Слайд 2
Языки программирования

Язык программирования — это формализованная система записи алгоритмов, предназначенная для передачи компьютеру информации о том, какие действия он должен выполнять. Язык должен быть понятен как человеку-программисту, так и машине после соответствующей обработки.

Языки программирования принято делить на несколько уровней:

- **машинные языки**, непосредственно соответствующие двоичным командам процессора;
- **языки низкого уровня** — ассемблеры, где команды описываются с использованием мнемоник, но сохраняется прямая связь с машинными инструкциями;
- **языки высокого уровня**, предоставляющие более абстрактные конструкции: операторы ветвления, циклы, функции, структуры данных.

К языкам высокого уровня относятся Fortran, Pascal, C, C++, Java и многие другие. Они позволяют программисту сосредоточиться на логике задачи, а не на деталях работы процессора.

Слайд 3
История возникновения языка Си

Язык Си был создан в начале 1970-х годов в исследовательских лабораториях Bell Laboratories. Его автором считается Деннис Ритчи. Предпосылкой появления Си послужила разработка операционной системы UNIX. Для её реализации требовался язык, обеспечивающий эффективный доступ к возможностям оборудования, но при этом более удобный, чем ассемблер.

Прямым предшественником Си был язык B, созданный Кеном Томпсоном. B в свою очередь развивался из языка BCPL. Однако язык B оказался недостаточно мощным для задач системного программирования. В результате Деннис Ритчи разработал Си как более универсальный и эффективный инструмент.

Первоначально Си использовался для переписывания ядра UNIX, что позволило сделать систему переносимой между различными аппаратными платформами. Этот факт сыграл решающую роль в популяризации как UNIX, так и языка Си.

С середины 1970-х годов Си стал распространяться за пределы Bell Labs. Появилось множество его реализаций для разных машин. Однако отсутствовал единый стандарт, и программы, написанные для одного компилятора, нередко требовали значительной адаптации для другого.

В 1983 году Американский национальный институт стандартов (ANSI) начал работу по унификации языка. В 1989 году был утверждён стандарт ANSI C, позже ставший международным (ISO C). В дальнейшем язык развивался: появились версии C99, C11, C17 и C23, каждая из которых вносила уточнения и новые возможности, сохраняя при этом преемственность и обратную совместимость.

Слайд 4

Особенности языка Си

Си занимает особое место среди языков программирования. Его отличает сочетание черт низкоуровневого и высокоуровневого подхода.

К характерным особенностям языка относятся:

1. **Эффективность.** Программы на Си обычно компилируются в компактный и быстрый машинный код, близкий по эффективности к ассемблеру.
2. **Близость к оборудованию.** Си предоставляет программисту средства управления памятью, работы с адресами, битовыми операциями. Это делает его удобным для системного программирования, разработки драйверов и операционных систем.
3. **Структурность.** Язык поддерживает разбиение программы на функции, использование управляющих структур — условных операторов и циклов, что способствует созданию читаемого и организованного кода.
4. **Переносимость.** При соблюдении стандарта программы на Си можно переносить на разные платформы, изменяя лишь минимальные детали.
5. **Универсальность.** На Си можно писать как системные программы, так и прикладные приложения.

Слайд 5

В то же время у Си отсутствуют некоторые удобные средства, характерные для более поздних языков. Например, управление памятью полностью возлагается на программиста. Однако именно эта особенность позволяет глубже понять устройство компьютера и формирует у студентов строгую дисциплину программирования.

Слайд 6

Алгоритм. Программа. Исполнитель

В чем разница между понятиями алгоритма и программы? Алгоритм – концептуальное описание шагов, которые необходимо выполнить для решения той или иной задачи, в то время как программа – это запись алгоритма на специальном языке (программирования), который понятен исполнителю. Исполнитель – нечто, что умеет шаг за шагом выполнять программу, написанную на понятному ему языке.

Слайд 7

Рассмотрим пример алгоритма нахождения минимального элемента в заданной последовательности чисел:

1. Запомним значение первого элемента последовательности как минимальное.

2. Перебираем все элементы последовательности, выполняя для каждого:

а) если значение текущего элемента меньше минимального, то примем его за минимальное.

