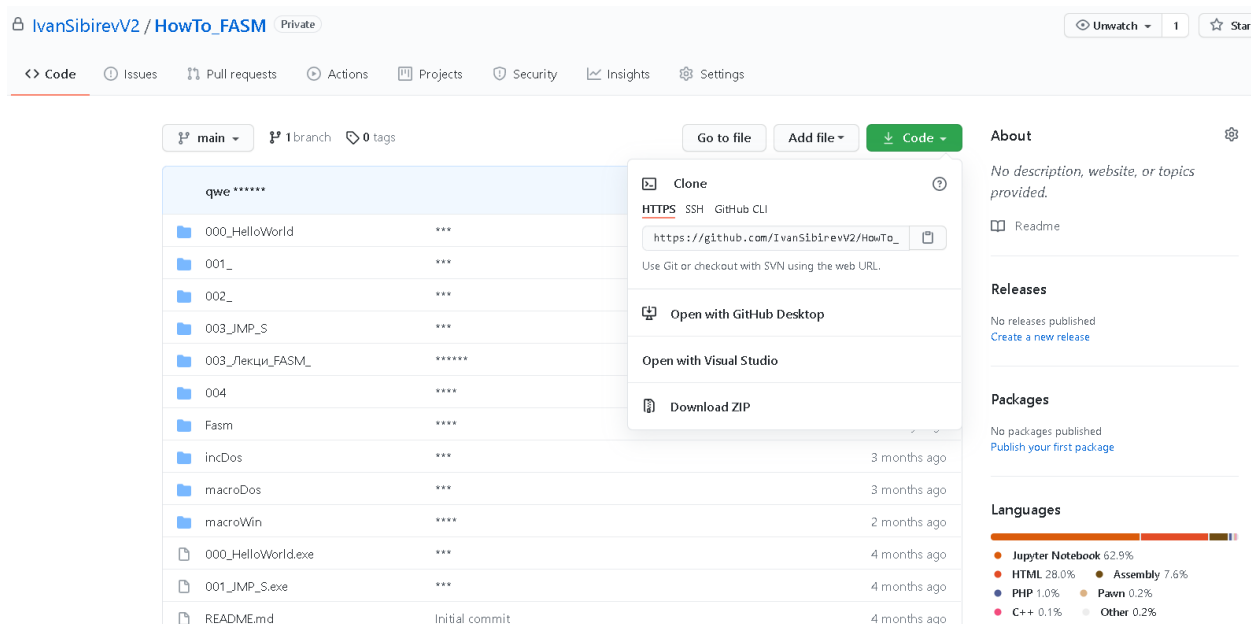


Рис. 29.

А теперь попробуем при помощи **GitExtensions-Portable** получить репозиторий с **GitHub**.

<https://github.com/KIPFINDemoExam/ONMurashkin.git>

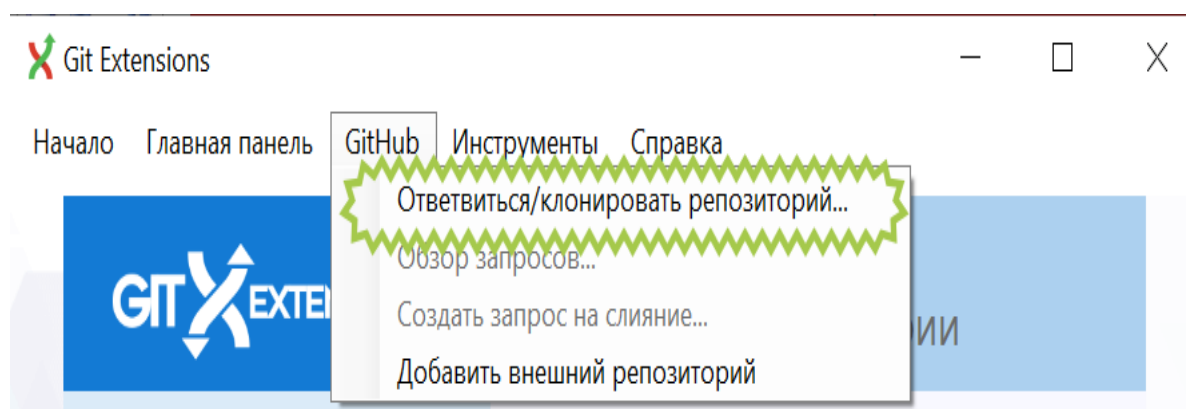


Рис.30. Получим репозиторий с GitHub

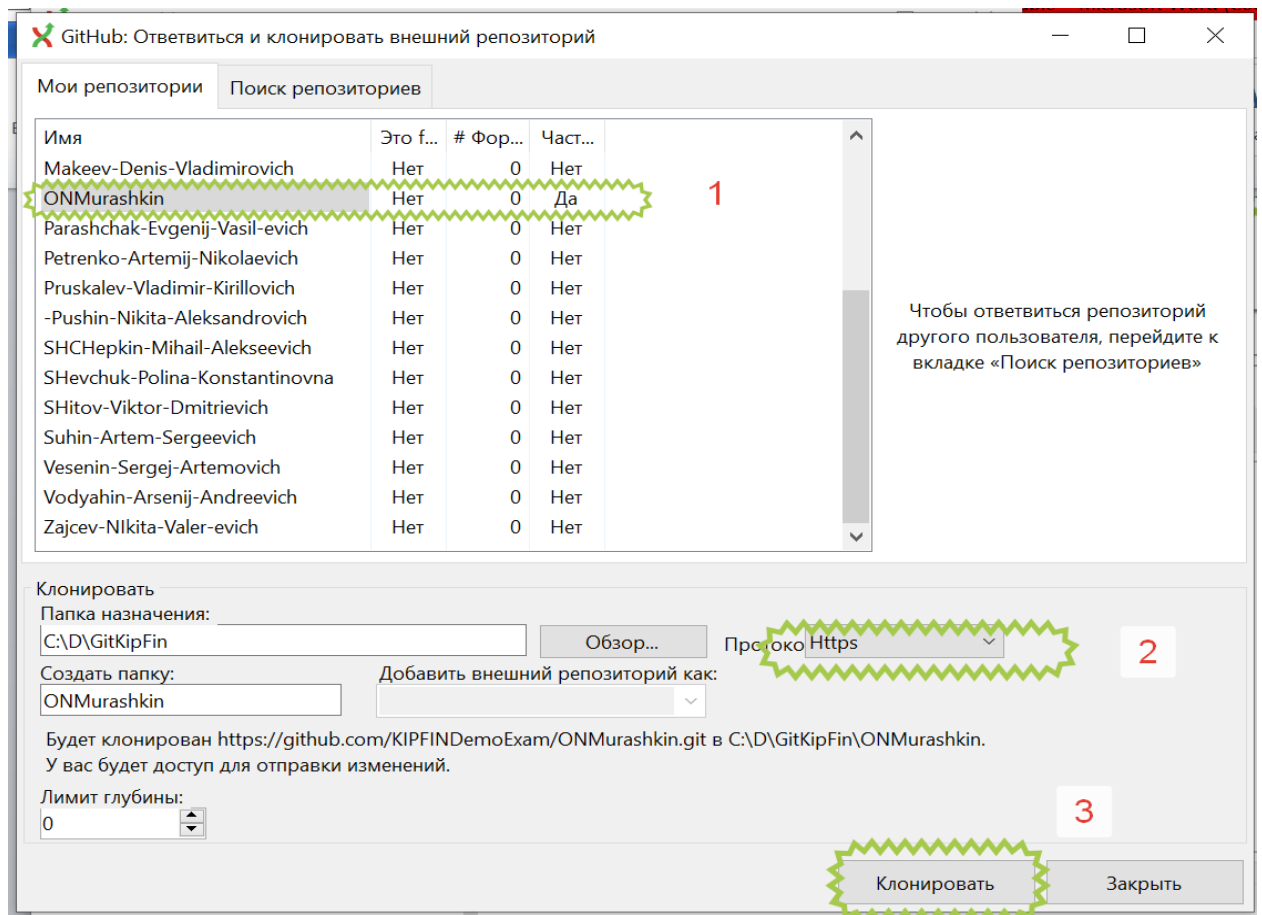


Рис. 31. Ответвиться и клонировать внешний репозиторий

Затем в консольном окне нужно ввести логин и пароль.

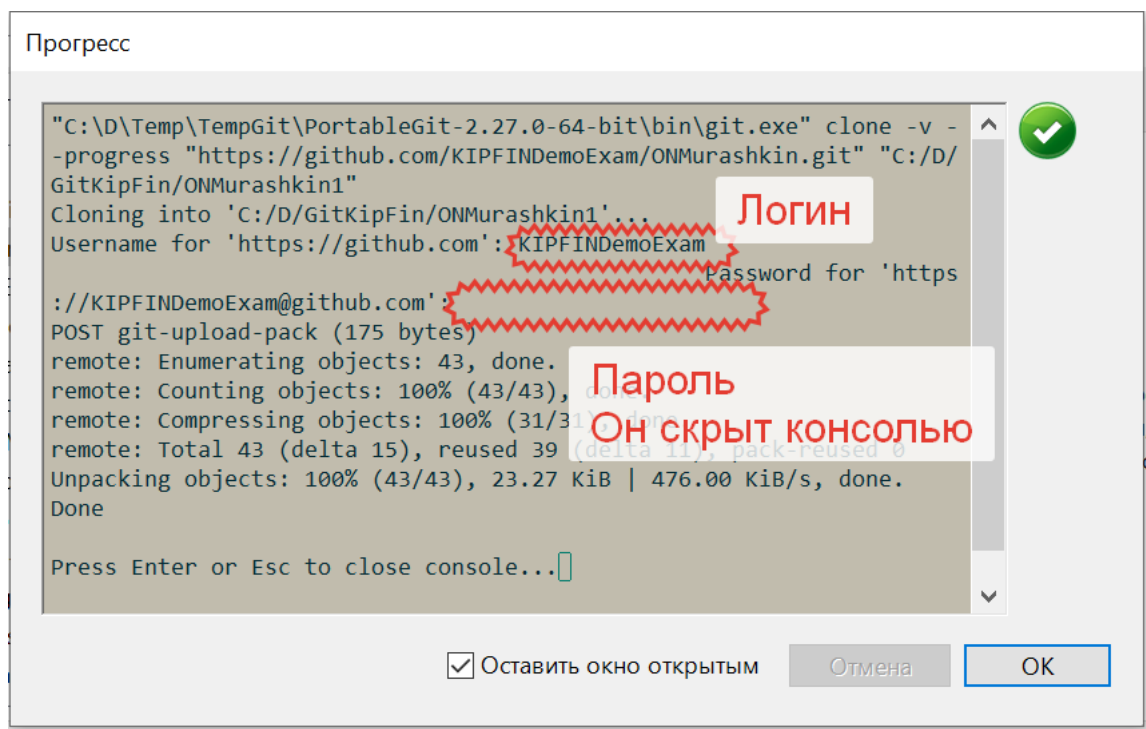


Рис. 32. Введем логин и пароль

Теперь закидываю туда эту инструкцию и заливаю на GitHub
<https://github.com/KIPFINDemoExam/ONMurashkin.git>

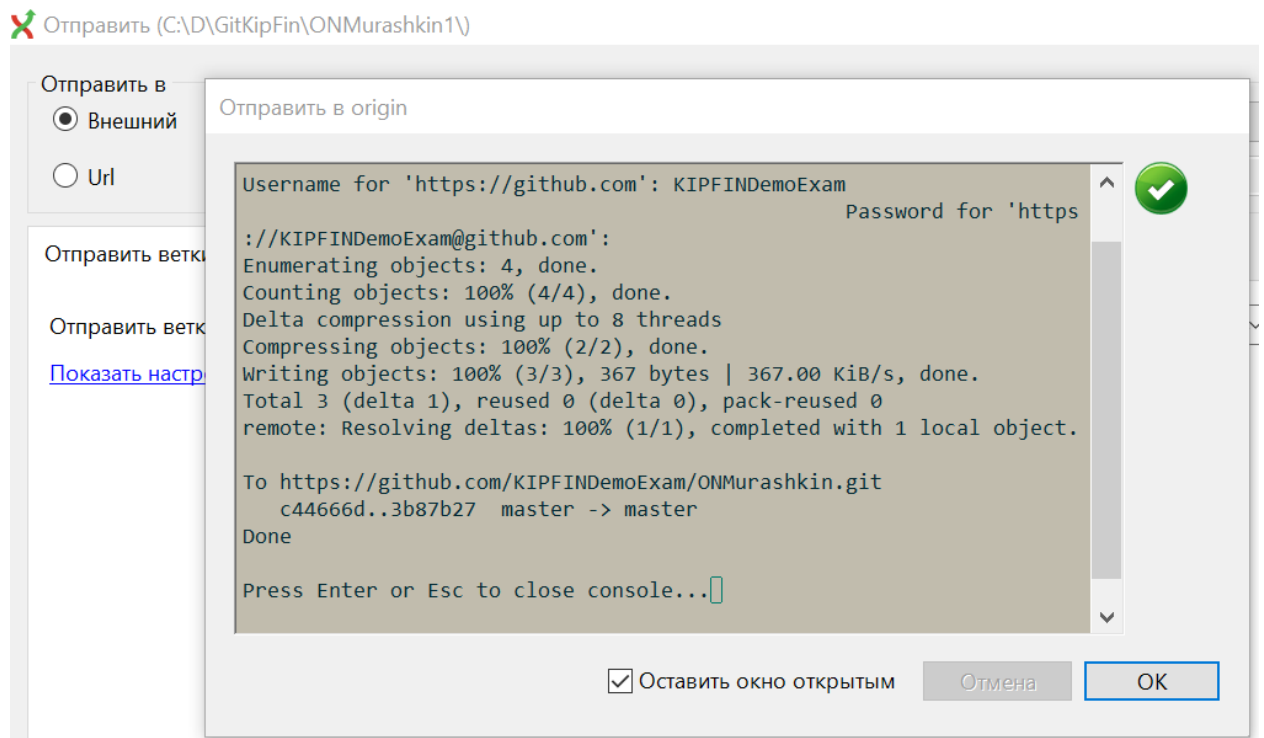


Рис. 33. Заливаем на GitHub

Выводы

Предлагаемые инструменты полностью -Portable- и подходят как для основного, так и резервного использования.

<https://git-scm.com/download/win>

PortableGit-2.27.0-32-bit

PortableGit-2.27.0-64-bit

<https://github.com/gitextensions/gitextensions/releases/tag/v3.4.1>

GitExtensions-Portable-3.4.1.9675-f49b4d059

Дистрибутивы и инструкцию можно скачать по ссылке.

<https://yadi.sk/d/0lFgxyHg4BTcRQ/TempGit>

Про KDiff3Portable, DiffMerge и DiffusePortable – судить насколько они Portable не берусь, изначально оба инструмента требовали установки, но они пока вроде работают, что называется «с флешки».

Разбор ошибок

Наиболее распространенная ошибка – трата времени на прочтение всего материала сразу. Ожидается, что студент получает инструкции и последовательность действий либо от преподавателя, либо из задания на лабораторную работу.

