Языки и методы программирования

Лекции 1-16

Лекция **1.** Язык программирования

- **Язык программирования** — формальная — формальная знаковая система — формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических — формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических —

- Со времени создания первых программируемых машин человечество придумало **более двух с половиной тысяч языков программирования** (включая абстрактные и нестандартные языки). Каждый год их число увеличивается. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие становятся известны миллионам людей. Профессиональные программисты иногда применяют в своей работе более десятка разнообразных языков программирования.

Создатели языков по-разному толкуют понятие *язык программирования*. К наиболее распространённым утверждениям, признаваемым большинством разработчиков, относятся следующие:

- *Функция:* язык программирования предназначен для написания компьютерных программ, которые применяются для передачи компьютеру инструкций по выполнению того или иного вычислительного процесса и организации управления отдельными устройствами.
- *Задача:* язык программирования отличается от естественных языков тем, что предназначен для передачи команд и данных от человека к компьютеру, в то время как естественные языки используются для общения людей между собой. Можно обобщить определение «языков программирования» — это способ передачи команд, приказов, чёткого руководства к действию; тогда как человеческие языки служат также для обмена информацией.
- *Исполнение:* язык программирования может использовать специальные конструкции для определения и манипулирования структурами данных и управления процессом вычислений.

- **Ранние этапы развития**
- Можно сказать, что первые языки программирования возникали еще до появления современных электронных вычислительных машин: уже в XIX веке были изобретены устройства, которые можно с долей условности назвать программируемыми — к примеру, механические пианино и ткацкие станки. Для управления ими использовались наборы инструкций, которые в рамках современной классификации можно назвать предметно-ориентированными языками программирования. К началу XX века для кодирования данных и управления разнообразными механическими операциями начали применяться перфокарты. Позднее, в 1930—1940 годах, А. Чёрч и А. Тьюринг разработали математические абстракции — лямбда-исчисление и машину Тьюринга соответственно — для формализации алгоритмов; первая из упомянутых абстракций сохраняет свое влияние на построение языков программирования и по сей день

- В это же время, в 1940-е годы, появились электрические цифровые компьютеры и был разработан язык, который можно считать первым высокоуровневым языком программирования для ЭВМ — «PlankalkülВ это же время, в 1940-е годы, появились электрические цифровые компьютеры и был разработан язык, который можно считать первым высокоуровневым языком программирования для ЭВМ — «Plankalkül», созданный немецким инженером К. ЦузеВ это же время, в 1940-е годы, появились электрические цифровые компьютеры и был разработан язык, который можно считать первым высокоуровневым языком программирования для ЭВМ —

- Программисты ЭВМ начала 1950-х годовПрограммисты ЭВМ начала 1950-х годов, в особенности таких, как UNIVACПрограммисты ЭВМ начала 1950-х годов, в особенности таких, как UNIVAC и IBM 701, при создании программПрограммисты ЭВМ начала 1950-х годов, в особенности таких, как UNIVAC и IBM 701, при создании программ пользовались непосредственно машинным языкомПрограммисты ЭВМ начала 1950-х годов, в особенности таких, как UNIVAC и IBM 701, при создании программ пользовались непосредственно машинным языком — то есть писали на языке первого поколения. Вскоре на

- Позднее, к концу десятилетия, языки второго поколения были усовершенствованы: в них появилась поддержка макрокомандПозднее, к концу десятилетия, языки второго поколения были усовершенствованы: в них появилась поддержка макрокоманд. Одновременно с этим начали появляться уже и языки третьего поколения — такие, как ФортранПозднее, к концу десятилетия, языки второго поколения были усовершенствованы: в них появилась поддержка макрокоманд. Одновременно с этим начали появляться уже и языки третьего поколения — такие, как Фортран, ЛиспПозднее, к концу десятилетия, языки второго поколения были усовершенствованы: в них появилась поддержка макрокоманд. Одновременно с этим начали появляться уже и языки третьего поколения — такие, как Фортран, Лисп и Кобол[4]. Языки программирования этого типа более абстрактны и универсальны, не имея жесткой зависимости от конкретной аппаратной платформы и используемых на ней машинных команд. Обновленные версии перечисленных языков до сих пор

Совершенствование

- В период 1960-хВ период 1960-х — 1970-х годов были разработаны основные парадигмы языков программирования, используемые в настоящее время, хотя во многих аспектах этот процесс представлял собой лишь улучшение идей и концепций, заложенных еще в первых языках третьего поколения.

- Язык APLЯзык APL оказал влияние на функциональное программированиеЯзык APL оказал влияние на функциональное программирование и стал первым языком, поддерживавшим обработку массивов[7].
- Язык ПЛ/1 (NPL) был разработан в 1960-х годах как объединение лучших черт Фортрана и Кобола.
- Язык СимулаЯзык Симула, появившийся примерно в это же время, впервые включал поддержку объектно-ориентированного программированияЯзык Симула, появившийся примерно в это же время, впервые включал поддержку объектно-ориентированного программирования. В середине 1970-х группа специалистов представила язык Smalltalk, который был уже всецело объектно-ориентированным.
- В период с 1969В период с 1969 по 1973 годыВ период с 1969 по 1973 годы велась разработка языка Си, популярного и по сей день [8].
- В 1972 годуВ 1972 году был создан ПрологВ 1972 году был создан Пролог — первый язык логического программирования.
- В 1978 годуВ 1978 году в языке MLВ 1978 году в языке ML была реализована расширенная система полиморфнойВ 1978 году в языке ML была реализована расширенная система полиморфной типизацииВ 1978 году в языке ML была реализована расширенная

- Кроме того, в 1960 — 1970х годах активно велись споры о необходимости поддержки структурного программирования в тех или иных языках[9]. В частности, голландский специалист Э. Дейкстра выступал в печати с предложениями о полном отказе от использования инструкций GOTO во всех высокоуровневых языках. Развивались также приемы, направленные на сокращение объема программ и повышение продуктивности работы программиста и пользователя; в итоге наборы инструкций на языках четвертого поколения уже требовали существенно меньшего количества перфокарт для их записи, нежели аналогичные программы на языках третьего поколения.

Объединение и развитие

- В 1980-е годыВ 1980-е годы наступил период, который можно условно назвать временем консолидации. Язык C++В 1980-е годы наступил период, который можно условно назвать временем консолидации. Язык C++ объединил в себе черты объектно-ориентированного и системного программирования, правительство СШАВ 1980-е годы наступил период, который можно условно назвать временем консолидации. Язык C++ объединил в себе черты объектно-ориентированного и системного программирования, правительство США стандартизировало язык АдаВ 1980-е годы наступил период, который можно условно назвать временем консолидации. Язык C++ объединил в себе черты объектно-ориентированного и системного программирования, правительство США стандартизировало язык Ада, производный от ПаскаляВ 1980-е годы наступил период, который можно условно назвать временем консолидации. Язык C++ объединил в себе черты объектно-ориентированного и системного программирования, правительство США стандартизировало язык Ада, производный от Паскаля и

- Важной тенденцией, которая наблюдалась в разработке языков программирования для крупномасштабных систем, было сосредоточение на применении модулей — объемных единиц организации кода. Хотя некоторые языки, такие, как ПЛ/1, уже поддерживали соответствующую функциональность, модульная система нашла свое отражение и применение также и в языках Модула-2Важной тенденцией, которая наблюдалась в разработке языков программирования для крупномасштабных систем, было сосредоточение на применении модулей — объемных единиц организации кода. Хотя некоторые языки, такие, как ПЛ/1, уже поддерживали соответствующую функциональность, модульная система нашла свое

- В 1990-х годах в связи с активным развитием Интернета распространение получили языки, позволяющие создавать сценарии для веб-страниц — главным образом Perl, развившийся из скриптового инструмента для Unix-систем, и Java. Возрастала также и популярность технологий виртуализации. Эти изменения, однако, также не представляли собой фундаментальных новаций, являясь скорее совершенствованием уже существовавших парадигм и языков (в последнем случае — главным образом семейства Си).
- В настоящее время развитие языков программирования идет в направлении повышения безопасности и надежности, создания новых форм модульной организации кода и интеграции с базами данных.

Стандартизация языков программирования

- Язык программирования может быть представлен в виде набора спецификаций, определяющих его синтаксис и семантику.
- Для многих широко распространённых языков программирования созданы международные стандарты. Специальные организации проводят регулярное обновление и публикацию спецификаций и формальных определений соответствующего языка. В рамках таких комитетов продолжается разработка и модернизация языков программирования и решаются вопросы о расширении или поддержке уже существующих и новых языковых конструкций.

Лекция 2. Типы и структура данных. Семантика языков программирования.

Типы данных.

- Современные цифровые компьютеры являются двоичными и данные хранят в двоичном (бинарном) коде (хотя возможны реализации и в других системах счисления). Эти данные как правило отражают информацию из реального мира (имена, банковские счета, измерения и др.), представляющую высокоуровневые концепции.
- Особая система, по которой данные организуются в программе, — это *система типов* языка программирования; разработка и изучение систем типов известна под названием теория типов. Языки можно поделить на имеющие *статическую типизацию* и *динамическую типизацию*, а также *бестиповые языки* (например, *Forth*).
- Статически типизированные языки могут быть в дальнейшем подразделены на языки с *обязательной декларацией*, где каждая переменная и объявление функции имеет обязательное объявление типа, и языки с *выводимыми типами*. Иногда динамически типизированные языки называют *латентно типизированными*.