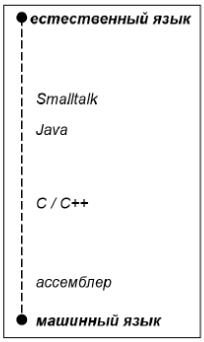
Данный алгоритм показывает идею (рецепт, описание) пошагового решения поставленной задачи. Исполнителем не обязательно должен быть компьютер. Им может быть человек, а инструкции, приведенные выше, можно рассматривать как программу для данного человека-исполнителя. Если же исполнителем является компьютер, то этот алгоритм можно записать на любом языке программирования: Си, Паскаль, Java, Бейсик, Питон и т.д. Программ получится много, но делать они будут одно и то же – реализовывать алгоритм поиска минимального элемента в последовательности. Для того чтобы стать квалифицированным программистом, недостаточно выучить какой-либо из языков программирования – необходимо еще научиться составлять алгоритмы, позволяющие эффективно решать поставленную задачу.

Слайд 8

Задачу сортировки можно решить большим количеством концептуально разных алгоритмов, при этом каждый из них можно запрограммировать на одном из множества языков программирования. В итоге мы имеем огромное количество вариантов программного решения одной поставленной задачи. Поэтому хороший программист не просто должен уметь писать код на конкретном языке программирования – он еще должен уметь разрабатывать эффективные алгоритмы для решения любой поставленной задачи.

Слайд 9

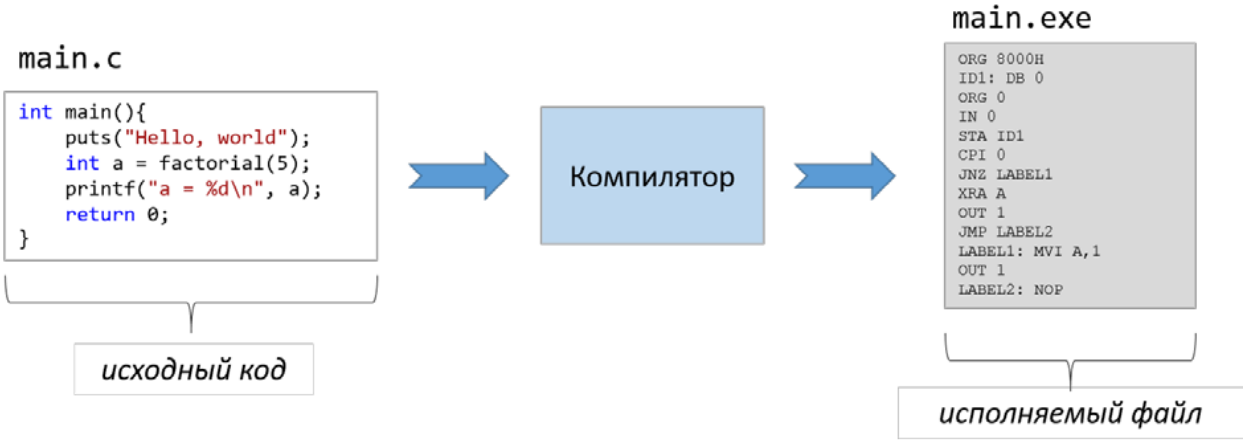
Уровень языка – его позиция в шкале «компьютер-человек». Чем ниже уровень – тем более он понятен компьютеру и менее понятен человеку, и наоборот.



Программы, написанные на языках высокого уровня, похожи на тексты на естественном (чаще всего английском) языке. Например, смысл строки кода «`if (a > b) then min = b`» несложно уловить даже не программисту.
Машинный же язык является языком низкого уровня. Машина не понимает языков высокого уровня, поэтому, прежде чем программу, написанную на языке Си (и любом другом не машинном), сможет выполнить компьютер, ее необходимо перевести на язык, понятный компьютеру. Этим занимается *компилятор*.

Слайд 10

Процесс создания программы на сегодняшний день в самом общем виде изображен на Рис сначала пишется код на высокоуровневом языке программирования, который сохраняется в обычном текстовом файле.