Вторая по распространенности ошибка – скачивание не всего программного обеспечения. GitExtensions – без Git работать не будет. При первом запуске потребуется указать путь к Git.exe, который обычно лежит в папке Bin. Если после обновления Git, или после скачивания более поздней версии Git учащийся не может найти Git.exe, рекомендуется обратиться к тому, кто это уже сделал, или к преподавателю.

Третья ошибка – сетевое расположение рабочего стола. Все скаченное программное обеспечение должно быть распаковано или установлено на локальный жесткий диск.

Следующая по распространенности ошибка – обновление GitExtensions-Portable. С этим лучше не связываться. Установите полноразмерную версию, а затем просто перенесите всю папку в тот же путь, но на другой машине.

Далее следует ошибка: «Это не мой логин, пароль». Нажмите Win+R, затем введите “appdata“, нажмите Enter. Откроется папка для временного хранения файлов Windows. При помощи поиска найдите временную папку Git – удалите её.

Ошибка – забыли закрыть GitExtensions перед удалением временной папки.

Бывает и так: «Сижу, жду, программа ничего не делает...», например, после нажатия на кнопку». Программа периодически в ответ на пользовательские действия открывает свои окна. Иногда они открываются

позади всех окон или в фоновом свернутом режиме. Не ждите у моря погоды, просто найдите их!

А теперь: «Я все сломал». Все программное обеспечение удаляем, удаляем временную папку. Распаковываем/ устанавливаем, запускаем заново.

При ситуации: «Я забыл указать имя пользователя и почту в настройках Git/ GitExtensions,» – все программное обеспечение удаляем, удаляем временную папку. Распаковываем/ устанавливаем, запускаем заново.

Что делать, если: «Я сломал свой репозиторий, создав конфликт слияния»? Ответ. Для текущей лабораторной работы разрешать конфликт не требуется, возьмите предыдущую версию репозитория из центрального хранилища и просто перенесите туда файлы. С разрешением конфликтов слияния можно работать, но не в первый день. Бывает так, что человек много лет работает программистом и ни разу не доводил до разрешения конфликтов слияния свой репозиторий.

Следующая ошибка – с заливкой репозитория на GitHub. Причины: не подтвержден аккаунт, имя репозитория дает сбой при консольных командах (Только английские буквы!), по сети закрыт порт для Git, на GitHub профилактические работы, Ваш аккаунт побит или «забанен». В таких случаях помогает, если сменить пользовательский графический интерфейс. В крайнем случае, попробуйте работать через сайт GitHub, заливая изменения пофайлово. Если и это не помогло, то попросите логин, пароль от аккаунта коллеги и работайте через его профиль со своим новым репозиторием.

В ситуации: «У меня все сделано дома, в классе я работать не смогу, »– начинайте делать заново. Не забываем заливать изменения в центральный репозиторий, у нас все всегда лежит в Интернете, отговорки не принимаются.

Ситуация: «Я забыл свой логин, пароль». Восстанавливаем. «Я забыл свой логин пароль от GitHub и почты», – восстанавливайте логин, пароль от почты. Тот, кто не может восстановить логин, пароль от почты – работает на аккаунте более ответственного коллеги.

Редко случается ситуация: «У меня на GitHub все было, а теперь там пусто». Возможно, Вы забыли залить изменения на сервер, или кто-то зашел на GitHub и удалил все репозитории. Что же – выходите из своих аканутов и чистите за собой временные папки Windows.

Редко возникают «баги непонятного происхождения после файлового копирования репозитория». Репозитории не копируют, их только клонируют.

Примерный план проблемной лекции

Введение. Эзотерические языки программирования на примере BF

Время изучения темы – от 2 до 4 часов, из них 2 часа- лекция

Учебные цели занятия: изучить представителя группы ассемблеро-подобных машинно-ориентированных эзотерических языков программирования **BF**, познакомиться со стеком вызова функций и особенностями программирования на его вершине.

Форма проведения занятия: проблемная лекция.

Учебные задания: изучить истоки, актуальность вопроса, которая не очевидна, операции инкремент декремент, особенности написания ветвлений через циклы с предусловием, "процедурный" **BF** на основе вложенных циклов с постусловием на примере операции умножения чисел, посимвольный вывод в консоль своего ФИО в ASCII или UTF кодировке.

Инструкции по проведению и ходу занятия: В начале занятия обосновываются цель и задачи проблемной лекции, определяется проблема для обсуждения. Преподаватель излагает материал, задает вопросы студентам, посвященные проблематике лекции, затем начинается дискуссия по поставленной проблеме; вторая половина занятия - групповая практика на ПК или онлайн, **BF** - через телефон.

Методические рекомендации: ожидается, что учащийся найдет в Википедии страничку языка **BF** (и некоторых других языков по рекомендации преподавателя), изучит 8 базовых команд, получит шаблоны написания кодов от преподавателя, затем приступит к групповому написанию программных кодов. Задание хоть и шуточное, но его выполнение обычно дается не с первого раза, что приводит к алгоритмическому росту учащихся.

Вопрос для самоконтроля:

При сдаче задания его удастся запустить?

Нет семантических ошибок?

Почему выбран именно такой путь реализации алгоритма?

Главное, чтобы программный код работал!

Введение

Ассемблер - машинно-ориентированный язык программирования низкого уровня. Иными словами, это язык управления битами в регистрах и оперативной памяти. Исполнитель Ассемблера - микропроцессор, реже высокоуровневая прослойка операционной системы. Ассемблер - язык старый, датируется 1947 годом, содержит множество диалектов, описать которые можно как множество частных случаев реализации идеи машинно-ориентированного языка. Литература встречается, вся она, скорее всего, пролежала несколько десятилетий на

полке библиотеки, прежде чем попасть в руки. Поэтому примеры из книг часто не работают или требуют адаптации под современные реалии.

Чтобы что-нибудь написать, приходится пройти через поиск на просторах Интернета. При этом то, что Вам нужно, попадает только каждый 10 или 20 раз. Периодически встречаются посты с заведомо ложной информацией, распознать которую новичку не под силу. Программные коды, взятые из литературы, обладают высокой скоростью роста сложности и потому не пригодны для начинающих.

Все это создает высокий входной порог сложности материала данной темы.

Эзотерические языки программирования

Мы постараемся описать траекторию, по которой удастся преодолеть большую часть подобных трудностей.

Материалы изложены в порядке возрастания сложности.

Сразу писать на Ассемблере не удастся. Смиритесь. Для начала требуется познакомиться с чем-то попроще. Эзотерические языки - это языки программирования, созданные ради шутки или для проведения эксперимента по расширению программистских возможностей. Изучение более простых, чем Ассемблер, представителей этой языковой группы позволит упростить последующее понимание Ассемблера.

Чтобы познакомиться с творчеством разума земного и не очень можно прочесть статью ["Примеры кода на 39 эзотерических языках программирования"](#). Многого от них не ждите. **BF** - самый адекватный из них.

BrainF

Познакомьтесь с младшим братом Ассемблера: BrainF. [BrainF Developer](#).

В Интернете пишут, что **BF** - это язык программирования низкого уровня, который обладает высокой производительностью. Все это позволяет его запускать вне зависимости от платформы, лишь бы был интерпретатор языка. На JavaScript, Python или VBA - не важно. Интерпретатор, как правило, крайне легковесный. [Интерпретатор BF размером 160 байт](#).

Периодически задаются вопросы: "А есть ли в таком языке какой-то практический смысл?"

Ответы на подобные вопросы, как ни странно, замалчивают. Постараемся, как можем, на них ответить.

Начнем с анекдота.

"На просторах социальной сети ВКонтакте была высказана мысль: "В **BF** проще войти, чем в Ассемблер. Поэтому можно писать на framework, который компилирует из **BF** в конкретный Ассемблер. Такой путь от собеседования до первого сданного проекта намного короче."

В этот момент, где-то в далекой-далекой галактике появилась вакансия машинно-ориентированного программиста на **Java/BF**."

Почему именно **BF**?

- Максимальная простота: **BF** прост, учить в нем, в принципе, нечего, в нем всего 8 команд.
- Дает возможность поработать со стекком вызова функций и понять - что же это такое.

- Дает сильный стимул для развития алгоритмического мышления и неожиданно упрощает последующее понимание Ассемблера.
- Язык крайне недооценен, так как он очень прост в схемотехнической реализации. В теории, работающие на нем микропроцессоры будут ещё компактнее и проще.

Посмотрим - как выглядит **BF** :

- +- - инкремент, декремент,
- >< - смещение на ячейку выше, ниже,
- [] - цикл открыть, закрыть,
- „ - ввод, вывод одного символа.

Все! Можно начинать "ваять"! Если по каким-то причинам нас не устраивает [BrainF Developer](#), то всегда можно попробовать написать свой интерпретатор языка. Предупреждаю, существуют ASCII и Unicode модели символов, программы с их учетом могут различаться. В Интернете можно встретить Вызов/Чёллендж/Challenge, брошенный другим программистам: "Напиши интерпретатор **BF** на каждом языке программирования, который ты знаешь". Встречаются программные коды на C++ и C#. Реже на Java. Пополняю копилку **BF** интерпретаторов на языке Julia. Не стоит воспринимать это, как сумасшествие. Это всего лишь одна из задач на ночь, которые, ради тренировки, исполняют "кодеры".

[Try Jupyter with Julia](#). Вот программа, язык Julia:

```
In [7]: function BrainF(
    _Code::Base.String=
    """+++++
    +++++
    +++++
    +++++
    +++++
    """, _IsNeedListing::Core.Bool=false
    , _IsNeedFinListing::Core.Bool=false
    )
    ###Исправляем ошибку индексирования строки с разным типом Code.Char###
    ###Ошибка исправляется как ни странно преобразованием в массив.
    ###Потрачено 4 часа.
    _Code=_Code|>a->(_arr=Core.Char[];foreach((b::Core.Char)->push!(_arr,b),a);_arr;)
    #####
    _StrIn="";_StrOut="";
    ###Def Code###
    #_Code::Base.String;
    _CodeId=1; _CodeIdMax=_Code|>length;
    ###Def Mem###
    _MemId=1; _Mem=Core.UInt32[]; push!(_Mem,0);
    ###LoopStack###
    _LoopStack=Core.Any[]
    ###Listing###
    function BrainFackListing()
        "###BrainFListing###TargetOperatorWasExecuted###"|>println;
        "###Code###"|>println;
        for i=1:_CodeIdMax;if(i==_CodeId);print("{*_Code[i]*"}");else;print(_Code[i]);end
        "###Memory###"|>println;
        _MemIdMax=_Mem|>length;
```

```

for i=1:_MemIdMax; _p="";if(i==_MemId);_p=" "else;_p=" "end;
println("$(_p)_Mem[$(i)]=$( _Mem[i])<=$(convert(Core.Char, _Mem[i]))>");
end;
if(length(_StrOut)!=0);println("###Out###\n"*_StrOut);end;
if(length(_StrIn)!=0);println("###In###\n"*_StrIn);end;
if(length(_LoopStack)!=0);println("###LoopStack###");for i in _LoopStack; println
end;
###Work Part###
while (_CodeId <= _CodeIdMax)
    _char =convert(Core.Char, _Code[_CodeId]);
    if(
        (_char=='+')||(_char=='-')||(_char=='>')||(_char=='<')
        ||(_char=='.')||(_char==',' )||(_char=='[')||(_char==']')
    );
        if(_char=='+');_Mem[_MemId]=_Mem[_MemId]+1;end;
        if(_char=='-');_Mem[_MemId]=_Mem[_MemId]-1;end;
        if(_char=='>');_MemId=_MemId+1;while(_MemId>length(_Mem));push!(_Mem,0);end;e
        if(_char=='<');_MemId=_MemId-1;if(_MemId<1);_MemId=length(_Mem) end;end;
        if(_char=='.');
            _charPrint=convert(Core.Char, _Mem[_MemId])
            print(_charPrint);
            _StrOut=_StrOut*_charPrint;
        end;
        if(_char==',' );
            while(length(_StrIn)==0);_StrIn= readline();end;
            _Mem[_MemId]=convert(Core.UInt32, _StrIn[1])
            _StrIn=_StrIn[2:length(_StrIn)]
        end;
        if(_char=='[');push!(_LoopStack,(_CodeId=_CodeId,_MemId=_MemId))end;
        if(_char==']');
            var=pop!(_LoopStack)
            if(_Mem[var._MemId]!=0);push!(_LoopStack,var);_CodeId=var._CodeId;end;
        end;
        ###Listing###
        if(!false);if(_IsNeedListing);BrainFackListing();end;end;
    end;
    _CodeId=_CodeId+1;
end;
if(!false);if(_IsNeedFinListing);BrainFackListing();end;end;
end;
BrainF("">>+++++++[-<+++++]<+++++++.....++++
+++-----.....+++++-----
[-]+++++++.....")

```

HelloWorld!!!

Как видите, все символы, кроме тех 8 команд игнорируются. Давайте разберемся с синтаксисом. У BF, как правило, есть режим пошаговой отладки и вывод конечного снимка памяти. В нашем случае это 2 флага, соответственно.

Для упрощения интерпретации снимка памяти выводятся все её ячейки в виде индекса, числового содержимого и соответствующего этому коду символа.

В случае нехватки ячеек памяти осуществляется динамическое их выделение. Ячейки не высвобождаются!!! Это заложено в стандартах языка.

Текущая ячейка помечается либо звёздочкой, либо обратным отступом.

In [9]: BrainF("">>+++++++[-<+++++]<+++++++.....++++

```

###BrainFListing###TargetOperatorWasExecuted###
###Code###

```

```
###Memory###
_Mem[1]==0==<>
```

Вот полный листинг программы занесения двух единиц в ячейку с последующим обнулением при помощи цикла.

```
In [11]: BrainF("""+[-]""", !false, true)

###BrainFListing###TargetOperatorWasExecuted###
###Code###
{+}+[-]
###Memory###
_Mem[1]==1==<>
###BrainFListing###TargetOperatorWasExecuted###
###Code###
+{+}[-]
###Memory###
_Mem[1]==2==<>
###BrainFListing###TargetOperatorWasExecuted###
###Code###
++{[]-}
###Memory###
_Mem[1]==2==<>
###LoopStack###
(_CodeId = 3, _MemId = 1)
###BrainFListing###TargetOperatorWasExecuted###
###Code###
++[{ -}]
###Memory###
_Mem[1]==1==<>
###LoopStack###
(_CodeId = 3, _MemId = 1)
###BrainFListing###TargetOperatorWasExecuted###
###Code###
++{[]-}
###Memory###
_Mem[1]==1==<>
###LoopStack###
(_CodeId = 3, _MemId = 1)
###BrainFListing###TargetOperatorWasExecuted###
###Code###
++[{ -}]
###Memory###
_Mem[1]==0==<>
###LoopStack###
(_CodeId = 3, _MemId = 1)
###BrainFListing###TargetOperatorWasExecuted###
###Code###
++[-{]}
###Memory###
_Mem[1]==0==<>
###BrainFListing###TargetOperatorWasExecuted###
###Code###
++[-]
###Memory###
_Mem[1]==0==<>
```

Пример задачи, копирование числа через цикл.

```
In [18]: BrainF("""+++[->+>+<<]""", false, true)

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+++[->+>+<<]
```

```

###Memory###
_Mem[1]==0==<>
_Mem[2]==3==<2>
_Mem[3]==3==<2>

```

Обратите внимание, что текущий экземпляр числа при копировании уничтожается. Поэтому при копировании нужно создавать минимум две копии, одну из которых позже возвращать на исходное место.

```
In [19]: BrainF("""+++[->+>+<<]>>[-<<+>>]""",false,true)
```

```

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+++[->+>+<<]>>[-<<+>>]
###Memory###
_Mem[1]==3==<2>
_Mem[2]==3==<2>
_Mem[3]==0==<>

```

Обратите внимание, что данный цикл - это цикл с предусловием. Он же используется вместо ветвления. Блок else у ветвления не предусмотрен.