Структуры данных

- Системы типов в языках высокого уровня позволяют определять сложные, составные типы, так называемые структуры данных. Как правило, структурные типы данных образуются как декартово произведение базовых (атомарных) типов и ранее определённых составных типов.
- Основные структуры данных (списки, очереди, хеш-таблицы, двоичные деревья и пары) часто представлены особыми синтаксическими конструкциями в языках высокого уровня. Такие данные структурируются автоматически.

Семантика языков программирования

- Существует несколько подходов к определению семантики языков программирования.
- Наиболее широко распространены разновидности следующих трёх: операционного, деривационного (аксиоматического) и денотационного (математического).
- При описании семантики в рамках *операционного* подхода обычно исполнение конструкций языка программирования интерпретируется с помощью некоторой воображаемой (абстрактной) ЭВМ.
- *Деривационная* семантика описывает последствия выполнения конструкций языка с помощью языка логики и задания пред- и постусловий.
- *Денотационная* семантика оперирует понятиями, типичными для математики — множества, соответствия, а также суждения, утверждения и др.

Парадигма программирования

- Язык программирования строится в соответствии с той или иной базовой моделью вычислений и парадигмой программирования.
- Несмотря на то, что большинство языков ориентировано на императивную модель вычислений, задаваемую фон-неймановской архитектурой ЭВМ, существуют и другие подходы. Можно упомянуть языки со стековой вычислительной моделью (Форт, Factor, PostScript и др.), а также функциональное (Лисп, Haskell, ML, F#, РЕФАЛ, основанный на модели вычислений, введённой советским математиком А. А. Марковым-младшим и др.) и логическое программирование (Пролог).
- В настоящее время также активно развиваются проблемно-ориентированные, декларативные и визуальные языки программирования

Способы реализации языков

- Языки программирования могут быть реализованы как компилируемые и интерпретируемые.
- Программа на компилируемом языке при помощи компилятора (особой программы) преобразуется (компилируется) в машинный код (набор инструкций) для данного типа процессора и далее собирается в исполнимый модуль, который может быть запущен на исполнение как отдельная программа. Другими словами, компилятор переводит исходный текст программы с языка программирования высокого уровня в двоичные коды инструкций процессора.
- Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) исходный текст без предварительного перевода. При этом программа остаётся на исходном языке и не может быть запущена без интерпретатора. Процессор компьютера, в этой связи, можно назвать интерпретатором для машинного кода.

- Разделение на компилируемые и интерпретируемые языки является условным. Так, для любого традиционно компилируемого языка, как, например, Паскаль, можно написать интерпретатор. Кроме того, большинство современных «чистых» интерпретаторов не исполняют конструкции языка непосредственно, а компилируют их в некоторое высокоуровневое промежуточное представление (например, с разыменованием переменных и раскрытием макросов).
- Для любого интерпретируемого языка можно создать компилятор — например, язык Лисп, изначально интерпретируемый, может компилироваться без каких бы то ни было ограничений. Создаваемый во время исполнения программы код может так же динамически компилироваться во время исполнения.

- Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных программ, так как уже переведены на машинный язык. Вместе с тем, при каждом изменении текста программы требуется её перекомпиляция, что замедляет процесс разработки. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной системой, на которую был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа, требуется новая компиляция.

- Интерпретируемые языки обладают некоторыми специфическими дополнительными возможностями (см. выше), кроме того, программы на них можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке может быть зачастую запущена на разных типах машин и операционных систем без дополнительных усилий.
- Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, кроме того, они не могут выполняться без программы-интерпретатора.

- Некоторые языки, например, Java и C#, находятся между компилируемыми и интерпретируемыми. А именно, программа компилируется не в машинный язык, а в машинно-независимый код низкого уровня, байт-код. Далее байт-код выполняется виртуальной машиной. Для выполнения байт-кода обычно используется интерпретация, хотя отдельные его части для ускорения работы программы могут быть транслированы в машинный код непосредственно во время выполнения программы по технологии компиляции «на лету» (Just-in-time compilation, JIT). Для Java байт-код исполняется виртуальной машиной Java (Java Virtual Machine, JVM), для C# — Common Language Runtime.
- Подобный подход в некотором смысле позволяет использовать плюсы как интерпретаторов, так и компиляторов. Следует упомянуть, что есть языки, имеющие и интерпретатор, и компилятор (Форт).

Языки программирования низкого уровня

- Первые компьютеры приходилось программировать двоичными машинными кодами. Однако программировать таким образом - довольно трудоемкая и тяжелая задача. Для упрощения этой задачи начали появляться языки программирования низкого уровня, которые позволяли задавать машинные команды в понятном для человека виде. Для преобразования их в двоичный код были созданы специальные программы - трансляторы.
- Трансляторы делятся на:
- компиляторы - превращают текст программы в машинный код, который можно сохранить и после этого использовать уже без компилятора (примером является исполняемые файлы с расширением *.exe) .
- интерпретаторы - превращают часть программы в машинный код, выполняют его и после этого переходят к следующей части. При этом каждый раз при выполнении программы используется интерпретатор .

- Примером языка низкого уровня является ассемблер. Языки низкого уровня ориентированы на конкретный тип процессора и учитывают его особенности, поэтому для переноса программы на ассемблере на другую аппаратную платформу ее нужно почти полностью переписать. Определенные различия есть и в синтаксисе программ под разные компиляторы. Правда, центральные процессоры для компьютеров фирм AMD и Intel практически совместимы и отличаются лишь некоторыми специфическими командами. А вот специализированные процессоры для других устройств, например, видеокарт и телефонов содержат существенные различия.
- Языки низкого уровня, как правило, используют для написания небольших системных программ, драйверов устройств, модулей стыков с нестандартным оборудованием, программирование специализированных микропроцессоров, когда важнейшими требованиями являются компактность, быстродействие и возможность прямого доступа к аппаратным ресурсам.
- Ассемблер - язык низкого уровня, широко применяется до сих пор.

Языки программирования высокого уровня

- Особенности конкретных компьютерных архитектур в них не учитываются, поэтому созданные приложения легко переносятся с компьютера на компьютер. В большинстве случаев достаточно просто перекомпилировать программу под определенную компьютерную архитектурную и операционную систему. Разрабатывать программы на таких языках значительно проще и ошибок допускается меньше. Значительно сокращается время разработки программы, что особенно важно при работе над большими программными проектами .
- Сейчас в среде разработчиков считается, что языки программирования, которые имеют прямой доступ к памяти и регистров или имеют ассемблерные вставки, нужно считать языками программирования с низким уровнем абстракции. Поэтому большинство языков, считавшихся языками высокого уровня до 2000 года сейчас уже таковыми не считаются.

- Адресный язык программирования
- Фортран

- Кобол
- Алгол
- Pascal
- Java
- C
- C++
- Objective C
- Smalltalk
- C#
- Delphi

- Недостатком языков высокого уровня является большой размер программ по сравнению с программами на языках низкого уровня. Сам текст программ на языке высокого уровня меньше, однако, если взять в байтах, то код, изначально написанный на ассемблере, будет более компактным. Поэтому в основном языки высокого уровня используются для разработки программного обеспечения компьютеров и устройств, которые имеют большой объем памяти. А разные подвиды ассемблера применяются для программирования других устройств, где критичным является размер программы.

Используемые символы

- Современные языки программирования рассчитаны на использование ASCII, то есть доступность всех *графических* символов ASCII является необходимым и достаточным условием для записи любых конструкций языка. *Управляющие* символы ASCII используются ограниченно: допускаются только возврат каретки CR, перевод строки LF и горизонтальная табуляция HT (иногда также вертикальная табуляция VT и переход к следующей странице FF).
- *Подробнее по этой теме см.: Переносимый набор символов.*
- Ранние языки, возникшие в эпоху 6-битных символов, использовали более ограниченный набор. Например, алфавит Фортрана включает 49 символов (включая пробел): A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 = + - * / () . , $ ' :

- Заметным исключением является язык APL, в котором используется очень много специальных символов.
- Использование символов за пределами ASCII (например, символов KOI8-R или символов Юникода) зависит от реализации: иногда они разрешаются только в комментариях и символьных/строковых константах, а иногда и в идентификаторах. В СССР существовали языки, где все ключевые слова писались русскими буквами, но большу́ю популярность подобные языки не завоевали (исключение составляет Встроенный язык программирования 1С:Предприятие).
- *Подробнее по этой теме см.: Языки программирования с ключевыми словами не на английском.*

- Расширение набора используемых символов сдерживается тем, что многие проекты по разработке программного обеспечения являются международными. Очень сложно было бы работать с кодом, где имена одних переменных записаны русскими буквами, других — арабскими, а третьих — китайскими иероглифами. Вместе с тем, для работы с текстовыми данными языки программирования нового поколения (Delphi 2006Расширение набора используемых символов сдерживается тем, что многие проекты по разработке программного обеспечения являются международными. Очень сложно было бы работать с кодом, где имена одних переменных записаны русскими буквами, других — арабскими, а третьих — китайскими иероглифами. Вместе с тем, для работы с текстовыми данными языки программирования нового поколения (Delphi 2006, C#Расширение набора используемых

Классы языков программирования

- Функциональные
- Процедурные (императивные)
- Стековые
- Аспектно-ориентированные
- Декларативные
- Динамические
- Учебные
- Описания интерфейсов
- Прототипные
- Объектно-ориентированные
- Рефлексивные — поддерживающие отражение
- Логические
- Скриптовые (сценарные)
- Эзотерические

Лекция **3.** Общая характеристика языка Си

- Си - язык системного программирования; его принято относить к языкам *среднего уровня*, позволяющим выполнять как стандартные высокоуровневые подпрограммы, так и ассемблерно-ориентированный код.
- Можно выделить следующие основные особенности Си:
- легкий доступ к аппаратным средствам компьютера, позволяющий писать высокоэффективные программы;
- высокая переносимость написанных на Си программ - как между компьютерами с различной архитектурой, так и между различными операционными средами;
- принцип построения "что пишем, то и получаем", т. е., в состав компилятора не включен код, который мог бы проверить корректность работы программы в процессе ее выполнения;
- в транслятор не включена информация о стандартных функциях, отсутствуют операции, имеющие дело непосредственно с составными объектами;
- компактный синтаксис, потенциально приводящий к трудноуловимым ошибкам.