Слайд 11

Часто (в том числе и в нашем случае) компилятор не существует в одиночестве. Вместе с ним прилагается еще ряд программ, призванных упросить процесс разработки приложений – например, специальный текстовый редактор, который знает синтаксис языка Си и умеет подсвечивать разными цветами ключевые слова, имена переменных и другие элементы кода, средства для отладки программ, справочная документация и т.д. Весь этот комплекс программ называется *IDE* (Integrated Development Environment)

интегрированная среда разработки. В отличие от набора дискрет- ных (отдельных) приложений, как например, обычного текстового редактора, компилятора, файла со справочной информацией – IDE представляет собой совокупность очень тесно взаимосвязанных программ, которые с точки зрения программиста могут выглядеть как одно приложение, из которого доступны все необходимые функции. Современный IDE состоит из:

1. интеллектуального текстового редактора с функциями структурирования и подсветки кода, автодополнения и т.д.,
2. встроенной справки,
3. компилятора,
4. линковщика,
5. библиотек кода,
6. средств отладки,
7. средств автоматизированной сборки приложения,
8. средств для интеграции с системами управления версиями кода,
9. средств для совместной разработки,
10. средств для взаимодействия с базами данных,
11. всевозможных утилит, упрощающие разработку и отладку кода

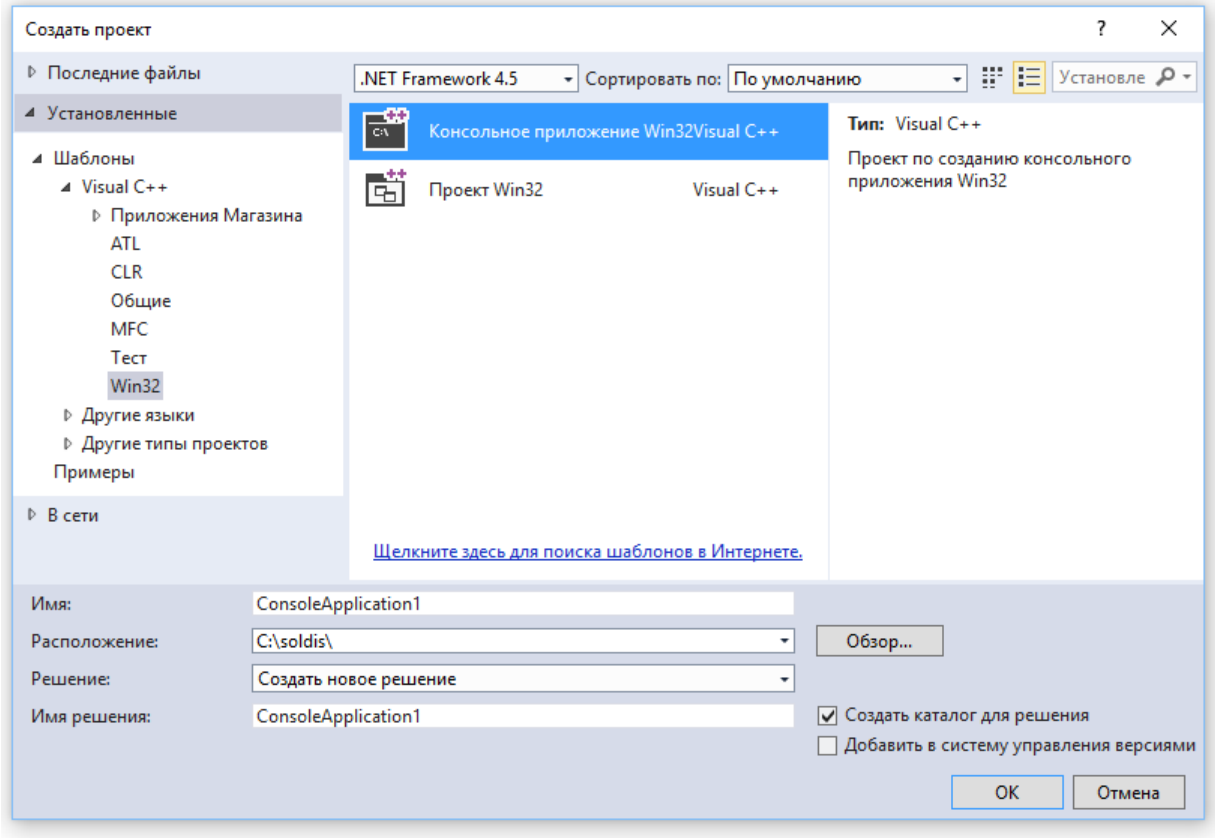
В рамках данного учебного пособия мы будем использовать бесплатно распространяемую среду разработки Microsoft Visual Studio Community