Следует отметить, что есть отдельная группа часто встречающихся задач, которая требует умения перемножать числа.

Пример, получение в ячейке числа 15. Какой из вариантов короче?

```
In [12]: BrainF(""">+++[-<++++>]<""",false,true)
```

```

###BrainFListing###TargetOperatorWasExecuted###
###Code###
>+++[-<++++>]<
###Memory###
_Mem[1]==10==<
>
_Mem[2]==0==<>

```

```
In [13]: BrainF("""+++++ +++++ +++++""",false,true)
```

```

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+++++ +++++ +++++
###Memory###
_Mem[1]==15==<2>

```

Таким образом, Вы можете довольно компактным программным кодом вычислить $2 * 3 * 5 * 7$, но для этого потребуется соответствующее число ячеек памяти.

```
In [32]: BrainF("""+[->+++[->+++++[->++++++<]<]<]""",false,true)
```

```

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+[->+++[->+++++[->++++++<]<]<]

###Memory###
_Mem[1]==0==<>
_Mem[2]==0==<>
_Mem[3]==0==<>
_Mem[4]==210==<0>

```

```
In [34]: BrainF("""+[->+++[->+++++[->++++++<]<]<]
>>>[-<<<+>>>]<<<
""",false,true)
```

```

###BrainFListing###TargetOperatorWasExecuted###
###Code###
++[->+++[->+++++[->++++++<]<]<]
>>>[-<<<+>>>]<<<

###Memory###
_Mem[1]==210==<0>
_Mem[2]==0==<>
_Mem[3]==0==<>
_Mem[4]==0==<>

```

Ещё одна задача. Требуется найти в тексте первую букву русского алфавита.

Есть несколько подходов к её решению. Подход первый: метод простого перебора.

Формируем окно для просмотра из "+.". Предварительно при помощи крупных множителей прибавляем к точке старта.

Процесс сильно ускоряется, если заранее посмотреть искомое значение по таблице, например, ASCII кодов.

In [70]:

```

BrainF("""
Предварительно с помощью крупных множителей компактно приближаемся к искомому числу
+++++++[->+++++++[->+++++++<]<]
>>[-<<+>>]<<
Осуществляем приближение с небольшим шагом
>++++[-<+++++++>]<
Даем мелкую доводку без циклов

Выводим окно для просмотра результата
.+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
.+.+.+.+.+.+.
""", false, true)

```

```

АБВГДЕЖЗИЙКЛМНОПРСТУФХЦШЩЪЫЬЭЮЯабвгдезийклмнопрстуфхцшщъыьэяё###BrainFListing###Target
OperatorWasExecuted###
###Code###
Предварительно с помощью крупных множителей компактно приближаемся к искомому числу
+++++++[->+++++++[->+++++++<]<]
>>[-<<+>>]<<
Осуществляем сриближение с небольшим шагом
>++++[-<+++++++>]<
Даем мелкую доводку без циклов

Выводим окно для просмотре результата
.+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
.+.+.+.+.+.+.

###Memory###
_Mem[1]==1104==<è>
_Mem[2]==0==<>
_Mem[3]==0==<>
###Out###
АБВГДЕЖЗИЙКЛМНОПРСТУФХЦШЩЪЫЬЭЮЯабвгдезийклмнопрстуфхцшщъыьэяё

```

В подобных случаях нередко прибегают к макросам, которые выполняют часть работы за нас. Напишем макрос для получения русского символа А, код 1040 = 10813.

In [89]:

```

#Вставка получения русского символа А
BF_Get_RussianA="""BF_Get_RussianA(10+++++++[->8+++++++[->13+++++++<]<])ВозвратВ

```

```
BrainF("""$(BF_Get_RussianA)""", false, true)
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
BF_Get_RussianA(10+++++++[->8+++++++[->13+++++++<]<]ВозвратВНачало>>[-<<+>>]<<)
###Memory###
_Mem[1]==1040==<A>
_Mem[2]==0==<>
_Mem[3]==0==<>
```

Можно написать макрос вывода вообще любых текстов. Только он будет работать в разы медленнее, так как работает не по принципу "смещение от предыдущего", а по принципу "каждый символ печатаем заново и в лоб".

Можно написать разложение каждой разницы относительно предыдущего символа на простые множители. И сделать вывод на экран. Только займет это неразумно много времени...

```
In [100]: function Get_TXT(_str)
    _codeBF="";
    foreach(
        (x::Core.Char)->(
            rez="";
            for i=1:convert(Core.Int64, x);
                rez=rez*"+";
            end;
            _codeBF=_codeBF*rez*"."[-]$(x)\n";
        )
    ,_str
    );
    return _codeBF;
end;
BrainF("""$(Get_TXT("Привет мир!!!"))""", false, true)
```

```
Привет мир!!!###BrainFListing###TargetOperatorWasExecuted###  
###Code###
```

[illegible]

[illegible]


```
-----
-----
-.5----.1+++++.9[-]
)
```

```
###Memory###
_Mem[1]==0==<>
###Out###
Калашников Сергей 2ИСИП_519
```

На этом знакомство с **BF** можно считать состоявшимся.

BF - может стать хобби некоторых учащихся.

Для достижения цели "познакомиться с **BF**" достаточно одного или двух заданий.

Дополнительные задачи - на усмотрение преподавателя.

Практическая работа №2. **BF**, как меня зовут

- Уровень сложности низкий.
Написать программу вывода ФИО номер группы в консоль. # Творческие задания повышенной сложности
- Уровень сложности ниже среднего.
Нарисовать заданный рисунок псевдографикой и вывести в консоль.
- Уровень сложности средний.
Написать шифрование/дешифрование текстового файла в BrainF файл и обратно.
- Уровень сложности достаточный.
Сложение двух одноразрядных чисел с вводом из консоли. Результат вернуть в начальную ячейку, остальные ячейки за собой подчистить. Перенос на старший разряд не учитывается.
- Уровень сложности высокий.
Сложение двух двухразрядных чисел с вводом из консоли. Результат вернуть в 2 начальные ячейки, остальные ячейки за собой подчистить. Перенос на старший разряд не учитывается.
- Уровень сложности очень высокий.
Реализовать бесконечный четырехразрядный счетчик с прибавлением единицы и выводом в консоль.
- Уровень сложности "Мечта программистов из Интернета".
Написать морской бой.

Распространенные ошибки

Какие-то конкретные ошибки выявить сложно, по причине того, что исполнитель языка не подразумевает синтаксических ошибок, только семантические. Семантические ошибки возникают как следствие невнимательности или недостатка опыта.

In []:

Примерный план проблемной лекции

ЛамПанель, или школьный ассемблер.

Время – от 2 до 4 часов, из них лекция - 2 часа.

Учебные цели занятия: познакомиться с обучающей программой ЛамПанель для языка Ассемблер. По образцу, в 3 этапа, начать писать программный код, получить положительный опыт от общения и написания несложных программ в ЛамПанель.

Форма проведения занятия: проблемная лекция.

Учебные задания:

- Этап первый.
 - 1) Пошагово разобрать у доски или на проекторе приведенный пример.
 - 2) Нарисовать в тетради или в EXCEL свой пиксельный рисунок 8 на 16 клеток.
 - 3) При помощи таблицы перевода перевести из бинарной системы исчисления каждую строку в шестнадцатеричную систему исчисления.
 - 4) При помощи таблицы перевода перевести из бинарной системы исчисления каждую строку в шестнадцатеричную систему исчисления. Вспомнить таблицу перевода систем исчисления.
- Этап второй
 - 1) Прочитать теоретическую справку или мануал ЛамПанель.
 - 2) Подставить полученные шестнадцатеричные коды в пример, запустить, получить рисунок в лампочках портов, исправить ошибки.
- Этап третий
 - 1) Прочитать теоретическую справку или мануал ЛамПанель.
 - 2) При помощи сдвигов, перестановок портов с использованием любого регистра сделать анимацию. Одновременное циклическое движение, на выбор: снизу вверх или сверху вниз; слева направо или справа налево.

Инструкции по проведению и ходу занятия: В начале занятия обосновываются цель и задачи проблемной лекции, определяется проблема для обсуждения. Преподаватель излагает материал, задает вопросы студентам, посвященные проблематике лекции, затем начинается дискуссия по поставленной проблеме; вторая половина занятия - групповая практика на ПК.

Методические рекомендации: соблюдайте последовательность действий; не перескакивайте на следующий этап, не пройдя предыдущий; обратите внимание - есть секретный компактный способ решения поставленной задачи через системные процедуры (system [16 ричный код процедуры]), об этом не подсказывайте учащимся, это награда за внимательное прочтение методички ЛамПанели.

Вопрос для самоконтроля:

Учим команды: mov,in, out, jmp, :metko,system,ror,rol,shr,shl и так далее... - все команды, что

встречаются в программном коде.

С чего начать?

Большой проблемой является поиск актуальной литературы на тему "Программирование на Ассемблере".

Для решения этого вопроса рекомендую посетить сайты, названные ниже.

Изучать будем FASM.

Как писать на Ассемблере в 2018 году?

В этой статье приводится соблазнительно большой список литературы, статья может претендовать на краткий обзор вопроса.

Вдобавок, на основании этой статьи можно сказать, что приличных пакетов под Ассемблер осталось "раз два и обчелся"...

Среди них:

- **MASM** - много версий, документация запутана, требует установки, нужна лицензия... Нередко можно встретить источник литературы по уже несуществующей версии, так что: "Удачи!"...
- **TASM** - когда-то был в приличном состоянии. Но сейчас ... он не обновлялся со времен 32 битного Borland, а это примерно 2008 год.
- Следующий в списке - **FASM** - когда-то был темной лошадкой. На сегодняшний момент заслуженно теснит MASM. FASM распространяется свободно, не требует установки, и, что удивительно, по сей день активно обновляется. К тому же он невероятно складно и просто синтаксически устроен. При скачивании с официального сайта идет вместе с примерами. Есть актуальный, недавно обновленный, программистский мануал. Всего этого по-прежнему недостаточно для начала работы. Потому попробуем создать начальное ускорение.

Поэтому предложим свой взгляд на ситуацию...

- Имеет смысл посетить сайт [FasmWorld](#). Он посвящен тренировке навыков программирования на FASM под DosBox. Там высказана мысль: "Прежде чем браться за Windows, можно поиграть с DosBox". Из этого этапа изучения Ассемблера получается неплохой экскурс в историю. Тем не менее, материал хоть и обладает редким качеством достоверности изложения, на практике половине учащихся трудно и почти невозможно изучить его в том варианте, в котором он предложен на сайте. Большие объемы теории, резко "сваливающиеся на голову", демотивируют. По этой причине будем FasmWorld использовать как дополнительный материал высокого качества.
- На одном из форумов был задан вопрос: "А есть ли руссифицированный мануал программиста под FASM?" Ответ был таков: "Нет, но подожди, сейчас перегоню через онлайн переводчик..." После этого энтузиасты, с различной степенью отклонения от исходного текста, с применением личной интерпретации, с минимальным объяснением -

как этим пользоваться, стали выкладывать в Интернет свои варианты переводов... Я насчитал 2,3 или 4 таких варианта. Каждый из них не охватывает общую картину, но тем не менее хочется предложить один из наиболее приличных вариантов. [Мануал программера.flat assembler 1.71](#)

- В тот момент, когда программки в стиле "Привет, мир" остались позади, возникает вопрос: "Как со всем этим обращаться? Как не тратить время на часто повторяющиеся проблемы?" Именно в этот момент в дело вступают макросы. [Руководство по препроцессору FASM](#). В программировании 2000 годов в литературе часто встречается антипатерн "GoldenHammer" или "золотой молоток", что означает решение всех проблем одним способом. Антипатерны являются обратной стороной медали паттернов проектирования. Хотя в литературе и повторяют заученную фразу: "Только паттерны и никаких антипаттернов", - разумное применение любых приемов ещё никому не вредило! В варианте FASM, судя по всему, пошли по пути максимального упрощения, и очень талантливо решили почти все проблемы при помощи макросов. Да, FASM - только компилятор, не литнкер. Не объединяет заранее скомпилированные файлы. Зачем такие сложности! Проблема решается через препроцессинг, макросы и подшивание *.Inc файлов в один на этапе перед подачей на компиляцию. Получился поражающий простотой и универсальностью инструментарий. Можно сказать, что налицо неканоническое применение антипатерна "золотой молоток". Если освоение макросов удастся, то оно превратит процесс написания программных кодов Ассемблера из решения ребуса в увлекательное программирование на близком аналоге императивного языка высокого уровня. Вы удивитесь, узнав, что так было не всегда. Копнув чуть глубже Интернет-форумы, можно узнать, что MASM пестрит разношерстным синтаксисом в различных версиях на тему линкинга, предкомпиляции и сегментации памяти программы. Возможно, под FASM встречается куда больше примеров работающего программного кода, чем под MASM. В этом плане на экзамене легко узнать, кто из студентов отлынивал от работы. Некоторые граждане находят программный код, тасуют его, не глядя, и пытаются сдавать... При этом очень сильно удивляются на то, что он ещё должен компилироваться без ошибок и работать. Дело доходит до того, что сразу видно - какие блоки размером примерно в 10-15 строк взяты из FASM, а какие из MASM. **Делаем вывод. Программный код ничего не стоит, если он не компилируется и не запускается!!!** Именно этому вопросу мы и собираемся уделить особое внимание!!!

Практическое задание №3. Мой первый рисунок в ЛамПанель.

Основная проблема юного ассемблериста - это вывод на экран состояний регистров.

Чтобы это "провернуть", требуется средний уровень владения языком (не меньше).