Алфавит языка Си

- *Константа* в Си может представлять собой число, символ или строку символов.
- *Целочисленные* константы записываются, в зависимости от используемой системы счисления, в одной из следующих форм:
- десятичная: цифры от 0 до 9 со знаком "+", "-" или без знака. Примеры: 15, -305.
- восьмеричная: лидирующий 0, далее цифры от 0 до 7. Примеры: 0777, 0150.
- шестнадцатеричная: лидирующий 0, далее символ "x" или "X", затем цифры от 0 до 9 и/или символы A-F или a-f. Примеры: 0x10, 0XFF.
- Целочисленные константы могут иметь тип данных int (целочисленный) или long (длинный целый).

- *Константы с плавающей точкой* имеют следующую общую форму записи:
- [+ или -][цифры].[цифры][E][+ или -][цифры]
- Здесь E - признак экспоненциальной формы записи, задаваемый символом E или e. Либо целая, либо дробная часть константы могут быть опущены, но не обе сразу. Либо десятичная точка с дробной частью, либо экспонента могут быть опущены, но не обе сразу. Примеры: -2.251e6, .45, 1.E-03, 1e-30.
- *Символьная* константа - это буква, цифра, знак пунктуации или специальный символ, заключенный в апострофы: 'c'. Значение символьной константы равно ASCII-коду представляемого ею символа. Символ с может быть любым, за исключением апострофа ' (записывается как '\''), обратного слеша \ ('\\') и новой строки ('\n'). Примеры символьных констант приведены в табл. 2.1.

- *Таблица 2.1 Примеры символьных констант*

Константа	Значение
'a'	Малая латинская буква a
'\007'	Символ с кодом 7 ("звонок")
'\b'	Символ "забой" (BackSpace)
'\x1B'	Символ ESC в коде ASCII

- Символьные константы имеют тип char или int. Младший байт хранит код символа, а старший байт, если он есть, - знаковое расширение младшего байта.
- *Множество символов* языка Си включает символы ASCII-кода, при этом прописные и строчные буквы *различаются* компилятором в любом контексте.
- *Разделителями* языка являются символы пробела, табуляции, перевода строки, возврата каретки, новой страницы, вертикальной табуляции и комментариев (см. табл. 2.2).
- Специальные символы предназначены для представления пробельных и неграфических знаков в символьных константах и строках. Специальный символ состоит из обратного слэша, за которым следует либо буква, либо знаки пунктуации, либо комбинация цифр. Специальные символы языка Си перечислены в табл.2.2.

- Таблица 2.2. Специальные символы языка Си

Специальный символ	16-ричная запись в коде ASCII	Наименование
\n	0A	Новая строка (перевод строки)
\t	09	Горизонтальная табуляция
\v	0B	Вертикальная табуляция
\b	08	Backspace
\r	0D	Возврат каретки
\f		Новая страница
\a	07	Звуковой сигнал
\'	2C	Апостроф
\"	22	Двойная кавычка
\\		Обратный слеш
\ddd		Байтовое восьмеричное значение
\xdd		Байтовое шестнадцатеричное значение

- Стандартные библиотечные функции ввода и вывода текстовой информации обычно рассматривают пару символов \r\n как один символ.
- Конструкция \ddd позволяет задать произвольное байтовое значение как последовательность от одной до трех восьмеричных цифр. Конструкция \xdd позволяет задать произвольное байтовое значение как последовательность от одной до двух шестнадцатеричных цифр
- Нулевой код может быть записан как \0 или \x0.
- Символ <Ctrl>+<Z> (шестнадцатеричный код 1A) рассматривается как индикатор конца файла (символ EOF).

- *Комментарии* компилятор Си также рассматривает как пробельные символы. Определены комментарии двух видов:
- /* многострочный */. Комментируется весь текст после комбинации символов /* до первой встретившейся комбинации */. Вложение многострочных комментариев опционально, т.е., зависит от настройки компилятора, поэтому не рекомендуется.
- // однострочный. Комментируется текст после комбинации символов // до конца строки.

- *Символьная строка* - это последовательность символов, заключенная в двойные кавычки. В Си строка рассматривается как массив символов, каждый элемент которого представляет отдельный символ. Строка может содержать произвольное (в том числе нулевое) количество представимых символов, за исключением двойной кавычки ("), обратного слэша (\) и новой строки. Примеры: "Это символьная строка\n", "Первый \\ Второй".
- Для формирования символьных строк, занимающих несколько строк текста программы, используется комбинация символов "обратный слеш" и "новая строка":
- printf ("\nHello,\
- world");

- Нулевой символ ('\0') автоматически добавляется в качестве последнего байта символьной строки и служит признаком ее конца. Таким образом, строка из N символов занимает N+1 байт памяти. В отличие от Паскаля, длина строки нигде не хранится. Каждая символьная строка в программе рассматривается как отдельный объект. Тип строки - массив элементов символьного типа данных char.

- К *идентификаторам* относятся имена переменных, функций и меток в программе на Си.
- Идентификатор Си - это последовательность из одной или более латинских букв, цифр и символов подчеркивания, которая начинается с буквы или символа подчеркивания. Допускается любое число символов в идентификаторе, однако только первые 32 символа рассматриваются компилятором как значащие.
- При использовании подчеркивания в качестве первого символа идентификатора необходимо соблюдать осторожность, поскольку такие идентификаторы могут совпасть (войти в конфликт) с именами "скрытых" библиотечных функций.

- Компилятор языка Си не допускает использования идентификаторов, совпадающих по написанию с ключевыми словами. Так, идентификатор do недопустим, однако Do или DO возможен.
- Ключевые слова - это предопределенные идентификаторы, которые имеют специальное значение для компилятора Си. Их использование строго регламентировано. При необходимости можно с помощью *директивы препроцессора* #define определить для ключевых слов другие имена. В общем случае директива #define располагается на отдельной строке и имеет вид
- #define НовоеКлючевоеСлово ИдентификаторСи
- Примеры:
- #define boolean int
- #define begin {
- #define word unsigned int

Лекция 4. Структура программы

- Программа на Си включает следующие элементы:
- *директивы препроцессора* - определяют действия по преобразованию программы *перед* компиляцией, а также включают инструкции, которым компилятор следует *во время* компиляции;
- *объявления* - описания переменных, функций, структур, классов и типов данных;
- *определения* - тела выполняемых функций проекта.

Объявление переменной

- Объявление переменной – задает имя и атрибуты переменной, приводит к выделению для нее памяти, а также может явно или неявно задавать начальное значение:
-
- int x,y; float r=0;
- Все переменные в языке Си должны быть явно объявлены перед использованием. Объявления имеют следующий общий синтаксис:
- <КлассПамяти> <Знаковость> <Длина> Тип СписокПеременных;
-
- Все указания, перечисленные в треугольных скобках, могут быть опущены. Список состоит из одной переменной или имен переменных, перечисленных через запятую.

- Класс памяти может принимать следующие значения:
- отсутствует или **auto** - переменная определена в том блоке { }, в котором описана, и вложенных в него блоках. Определенная вне всех блоков переменная видима до конца файла. Класс памяти принят по умолчанию в объявлении переменной на внутреннем уровне. Переменные класса auto автоматически не инициализируются. Память отводится в стеке. Как правило, ключевое слово auto опускается.
- **static** - переменная существует глобально независимо от того, на каком уровне блоков определена. Область действия - до конца файла, в котором она определена. По умолчанию имеет значение 0.
- **extern** - переменная или функция определена в другом файле, объявление представляет собой ссылку, действующую в рамках одного проекта;
- **register** - (только для типов данных **char** и **int**) -переменная при возможности хранится в регистре процессора.

- ***Знаковость*** может быть указана только для перечислимых (порядковых) типов, таких как char, int. Знаковость может принимать одно из двух значений:
- signed (по умолчанию) - переменная со знаком;
- unsigned - переменная без знака.
- ***Длина*** определена для типов **int, float, double*:***
- **short** - короткий вариант типа;
- отсутствует - вариант типа по умолчанию;
- **long** - длинный вариант типа.
- Действие этих модификаторов зависит от компилятора и аппаратной платформы. Например, на IBM-PC совместимых компьютерах типы short int и int совпадают и занимают по 2 байта оперативной памяти, long int требует выделения 4 байт.

- Базовыми *типами данных* являются:
- char - символьный;
- int - целочисленный;
- float - вещественный (плавающий) одинарной точности;
- double - вещественный (плавающий) двойной точности;
- void - "пустой" тип, имеет специальное назначение. Указание void в объявлении функции означает, что она не возвращает значений, а в списке аргументов объявления функции - что функция не принимает аргументов. Нельзя создавать переменные типа void, но можно создавать указатели.
- Переменные любых типов могут быть объявлены в любом месте проекта.