Слайд 12

Создание проекта

Создадим свое первое приложение. Для этого запускаем студию. Нас встречает стартовая страница с ссылками на последние проекты, с которыми мы работали, новости от Microsoft и другая информация справочного характера. Первое, что мы должны сделать – это создать *проект*. Файлы с исходным кодом в студии не существуют сами по себе – они всегда являются частью проекта. Проект объединяет в себе все файлы с исходными кодами и другими ресурсами, необходимыми

для сборки одного приложения. Как мы скоро увидим, программа может собираться из нескольких файлов (крупные приложения могут собираться из сотен самых различных файлов). Например, если вы разрабатываете графическое приложение, то здесь будут собраны все оконные формы вашего приложения, все графические изображения, иконки, звуки и другие медиа-ресурсы.
Сложный программный продукт может состоять из нескольких отдельных программ, то есть являться *программным комплексом*. Примером такового является сама IDE. Для нас это одна программа, хотя в действительности их около сотни. Поэтому в студии все проекты объединяются в так называемые *решения* (solutions). Таким образом, решение – это совокупность проектов, где каждый проект представляет собой отдельную программу.
Нажимаем на ссылку «Создать проект...». Мы увидим мастер по созданию проектов (Рис. 1.4).



Студия позволяет выбрать язык программирования, на котором вы будете вести разработку – C++, C#, Бейсик, Питон и другие. Более того, в студии уже есть ряд шаблонов, например, для разработки веб-приложений, или приложений под мобильные платформы.
Нас интересует шаблон Visual C++, подшаблон Win32, тип «Консольное приложение Win32». Теперь нам необходимо настроить параметры шаблона. Прежде всего, указываем место, где будет создан проект, имя проекта и имя решения. По умолчанию, имя решения

совпадает с именем проекта. Нажимаем ОК и настраиваем дополнительные параметры.

Нам необходимо выбрать *пустой проект* для *консольного приложения* (Рис. 1.5). Нажимаем «Готово».

Слайд 13

На первый взгляд ничего не изменилось. В действительности, на диске по тому пути, который вы указали, было создано несколько папок с файлами для сопровождения вашего проекта, однако главный файл еще отсутствует – файл с исходным кодом.
Посмотрим на структуру проекта в *обозревателе решения* (Solution Explorer, если обозреватель решений не виден, его можно включить в меню «Вид»). Здесь изображена логическая структура нашего проекта (Рис.) – на верхнем уровне представлено решение, в нем перечислены проекты, в которых идут папки для заголовочных файлов, файлов с исходным кодом и файлами ресурсов.

Слайд 14

Нажимаем правой кнопкой на папку «Файлы исходного кода» и выбираем пункт «Создать новый элемент» (Рис.).

Открывается окно с набором шаблонов файлов, которые мы можем создать/добавить в проект – от элементов графического интерфейса до файлов ресурса. Нам надо выбрать Код –> Файл C++
Почему мы выбрали шаблон C++, хотя изучаем язык Си? Дело в том, что язык C и является подмножеством языка C++, то есть компилятор C++ естественным образом транслирует код, написанный на языке Си.

Указываем его имя и нажимаем «Добавить».

Слайд 15

Перепишем в окно текстового редактора следующий код:

```
#include <stdio.h>
int main(){
puts("Hello, world");
int a = 2;
printf("a = %d\n", a);
return 0;
}
```

Прежде всего, обратим внимание, как студия выделяет цветом ключевые слова и автоматически выравнивает код, разбивая его на блоки, которые вы можете сворачивать и разворачивать по мере надобности. Также мы можем заметить многочисленные подсказки, «всплывающие» при наборе текста и при наведении мышкой на различные элементы кода. Все это – работа встроенного в студию интеллектуального текстового редактора.
Смысл написанного кода мы разберем в следующем разделе, а сейчас мы можем скомпилировать и запустить нашу первую программу, которая, при удачном стечении обстоятельств, должна вывести на экран строку приветствия и значение переменной `a`.

Слайд 16

Компилируем решение. Для этого можно выбрать в меню «Сборка» пункт «Собрать решение» или просто нажать на клавиатуре <F7>. В окне «Вывод» снизу экрана мы видим сначала ход, а затем и результат компиляции (Рис. 1.8).