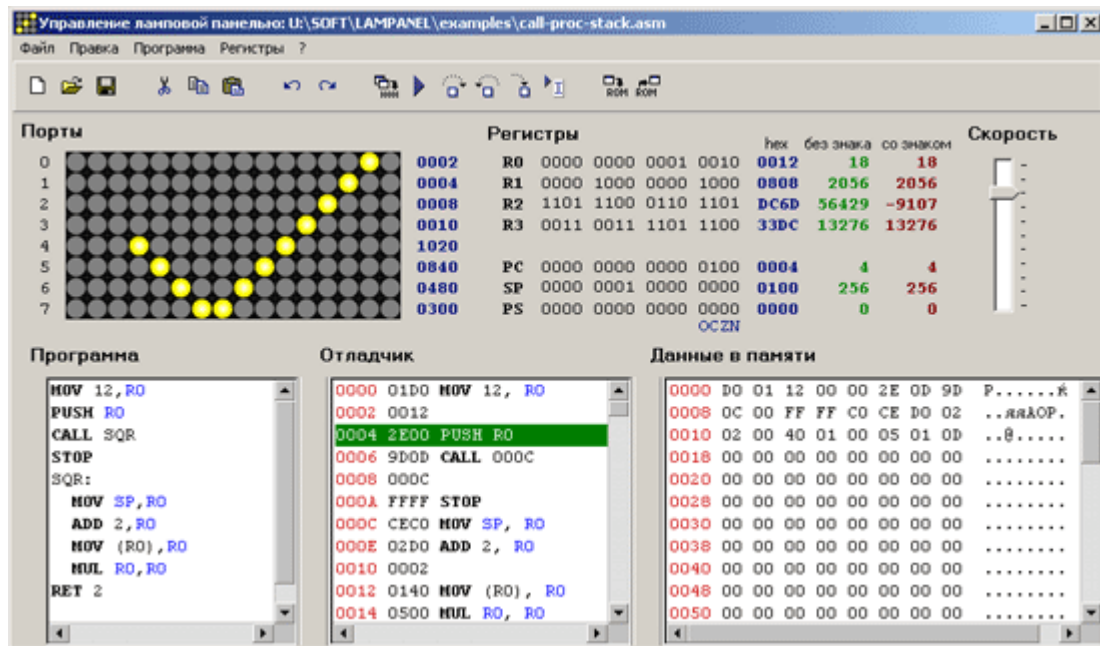
Это первая из преград, которая преодолевается нахождением адекватной среды разработки.

EMU8086 - хотелось бы сказать, но за прошедшее десятилетие среда стала платной.

Free Pascal 3.0/Lazarus/Delphi - хотелось бы сказать, но среда не является мейнстримом дня сегодняшнего.

Visual Studio 2019/C++ - тоже обладает своими неочевидными с первого взгляда недостатками, но пользоваться мы ей будем, хоть и не с самого начала...

А потому, встречайте: [ЛамПанель](#).



Сразу скажу, ЛамПанель ни на что не способна, кроме визуализации... Что на определенном этапе себя оправдывает.

Есть русскоязычный хелп на 2 страницы.

Есть *.PDF на 14 страниц.

Есть все, что требуется для изучения.

- Задание. Часть первая.

Нарисовать авторскую картинку.

- Задание. Часть вторая.

Сделать анимацию. Циклический сдвиг снизу вверх, слева направо.

Студентам можно сказать, что есть:

- 4 регистра, r0, r1, r2, r3;
- 8 портов для работы с внешними устройствами (имитация), p0, p1, p2, p3, p4, p5, p6, p7;
- in/out - пересылка машинных слов между регистрами и портами;
- mov - команда пересылки между регистрами и оперативной памятью (об этом подробнее позже);
- есть системные процедуры, которые можно посмотреть как "программа посмотреть ПЗУ".

Подробнее о том, как сделать это...

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | | DEC | HEK | BIN | | 0 | 1 | 2 | 3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-----|-----|------|------|---|---|---|---|---|-------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0000 | | 0 | 0 | 1 | 8 | 0 | mov 0180,r0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0001 | | 1 | 0 | 2 | 4 | 0 | out r0,p0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | 2 | 0010 | | 2 | 0 | 5 | A | 0 | mov 0240,r1 |
| 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | | 3 | 0011 | | 3 | 0 | 8 | 1 | 0 | out r1,p1 |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | | 4 | 0100 | | 4 | 1 | F | F | 8 | mov 05A0,r2 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | 5 | 0101 | | 5 | 1 | 0 | 0 | 8 | out r2,p2 |
| 6 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | 6 | 0110 | | 6 | 1 | 0 | 0 | 8 | mov 0810,r3 |
| 7 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | | 7 | 0111 | | 7 | 1 | F | F | 8 | out r3,p3 |
| | | | | | | | | | | | | | | | | | | | | 8 | 1000 | | | | | | mov 1FF8,r0 |
| | | | | | | | | | | | | | | | | | | | | 9 | 1001 | | | | | | out r0,p4 |
| | | | | | | | | | | | | | | | | | | | | 10 | A | | | | | | mov 1008,r1 |
| | | | | | | | | | | | | | | | | | | | | 11 | B | | | | | | out r1,p5 |
| | | | | | | | | | | | | | | | | | | | | 12 | C | | | | | | mov 1008,r2 |
| | | | | | | | | | | | | | | | | | | | | 13 | D | | | | | | out r2,p6 |
| | | | | | | | | | | | | | | | | | | | | 14 | E | | | | | | mov 1FF8,r3 |
| | | | | | | | | | | | | | | | | | | | | 15 | F | | | | | | out r3,p7 |

0123456789ABCDEF

0

1

2

3

4

5

6

7

0

1

2

3

4

5

6

7

Создаем файл Excel, делаем эскиз рисунка из нулей и единиц, переводим при помощи таблиц в 16-ричный формат (слева направо). Осталось подставить в программный код.

Пример программного кода

```

mov 0180,r0
out r0,p0
mov 0240,r1
out r1,p1
mov 05A0,r2
out r2,p2
mov 0810,r3
out r3,p3
mov 1FF8,r0
out r0,p4
mov 1008,r1
out r1,p5
mov 1008,r2
out r2,p6
mov 1FF8,r3
out r3,p7

```

```

m:
;ДВИГАЕМ КРЫШУ
IN P0,R0
ROR 1,r0
OUT R0,P0
;
IN P1,R0
ROR 1,r0
OUT R0,P1
;
IN P2,R0
ROR 1,r0
OUT R0,P2

```

```
;
IN P3,R0
ROR 1,r0
OUT R0,P3
;
IN P4,R0
ROR 1,r0
OUT R0,P4
;
IN P5,R0
ROR 1,r0
OUT R0,P5
;
IN P6,R0
ROR 1,r0
OUT R0,P6
;
IN P7,R0
ROR 1,r0
OUT R0,P7
;
JMP m
```

stop

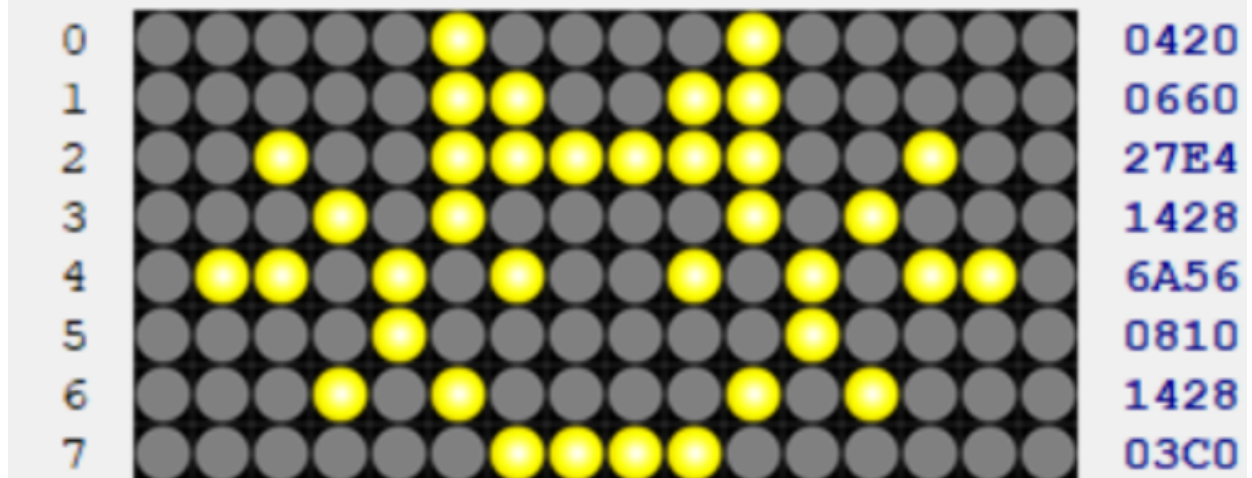
Творческая составляющая

Приветствуется реализация своей оригинальной картинки.

Кто-то из студентов подходит к вопросу творчески. Вот, например, котик, при циклическом сдвиге которого на отдельном кадре появляются "рожки". Это особенность картинки. Список

возможных картинок ограничен только фантазией.

Порты



Если возникают трудности, то задание может выполняться парами студентов.

ЛамПанель также может работать с оперативной памятью, но оставим подобные вопросы до более интересного языка.

Задания хватит на несколько часов.

В дальнейшем работает правило: если что-то не получается, то можно вернуться назад, к предыдущей среде разработки, что-либо отработать там и идти дальше.

Распространенные ошибки.

In []:

- Попытка "закодировать" рисунок методом перебора 16-ричных кодов - превращает работу из одночасовой в 3 часовую.
- Попытка запуска программы из архива, не распаковывая. Как следствие ошибки при выполнении системных процедур нет возможности загрузки русскоязычной справки.
- Попытка использовать несуществующие регистры, их только 4: r0-r3. Вызывать r4 уже бесполезно. # Хорошая практика
- Группировка однотипно используемых команд за счет использования большего числа регистров. В теории 64 битные системы должны приводить к параллельному исполнению независимых ветвей кода. К тому же, работа с мелкими блоками команд затрудняет понимание кода.

```
mov 0180,r0  
mov 0240,r1  
mov 05A0,r2  
mov 0810,r3
```

```
out r0,p0  
out r1,p1  
out r2,p2  
out r3,p3
```

```
mov 1FF8,r0  
mov 1008,r1  
mov 1008,r2  
mov 1FF8,r3
```

```
out r0,p4
```

```
out r1,p5
```

```
out r2,p6
```

```
out r3,p7
```

</div>

- Группировка однотипно используемых команд за счет использования большего числа регистров. Работа с мелкими блоками команд затрудняет понимание.

Примерный план проблемной лекции

C++, ASM вставки, команды пересылки данных, дизассемблер, арифметические операции

Время на изучение темы – от 6 до 8 часов, из них лекция - 2 часа.

Учебные цели занятия: освоение написания ASM вставок C++, с использованием команд пересылки данных, обращение к оперативной памяти по указателю, смещения, арифметика указателей.

Форма проведения занятия: проблемная лекция.

Учебные задания:

- Этап предварительной подготовки.
 - Дается задание сделать конспект и доклад, каждому из учащихся достаются 2 или 3 команды языка Ассемблер. После ЛамПанели у учащихся есть некоторый опыт. Поэтому прежде чем что-то обсуждать на паре, хорошо бы было самостоятельно найти и разобрать материал из Интернета, желательно аккумулировать его из 2 или 3 источников. Не ставится цель сделать все идеально, ставится цель просто начать поиск и разбор "самостоятельно".
- Этап проблемной лекции. Преподавателем дается теоретический материал, работающие программные коды - как ответы на вопросы, поставленные в процессе обсуждения. От учащегося требуется запустить код, разобраться в его работе, найти в Интернете ответы на возникшие вопросы или задать вопрос преподавателю.
 - Пересылка значений между регистрами.
 - Пересылка значений между регистрами и ячейкой оперативной памяти по указателю.
 - C++, работа с указателями, арифметика указателей, работа с массивами.
 - ASM, работа с указателями, арифметика указателей, работа с массивами. Помните, в ASM нет локальных переменных, все они заменяются на указатель, адрес оперативной памяти, что хранит значение.
 - ASM/C++, примеры программ обмена значениями в разных вариациях.
 - ASM, размещение значений регистра флагов в стеке, извлечение значения с вершины стека.
- Этап практического задания. -Выполняются по 2 варианта на учащегося. Допустимо командное выполнение, например, команда из 2 человек делает 4 варианта. .

Инструкции по проведению и ходу занятия: В начале занятия обосновываются цель и задачи проблемной лекции, определяется проблема для обсуждения. Преподаватель излагает материал, задает вопросы студентам, посвященные проблематике лекции, затем начинается дискуссия по поставленной проблеме; вторая половина занятия - групповая практика на ПК.

Методические рекомендации: есть два варианта: или спрашивать наизусть команды пересылки данных по списку, что приведен ниже, или идти в сторону большего объема практики, то есть выполнять по 2 варианта заданий каждому.

Вопрос для самоконтроля:

Для чего используется команда: mov, push, pop, xchg, pushf, popf, xlat, lea, lds, les, lahf, sahf ?

Что такое арифметика указателей?

Как из указателя получить значение переменной, на которую он указывает на ASM?

Как получить 3 элемент массива на asm?

Как узнать адрес переменной в оперативной памяти на с++?

Машинные слова и их размеры?

Арифметические операции: ADD, ADC, INC, SUB, SBB, DEC, CMP, MUL, IMUL, DIV, IDIV, NEG, CBW, CWD.

Начнем

Наверняка, у каждого есть "любимая книга, с которой все началось".

Для меня по данному направлению такой книгой стала:

- Архитектура ЭВМ. Задания и примеры выполнения лабораторных работ. Методические указания/ сост.: А.Е. Докторов, Е. А. Докторова. - Ульяновск : УлГТУ, 2008. - 32 с.

Чем примечательны эти методические указания?

- В них предлагается короткий путь вхождения в Ассемблер. Всего 32 с., из них 18 страниц посвящено теории написания ассемблерных вставок на Pascal 2.6. Остальное же - инструкции и задания к лабораторным работам.
- Подобный подвиг компактного изложения материала не удалось повторить ни одному автору, чьи книги мне попадались.
- Хотя Pascal 2.6. потерял свою актуальность, но теория и задания переносимы и на другие среды.

Архитектура ЭВМ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

**ДЛЯ СТУДЕНТОВ СПЕЦИАЛЬНОСТИ
ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ**

Сост.: А. Е. ДОКТОРОВ
Е. А. ДОКТОРОВА

Эта книга содержит таблицы команд Ассемблера, по которым вполне можно строить дальнейшие планы по изучению языка. Только не все из этих команд ныне работают...

| | | | | | | |
|---|---------|-------|--------|-------|------|------|
| ПЕРЕСЫЛКА ДАННЫХ | | | | | | |
| MOV | PUSH | POP | XCHG | PUSHF | POPF | |
| XLAT | LEA | LDS | LES | LAHF | SAHF | |
| АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ | | | | | | |
| ADD | ADC | INC | SUB | SBB | DEC | CMP |
| MUL | IMUL | DIV | IDIV | NEG | CBW | CWD |
| ЛОГИЧЕСКИЕ ОПЕРАЦИИ | | | | | | |
| NOT | SHL/SAL | SHR | SAR | ROL | ROR | |
| RCL | RCR | AND | TEST | OR | XOR | |
| ОБРАБОТКА БЛОКОВ ДАННЫХ | | | | | | |
| REP | REPE | REPNE | REPZ | REPNZ | | |
| CMPSB | LODSB | MOVSB | SCASB | STOSB | | |
| CMPSW | LODSW | MOVSW | SCASW | STOSW | | |
| КОМАНДЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ | | | | | | |
| CALL | JMP | RET | | | | |
| КОМАНДЫ УСЛОВНОГО ПЕРЕХОДА | | | | | | |
| JZ | JO | JP | JS | JC | JA | JB |
| JNZ | JNO | JNP | JNS | JNC | JNA | JNB |
| JE | | JPE | | | JNAE | JBE |
| JNE | | JPO | | | JAЕ | JNBE |
| | | | LOOP | JCXZ | | |
| JL | JG | | LOOPE | | | |
| JNL | JGE | | LOOPNE | | | |
| JLE | JNGE | | LOOPZ | | | |
| JNLE | | | LOOPNZ | | | |
| УПРАВЛЕНИЕ СОСТОЯНИЕМ ПРОЦЕССОРА | | | | | | |
| CLC | CMC | STC | CLD | STD | NOP | |

Команды пересылки данных

Итак, приступим. Первая серьёзная лабораторная работа. Если материал изучается на том или ином этапе, знайте, что позже вернуться к нему возможности не будет.