Объявление функции

- Объявление функции (описание прототипа) задает ее имя, тип возвращаемого значения и может задавать атрибуты ее формальных параметров. Общий вид объявления следующий:
-
- ТипФункции имя (тип1 параметр1, :, типN параметрN);

- Объявление необходимо для функций, которые описаны ниже по тексту, чем вызваны или описаны в другом файле проекта:

```
float f1(double t, double v) {
 return t+v;
 //Для этой функции прототип не указан
}
extern char f0 ();
 //f0 определена в другом файле проекта
int f2(); //Прототип нужен т.к. тело функции
 //определено ниже ее вызова
void main () {
 f1 (1,2); f2 ();
}
int f2(void){//в прототипе не указан тип
  //функции, предполагается int
 return 0;
}
```

- Функции могут быть объявлены в любом месте проекта. Для подключения функции из другого файла проекта можно использовать:
- модификатор extern;
- включение заголовочного файла внешнего модуля директивой препроцессора
- #include <ИмяФайла.h>

- В примере ниже функция f4() определена во внешнем файле и должна быть доступна к моменту сборки приложения:
-
- int f1 (int n) {
- extern int f4(int);
- return f4(n);
- }
-

- Обратите внимание, что если у функции опущен тип, то предполагается int:
- main () {
- return 0;
- }
-
- но
-
- void main () { : }

В следующем примере в проект включены 2 файла, file1.cpp и file2.cpp. При этом прототип функции f4(), определенной в файле file2.cpp, включен в заголовочный файл file2.h и подключен к файлу file1.cpp с помощью директивы #include:

---------- листинг file1.cpp

```
#include <stdio.h>
#include <file2.h>

int f3(int n) {
 return f4(n);
}

void main () {
 printf ("\n%d",f3(3));
}
```

---------- листинг file2.cpp

```
int f4 (int k) {
 return ++k;
}
```

---------- листинг file2.h

```
int f4 (int);
```

- **Директива #include** предлагает компилятору включить другой исходный файл, имя которого указывается после директивы. Имя файла заключается в двойные кавычки или в <>. Например, следующие две директивы указывают компилятору на необходимость подключить заголовочные файлы стандартной библиотеки ввода/вывода:
-
- #include "stdio.h"
- #include <stdio.h>

- Подключаемые файлы также могут иметь директивы #include. Если это имеет место, то говорят о вложенных подключениях. Например, следующая программа подключает файл, который, в свою очередь, подключает другой файл:

```
/* файл программы */
#include <stdio.h>
int main(void)
{
#include "one"
return 0;
}
/* подключаемый файл ONE */
printf("This is from the first include file.\n");
#include "two"
/* подключаемый файл TWO */
printf("This is from the second include file.\n");
```

- Если подключаемый файл указан в <>, то поиск будет происходить в стандартных каталогах, предназначенных для хранения заголовочных файлов. В случае, если подключаемый файл заключен в двойные кавычки, поиск будет происходить в текущем рабочем каталоге. Если файл не найден, то поиск продолжается в стандартных каталогах.

Лекция 5. Объявление типа

- Объявление типа позволяет создать собственный тип данных. Оно состоит в присвоении имени некоторому базовому или составному типу языка Си. Для типа понятия объявления и определения совпадают.
- Первый вид объявления позволяет определить тег (наименование типа) и ассоциированные с тегом элементы структуры, объединения или перечисления. После такого объявления имя типа может быть использовано в объявлениях переменных и функций для ссылки на него:
- struct list {
- char name [20];
- long int phone;
- }
- struct list mylist [20];
- Здесь объявлен структурный тип list, а затем описан массив структур типа list с именем mylist, состоящий из 20 элементов.

- Второй вид объявления типа использует ключевое слово typedef. Это объявление позволяет присвоить осмысленные имена типам, уже существующим в языке или создаваемым пользователем:

```
typedef float real;

typedef long int integer;

typedef struct {
 float x,y;
} Point;
Point r;
```

- Обратите внимание, что в отличие от директивы #define, имеющей синтаксис
-
- #define новое_слово старое_слово
-
- определение существующего типа через typedef имеет вид
-
- typedef старый_тип новый_тип;
-
- Типы могут быть объявлены в любом месте проекта, но для надежности следует делать их объявления глобальными.

- Пример ниже показывает особенности, связанные с объявлением типов внутри блока:

```
#include <stdio.h>
typedef unsigned int word;
void f() {
 typedef unsigned long longint;
 #define byte unsigned char
 byte b;
}
void main () {
 word w=65535;
 byte b=65; //ошибки нет - #define
 //действует и вне блока, в котором указан
 longint l; //ошибка - тип longint
 //определен только внутри функции f()
 printf ("\n%u,%c",w,b);
}
```

Определение функции

- **Определение функции** задает ее тело, которое представляет собой составной оператор (блок), содержащий другие объявления и операторы. Определение функции также задает имя функции, тип возвращаемого значения и атрибуты ее формальных параметров:

```
int f (int a, int b) {
 return (a+b)/2;
}
```

- В определении функции допускается указание спецификации класса памяти static или extern, а также модификаторов типа функций рассматриваемых позже.
- Тип возвращаемого значения, задаваемый в определении функции перед ее именем, должен соответствовать типу возвращаемого значения во всех объявлениях этой функции, если они имеются в программе.
- При вызове функции ее выполнение начинается с первого оператора. Функция возвращает управление при выполнении оператора return значение;, либо когда выполнение доходит до конца тела функции.

- В первом случае значение вычисляется, преобразуется к типу возвращаемого значения и возвращается в точку вызова функции. Если оператор return отсутствует или не содержит выражения, то возвращаемое значение функции не определено. Если в этом случае вызывающая функция ожидает возвращаемое значение, то поведение программы непредсказуемо.
- Список объявлений формальных параметров функции содержит их описания через запятую. Тело функции (составной оператор) начинается непосредственно после списка. Список параметров может быть пустым, но и в этом случае он должен быть ограничен круглыми скобками. Если функция не имеет аргументов, рекомендуется указать это явно, записав в списке объявлений параметров ключевое слово void.

- Формальные параметры могут иметь базовый тип, либо быть структурой, объединением, указателем или массивом. Указание первой (или единственной) размерности для массива не обязательно. Массив воспринимается как указатель на тип элементов массива.
- Параметры могут иметь класс памяти auto (по умолчанию) или register. По умолчанию формальный параметр имеет тип int.
- Идентификаторы формальных параметров не могут совпадать с именами переменных, объявляемых внутри тела функции, но возможно локальное переобъявление формальных параметров внутри вложенных блоков функции.
- Тип каждого формального параметра должен соответствовать типу фактического аргумента и типу соответствующего аргумента в прототипе функции, если таковой имеется.
- После преобразования все порядковые формальные параметры имеют тип int, а вещественные - тип double.

- После последнего идентификатора в списке формальных параметров может быть записана запятая с многоточием (,:). Это означает, что число параметров функции переменно, однако не меньше, чем следует идентификаторов до многоточия.
- Фактический аргумент может быть любым значением базового типа, структурой, объединением или указателем. По умолчанию фактические аргументы передаются по значению. Массивы и функции не могут передаваться как параметры, но могут передаваться указатели на эти объекты. Поэтому массивы и функции передаются по ссылке. Значения фактических аргументов копируются в соответствующие формальные параметры. Функция использует только эти копии, не изменяя сами переменные, с которых копия была сделана.

- Стандарт языка не предполагает размещения тела одной функции внутри другой, хотя прототип функции может быть указан внутри другой функции. Т.е., определение функции возможно только *вне* всех блоков.
- Программа на Си должна содержать хотя бы одно определение функции - функции с именем main**.** Функция main является единственной точкой входа в программу. На весь проект может быть только одна функция с именем main.

- Текст программы может быть разделен на несколько исходных файлов. При компиляции программы каждый из исходных файлов должен быть скомпилирован отдельно, а затем связан с другими файлами компоновщиком. Исходные файлы можно объединять в один файл, компилируемый как единое целое, посредством директивы препроцессора #include. В тексте примера ниже файл file3.cpp включает file2.cpp, используя из него функцию f4():

```
#include <stdio.h>
#include "file2.cpp"
void main () {
 printf ("\n%d",f4(0));
}
```

- Исходный файл не обязательно содержит выполняемые операторы. Обычно удобно размещать объявления переменных, типов и функций в файлах типа *.h, а в других файлах использовать эти объекты путем их определения. Подключение заголовочного файла *.h выполняется директивой **#include "ИмяФайла.h"** предполагающей поиск заголовочного файла в текущей папке проекта.
- Директива **#include <ИмяФайла.h>** предназначена для подключения стандартных заголовочных файлов и ищет их в папках, указанных в настройке Include directories компилятора.

- Функция main также может иметь формальные параметры. Значения формальных параметров main могут быть получены извне - из командной строки при вызове программы и из таблицы контекста операционной системы. Таблица контекста заполняется системными командами SET и PATH:
- int main (int argc, char *argv[], char *envp []) { : }
- Доступ к первому аргументу, переданному программе, можно осуществить с помощью выражения argv[1], к последнему аргументу - argv[argc-1]. Аргумент argv[0] содержит строку вызова самого приложения.
- Параметр envp представляет собой указатель на массив строк, определяющих системное окружение, т.е. среду выполнения программы. Стандартные библиотечные функции getenv и putenv (библиотека stdlib.h) позволяют организовать удобный доступ к таблице окружения.
- Существует еще один способ передачи аргументов функции main - при запуске программы как независимого подпроцесса из другой программы, также написанной на Си (функции семейства exec и spawn, бибилиотека process.h).