Если все прошло без ошибок, то в последней строке будет написано «`успешно: 1, с ошибками: 0, пропущено: 0`». В этом окне мы также можем увидеть путь, по которому был создан исполняемый файл нашего приложения. Заметим, что исполняемый файл называется не так, как назван файл с исходным кодом, а по имени проекта.

Слайд 17

Изучим содержимое каталога решения:

Прежде всего, мы найдем в нем файл с расширением `*.sln`, который содержит настройки решения. Для открытия нашего проекта в студии достаточно дважды щелкнуть мышью на этом файле. Помимо этого, в этой папке есть отдельный подкаталог для каждого проекта, а также каталоги Debug и Release, куда для удобства после компиляции выносятся исполняемые файлы всех проектов. Все пути можно изменить в настройках решения.
В каталоге проекта мы также видим подкаталог Debug, где хранятся промежуточные результаты компиляции, файл с расширением *.vcxproj с описанием проекта, а также все файлы, добавленные нами к проекту (в том числе и с исходным кодом).

По умолчанию, студия собирает проекты в так называемом *отладочном режиме* (Debug mode), в котором после компиляции создается исполняемый файл, содержащий большое количество отладочной информации. Благодаря этому мы можем использовать предоставляемый студией мощный инструментарий отладки. Когда приложение будет готово, и мы захотим собрать его «начисто», нам надо будет выбрать в панели инструментов студии (Рис.) *режим релиза* (Release mode). После сборки решения в этом режиме рядом с каталогами Debug будут созданы каталоги Release с результатами компиляции. Мы можем заметить, что релиз-версия программы занимает существенно меньше места на диске, чем отладочная версия, потому что из нее убрана вся избыточная информация. Всюду далее будем предполагать, что используется отладочный режим.
Рис. : Выбор режима сборки проекта

Слайд 18

Осталось самое последнее – запустить наше приложение. Мы можем это сделать, нажав кнопку <F5>, однако в этом случае приложение запустится в новом консольном окне, отработает и исчезнет, не дав нам увидеть результат своей работы. Чтобы задержать консольное окно, приложение надо запускать с помощью комбинации клавиш <Ctrl>-<F5>. В этом случае после выполнения программы консольное окно будет держаться на экране до тех пор, пока мы не нажмем любую кнопку на клавиатуре (Рис.).
Рис.: Консольное окно приложения с результатом работы

Слайд 19
Отладка

Вкратце познакомимся с основными средствами и базовыми приемами отладки кода, предоставляемыми студией.
Во-первых, приложение может не скомпилироваться в виду наличия синтаксических ошибок. Соответствующая информация будет выведена в окне «Вывод» внизу рабочего экрана. Попробуем в качестве эксперимента стереть в нашей программе в шестой строке первую кавычку:

```
#include <iostream>

int main(){
    puts("Hello, world");
    int a = 2;
    printf(a = %d\n", a);
    return 0;
}
```

Обратим внимание, что еще до компиляции студия выделила красной волнистой чертой те фрагменты кода, которые вызывают у нее вопросы. Это происходит благодаря тому, что студия делает подробный анализ кода уже тогда, когда вы его только набираете в текстовом редакторе, практически осуществляя его «пробную» компиляцию на лету. Это позволяет заметить ошибки еще до процесса сборки решения, который для сложных программных продуктов может длиться часами. Если же мы все-таки попытаемся откомпилировать приложение, то получим следующий результат в окне вывода:

```
1>------ Сборка начата: проект: ConsoleApplication1, Конфигурация: Debug Win32 ---
1>  main.cpp
1>…\main.cpp(6): error C2017: недопустимая escape-последовательность
1>…\main.cpp(6): error C2065: d: необъявленный идентификатор
1>…\main.cpp(6): error C2001: newline в константе
1>…\main.cpp(6): error C2146: синтаксическая ошибка: отсутствие ")" перед
идентификатором "n"
========== Сборка: успешно: 0, с ошибками: 1, без изменений: 0, пропущено: 0 ========
```

Мы стерли всего один символ, а получили целых четыре сообщения об ошибке! В этом нет ничего удивительного. Очень частое явление при компиляции кода с синтаксическими ошибками – одна ошибка в коде может порождать сразу несколько диагностических сообщений от компилятора.