Почему? Потому что изучение глубин Ассемблера перспективами уходит в бесконечность.

Материала много, самостоятельно его почти не изучить, малейшая безалаберность на уроке приводит к почти полному выпаданию из последующих тем курса. Дабы это предотвратить выполняется по 2 задания на человека. Разрешено командное выполнение, при условии - по 2 задачи на каждого члена команды.

Почему так? Если этого не сделать, то можно можно почти сразу получить стадию бесконечного повторения предыдущего материала.

Если на данной лабораторной работе в группе это не предотвратить, то до конца курса добросовестно доходят единицы.

Выполняется на Visual Studio/c++

Практическая работа №4. Команды пересылки данных.

Варианты заданий.

- 1) Обменять значения в переменных $\text{int } x$ и $\text{int } y$, где $\text{int } y$ - указатель.
- 2) Обменять значения в переменных $x[4]$ и $y[3]$, где y - указатель на элемент массива.
- 3) Обменять значения в переменных $x[4]$ и $y[3]$, где y - указатель на элемент массива. Используйте команды PUSH и POP для временного хранения элементов массива в стеке.
- 4) Сделать то же самое с использованием команды LEA.
- 5) Используя команды пересылок, покажите, как работает команда CMC.
- 6) Содержимое регистра флагов поместить в переменную $\text{int } x$.
- 7) Обменять значения в переменных $\text{int } x$ и $\text{int } y$, где $\text{int } y$ - указатель. При этом использовать команду XCHG.

Перед лабораторной работой рекомендуется дать задание студентам на самостоятельный поиск информации по командам пересылки данных в Интернете или других источниках.

ПЕРЕСЫЛКА ДАННЫХ

MOV
XLAT

PUSH
LEA

POP
LDS

XCHG
LES

PUSHF
LAHF

POPF
SAHF

Теория

Предком современного Ассемблера является Intel 8086. Поэтому регистры общего назначения называются A, B, C, D. При этом они хранят машинные слова Byte (8 бит), Word (16 бит), DWord (32 бит).

Вот более полная таблица, взятая из "Tomasz Grysztar. Flat assembler 1.73 Programmer's Manual."

| Operator | Bits | Bytes |
|----------|------|-------|
| byte | 8 | 1 |
| word | 16 | 2 |
| dword | 32 | 4 |
| fword | 48 | 6 |
| pword | 48 | 6 |
| qword | 64 | 8 |
| tbyte | 80 | 10 |
| tword | 80 | 10 |
| dqword | 128 | 16 |
| xword | 128 | 16 |
| qqword | 256 | 32 |
| yword | 256 | 32 |
| dqqword | 512 | 64 |
| zword | 512 | 64 |

Table 1.8: Size operators.

[Официальный сайт.](#)

Ассемблерные команды могут быть без операндов, могут обладать одним операндом, могут обладать двумя операндами.

mov[приемник], [источник]

.

[приемник] - операнд, в который помещается машинное слово.

[приемник] - может быть только регистром.

[источник] - операнд, из которого изымается машинное слово.

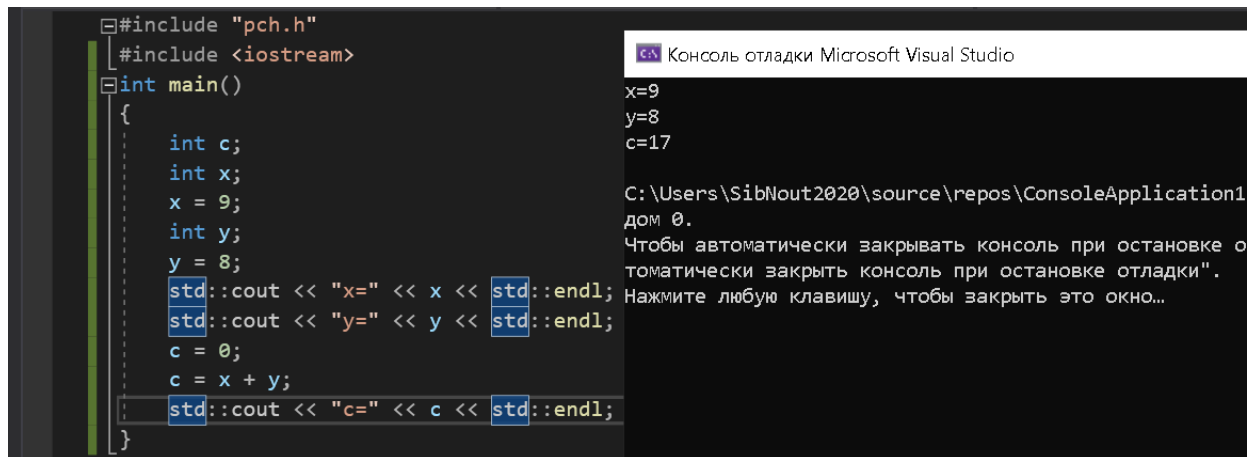
Рассмотрим регистр А.

- RAX - 64 битный регистр
- EAX - 32 битный регистр. Нижняя половина регистра RAX.
- AX - 16 битный регистр. Нижняя половина регистра EAX.
- AL - 8 битный регистр. Нижняя половина регистра AX.
- AH - 8 битный регистр. Верхняя половина регистра AX.

С остальными регистрами общего назначения - по аналогии.

Ожидается, что учащийся обладает навыками C++.

Пример первый



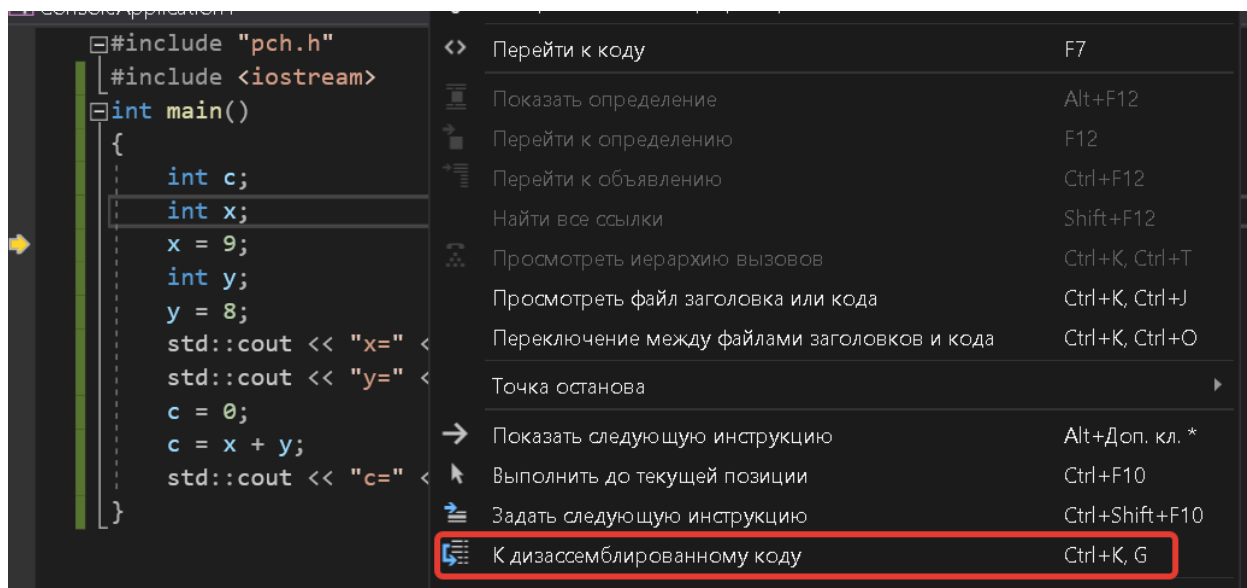
```
#include "pch.h"
#include <iostream>
int main()
{
    int c;
    int x;
    x = 9;
    int y;
    y = 8;
    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
    c = 0;
    c = x + y;
    std::cout << "c=" << c << std::endl;
}
```

Консоль отладки Microsoft Visual Studio

```
x=9
y=8
c=17

C:\Users\SibNout2020\source\repos\ConsoleApplication1
дом 0.
Чтобы автоматически закрывать консоль при остановке о
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно...
```

Этот программный код обменивает местами значения двух обычных переменных. Поскольку на начальных этапах у любого юного ассемблера возникают многочисленные проблемы, мы пойдем по пути дезассемблера. Если не знаете как писать что-то, то напишите это на с++ (как мы сделали сейчас). Нажмите F10 (пошаговое исполнение программного кода), правой кнопкой мыши на самом программном коде вызываем контекстное меню, в котором выбираем "Перейти к дезассемблированному коду".



Откроется вкладка "Дизассемблированный код".

```
In [ ]: #include "pch.h"
#include <iostream>
int main()
{
    int c=0;
012428C8 push     ebp
012428C9 mov      ebp,esp
012428CB sub      esp,30h
012428CE cmp      dword ptr ds:[11B42F0h],0
012428D5 je      <Module>.main()+014h (012428DCh)
012428D7 call    72DEFD80
012428DC xor      edx,edx
012428DE mov     dword ptr [ebp-4],edx
012428E1 xor      edx,edx
```

```

012428E3 mov     dword ptr [ebp-0Ch],edx
012428E6 xor     edx,edx
012428E8 mov     dword ptr [ebp-8],edx
012428EB xor     edx,edx
012428ED mov     dword ptr [ebp-4],edx
        int x;
        x = 9;
012428F0 mov     dword ptr [ebp-0Ch],9
        int y;
        y = 8;
012428F7 mov     dword ptr [ebp-8],8
        std::cout << "x=" << x << std::endl;
012428FE mov     ecx,dword ptr [__imp_std::cout (0767084h)]
01242904 mov     edx,767450h
01242909 call    dword ptr [Указатель на CLRStub[MethodDescPrestub]@352c5db101240851]
0124290F mov     dword ptr [ebp-10h],eax
01242912 mov     ecx,dword ptr [ebp-10h]
01242915 mov     edx,dword ptr [ebp-0Ch]
01242918 call    <Module>.std.basic_ostream<char,std::char_traits<char> >.<<(std.bas
0124291D mov     dword ptr [ebp-14h],eax
01242920 mov     edx,dword ptr [__unep@??$endl@DU?$char_traits@D@std@@@std@@$FYAAAV
01242926 mov     ecx,dword ptr [ebp-14h]
01242929 call    <Module>.std.basic_ostream<char,std::char_traits<char> >.<<(std.bas
0124292E mov     dword ptr [ebp-18h],eax
01242931 nop
        std::cout << "y=" << y << std::endl;
01242932 mov     ecx,dword ptr [__imp_std::cout (0767084h)]
01242938 mov     edx,767454h
0124293D call    dword ptr [Указатель на CLRStub[MethodDescPrestub]@352c5db101240851]
01242943 mov     dword ptr [ebp-1Ch],eax
01242946 mov     ecx,dword ptr [ebp-1Ch]
01242949 mov     edx,dword ptr [ebp-8]
0124294C call    <Module>.std.basic_ostream<char,std::char_traits<char> >.<<(std.bas
01242951 mov     dword ptr [ebp-20h],eax
01242954 mov     edx,dword ptr [__unep@??$endl@DU?$char_traits@D@std@@@std@@$FYAAAV
0124295A mov     ecx,dword ptr [ebp-20h]
0124295D call    <Module>.std.basic_ostream<char,std::char_traits<char> >.<<(std.bas
01242962 mov     dword ptr [ebp-24h],eax
01242965 nop
        c = 0;
01242966 xor     edx,edx
01242968 mov     dword ptr [ebp-4],edx
        c = x + y;
0124296B mov     eax,dword ptr [ebp-0Ch]
0124296E add     eax,dword ptr [ebp-8]
01242971 mov     dword ptr [ebp-4],eax
        std::cout << "c=" << c << std::endl;
01242974 mov     ecx,dword ptr [__imp_std::cout (0767084h)]
0124297A mov     edx,767458h
0124297F call    dword ptr [Указатель на CLRStub[MethodDescPrestub]@352c5db101240851]
01242985 mov     dword ptr [ebp-28h],eax
01242988 mov     ecx,dword ptr [ebp-28h]
0124298B mov     edx,dword ptr [ebp-4]
0124298E call    <Module>.std.basic_ostream<char,std::char_traits<char> >.<<(std.bas
01242993 mov     dword ptr [ebp-2Ch],eax
01242996 mov     edx,dword ptr [__unep@??$endl@DU?$char_traits@D@std@@@std@@$FYAAAV
0124299C mov     ecx,dword ptr [ebp-2Ch]
0124299F call    <Module>.std.basic_ostream<char,std::char_traits<char> >.<<(std.bas
012429A4 mov     dword ptr [ebp-30h],eax
012429A7 nop
}

```

```

012429A8 xor     eax,eax
012429AA mov     esp,ebp
012429AC pop     ebp
012429AD ret
012429AE add     byte ptr [eax],al
012429B0 mov     ah,63h
012429B2 add     dword ptr ds:[eax],eax
012429B5 add     byte ptr [eax],al
012429B7 add     byte ptr [eax+28013E63h],ch
012429BD test    al,1Bh
012429BF ?? ??????

```

Из всего этого нам нужен лишь вот этот кусок программного кода.

```

In [ ]: int x;
        x = 9;
012428F0 mov     dword ptr [ebp-0Ch],9
        int y;
        y = 8;
012428F7 mov     dword ptr [ebp-8],8
        c = 0;
01242966 xor     edx,edx
01242968 mov     dword ptr [ebp-4],edx
        c = x + y;
0124296B mov     eax,dword ptr [ebp-0Ch]
0124296E add     eax,dword ptr [ebp-8]
01242971 mov     dword ptr [ebp-4],eax

```

```

In [ ]: # Мы видим, что int в этой системе 32 битный
# Это следствие оптимизации компилятора...
        int x;
        x = 9;
012428F0 mov     dword ptr [ebp-0Ch],9
# [ ] - обращение к ячейке памяти по указателю
# dword ptr [ebp-0Ch] - обращение к ячейке памяти по указателю,
# адрес ebp(вершина стека) -0Ch (смещение до ячейки памяти переменной x)
        int y;
        y = 8;
012428F7 mov     dword ptr [ebp-8],8
# dword ptr [ebp-8] - обращение к переменной y
        c = 0;
01242966 xor     edx,edx
# очистка регистра edx, а как по другому быстро получить ноль?
01242968 mov     dword ptr [ebp-4],edx
        c = x + y;
0124296B mov     eax,dword ptr [ebp-0Ch]
# Кладем машинное слово из переменной X в регистр eax
0124296E add     eax,dword ptr [ebp-8]
# Прибавляем к регистру машинное слово из переменной Y
01242971 mov     dword ptr [ebp-4],eax
# Возвращаем результат в переменную C.