Пример ниже представляет собой законченную программу на Си, состоящую из трех функций.

```
#include <stdio.h>
int long power(int,int); //прототип функции
//необходим, т.к. тело функции ниже вызова
void printLong (char *s, long int l) {
//без прототипа
printf ("%s%ld",s,l);
}
void main() {
for (int i = 0; i <= 30; i++) {
 int long l1=power(2,i);
 printf("%d",i);
 printLong (" ",l1);
 printf("\n");
}
}
int long power(int x, int n) {
int i; long int p;
for (i=1,p=1L; i <= n; ++i) p *= x;
return (p);
}
```

Функции на Си могут быть рекурсивными:

```
int long power(int x, int n) {
return (n>1 ?
 (long int)x*power(x,n-1) : x);
}
```

Лекция **6. Передача параметров по значению и по ссылке.**

- По умолчанию аргументы функций передаются по значению. При передаче по ссылке перед аргументом в прототипе и в заголовке указывается операция "адрес" (&):
- void swap (int &,int &);
- //...
- void swap (int &a, int &b) {
- int c=a; a=b; b=c;
- }
- //...
- swap (x1,x2);

- Здесь функция swap поменяла местами фактические значения аргументов x1 и x2.
- Так как компилятор на первом этапе формирует внешние имена функций, в одном проекте возможны функции, имеющее одинаковые имена, но отличающиеся списком параметров:
- int f(int,int);
- float f(float,float);
- но не
- float f(int,int);

- **Время жизни и область действия объектов.**
- Итак, описания переменных в языке Си не предполагают их объединения в отдельные разделы описаний. В этой связи огромную важность приобретают правила, касающиеся времени жизни и области действия объектов. Эти правила описаны в табл. 5.1.

Объект	Время жизни	Область действия
функция	глобально для всех функций (все время выполнения программы)	ниже по тексту от объявления или определения
переменная, определенная на внешнем уровне (вне всех блоков)	глобально	от точки программы, в которой объявлена, до конца исходного файла, включая все функции и вложенные блоки. При этом может быть вытеснена одноименной переменной, объявленной внутри блока. Если указан класс памяти static - только до конца исходного файла, содержащего объявление
переменная, определенная внутри блока	локально, пока выполняется этот и вложенные блоки (если не указан класс памяти static)	в этом и вложенных блоках, может быть переопределена во вложенных блоках

- Пример ниже наглядно демонстрирует переопределение глобальных и локальных переменных одноименными переменными более низких уровней.

```
#include <stdio.h>
static int i=12;
 //глобальная статическая переменная
void f() {
 printf ("\n%d",i); //i=12
}
void main () {
 int i=5;
 printf ("\n%d\n",i); //i=5
 {//без этого блока компиляторы выведут
 //ошибку "множественная декларация
 //переменной", т.к. открывающая часть
 //цикла for выполняется до его тела
 for (int i=0; i<10; i++)
 printf ("%d ",i); //i=0,1,:,9
 }
 printf ("\n%d",i); //i=5
 f();
}
//************************************************************
```

```
#include <stdio.h>
void f(int,float);
int main(void) {
int a,b;
a=8;b=0;
//if(a==8)return 0;
switch (a) {
 case 1: b=1;break;
 case 2: b=2;break;
 case 3: b=3;break;
 default: b=5;
};
printf("b= %d",b);
f(66,66.66);

return 0;
}
void f(int a, float b){
  printf("\n*****    a=%d    b=%6.3f",a,b);
};
```

Лекция **7. Операции и выражения**

- *Операции* - это комбинации символов, определяющие действия по преобразованию значений.
- В Си определены 5 *арифметических* операций: сложение (знак операции "+"), вычитание ("-"), умножение ("*"), деление ("/") и взятие остатка от деления ("%"). Приоритеты и работа операций обычные: умножение, деление и взятие остатка от деления равноправны между собой и старше, чем сложение и вычитание. *Ассоциативность* (порядок выполнения) арифметических операций принята слева направо.
- Операция % определена только над целыми операндами, а результат операции деления зависит от типа операндов. Деление целых в Си дает всегда целое число, если же хотя бы один из операндов вещественный, результат также будет вещественным:
- 3/2 //результат=1
- 3./2 //результат=1.5

- В Си существует богатая коллекция разновидностей оператора присваивания. Обычное присваивание выполняется оператором "=":
- x=y+z;
- Возможно *объединение присваивания с другой операцией*, используемое как сокращенная форма записи для присваивания, изменяющего значение переменной:
- x+=3; //эквивалентно x=x+3;
- p*=s; //эквивалентно p=p*s;
- Присваивание также может быть *составным*, при этом цепочка вычислений выполняется справа налево:
- i=j=0;
- c=1; a=b=c+1; //a=b=2;

- Присваивание начального значения переменной (а также элементов массива) может быть выполнено непосредственно при описании:
- int k=5;
- Для распространенных операций *инкремента* (увеличения на 1) и *декремента* (уменьшения на 1) есть специальные обозначения ++ и -- соответственно:
- i++; //эквивалентно i+=1; или i=i+1;
- Операнд инкремента и декремента может иметь целый или вещественный тип или быть указателем.

- Операции инкремента и декремента могут записываться как перед своим операндом (*префиксная* форма записи), так и после него (*постфиксная* запись). Для операции в префиксной форме операнд сначала изменяется, а затем его новое значение участвует в дальнейшем вычислении выражения. Для операции в постфиксной форме операнд изменяется после того, как его старое значение участвует в вычислении выражения:
- int i=3;
- printf ("\n%d",i++); //напечатает значение i=3
- printf ("\n%d",++i); //напечатает значение i=4
- *Присваивание с приведением типа* (тип)(выражение) использует заключенное в круглые скобки название типа, к которому нужно привести результат:
- float f1=4,f2=3;
- int a = (int)(f1/f2); //a=1
- f2=(float)a+1.5; //f2=2.5
- При этом разрешены преобразования типов, приводящие к потере точности, ответственность за это целиком лежит на программисте.

- ***Логические операции*** в языке Си делятся на 2 класса. Во-первых, это *логические функции*, служащие для объединения условий, во-вторых, *поразрядные* логические операции, выполняемые *над отдельными битами* своих операндов. Операнды логических операций могут иметь целый, вещественный тип, либо быть указателями. Типы первого и второго операндов могут различаться. Сначала всегда вычисляется первый операнд; если его значения достаточно для определения результата операции, то второй операнд не вычисляется.

- Логические операции не выполняют каких-либо преобразований по умолчанию. Вместо этого они вычисляют свои операнды и сравнивают их с нулем. Результатом логической операции является либо 0, понимаемый как ложь, либо ненулевое значение (обычно 1), трактуемый как истина. Существенно то, что в языке Си нет специального логического типа данных и тип результата логической операции - целочисленный.
- Логическая функция "И" (соответствует and в Паскале) обозначается как &&, "ИЛИ" (or) как ||, унарная функция отрицания (not в Паскале) записывается как ! перед своим операндом:
- if (x<y && y<z) min=x;
- if (!(x>=a && x<=z))
- printf ("\nx не принадлежит [a,b]");

- Приоритеты логических функций традиционны: операция ! старше чем &&, которая, в свою очередь, старше ||. При необходимости приоритеты могут быть изменены с помощью круглых скобок.
- Как отмечено ранее, *поразрядные* логические операции выполняются над отдельными битами (разрядами) своих операндов. Имеется три бинарных и одна унарная поразрядная операция. Они описаны в табл. 3.1.

Операнд x	**0**	**0**	**1**	**1**	**Описание**
Операнд y	**0**	**1**	**0**	**1**	
x|y	0	1	1	1	Побитовое ИЛИ
x&y	0	0	0	1	Побитовое И
x^y	0	1	1	0	Побитовое исключающее ИЛИ
~x	1		0		Побитовое отрицание

- Примеры:
- char x=1,y=3; char z=x&y; //z=1
- char x=0x00; x=x^0x01; //либо x^=1;
- //организует "флажок", переключающийся между состояниями 0 и 1
-

Операция	Проверяемое отношение
<	Первый операнд меньше, чем второй
>	Первый операнд больше, чем второй
<=	Первый операнд меньше или равен второму
>=	Первый операнд больше или равен второму
==	Первый операнд равен второму
!=	Первый операнд не равен второму

Знаки операций	Наименование	Ассоциативность
() [] . ->	Первичные	Слева направо
+ - ~ ! * & ++ -- sizeof(тип) приведение типа	Унарные	Справа налево
* / %	Мультипликативные	Слева направо
+ -	Аддитивные	Слева направо
>> <<	Сдвиг	Слева направо
< > <= >=	Отношение	Слева направо
== !=	Отношение	Слева направо
&	Поразрядное И	Слева направо
^	Поразрядное исключающее ИЛИ	Слева направо
|	Поразрядное включающее ИЛИ	Слева направо
&&	Логическое И	Слева направо
||	Логическое ИЛИ	Слева направо
?:	Условная	Справа налево
= *= /= %= += -= <<= >>= &= |= ^=	Простое и составное присваивание	Справа налево
,	Последовательное вычисление	Слева направо

Лекция **8.** Операторы

- Операторы управляют процессом выполнения программы. Набор операторов Си содержит все типовые управляющие конструкции структурного программирования.
- Программа на Си выполняется последовательно, оператор за оператором, за исключением случаев, когда какой-либо оператор явно передает управление в другую часть программы, например при вызове функции или возврате из функции.