В нашем пример их четыре, и они все разные. Причина этого заключается в том, что современный компилятор, встретив первую ошибку в коде, не завершает работу, а пытается продолжить компиляцию дальше. Для этого он делает «предположение», каким образом надо исправить код, чтобы он, по его мнению, не содержал ошибок. Однако, к сожалению, программным способом невозможно абсолютно точно определить, где и в чем именно была совершена ошибка, поэтому «предположение» компилятора часто оказывается неверным и его «исправление» порождает новые ошибки, которые компилятор также пытается исправить и т.д.

В большинстве случаев компилятору удается выйти на участок кода, не затрагиваемый хаосом, вызванным допущенной пользователем ошибкой и попытками компилятора «сделать как лучше», и завершить компиляцию. Этот процесс называется восстановлением после ошибки, и он позволяет локализовать некорректные фрагменты кода. Для чего же компилятору это нужно? Для того, чтобы за один проход попытаться найти по возможности большую часть ошибок в коде и

сразу же предоставить эту информацию программисту. Ведь, как уже было сказано ранее, сборка серьезных сложных приложений может занимать по времени даже не минуты, а часы, поэтому перекомпилировать приложение после каждой исправленной одиночной ошибки не всегда удобно.
Для нас же главное – запомнить следующее правило: если в окне вывода мы видим сообщения о нескольких десятках ошибок, это вовсе не означает, что именно столько их присутствует в нашем коде. Скорее всего, после исправления одной-двух, количество диагностических сообщений существенно убавится, если не сведется к нулю. Для локализации предполагаемого места нахождения ошибки в коде, надо дважды щелкнуть мышью на самое первое диагностическое сообщение – курсор автоматически перейдет к нужной строке. Как правило, внимательный анализ строки сразу выявит, что нужно исправить.
Помимо ошибок, компилятор также будет иногда выводить *предупреждения*, которые можно распознать по ключевому слову «warning» в диагностическом сообщении. Предупреждения не мешают компиляции приложения, а предупреждают вас о возможных проблемах в коде: возможно вы используете устаревшую или небезопасную функцию, которая в последней версии среды разработки была заменена, или компилятору «кажется», что вы написали не совсем корректный или безопасный код. В зависимости от ситуации, вы сами принимаете решение, как реагировать на эти предупреждения.
Если приложение скомпилировалось, это еще не значит, что оно корректно. Во-первых, во время работы приложения могут возникнуть так называемые *ошибки времени выполнения* (runtime errors), когда приложение аварийно завершается с сообщением, что оно где-то куда-то неправильно обратилось к памяти. Во-вторых, оно может отработать до конца без каких-либо сообщений об ошибках, но при этом выдать совсем не тот результат, который хотелось бы.

Слайд 20

В этих случаях можно прибегнуть к механизмам отладки, основным из которых является *пошаговое выполнение программы*.

Для этого мы нажимаем кнопку <F10> – приложение запускается, открывается и «зависает» консольное окно, а в текстовом редакторе студии напротив одной из строк появляется желтая стрелка:

Рис.: Запуск пошагового выполнения приложения

Эта стрелка указывает на ту инструкцию, которая будет выполнена на следующем шаге. Чтобы сделать этот самый следующий шаг, надо

снова нажать <F10> и т.д. Так, шаг за шагом, нажимая <F10>, можно «пройтись» по каждой строке кода, смотря на вывод в консольном окне. Если мы хотим закончить пошаговое выполнение, то можно нажать <F5>.
Во время отладки мы можем делать с кодом много интересных вещей. Можем навести на переменную и посмотреть ее текущее значение. Это значение также можно увидеть в специальном окне «Видимые» внизу рабочего экрана. Можем посмотреть на всю цепочку вызовов функций, если таковые были. Если желтая стрелка указывает на строку, содержащую вызов функции, то, чтобы зайти «внутрь», нужно нажать < F11> Таким образом, можно «ходить» по сколь угодно сложному коду, просматривая значения всех переменных в каждый момент времени. В результате, если логика приложения позволяет, можно найти то место в коде, где возникает ошибка.