```

Как видите, машина в одной ветви программного кода старается работать через один регистр общего назначения.

```
#include "pch.h"
#include <iostream>
int main()
{
    int c=0;
    int x;
    x = 9;
    int y;
    y = 8;
    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
    c = 0;
    __asm
    {
        mov     eax, dword ptr[ebp - 0Ch]
        add     eax, dword ptr[ebp - 8]
        mov     dword ptr[ebp - 4], eax
    }
    std::cout << "c=" << c << std::endl;
}
```

Консоль отладки Microsoft Visual Studio

x=9
y=8
c=17

C:\Users\SibNout2020\source\repos\ConsoleApplication1
дом 2061142408.
Чтобы автоматически закрывать консоль при остановке отладки, нажмите любую клавишу, чтобы закрыть это окно...

Пример второй

```
#include "pch.h"
#include <iostream>
int main()
{
    int x= 9;
    int y= 8;
    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
    int c = 0;
    c = x;
    x = y;
    y = c;
    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
}
```

Консоль отладки Microsoft Visual Studio

x=9
y=8
x=8
y=9

C:\Users\SibNout2020\source\repos\ConsoleApplication1
дом 0.
Чтобы автоматически закрывать консоль при остановке отладки, нажмите любую клавишу, чтобы закрыть это окно...

```
In [ ]: c = x;
01522966 mov     eax,dword ptr [ebp-8]
01522969 mov     dword ptr [ebp-0Ch],eax
        x = y;
0152296C mov     eax,dword ptr [ebp-4]
0152296F mov     dword ptr [ebp-8],eax
        y = c;
01522972 mov     eax,dword ptr [ebp-0Ch]
01522975 mov     dword ptr [ebp-4],eax
```

```

int x= 9;
int y= 8;
std::cout << "x=" << x << std::endl;
std::cout << "y=" << y << std::endl;
int c = 0;
__asm {
    mov     eax, dword ptr[ebp - 8]
    mov     dword ptr[ebp - 0Ch], eax
    mov     eax, dword ptr[ebp - 4]
    mov     dword ptr[ebp - 8], eax
    mov     eax, dword ptr[ebp - 0Ch]
    mov     dword ptr[ebp - 4], eax
}
std::cout << "x=" << x << std::endl;
std::cout << "y=" << y << std::endl;

```

Консоль отладки Microsoft Visual Studio

```

x=9
y=8
x=8
y=9
C:\Users\SibNout2020\source\repos\ConsoleApplica
дом 2061142408.
Чтобы автоматически закрывать консоль при остано
томатически закрыть консоль при остановке отладк
Нажмите любую клавишу, чтобы закрыть это окно...

```

Можно и короче

```

int x= 9;
int y= 8;
std::cout << "x=" << x << std::endl;
std::cout << "y=" << y << std::endl;
int c = 0;
__asm {
    mov     eax, dword ptr[ebp - 8]
    mov     ebx, dword ptr[ebp - 4]
    mov     dword ptr[ebp - 8], ebx
    mov     dword ptr[ebp - 4], eax
}
std::cout << "x=" << x << std::endl;
std::cout << "y=" << y << std::endl;

```

Консоль отладки Microsoft Visual Studio

```

x=9
y=8
x=8
y=9
C:\Users\SibNout2020\source\repos\ConsoleApplica
дом 2061142408.
Чтобы автоматически закрывать консоль при остан
томатически закрыть консоль при остановке отлад
Нажмите любую клавишу, чтобы закрыть это окно...

```

```

int x= 9;int y= 8;
std::cout << "<" << x<<";"<< y << ">" << std::endl;
int c = 0;
__asm {
    mov     eax, dword ptr[ebp - 8]
    mov     ebx, dword ptr[ebp - 4]
    XCHG    eax, ebx
    mov     dword ptr[ebp - 8], eax
    mov     dword ptr[ebp - 4], ebx
}
std::cout << "<" << x << ";" << y << ">" << std::endl;

```

Кон

```

<9;8>
<8;9>
C:\Use
дом 20
Чтобы
томати
Нажмит

```

```

int x= 9;int y= 8;
std::cout << "<" << x<<";"<< y << ">" << std::endl;
int c = 0;
__asm {
    mov     eax, dword ptr[ebp - 8]
    XCHG    eax, dword ptr[ebp - 4]
    mov     dword ptr[ebp - 8], eax
}
std::cout << "<" << x << ";" << y << ">" << std::endl;

```

Консоль отладки Micr

```

<9;8>
<8;9>
C:\Users\SibNout2020
дом 2061142408.
Чтобы автоматическ
томатически закрыть

```

In []: Совет. Если пишете программный код, сначала очищайте используемые регистры. ****XOR EAX,EAX**

Пример третий

```
In [ ]: #include "pch.h"
#include <iostream>
int main()
{
    //переменная x
    int x = 9;
    //переменная y
    int y = 8;
    //Указатель на переменную yy
    int* yy = &y;
    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
    //Обращение к адресу оперативной памяти переменной
    std::cout << "&y=" << &y << std::endl;
    //Просто посмотреть - что внутри указателя
    std::cout << "yy=" << yy << std::endl;
    //Обращение к значению по адресу из указателя
    std::cout << "*yy=" << *yy << std::endl;
    _asm
    {
        xor eax, eax
        xor ebx, ebx
        mov     eax, dword ptr[x]
        mov     ecx, dword ptr[yy]
        // Обращаемся к переменной указателя
        // Получаем содержимое переменной указателя в регистр ecx
        mov     ebx, dword ptr[ecx]
        //Теперь ecx содержит указатель на переменную y
        //Получаем значение переменной y в ebx
        XCHG eax, ebx
        mov     dword ptr[x], eax
        mov     dword ptr[ecx], ebx
        //Кладем значение в переменную y
        //ecx содержит указатель на переменную y
    }
    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
}
```



```

//переменная x
int x = 9;
//переменная y
int y = 8;
//Указатель на переменную yy
int* yy = &y;
std::cout << "x=" << x << std::endl;
std::cout << "y=" << y << std::endl;
//Обращение к адресу оперативной памяти по указателю
std::cout << "&y=" << &y << std::endl;
//Просто посмотреть что внутри указателя
std::cout << "yy=" << yy << std::endl;
//Обращение к значению по адресу из указателя
std::cout << "*yy=" << *yy << std::endl;

```

Внимание, при копировании текста в Visual Studio 2019 некоторые символы вставляются как символы других кодов, но с тем же внешним видом. В таких случаях рекомендуется либо создать новый проект, либо в Visual Studio набрать код с клавиатуры.

Пример четвертый

Работа с массивами на Ассемблере ведется по тем же принципам, что и работа с массивами в ++ через арифметику указателей. Вот пример.

```

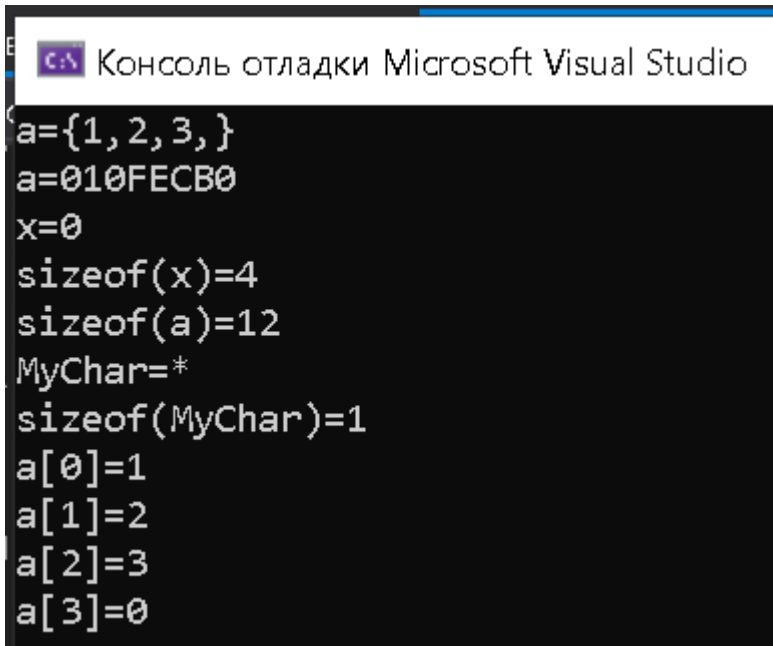
In [ ]: #include "pch.h"
#include <iostream>
int main()
{
    //переменная x
    int x = 0;
    //Создали одномерный массив из 3 элементов
    int a[3] = { 1, 2, 3 };
    //Создали символ
    char MyChar = '*';
    //Вывели наш массив на экран в стиле аля питон
    std::cout << "a={" << a[0] << "," << a[1] << "," << a[2] << "," << "}" << std::endl;
    //Вывели на экран весь массив, и вдруг выяснилось, что это указатель
    std::cout << "a=" << a << std::endl;
    std::cout << "x=" << x << std::endl;
    //Получили размер переменной x, которая int
    std::cout << "sizeof(x)=" << sizeof(x) << std::endl; //4
    //Получили размер всего нашего массива (он 12) из 3 элементов размером по 4
    std::cout << "sizeof(a)=" << sizeof(a) << std::endl;
    std::cout << "MyChar=" << MyChar << std::endl;
    //Вывели размер символа
    std::cout << "sizeof(MyChar)=" << sizeof(MyChar) << std::endl;
    //Вывели на экран первый элемент массива, обратившись к нему через Ассемблер
    std::cout << "a[0]=";
    _asm
    {
        xor eax, eax
        mov eax, dword ptr[a]
    }
}

```

```

        mov dword ptr[x], eax
    }
    std::cout << x << std::endl;
    //Вывели на экран первый элемент массива, обратившись к нему через Ассемблер
    //НЕ ЗАБЫЛИ У INT ПРО РАЗМЕР И ДОПИСАЛИ СМЕЩЕНИЕ В ОПЕРАТИВНОЙ ПАМЯТИ +4
    std::cout << "a[1]=";
    _asm
    {
        xor eax, eax
        mov eax, dword ptr[a + 4 * 1]
        mov dword ptr[x], eax
    }
    std::cout << x << std::endl;
    //+4 2 раза
    std::cout << "a[2]=";
    _asm
    {
        xor eax, eax
        mov eax, dword ptr[a + 4 * 2]
        mov dword ptr[x], eax
    }
    std::cout << x << std::endl;
    //Обратились к 4, несуществующему элементу массива
    //Получили мусор
    std::cout << "a[3]=";
    _asm
    {
        xor eax, eax
        mov eax, dword ptr[a + 4 * 3]
        mov dword ptr[x], eax
    }
    std::cout << x << std::endl;
}

```



Консоль отладки Microsoft Visual Studio

```

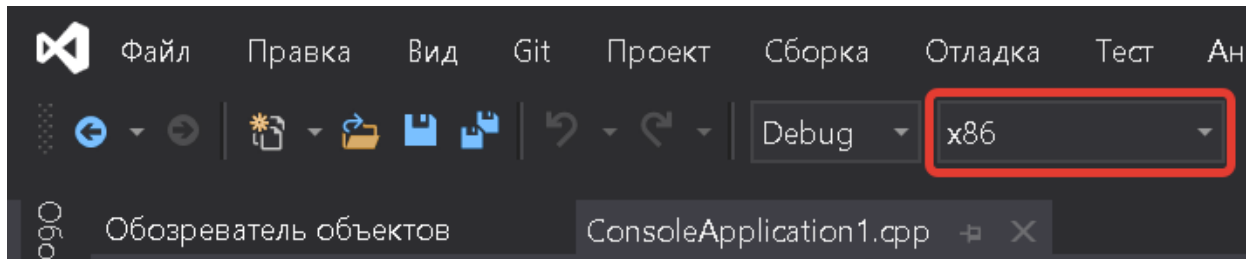
a={1, 2, 3, }
a=010FECB0
x=0
sizeof(x)=4
sizeof(a)=12
MyChar=*
sizeof(MyChar)=1
a[0]=1
a[1]=2
a[2]=3
a[3]=0

```

Практическая работа №5. Арифметические операции

Задание взято из "Архитектура ЭВМ. Задания и примеры выполнения лабораторных работ. Методические указания/ сост. : А.Е. Докторов, Е. А. Докторова. - Ульяновск : УлГТУ, 2008. - 32 с".

Тема лабораторной работы. 64 битные операции на 32 битной машине Выставляем в Visual Studio "Debug x86". После этого программный код будет генерироваться 32 битным даже на 64 битной машине.



Далее рекомендуется дать на самостоятельный поиск по источникам из Интернета следующие команды. Также их краткие описания можно посмотреть в [А. Е. Докторов.2008]

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

| | | | | | | |
|-----|------|-----|------|-----|-----|-----|
| ADD | ADC | INC | SUB | SBB | DEC | CMP |
| MUL | IMUL | DIV | IDIV | NEG | CBW | CWD |

Требуется написать ассемблерную вставку сложения (вычитания) двух 64 битных чисел.

```
In [ ]: #include "pch.h"
#include <iostream>
int main()
{
    __int64 c=0;
    __int64 x;
    x = 4;
    __int64 y;
    y = 6;
    c = x + y;
    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
    std::cout << "c=" << c << std::endl;
}
```

```
In [ ]: __int64 x;
        x = 4;
0068290F mov     eax,4
00682914 cdq
00682915 mov     dword ptr [ebp-10h],eax
00682918 mov     dword ptr [ebp-0Ch],edx
        __int64 y;
        y = 6;
0068291B mov     eax,6
00682920 cdq
00682921 mov     dword ptr [ebp-8],eax
00682924 mov     dword ptr [ebp-4],edx
        __int64 c=0;
        c = x + y;
00682927 mov     eax,dword ptr [ebp-10h]
0068292A mov     edx,dword ptr [ebp-0Ch]
0068292D add     eax,dword ptr [ebp-8]
00682930 adc     edx,dword ptr [ebp-4]
00682933 mov     dword ptr [ebp-18h],eax
```

```

00682936 mov          dword ptr [ebp-14h],edx
          c = x - y;
00EE2927 mov          eax,dword ptr [ebp-10h]
00EE292A mov          edx,dword ptr [ebp-0Ch]
00EE292D sub          eax,dword ptr [ebp-8]
00EE2930 sbb          edx,dword ptr [ebp-4]
00EE2933 mov          dword ptr [ebp-18h],eax
00EE2936 mov          dword ptr [ebp-14h],edx

```

Задим студентам вопрос: "Почему сложение 64 битных чисел?" Возможно, кто-то из них догадается или найдет ответ в презентациях...

Ответ ожидается примерно следующий: 32 битная система оперирует машинными словами Word 32 бита.

Единственный способ эмулировать 64 битный код, это использовать два 32 битных слова.

Ранее подобный прием использовался на 16 битных машинах с 32 битным кодом.

Как бы выглядел 64 битный код на 8 битной машине?

Производительность в этом случае была бы просто "фантастическая".

Поддержка кода осуществляется за счет транслирования с макро-ассемблера в микро-ассемблер, родной для конкретной машины. Чтобы это осуществить, требуется написать множество подпрограмм, заменяющих базовые операции или другие подпрограммы.

Ранее базовые команды микропроцессора реализовывались на уровне железа. Сейчас некоторые базовые команды реализуются на программном уровне. Это приводит к снижению производительности, но позволяет крос-платформенность.