- В теле некоторых операторов могут содержаться другие операторы. Оператор, находящийся в теле другого оператора, в свою очередь может содержать операторы.
- Составной оператор ограничивается фигурными скобками { }. Все другие операторы заканчиваются точкой с запятой (;). Точка с запятой в языке Си является признаком конца оператора, а не разделителем операторов, как в ряде других языков.
- Перед любым оператором может быть записана метка, состоящая из имени и двоеточия. Метки распознаются только оператором goto.
- Далее приведен полный список операторов Си.

Пустой оператор

- Применяется там, где по правилам синтаксиса требуется указать оператор, например, для создания пустого тела цикла или пустой ветви условного оператора. Выполнение пустого оператора не меняет состояния программы.

Составной оператор

- ***Составной*** **оператор** или *блок* { }. Действие составного оператора заключается в последовательном выполнении содержащихся в нем операторов, за исключением тех случаев, когда какой-либо оператор явно передает управление в другое место программы. Типичное применение блока подобно другим языкам - ограничение тела функции, тела цикла, описания структурного типа данных или ветви условного оператора.
- В начале составного оператора могут содержаться объявления переменных, локальных для данного блока, либо объявления, служащие для распространения на блок области действия глобальных объектов. Возможны объявления переменных и в любом другом месте блока (см. п. 5.1).

- ***Условный* оператор** if записывается в общем виде следующим образом:
- if (УсловноеВыражение1) оператор1;
- else if (УсловноеВыражение2) оператор2;
- :
- else if (УсловноеВыражениеN) операторN;
- else оператор0;

- Как и в других языках, выполняется только один из операторов 1,2,:,N, в зависимости от того, какое из соответствующих условных выражений первым оценено как истинное. Если все условия ложны, выполняется оператор0. Любая из ветвей, кроме первой, необязательна. Круглые скобки и точки с запятой обязательны везде, где указаны. Еще раз акцентируем внимание на том, что в Си отсутствуют булевские выражения и в качестве условий применяются обычные выражения языка Си. Значение выражения считается истинным, если оно не равно нулю, и ложным, если равно нулю. Из этого следует, что условные выражения не обязательно должны содержать операции отношения. Условия могут быть записаны в обычном виде

- if (a < 0) :
- а могут выглядеть, например, так:
- if (a) : или if (a + b) : .
- Условия могут включать логические функции:
- if (x>0 && y>0) ch=1;

Лекция **9. Операторы цикла.**

- ***Оператор пошагового цикла*** **for** имеет следующий общий вид:
- for (НВ; УВ; ВП) Оператор;
- Здесь НВ - начальное выражение, служащее для инициализации параметров цикла, УВ - условное выражение, прямо или косвенно определяющее число повторений цикла (цикл выполняется, пока УВ не станет ложным), ВП - выражение приращения, используемое для модификации параметра или параметров цикла. Любое из трех выражений может быть опущено, а также любые 2 выражения или все сразу.

- Цикл работает следующим образом: сначала вычисляется НВ, если оно имеется. Затем вычисляется УВ и производится его оценка следующим образом:

- если УВ истинно (не равно нулю), то выполняется тело оператора. Затем вычисляется ВП, если оно есть, и процесс повторяется;
- если УВ опущено, то его значение принимается за истину и процесс выполнения продолжается, как описано выше. В этом случае цикл for бесконечен и может завершиться только при выполнении в его теле операторов break, goto, return;
- если УВ ложно, то выполнение цикла заканчивается и управление передается следующему за ним оператору программы. Цикл for может завершиться также при выполнении операторов break, goto, return в его теле.
- Пример: показанный ниже цикл выполняется 10 раз.
- for (x=1; x<11; x++) printf ("*");

- **Оператор *цикла с предусловием* while**:
- while (выражение) оператор;
- Тело цикла while выполняется до тех пор, пока значение выражения не станет ложным (равным нулю). Сначала вычисляется выражение, если оно изначально ложно, то тело цикла не выполняется и управление передается на следующий за ним оператор программы. Если выражение истинно, то выполняется тело цикла. Перед каждым следующим выполнением цикла выражение вычисляется заново. Процесс повторяется до тех пор, пока выражение не станет ложным. Оператор while может завершиться досрочно при выполнении break, goto или return внутри своего тела.
- Приведенный далее цикл while аналогичен показанному выше for:
- x=1;
- while (x<11) {
- printf ("*");
- x++;
- }

- **Оператор *цикла с постусловием* do** записывается в виде
- do оператор while (выражение);
- Тело цикла do выполняется один или несколько раз до тех пор, пока значение выражения не станет ложным (равным нулю). Сначала выполняется тело цикла - оператор, затем вычисляется условие - выражение. Если выражение ложно, цикл завершается и управление передается следующему за телом цикла оператору программы. Если выражение истинно (не равно нулю), то тело цикла выполняется вновь, и выражение вычисляется повторно. Выполнение цикла повторяется до тех пор, пока выражение не станет ложным. Цикл do может завершиться досрочно при выполнении в своем теле операторов break, goto, return. Показанный далее цикл по действию совпадает с примерами на for и while:
- x=1;
- do {
- printf ("*"); x++;
- } while (x<11);

- ***Оператор-переключатель*** **switch** записывается в виде
- switch (выражение) {
- объявление
- case КВ1: оператор1;
- :
- case КВN: операторN;
- default: оператор0;
- }

- ***Оператор продолжения*** **continue** передает управление на следующую итерацию в циклах do, for, while. Он может появиться только в теле этих операторов. Остающиеся в теле цикла операторы при этом не выполняются. В циклах do и while следующая итерация начинается с вычисления условного выражения. В цикле for следующая итерация начинается с вычисления выражения приращения, а затем происходит вычисление условного выражения.
-
- ***Оператор разрыва*** **break** прерывает выполнение операторов do, for, while или switch. Он может содержаться только в теле этих операторов. Управление передается оператору программы, следующему за прерванным.

- ***Оператор перехода* goto** имеет вид
- goto метка;

- и передает управление непосредственно на оператор, помеченный меткой:
- метка: оператор;
- Метка представляет собой обычный идентификатор. Область действия метки ограничивается функцией, в которой она определена. Каждая метка должна быть уникальна в пределах функции, где она указана. Нельзя передать управление по оператору goto в другую функцию. Метка оператора имеет смысл только для goto. Можно войти в блок, тело цикла, условный оператор, оператор-переключатель по метке. Нельзя с помощью goto передать управление на конструкции case и default в теле переключателя.

- *Оператор возврата* return выражение; заканчивает выполнение функции, в которой он содержится, и возвращает управление в вызывающую функцию.
- Управление передается в точку вызывающей функции, непосредственно следующую за оператором вызова функции. Значение выражения, если оно задано, вычисляется, приводится к типу, объявленному для функции, содержащей оператор, и возвращается в вызывающую функцию. Если выражение опущено, то возвращаемое функцией значение не определено (является пустым).
-

Лекция **10.** Статические массивы

- Массив объявляется одним из следующих способов:
-
- тип идентификатор [количество элементов];
- тип идентификатор [] = {*список элементов*};
-
- Во втором случае, элементы списка перечисляются через запятую, при этом размерность массива определяется по фактически указанному количеству элементов.

- **Примеры:**
-
- int x[10];
-
- float a[]={3.5,4.5,5.5}; //размерность=3
-
- char div[3]={' ','\n','\t'};

- Элементы массивов в Си всегда нумеруются с нуля. Синтаксис выражения для обращения к элементу массива имеет следующий вид:
- имя массива[номер элемента]
- **Примеры:**
- a[0] = 6;
- или
- int i=6;
- ...
- a[i] = 8;
- a[i+1] = 7;

- Элементы многомерного массива запоминаются построчно.
- **Примеры:**

```
char a[2][3];
a[0][1] = ‘g’;

float matrix[10][15];

int b[3][3]={
{1,2,3},
{4,5,6},
{7,8,9}};
```

В следующем примере выполняется определение, обработка и печать статической матрицы.

```
#include <stdio.h>

void main () {
int b[3][3]={
{1,2,3},
{1,2,3},
{1,2,3}
};

b[0][0]=2;
b[2][2]=b[0][0]*4;

for (int i=0; i<3; i++) {
printf ("\n");
for (int j=0; j<3; j++)
printf ("%d  ",b[i][j]);
}
}
```

Лекция 11. Структуры.

- Структурой в языке С называется совокупность логически связанных переменных различных типов, сгруппированных под одним именем для удобства дальнейшей обработки.
- Структура – это способ связать воедино данные разных типов и создать пользовательский тип данных. В языке Pascal подобная конструкция носит название записи.

- **Определение структуры**
- Структура – тип данных, задаваемый пользователем. В общем случае при работе со структурами следует выделить четыре момента:
- - объявление и определение типа структуры,
- - объявление структурной переменной,
- - инициализация структурной переменной,
- - использование структурной переменной.

- Определение типа структура представляется в виде
-
- struct ID
- {
- <тип> <имя 1-го элемента>;
- <тип> <имя 2-го элемента>;
- ……
- <тип> <имя последнего элемента>;
- };

- Определение типа структуры начинается с ключевого слова **struct** и содержит список объявлений, заключенных в фигурные скобки. За словом struct следует **имя типа**, называемое тегом структуры. Элементы списка объявлений называются **полями**. Каждое поле имеет уникальное для данного структурного типа имя. ***Однако следует заметить, что одни и те же имена полей могут быть использованы в различных структурных типах.***
- Определение типа структуры представляет собой шаблон, предназначенный для создания структурных переменных.
- Объявление переменной структурного типа имеет следующий вид:
-

- struct ID var1;
-
- при этом в программе создается переменная с именем var1 типа ID. При объявлении переменной происходит выделение памяти для размещения переменной. Тип структуры позволяет определить размер выделяемой памяти.
- В общем случае, под структурную переменную выделяется область памяти не менее суммы длин всех полей структуры, например,

- struct list
- {
- char name[20];
- char first_name[40];
- int year;
- } L;
- В данном примере объявляется тип структура с именем list, состоящая из трех полей, и переменная с именем L типа list,при этом для переменной L выделяется 64 байта памяти.