Слайд 21

Если код слишком большой и «шагать» до нужного места, которое хочется посмотреть повнимательнее, слишком долго, то можно воспользоваться механизмом *точек останова*. Для этого устанавливаем курсор на нужную строку, с которой мы хотели бы начать отладку, и нажимаем <F9>. Напротив этой строки появится красный маркер. Дальше нажимаем <F5> – приложение запускается, начинает работать и как только оно дойдет до помеченной красным кружком строки, оно перейдет в режим пошаговой отладки. Точек останова можно ставить произвольное количество. Чтобы снять точ- ку останова, нужно на соответствующей строке снова нажать <F9>.
Этих простейших методов отладки нам будет достаточно для изучения языка Си на наших упражнениях.

Слайд 22
Написание простейшей программы

Давайте теперь разберем код простейшей программы на языке Си.
Изменим код приложения на следующий (нумерация строк приведена для удобства, в коде программы ее быть не должно):

```
1. #include <stdio.h>
2.
3. int main(){
4. /* Выводим на экран строку */
5. printf("Hello, World!");
6.
7. return 0;
8. }
```

Строка 1 содержит команду препроцессора, подключающую заголовочный файл стандартной библиотеки Си, который дает нашей программе возможность печатать сообщения на экран. `stdio.h` – библиотека для работы с вводом/выводом информации (standard input/output – стандартный ввод/вывод). Если наша программа будет что-то печатать на экран или считывать с клавиатуры, то в ней обязательно должна присутствовать данная строка.
Препроцессор (от англ. предварительная обработка) – это программа, которая запускается до этапа компиляции, просматривает код и выполняет все инструкции, начинающиеся с символа решетки. В частности, инструкция `include` копирует содержимое указанного в ней файла на место этой строки. Первая строка необходима для того, чтобы мы могли пользоваться библиотечными функциями ввода/вывода, одна из которых – `printf`.
Строки 3-8 содержат определение главной функции программы. Функция языка Си, как и любого другого структурированного языка программирования, это именованный фрагмент кода, который можно вызывать по его имени (более подробно с функциями мы познакомимся в одной из следующих глав). Определение функции состоит из заголовка (строка 3) и тела функции (строки 4-7, заключенные в фигурные скобки). Главная функция `main` обязательна в любой программе на Си – это так называемая *точка входа* в программу. Именно она вызывается операционной системой в момент запуска нашей программы.

Слайд 22

Код на языке Си состоит из *операторов*. Оператор – это отдельная инструкция, заканчивающаяся точкой с запятой. Точка с запятой – обязательный разделитель! Самая распространенная ошибка при изучении языка Си – забыть поставить точку с запятой. Строки 5 и 7 содержат два оператора. Их можно было бы написать на одной строке, однако, делать этого не следует – код получится более сложным для восприятия.
На строке 5 указан оператор вызова библиотечной функции печати. Вызов функции в языке Си выглядит следующим образом: сначала пишется имя функции, а затем в круглых скобках через запятую перечисляются передаваемые в функцию аргументы. Функция `printf` (от англ. print formatted, форматированный вывод) печатает на экран переданную ей строчку. Строки в языке Си выделяются двойными кавычками. Круглые скобки – обязательный синтаксис вызова функции. Вводу/выводу также посвящена отдельная глава учебного пособия.
7-я строка – это оператор возврата из функции. Под «возвратом» понимается завершение работы данной функции, а так как main – это главная функция, то данный оператор завершает выполнение всей нашей программы.

Строка 4 – это пример многострочного комментария (который, правда, записан всего на одной строке). Эта строчка игнорируется компилятором. Она необходима программистам, чтобы понимать, что же делает программа. Хороший код – это хорошо оформленный и достаточно подробно комментированный код. Многострочность означает, что мы можем закомментировать подобным образом несколько строк кода. Если мы хотим закомментировать только часть какой-то одной строки, то можно воспользоваться синтаксисом однострочного комментария, поставив перед комментируемой частью два слеша //.
Пробуем скомпилировать и выполнить. Результат очевиден – на экране будет напечатано приветствие. Теперь усложним программу. Напишем второй оператор печати строки после первого. Пусть компьютер представится:

```
printf("Hello, World!");
printf("My name is John");
```

Что получилось? Ерунда! Строчки склеились друг с другом. Для того чтобы этого не происходило, курсор необходимо перевести на новую строку после печати первой фразы. Делается это путем добавления в выводимую строку специальной последовательности символов - \n3:

```
printf("Hello, World!\n");
printf("My name is John");
```

Функция printf, встретив данную комбинацию символов, перенесет курсор на новую строку.