Последующие темы по методическим указаниям [Докторов 2008]:

- Изучение логических команд и команд сдвигов.
- Изучение команд обработки блоков данных. Цикл LOOP. Обработка текстов.
- Изучение команд условного перехода.
- Изучение команд передачи управления.

Но не так все просто.

Команда LOOP работает только в Pascal ранних версий и DosBox.

Некоторых команд обработки данных нет в современном ассемблере. Их заменяют своими процедурами и работой через указатели.

Потому, с этого момента [Докторов 2008] будет использован нами как дополнительный материал.

Мы плавно переключаемся на программирование под DosBox на FlatAssembler1.71.

Сайт [FasmWorld](http://FasmWorld.com).

Продолжение следует...

Распространенные ошибки

- При создании проекта вместо c++ выбран любой другой язык.
- Ошибки в c++ коде.
- Слепое копирование ... неработающего кода в неподходящее для него место. Тут мы бессильны...

- Попытка реализовать задачу как на 32 или 16 битном типе данных. Внимательно прочитайте пример из лекции, используйте 64 битный тип данных и дезассемблер.

In []:

Примерный план проблемной лекции

FASM/DosBox. Изучение строк, ветвлений, циклов

Время на изучение темы 8 - 12 часов, из них 2 часа лекция.

Учебные цели занятия: по материалам под FASM/DosBox освоить написание полноценных консольных программ FASM/DosBox.

Форма проведения занятия: проблемная лекция.

Учебные задания:

- Скачать/распаковать/установить FASM/DosBox/Notepad++.
- Создать проект "Привет мир", настроить его компиляцию и запуск.
- Решить следующие учебные проблемы.
 - Этап первый. Изучить создание переменных и вывод на экран (1-2 ч.)
 - Этап второй. Изучить механику условных переходов (1-2 ч.)
 - Этап третий. Изучить создание циклов (2-4 ч.)
 - Этап четвертый. Собрать все воедино в виде моноблока исполняемых ассемблерных кодов (4 ч.)

Инструкции по проведению и ходу занятия: В начале занятия обосновываются цель и задачи проблемной лекции, определяется проблема для обсуждения. Преподаватель излагает материал, задает студентам вопросы, посвященные проблематике лекции, затем начинается дискуссия по поставленной проблеме; вторая половина занятия групповая практика на ПК.

Методические рекомендации: вовлечь учащихся в совместное, командное изучение отдельных этапов учебного задания. Используем перекрестные опросы. Отстающих включаем в команды, где им смогут помочь.

Вопрос для самоконтроля:

Компилируется ли без ошибок Ваша программа? Запускается ли без ошибок Ваша программа? Это два разных вопроса!

Знаете ли Вы каждый оператор из Вашей программы? Можете ли объяснить - что в программе происходит? Расплывчатые ответы не принимаются! Без четкого понимания того - что и зачем сделано, дальше продвигаться вперед не удастся.

Практическая работа №6. Изучение строк, циклов

Что требуется сделать: 1) Установить [Notepad++](#). 2) Установить [DosBox](#). 3) Создать папку "C:\D". 4) Работать будем, используя путь "C:\D\Git_Hub\HowTo_FASM". Туда скачать и распаковать [Flat_Assembler](#).

Объясним - что это и зачем. Flat_Assembler - ...

содержит компилятор "FASM.exe", только компилятор, причем пакетно запускаемый.

Параметров запуска немного, если не ошибаюсь, - только объемы оперативной памяти, поэтому сейчас об этом не заботимся.

Есть FASMW.exe - среда разработки. Позволяет писать текст, сохранять, компилировать, запускать скомпилированное. FASMW.exe - уже не обладает возможностями пакетного запуска, поэтому удастся запустить только программу, написанную под Windows.

"Выкручиваемся", используя Notepad++. Создаем 2 файла: "hello.asm" и "hello.bat".

```
In [ ]: ; Это комментарий на языке Ассемблер
; "_hello_.asm"
use16          ;Генерировать 16-битный код
org 100h       ;Программа начинается с адреса 100h
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
jmp start
hello db 'Hello, world!$'
_char db 'H$'
_The_Fin db '_The_Fin$'
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
start:
    mov dx,hello
    mov ah,9
    int 21h
    ;;;
    ;Вывод символа перехода на новую строку
    mov dl,0ah
    mov ah,2
    int 21h
    ;;;
    ;Вывод символа перехода на новую строку
    mov dl,0ah
    mov ah,2
    int 21h
    ;;;
    mov dx,_The_Fin
    mov ah,9
    int 21h
    ;;;
    ;Вывод символа перехода на новую строку
    mov dl,0ah
    mov ah,2
    int 21h
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;/Ожидание нажатия клавиши
    mov ah,01h
    int 21h
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    mov ax,4C00h ;
    int 21h      ;/ Завершение программы
;-----
```

```
In [ ]: :: Это комментарий на языке Dos
```

```
:: "_hello_.bat"
@echo off
echo %cd%
cd C:\D\Git_Hub\HowTo_FASM\000_HelloWorld
echo %cd%
set FileName=_hello_
..\Fasm\FASM.EXE %FileName%.ASM
"C:\Program Files (x86)\DOSBox-0.74-3\DOSBox.exe" %FileName%.COM
::pause
```

Как работать со всем этим?

Пишем код. Сохраняем. Нажимаем F5, выбираем *"hello.bat"*.

Последующая разработка выглядит так. Пишем код, сохраняем, нажимаем F5, нажимаем Enter.

При подобном запуске из Notepad++ адрес директории будет "C:\Program Files\Notepad++".

Об этом мы узнаем из команды "echo %cd%"

Следующей командой "cd C:\D\Git_Hub\HowTo_FASM\000_HelloWorld" мы меняем адрес директории на локальный (на папку, где лежат файлы).

"echo %cd%" - повторно убеждаемся, что находимся в нашей папке.

"C:\D\Git_Hub\HowTo_FASM\000_HelloWorld"

После этого пишем следующий программный код:

"(путь до компилятора FASM.exe)" "(путь до _hello_.asm)"

Это был пакетный запуск компилятора с нашим компилируемым файлом, как параметром.

После этого мы получаем в нашей локальной папке "_hello_.com".

Теперь уже этот файл можно запустить в DosBox строкой:

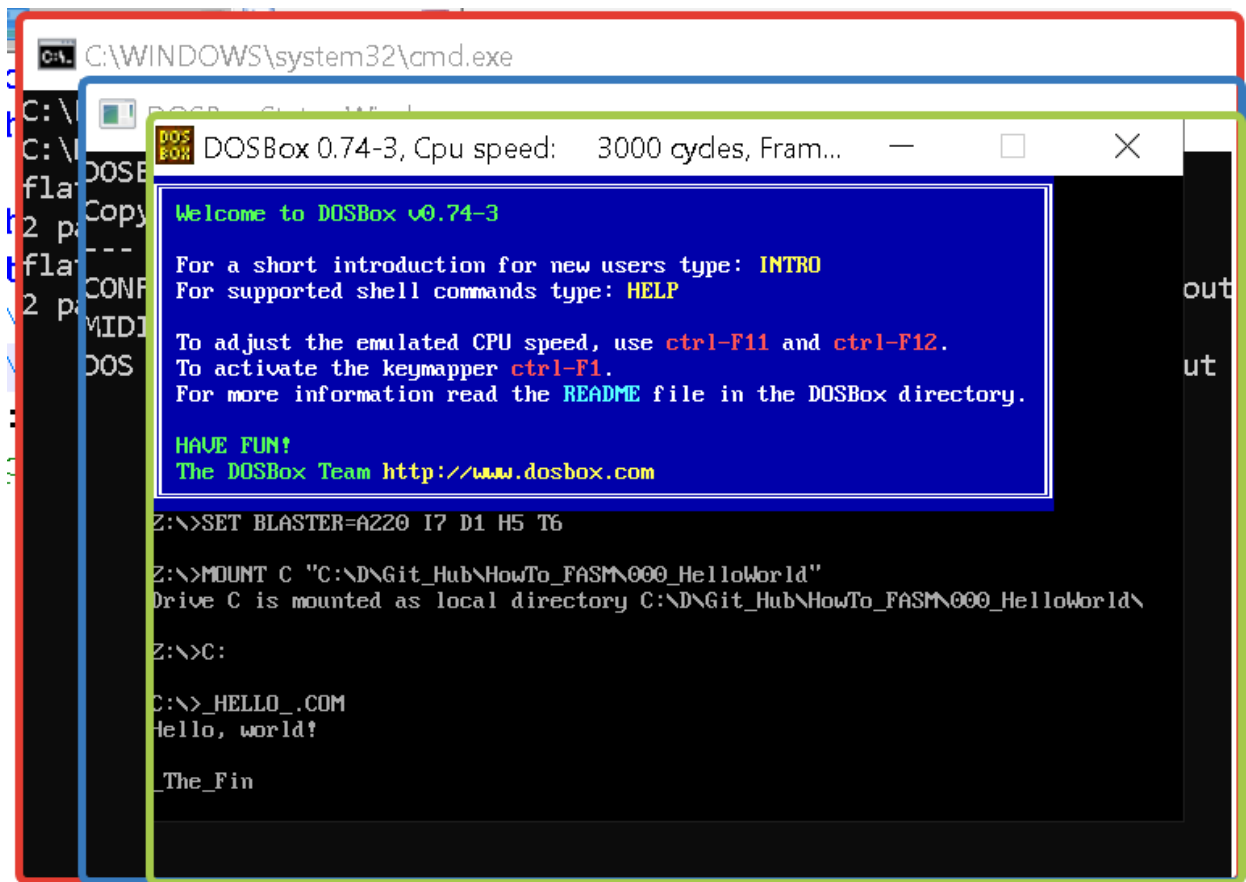
"C:\Program Files (x86)\DOSBox-0.74-3\DOSBox.exe" %FileName%.COM

Распространенные ошибки.

Сетевое расположение папки. Виртуальные логические диски, созданные запускаемыми приложениями, антивирусники, эмуляторы *.iso и просто установленное большое количество игрушек на компьютере (это был намек).

Ну и напоследок ":::pause" - снимаете комментарий и приложение будет работать с паузой в конце, если оно потребуется.

Приведем скрин того, как выглядит работающее приложение.



Задание

Примерные варианты контрольных задач.

- 1) Найти в строке позицию заданного символа.
- 2) Определить, есть ли в двух строках одинаковые символы на одинаковых позициях.
- 3) Найти позицию, на которой две строки символов отличаются.
- 4) Удалить из строки заданный символ.
- 5) Удалить из строки символ на заданной позиции.

Задание классифицирую как сложное. Выполняется в несколько этапов.

- Этап первый. Изучить создание переменных и вывод на экран.
- Этап второй. Изучить механику условных переходов.
- Этап третий. Изучить создание циклов.
- Этап четвертый. Собрать все воедино в виде моноблока исполняемых ассемблерных кодов.

Этап первый. Изучить создание переменных и вывод на экран.

Для создания локальных переменных при помощи команды безусловного перехода `jmp` создаем недостижимый карман программного кода между строками `"jmp start"` и `"start:"`.
`"start:"` - метка.

Это просто кусок оперативной памяти, в который в ASCII кодах укладывается текст. В Dos символом конца строки является доллар. Все процедуры работают, исходя из этого. По аналогии можно создать dw (word), dd (DWord)... Но ощутимой разницы мы не почувствуем... db - это директива создания данных типа byte по 8 бит, это означает, что 'Hello, world!\$' занимает 8*14 бит.

Отныне 'hello' - ключевое слово, которое компилятором будет заменено на указатель адреса начала строки... Да, Ассемблер работает только с указателями.

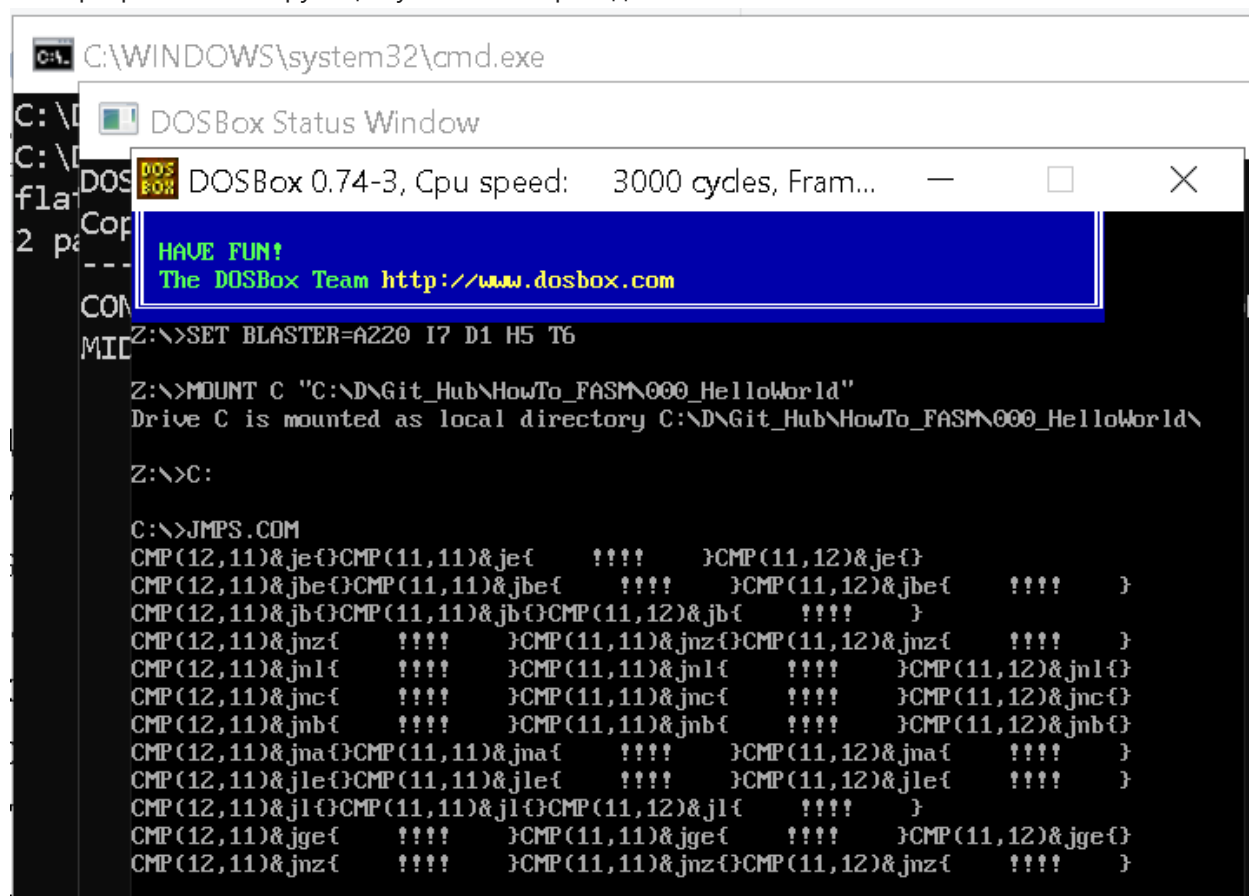
```
In [ ]: jmp start
        hello db 'Hello, world!$'
        _char db 'H$'
        _The_Fin db '_The_Fin$'
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        start:
```

Для изучения вывода на экран нам потребуется доступ к Интернету или книга-справочник по прерываниям dos.