- **Структуры и функции**
-
- Структуры могут быть переданы в функцию в качестве аргументов и могут служить в качестве возвращаемого функцией результата.
- Существует три способа передачи структур функциям:
- передача компонентов структуры по частям;
- передача целиком структуры;
- передача указателя на структуру.

- Например, в функцию передаются координаты двух точек:

```
void showrect(struct point p1, struct point p2)
{
printf("Левый верхний угол прямоугольника: %d %d\n", p1.x, p1.y);
printf("Правый нижний угол прямоугольника: %d %d\n", p2.x, p2.y);
}
```

- При вызове такой функции ей надо передать две структуры:

```
struct point pt1 = {5,5}, pt2={50,50};
showrect(pt1, pt2);
```

- Теперь рассмотрим функцию, возвращающую структуру:
-
- struct point makepoint (int x, int y) /*makepoint – формирует точку по компонентам x и y*/
- {
- struct point temp;
- temp.x = x;
- temp.y = y;
- return temp;
- }

Лекция **12.** Массивы структур

- При необходимости хранения некоторых данных в виде нескольких массивов одной размерности, но разного типа, возможна организация массива структур.
- Например, имеется два разнотипных массива:
- char c[N];
- int i[N];

- Объявление
- struct key
- {
- char c;
- int i;
- };
- struct key keytab[5];

- Инициализация массива структур выполняется следующим образом:

```
#include <stdio.h>
struct key
{
char c;
int i;
};

int main() {
  key keytab[3]={{'a',0},{'b',1},{'c',2}};
  for(int i=0;i<3;i++)printf("%c, %d\n", keytab[i].c, keytab[i].i);

  keytab[0].c='*';
  for(int i=0;i<3;i++)printf("%c, %d\n", keytab[i].c, keytab[i].i);

  return 0;
}
```

- **Вложенные структуры**
-
- Полем структурной переменной может быть переменная любого типа, в том числе другая структурная переменная. Поле, представляющее собой структуру, называется вложенной структурой.
- Тип вложенной структуры должен быть объявлен до того как будет использован. Кроме того, структура не может быть вложена в структуру того же типа.
-

- Пример объявления вложенной структуры:
- struct point
- {
- int x,y;
- };
-
- struct rect
- {
- struct point LUPoint, RDPoint;
- char BorderColor[20];
- };
-
- struct rect Rect;

- Пример создания и использования вложенной структуры:
-
- struct rect Rect={10,5,50,25,"White"};
- printf("Параметры прямоугольника:\n");
- printf("Координаты левого верхнего угла %d %d\n", Rect.LUPoint.x, Rect.LUPoint.y);
- printf("Координаты левого верхнего угла %d %d\n", Rect.RDPoint.x, Rect.RDPoint.y);
- printf("Цвет границы: %s\n", Rect.BorderColor);

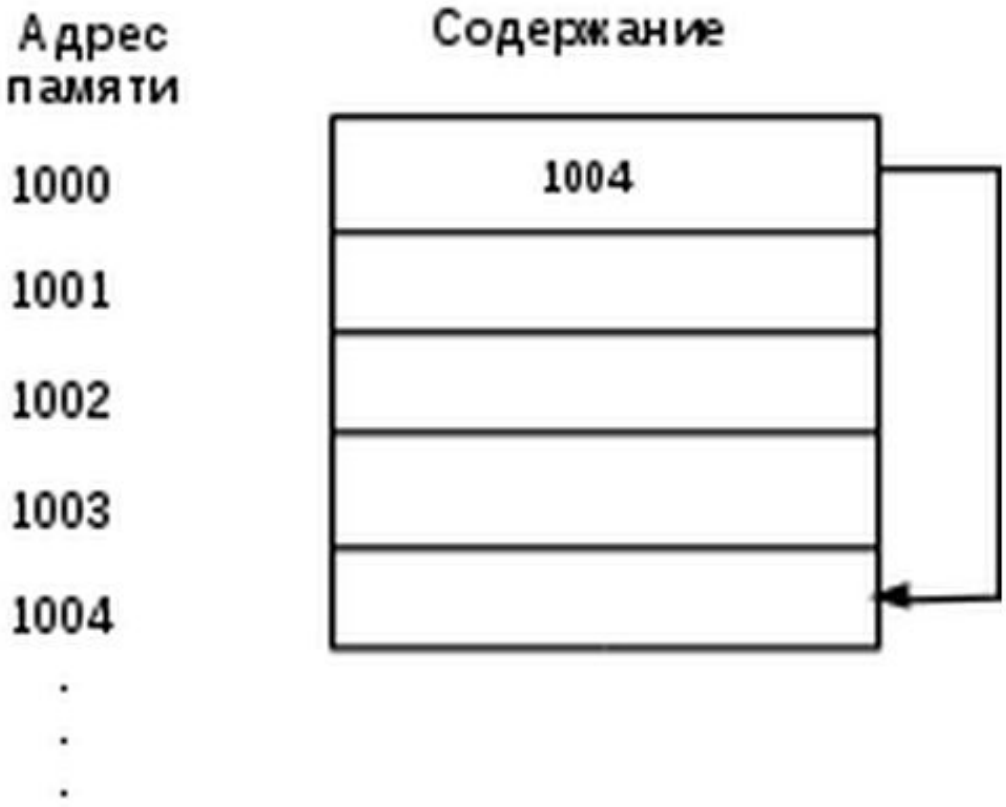
```
#include <stdio.h>
struct point
{
int x,y;
};
struct rect
{
struct point LUPoint, RDPoint;
char BorderColor[20];
};
int main() {
  //struct rect Rect;
  struct rect Rect={10,5,50,25,"White"};
  printf("Параметры прямоугольника:\n");
  printf("Координаты левого верхнего угла %d %d\n", Rect.LUPoint.x,
Rect.LUPoint.y);
  printf("Координаты левого верхнего угла %d %d\n", Rect.RDPoint.x,
Rect.RDPoint.y);
  printf("Цвет границы: %s\n", Rect.BorderColor);

  return 0;
}
```

Лекция **13. Указатели**

- Правильное понимание и использование указателей имеет большое значение при создании большинства C и C++-программ по четырем причинам:
- Указатели предоставляют способ, позволяющий функциям модифицировать передаваемые аргументы.
- Указатели используются для поддержки системы динамического выделения памяти.
- Использование указателей может повысить эффективность работы некоторых подпрограмм.
- Указатели, как правило, используются для поддержки некоторых структур данных типа связанные списки и двоичные деревья.

- Помимо того, что указатели - одна из самых сильных сторон C, они, в то же время, могут нанести большой ущерб. Например, неинициализированный или дикий указатель может привести к краху системы, может быть даже хуже, когда некорректное использование указателей приводит к трудноуловимым ошибкам.

- **Указатели - это адреса**
- Указатель содержит адрес памяти. Как правило, данный адрес содержит местоположение какой-либо переменной в памяти. Если одна переменная содержит адрес другой, то говорят, что первая переменная указывает на вторую. Например, если переменная по адресу 1000 указывает на переменную по адресу 1004, то по адресу 1000 будет находиться значение 1004. Данная ситуация продемонстрирована на рисунке.


Адрес памяти
Содержание
1000
1001
1002
1003
1004
1004

- **Переменные-указатели**
- Если переменная должна содержать указатель, она объявляется следующим образом. Объявление указателя включает базовый тип, * и имя переменной. Стандартный вид объявления указателя следующий:
-
- **<тип> *<имя переменной-указателя>;**
-
- где тип - это любой допустимый тип (базовый тип указателя).
-
- Например:
- int *a;
- char *str;
- double *pDoub;

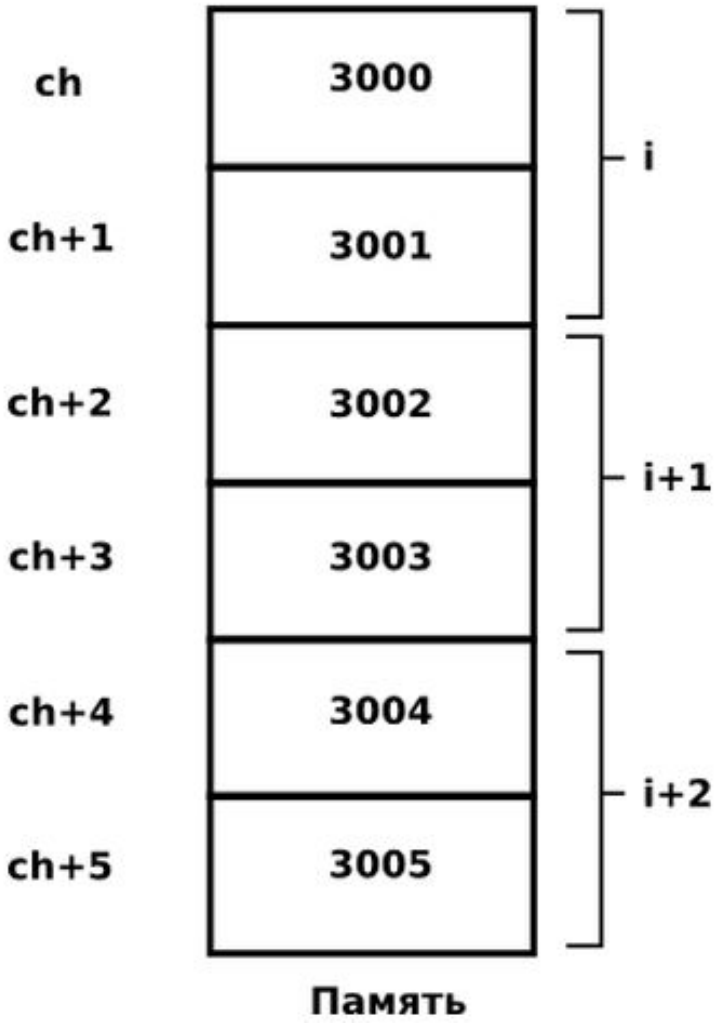
- **Операторы для работы с указателями**
- Имеется два специальных оператора для работы с указателями - * и **&**.
- Оператор **&** - это унарный оператор, возвращающий адрес операнда. Например:
-
- **p = #**

Лекция **14.** Арифметические действия с указателями.

- К указателям могут применяться только две арифметические операции: сложение и вычитание. Для понимания арифметических действий с указателями предположим, что p1 - это указатель на целое, равный 2000, и будем считать, что целые имеют длину 2 байта. После выражения
- **p1 ++;**
- содержимое p1 станет 2002, а не 2001! Каждый раз при увеличении p1 указатель будет указывать на следующее целое.

- Это справедливо и для уменьшения. Например:
-
- **p1 --;**
-
- приведет к тому, что p1 получит значение 1998, если считать, что раньше было 2000.
- Каждый раз, когда указатель увеличивается, он указывает на следующий элемент базового типа. Каждый раз, когда уменьшается - на предыдущий элемент. В случае указателей на символы это приводит к «обычной» арифметике. Все остальные указатели увеличиваются или уменьшаются на длину базового типа.

```
char *ch=3000;
int *i=3000;
```



- В большинстве случаев вычитание одного указателя из другого имеет смысл только тогда, когда оба указателя указывают на один объект, - как правило, массив. ***В результате вычитания получается число элементов базового типа, находящихся между указателями***. Помимо этих операций не существует других арифметических операций, применимых к указателям. Нельзя умножать или делить указатели, нельзя складывать указатели, нельзя применять битовый сдвиг или маски к указателям, нельзя добавлять или вычитать типы float или double.
-
- **Сравнение указателей**
- Возможно сравнивать два указателя. Например, имеются указатели p и q. Тогда справедлив следующий оператор:
-
- **if(p<q) printf("p points to lower memory than q\n");**

- **Динамическое выделение и указатели**
- У всех откомпилированных программ С при размещении в памяти можно выделить четыре области: код программы, глобальные данные, стек и куча. ***Куча*** **-** это область свободной памяти, управляемая функциями динамического выделения **malloc()** и **free()**.
- Функция **malloc()** выделяет память и возвращает указатель на начало выделенного блока. Функция **free()** возвращает ранее выделенную память в кучу для повторного использования. Ниже показаны прототипы функций **malloc()** и **free()**:
-
- **void *malloc(size_t число_байт);**
- **void free(void *p);**

- Обе функции используют заголовочный файл **stdlib.h**. Здесь число_число - это число требуемых байт. Если недостаточно свободной памяти для выполнения запроса, **malloc()** возвращает NULL. Тип **size_t** определен в **stdlib.h** и является беззнаковым целым типом, способным хранить размер памяти, выделяемой при одном вызове **malloc()**. Важно, что **free()** должна вызываться только с корректным, ранее выделенным указателем, иначе структура кучи может быть повреждена, что может привести к краху программы.
- Фрагмент кода, показанный ниже, выделяет 25 байт памяти:
-
- **char *p;**
- **p = (char *) malloc(25);**

Лекция **15. Указатели на константы**

- Модификатор **const** может использоваться для защиты объекта от возможного изменения со стороны функции. То есть, когда указатель передается в функцию, функция может модифицировать оригинал переменной, на которую указывает указатель. Если указатель определен как const, то функция не сможет модифицировать объект. Например, функция sp_to_dash() в следующей программе выводит дефис вместо каждого пробела в передаваемой строке. То есть строка «this is a test» будет напечатана как «this-is-a-test». Использование const при объявлении параметра гарантирует, что код функции не может модифицировать объект, на который указывает параметр.

```
#include <stdio.h>
void sp_to_dash(const char *str);
int main(void){
sp_to_dash ("this is a test");
return 0;
}

void sp_to_dash (const char *str)
{
while(*str) {
if(*str == ' ') printf ("%c", '-');
else printf("%c", *str);
str++;
}
}
```

- **Указатели и массивы**
- Массивы и указатели тесно связаны между собой. Рассмотрим следующий фрагмент:
-
- **char str [80], *p1;**
- **p1 = str;**

- **Указатели на символьные массивы**
- Многие строковые операции в С выполняются с помощью указателей, поскольку доступ к строкам осуществляется последовательно.

- Например, здесь показана версия стандартной библиотечной функции strcmp(), использующей указатели:
-
- /* Использование указателей. */
- int strcmp (const char *s1, const char *s2){
- while(*s1) if(*s1-*s2)
- return *s1-*s2;
- else {
- s1++;
- s2++;
- }
- return 0; /* равенство */

- Надо помнить, что все строки в С оканчиваются нулевым символом, который интерпретируется как ложь. Следовательно, оператор
-
- **while (*s1)**
-
- выдает истину, пока не достигнет конца строки. strcmp() возвращает 0, если s1 равно s2. Она возвращает число меньше 0, если s1 меньше s2. Иначе возвращает число больше нуля.

Лекция 16. Массивы указателей.

- Можно создавать массивы указателей. Для объявления массива целочисленных указателей из десяти элементов следует написать:
- **int *x[10];**
- Для присвоения адреса целочисленной переменной var третьему элементу массива следует написать:
- **x[2] = &var;**
- Для получения значения var следует написать:
- ***x [2]**

- Если необходимо передать массив указателей в функцию, можно использовать метод, аналогичный передаче обычных массивов. Просто надо вызвать функцию с именем массива без индексов. Например, функция, получающая массив x, должна выглядеть следующим образом:
-
- void display_array(int *q[]){
- int t;
- for(t=0; t<10; t++)
- printf ("%d ", *q[t]);
- }

- Надо помнить, что q - это не указатель на целое, а массив указателей на целые. Следовательно, необходимо объявить параметр q как массив целых указателей. Он не может объявиться как простой целочисленный указатель, поскольку он не является им.
- Типичным использованием массивов указателей является хранение сообщений об ошибках. Можно создать функцию, выводящую сообщение по полученному номеру, как показано ниже:

```
void serror(int num){
static char *err[] = {
"Cannot Open File\n",
"Read Error\n",
"Write Error\n",
"Media Failure\n"
};
printf ("%s", err[num]);
}
```

- Как можно видеть, printf() в serror() вызывается с указателем на символ, указывающим на одно из сообщений, номер которого передается в функцию. Например, если num приняла значение 2, будет выведено сообщение «Write Error».
- Интересно заметить, что аргумент командной строки argv является массивом указателей на символы.
-

- **Указатели на указатели - многочисленное перенаправление**
- Концепция массивов указателей открыта и проста, поскольку индексы имеют вполне определенное значение. Тем не менее, в случае, когда один указатель указывает на другой, могут возникнуть проблемы. Указатель на указатель является формой многочисленного перенаправления или цепочки указателей.

Одиночное перенаправление

Многочисленное перенаправление

- Переменная, являющаяся указателем на указателе должна быть описана следующим образом. Это выполняется путем помещения двух звездочек перед именем. Например, следующее объявление сообщается компилятору, что newbalance - это указатель на указатель типа float:
-
- **float **newbalance;**
- Важно понимать, что newbalance - это не указатель на число с плавающей точкой, а указатель на указатель на вещественное число.

- Для получения доступа к целевому значению, косвенно указываемому указателем на указатель, следует применить оператор * два раза, как показано в следующем примере:

```
#include <stdio.h>
int main(void){
int x, *p, **q;
x = 10;
p = &x;
q = &p;
printf ("%d", **q) ; /* вывод значения x */
return 0;
}
```

- Здесь p объявляется как указатель на целое, а q - это указатель на указатель на целое. Вызов printf() выводит число 10 на экран.

- **Инициализация указателей**
- После объявления указателя и до первого присвоения ему значения, указатель может содержать неизвестное значение. Если попытаться использовать указатель до сообщения ему значения, можно нарушить работу не только программы, но и всей операционной системы.
- По существующим соглашениям неиспользуемый указатель должен содержать нулевое значение. То, что указатель имеет нулевое значение, не обеспечивает безопасности. Если использовать нулевой указатель слева от оператора присваивания, это может привести к «зависанию» программы.

```
/* Ищем имя */
int search(char *p[], char *name){
register int t;
for(t=0; p[t]; ++t)
if(!strcmp(p[t], name)) return t;
return -1; /* не найдено */
}
```

Цикл for в search() запускается каждый раз при обнаружении нулевого указателя. Поскольку конец массива помечен нулем, условие, управляющее циклом, останавливает цикл (выдает ложное состояние) при достижении конца массива.