Предлагается книга Хитрово Н.Г. "Начала системного программирования в среде MS-DOS7".

Этап второй. Изучить механику условных переходов

Вот программа, тестирующая условные переходы



```
In [ ]: ;JMPS_Start.bat
        @echo off
```

```

echo %cd%
::cd C:\Fasm\Projects\001_JMP_S
cd C:\D\Git_Hub\HowTo_FASM\000_HelloWorld
echo %cd%
set FileName=JMPS
::C:\Fasm\FASM.EXE %FileName%.ASM
C:\D\Git_Hub\HowTo_FASM\Fasm\FASM.EXE %FileName%.ASM
"C:\Program Files (x86)\DOSBox-0.74-3\DOSBox.exe" %FileName%.COM
::pause

```

In []:

```

;JMPS.asm
use16                ;Генерировать 16-битный код
org 100h              ;Программа начинается с адреса 100h
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
jmp start
_ok db '!!!!!!$'
_The_Fin db '_The_Fin$'
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
include '..\macroDos\__Console_V0.inc'
start:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    __Test.CMP.011 je
    __Test.CMP.011 jbe
    __Test.CMP.011 jb
    __Test.CMP.011 jnz
    __Test.CMP.011 jnl
    __Test.CMP.011 jnc
    __Test.CMP.011 jnb
    __Test.CMP.011 jna
    __Test.CMP.011 jle
    __Test.CMP.011 jl
    __Test.CMP.011 jge
    __Test.CMP.011 jnz

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;Ожидание нажатия клавиши
    mov ah,01h
    int 21h
    ;Завершение программы
    mov ax,4C00h
    int 21h
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```

include '..\macroDos__ConsoleV0.inc' - вызов библиотеки макросов.

__Test.CMP.011 jl - вызов параметрического макроса, тестирующего заданный переход.

Примем соглашение, что далее: все макросы, написанные нами, будут начинаться с двух нижних подчеркиваний; все локальные переменные, написанные нами, будут начинаться с одного нижнего подчеркивания.

Из N++ , кликнув правой кнопкой на названии библиотеки, можно затем в контекстном меню выбрать и открыть файл. Не забудьте предварительно выделить полный путь и название через shift... ("..\\" - означает подъем на каталог выше).

Так мы увидим программный код библиотеки макросов.

__Console_V0.inc - эта библиотека написана нами для работы с консолью, тестирования условных переходов и решения подобных мелких прикладных задач. Работает под DosBox.

```

In [ ]: ;__Console_V0.inc
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
macro __WriteChar _Char
{
    push ax
    push dx
        mov dl, _Char
        ;mov dl,al;Вывод вывод одного символа из dl на экран
        mov ah,2
        int 21h
    pop dx
    pop ax
}
macro __WriteChar_N{__WriteChar 0ah}
macro __Write _str
{
    push ax
    push dx
        mov dx, _str;dx - положите указатель на строку;Например вот так;mov dx,he
        mov ah,9
        int 21h
    pop dx
    pop ax
}
macro __WriteLN _str
{
    __Write _str
    __WriteChar_N
}
macro __ReadKeyToAL
{;;Запись одного символа в регистр AL
    mov ah,1
    int 21h
}
macro __SharpX _count
{
    local m_1
    push ecx
    push dx
    mov ecx,_count
    m_1:
        __WriteChar 35
    loop m_1
    __WriteChar_N
    pop dx
    pop ecx
}
macro __SharpX79{__SharpX 79}
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
macro __TestCMP _A,_B,_JmpS
{
    ;;Макрос тестирует работу условных переходов в FASM ПОД DOS
    ;;Пример кода для запуска
    ;;__TestCMP 10,10,jnz
    ;;Вставка в текст прилетевших текстов кодов...
    __RValue.Write 'CMP('#`_A#','#`_B#')&#`_JmpS#'$'
        local m_1,m_2
        mov eax,_A
        cmp eax,_B
        __WriteChar '{'
        __JmpS m_1
}

```

```

        jmp m_2
m_1:
        __RValue.Write '    !!!!    $'

        m_2:
__WriteChar '}'
;__WriteChar_N
}
macro __Test.CMP.011 _JmpS
{
        __TestCMP 12,11,_JmpS
        __TestCMP 11,11,_JmpS
        __TestCMP 11,12,_JmpS
        __WriteChar_N
}
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
macro __RValue.Write _str
{
    ;;
; l-value - требует предварительного создания переменной
; - Все макросы и процедуры по умолчанию являются l-value
; r-value - не требует предварительного создания переменной
;;Терминология взята с сайта
;;https://ravesli.com/urok-190-ssylki-r-value/#toc-2
;;
local m_data,__str
jmp m_data
        __str db _str
m_data:
        push ax
        push dx
                mov dx,__str;dx - положите указатель на строку;Например вот так;mov dx,h
                mov ah,9
                int 21h
        pop dx
        pop ax
}
macro __RValue.WriteLN _str
{
        __RValue.Write _str
        __WriteChar_N
}
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```

Этап третий. Изучить создание циклов

Самое интересное, что для этого нам потребуется Visual Studio 2019 c++.

Пишем программный код с циклом с предусловием, с циклом с постусловием, с циклом со счетчиком и switch case.

Смотрим через дезассемблер, как все это устроено.

В этом нам поможет опять N++. В нем есть возможность выделить маркером сочетание букв.

После этого маркером выделяются все вхождения этого слова в текст. Есть 4 цвета маркера. Это проще, чем выискивать в тексте адрес, на который осуществляется переход.

После некоторой практики на Ассемблере станет заметно, что написание всех циклов и ветвлений очень похоже на цикл с предусловием, как в **BF**.

Этап четвертый. Собрать все воедино в виде моноблока исполняемых ассемблерных кодов

Приведу пример программного кода под ConsoleWindows64, решение одной из ранее описанных задач. По этому примеру можно написать код и под DosBox.

Для того чтобы начать писать на Ассемблере требуется не многим больше команд, чем есть в BF. Этот программный код копирует в новую строку все символы старой строки за исключением заглавной 'S', при этом используется только арифметика указателей и пересылка 8 битных byte машинных слов (без использования команд обработки блоков данных).

```
In [ ]: ;_22_.asm
format PE64 console
entry start

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
include 'C:\D\Git_Hub\HowTo_FASM\Fasm\INCLUDE\win64a.inc'

include '..\macroWin\__Console_V0.inc'

section '.idata' import data readable
__InitConsoleSectionImport
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
section '.data' data readable writeable
__InitConsoleSectionData
    _str db 'QWESSbblSSS<<sss>>',0
    _str2 db 255 dup(0)
    _char db '*,0

section '.code' code readable executable
start:
__setlocale_Russian
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    __RValue.WriteLine 'Удаление символа из строки'
    __RValue.WriteLine 'Введите строку символов длиной до 255 символов:'
    ;__RValue.Read '%s',_str
    ;__Write _N
    __WriteLN _str
    __WriteLN _str2
    ;rax - указатель первой строки
    ;rbx - Символ первой строки
    ;rcx - указатель второй строки
    xor rax,rax
    mov eax,_str
    xor rcx,rcx
    mov ecx,_str2
m_20210115_1152:
    mov bl,byte [eax]
    mov byte [_char],bl
    ;;;
    cmp bl,'S'
    jne m_20210115_1214
    JMP m_20210115_1213
    m_20210115_1214:
```

```

                                mov byte [ecx],bl
                                add ecx,1
                                m_20210115_1213:
                                ;;
                                add eax,1
                                mov bl,byte [eax]
                                cmp bl,0
                                jne m_20210115_1152
                                __WriteLN _str
                                __WriteLN _str2
                                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
cinvoke system,_Pause
                                jmp exit ;??????
exit:
                                ;invoke ExitProcess, 0
                                push 0
                                call [ExitProcess]

```

```

In [ ]: ::_22_Start.bat
@echo off
echo %cd%
cd C:\D\Git_Hub\HowTo_FASM\002_
echo %cd%
set FileName=_22_

C:\D\Git_Hub\HowTo_FASM\Fasm\FASM.EXE %FileName%.ASM
%FileName%.EXE
pause

```

Мы написали почти такую же библиотеку макросов под Windows. Консоль, тестирование условных переходов, вывод на экран состояний всех регистров без изменения состояния регистров.

Кстати, последняя задача - краеугольный камень программирования на Ассемблере на последующих этапах.

```

In [ ]: ;..\macroWin\__Console_V0.inc
;Макросы FASM консоль Winda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
macro __InitConsoleSectionImport
{
;;section '.idata' import data readable
    library kernel,'kernel32.dll',\
                msvcrt,'msvcrt.dll'

    import kernel,\
                ExitProcess,'ExitProcess'

    import msvcrt,\
                setlocale,'setlocale',\
                printf,'printf',\
                scanf,'scanf',\
                system,'system'
}
macro __InitConsoleSectionData
{
;;section '.data' data readable writeable
    _Russian db 'Russian',0
    _Pause db 'pause',0
    _N db 13,10,0
    _BraceCurlyBegin db '{',0
    _BraceCurlyEnd db '}',0
}

```

```

        _0 db '0',0
        _1 db '1',0
        _2 db '2',0
        _3 db '3',0
        _4 db '4',0
        _5 db '5',0
        _6 db '6',0
        _7 db '7',0
        _8 db '8',0
        _9 db '9',0
    }
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;; Консольный минимум                                     ;;
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    macro __setlocale_Russian{cinvoke setlocale,0,_Russian}
    macro __Write _Char{
    push rax
    push rbx
    push rcx
    push rdx
    pushf
        cinvoke printf,_Char
    popf
    pop rdx
    pop rcx
    pop rbx
    pop rax
    }
    macro __WriteLN _Char{
        __Write _Char
        __Write _N
    }
    macro __RValue.Write Char{
    local m_data,_Char
    jmp m_data
        _Char db Char,0
    m_data:
        __Write _Char
    }
    macro __RValue.WriteLN Char{
        __RValue.Write Char
        __Write _N
    }
    macro __RValue.Read fmat,[param]{
    COMMON
    ;COMMON - Директива, после которой программной
    ;код повторяется для всей групповой переменной
    ;[param]
    ;FORWARD/REVERSE (В Прямом/В Обратном порядке)- Директива, после которой программной
    ;код повторяется для каждого элемента
    ;групповой переменной [param]
    local m_data,_fmat
    jmp m_data
        _fmat db fmat,0
    m_data:
        cinvoke scanf,_fmat,param
    }
    macro __SharpX _count{
        local m_1
        push rcx
        xor ecx,ecx

```


[illegible]

```

m_y89ryd3:
;Циклические сдвиги выдвигают бит регистр флагов в EFLAGS.CF
;На EFLAGS.CF есть два перехода ;jc - если единица;jnc -если ноль
shl _Param,1
push rax
push rbx
push rcx
push rdx
jnc m_4234423
    __Write _1
jmp m_4723649
    m_4234423:
    __Write _0
m_4723649:
pop rdx
pop rcx
pop rbx
pop rax
sub cx,1
;Переход, если результат не отрицательный
jnz m_y89ryd3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pop rdx
pop rcx
pop rbx
pop rax
__RValue.WriteLine 'B'
}
macro __Test.BiTest_MonoScript{
xor rax,rax
mov al,10000000b
__BiTest_MonoScript "AL=",8, AL
;;;
xor rax,rax
mov ax,1000000000000000b
__BiTest_MonoScript "AX=",16, AX
;;;
xor rax,rax
mov eax,10000000000000000000000000000000b
__BiTest_MonoScript "EAX=",32, EAX
;;;
xor rax,rax
mov rax,10000000100000000000000000000000010000000000000000000000b
__BiTest_MonoScript "RAX=",64, RAX
;;;
}
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
macro __BiTest_Flag{
push rax
push rbx
push rcx
push rdx
pushf
;;;
pop AX
push AX
    __RValue.WriteLine 'Flag=**N*ODITSZ*A*P*C'
    __BiTest_MonoScript 'Flag=',16, AX
;;;
popf
pop rdx

```

```

pop rcx
pop rbx
pop rax
}
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
macro __BiTest_Oll{
;https://en.wikipedia.org/wiki/FLAGS_register
;https://prog-cpp.ru/asm-command/
push rax
push rbx
push rcx
push rdx
pushf
    __BiTest_MonoScript "RAX=",64, RAX
    __BiTest_MonoScript "RBX=",64, RBX
    MOV RAX,RCX
    __BiTest_MonoScript "RCX=",64, RAX
    __BiTest_MonoScript "RDX=",64, RDX
popf
pop rdx
pop rcx
pop rbx
pop rax
;;;
__BiTest_Flag
}

```

Распространенные ошибки

- При создании проекта выбрана не та шапка директив компиляций.
- Нет точки выхода из программы.
- Слепое копирование ... неработающего кода в неподходящее для него место. Тут мы бессильны... Иногда приходится один и тот же пример разбирать с каждым следующим учащимся. Чтобы этого избежать, рекомендуется командное исполнение заданий, когда сильные ученики консультируют отстающих. (Преподаватель при этом готов оказать помощь).
- Попытка реализовать задачу на машинных словах недопустимой битности. Dos - 16 битная система. Использование машинных слов большей битности может привести к потере нити происходящего учащимся.
- Попытка использовать ассемблерную команду, не предусмотренную операционной системой. Например для Windows такая команда Loop (Простой досовский цикл).
- Попытка использовать команду - подводный камень компилятора FASM. Когда мы беремся использовать чужие библиотеки, мы принимаем все допущенные в них ошибки и, по возможности, беремся абсорбировать их. Или просто стараемся не попадать в такую ситуацию. Подобные вещи выясняются только опытным путем. Например, DosBox выдает ошибку при излишней глубине стека вызова функций, значит отказываемся от функций и используем макросы... Например, Windows 64 Бита своеобразным образом осуществляет сдвиг 32 битных чисел, как будто его просто нет: затирает верхнюю половину регистра, как будто используется 64 битный сдвиг. Заметить это можно, только написав свой тестовый стенд. Исправить можно либо изменением логики программы, либо написанием своей

процедуры, которую можно применять для сдвига... Обо всем этом подробнее в следующих лекциях. Продолжение следует.

В следующих лекциях...

Далее будут рассмотрены проблемы написания процедур и функций.

- Вызов call и 3-4 способа передачи параметров внутрь функций.
- Макросы как средство повышения быстродействия. (Полный курс макросов).
- Стек вызова функций и размещение всех локальных переменных в нем.
- Компьютерная графика под DosBox.
- Подключение и вызов системных библиотек с++ Windows.
- Написание тестового стенда для изучения работы сдвигов под FASM Windows. -Ловим ошибки разработчиков FASM.
- Компьютерная графика под WinOpenGL.

Ближайшие перспективы.

- Ассемблерные команды MMX - аппаратное сложение массивов, используется обычно в криптографии и компьютерной графике.
- Работа с fword 48 бит. Аппаратная работа с дробными числами.
- Перебор материала [Мануал программера.flat assembler 1.71](#). Выясняем какие аппаратные процедуры работают под макроассемблер Windows FASM. Это настоящий ребус, разгадывание которого у нас впереди.

In []: