Министерство образования Республики Беларусь

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра «Вычислительные методы и программирование»

В.Л.Бусько, А.Г.Корбит, Т.М.Кривоносова

Конспект лекций по курсу

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

для студентов всех специальностей и всех форм обучения

Минск 2004

УДК 621.3.6 (075.8)
ББК 22.193 Я73
С 35

Рецензент: канд. техн. наук, доцент, зав. кафедры ПОИТ БГУИР В.В.Бахтизин

Бусько В.Л., Корбит А.Г., Кривоносова Т.М.
Конспект лекций по курсу «Основы алгоритмизации и программирования» для студентов всех специальностей и всех форм обучения. - Мн.: БГУИР, 2004. - 89 с.: ил. 4.

ISBN

Общий курс программирования предполагает знакомство с устройством ПЭВМ, теорией кодирования и хранения данных в памяти; изучение и закрепление навыков разработки вычислительных алгоритмов решения задач; изучение языка программирования Си; освоение операционных систем, изучение и освоение различных технологий программирования при решении конкретных задач.

Усвоение этих разделов осуществляется при подготовке студентов информационного профиля. Для студентов радиотехнических специальностей в первую очередь необходимо получить навыки алгоритмизации и программирования, освоить способы решения, в первую очередь, вычислительных задач.

Настоящее пособие по курсу «Основы алгоритмизации и программирования» представляет собой конспект лекций по темам, охватывающим основные конструкции языка Си. Неотъемлемой частью данного пособия является лабораторный практикум [1], при выполнении которого студенты осваивают основные конструкции среды программирования и получают навыки алгоритмизации и решения задач на персональном компьютере.

УДК 621.3.6 (075.8)
ББК 22.193 Я73

ISBN

СОДЕРЖАНИЕ

Стр.

1. Введение

1.1. История создания ЭВМ
1.2. Структура ПЭВМ
1.3. Классификация языков программирования
1.4. Размещение данных и программ в памяти ПЭВМ
1.5. Программные модули
1.6. Ошибки
1.7. Функциональная и модульная декомпозиции
1.8. Файловая система хранения информации
1.9. Операционная система

2. Основные понятия и определения

2.1. Этапы решения задач на ЭВМ
2.2. Понятие алгоритма и способы его записи
2.3. Свойства алгоритмов
2.4. Способы описания алгоритмов
2.5. Графическое описание алгоритма
2.6. Основные символы схемы алгоритма
2.7. Пример простейшего линейного алгоритма
2.8. Немного истории

3. Синтаксис языка Си

3.1. Алфавит языка
3.2. Лексемы
3.3. Идентификаторы и ключевые слова
3.4. Знаки операций
3.5. Литералы (константы)
3.6. Комментарии

4. Базовые типы объектов

4.1. Простейшая программа
4.2. Основные типы данных
4.3. Декларация (объявление) объектов
4.4. Данные целого типа (int)
4.5. Данные символьного типа (char)
4.6. Данные вещественного типа (float, double)

5. Константы в программах

5.1. Целочисленные константы
5.2. Константы вещественного типа
5.3. Символьные константы
5.4. Строковые константы

Стр.

6. Обзор операций

6.1. Операции, выражения
6.2. Арифметические операции
6.3. Операции присваивания
6.4. Сокращенная запись операции присваивания
6.5. Преобразование типов операндов арифметических операций
6.6. Операция приведения типа
6.7. Операции сравнения
6.8. Логические операции
6.9. Побитовые логические операции. Операции над битами
6.10. Операция , (запятая)

7. Обзор базовых инструкций языка С

7.1. Стандартная библиотека языка Си
7.2. Стандартные математические функции
7.3. Функции вывода данных на дисплей
7.4. Функции ввода информации
7.5. Ввод - вывод потоками
7.6. Дополнительные функции

8. Синтаксис операторов языка С

8.1. Условные операторы
8.2. Условная операция «? :»
8.3. Оператор выбора альтернатив (переключатель)

9. Составление циклических алгоритмов

9.1. Понятие цикла
9.2. Оператор с предусловием while
9.3. Оператор цикла с постусловием do – while
9.4. Оператор цикла с предусловием и коррекцией for

10. Операторы передачи управления

10.1. Оператор безусловного перехода goto
10.2. Оператор continue
10.3. Оператор break
10.4. Оператор return

11 . Указатели

11.1. Указатели
11.2. Операции над указателями (косвенная адресация)

12. Массивы

12.1. Понятие массива
12.2. Одномерные массивы
12.3. Многомерные массивы
12.4. Операция sizeof
12.5. Применение указателей
12.6. Указатели на указатели
12.7. Адресная функция

Стр.

13. Работа с динамической памятью

13.1. Пример создания одномерного динамического массива:
13.2. Пример создание двухмерного динамического массива:

4. Строки в языке Си

14.1. Русификация под Visual

15. Функции пользователя

15.1. Декларация функции
15.2. Вызов функции
15.3. Операция typedef
15.4. Указатели на функции
15.5. Параметры командной строки функции main
15.6. Функции с переменным числом параметров

16. Классы памяти и области действия объектов

16.1. Классы памяти объектов в языке Си
16.2. Автоматические переменные
16.3. Внешние переменные
16.4. Область действия переменных

17. Структуры, объединения, перечисления

17.1. Структуры
17.2. Декларация структурного типа данных
17.3. Создание структурных переменных
17.4. Вложенные структуры
17.5. Массивы структур
17.6. Размещение структурных переменных в памяти
17.7. Объединения
17.8. Перечисления

18. Файлы в языке С

18.1. Открытие файла
18.2. Закрытие файла
18.3. Запись - чтение информации
18.4. Текстовые файлы
18.5. Бинарные файлы

Список рекомендуемой литературы

Список используемой литературы

Приложение 1. **Таблицы символов ASCII**

Приложение 2. **Операции языка Си**

Приложение 3. **Возможности препроцессора**

1. Введение

1.1. История создания ЭВМ

Проблема вычислений сопровождает человечество на всем историческом отрезке его существования. Первый счетный инструмент ***абак*** был известен еще в V веке до нашей эры в Египте, Финикии, Греции и представлял дощечку, покрытую слоем песка, на которой острой палочкой проводили линии и в получавшихся колонках по позиционному принципу размещали камешки. В древнем Риме абак назывался Calculi. От этого слова произошло в дальнейшем латинское calculatore (вычислять).

Первую счетную машину для выполнения сложения и вычитания сконструировал в 1623г. профессор математики и астрономии Тюбингенского университета В.Шинкард. Она была изготовлена в одном экземпляре и, сгорев во время пожара в 1624г., не оказала влияния на развитие идей счетной техники.

Биография механических счетных машин ведется от арифметической машины французского математика, физика и философа Б.Паскаля, созданной в 1642г. Над счетной машиной Б.Паскаль работал 12 лет и сделал около 50 действующих моделей. Первый арифмометр, выполняющий все четыре арифметических действия, был предложен в 1670г. немецким ученым Г.В.Лейбницем. В Беларуси первая суммирующая машина была изобретена и изготовлена в 1770г. в г. Несвиже Евной Якобсоном, часовым мастером и механиком.

Идею ***универсальной вычислительной машины с программным управлением*** впервые предложил в своем неосуществленном проекте в 1834 г. английский ученый Ч.Бэббедж. Ее структура совпадала по существу со структурой современных ЭВМ.

Отличительной особенностью ***электронных вычислительных машин*** (ЭВМ) от счетных машин является наличие устройства управления вычислениями и принцип хранения программы. Еще одной особенностью современных ЭВМ является применение двоичной системы счисления.

Двоичную арифметику разработал Г.В.Лейбниц. Он также предложил арифметизацию логики за 200 лет до создания алгебры Дж.Буля (1815). Как двоичная арифметика представляет все числа с помощью двух символов (0,1), так и булева алгебра оперирует с двумя понятиями (истина, ложь) и тремя операциями (и, или, не).

С помощью этих понятий можно смоделировать любые логические цепочки и построить 16 логических функций. На этой основе строятся все современные логические схемы различной сложности, реализуемые в ЭВМ.

Первая ЭВМ была создана в 1945г. (США), она представляла огромное сооружение, содержащее 18000 электронных ламп, 1500 реле и выполняла около 3000 умножений в секунду. Мировой парк ЭВМ к 1965г. насчитывал порядка 50 тысяч компьютеров, к началу 1975г. – более 200 тысяч.

Первые ***персональные ЭВМ*** (ПЭВМ) появились в начале 70-х годов. Скорость вычислений достигает 10^8 операций в секунду.

1.2. Структура ПЭВМ

ПЭВМ содержат клавиатуру, системный блок, и дисплей. Схема ПЭВМ представлена на рис. 1.

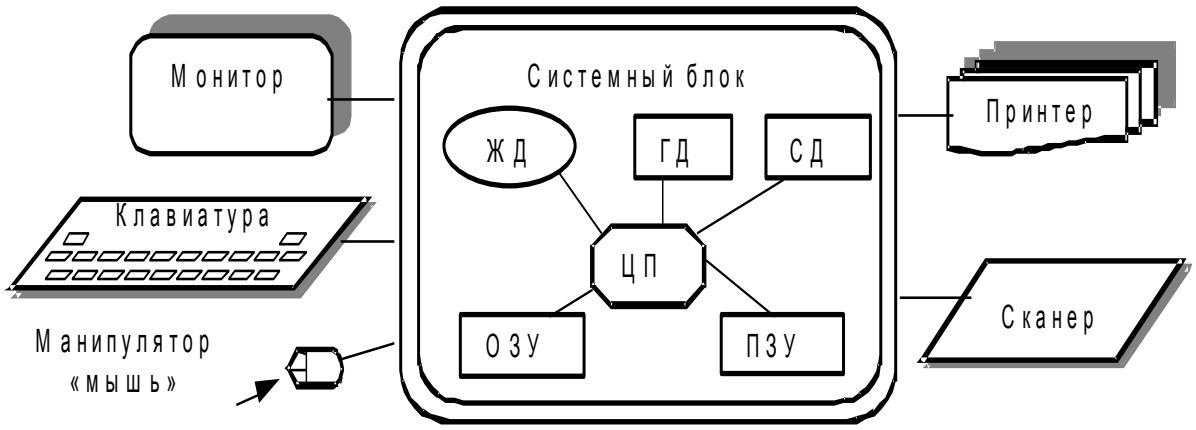


Рис.1.

В системном блоке ПЭВМ содержатся:

- ***центральный процессор*** (ЦП), который осуществляет управление работой и выполнение расчетов по программе;
- ***оперативное запоминающее устройство*** (ОЗУ), в котором во время работы компьютера располагаются выполняемые программы (при выключении компьютера - очищается);
- ***постоянное запоминающее устройство*** (ПЗУ), содержащее программы, необходимые для запуска компьютера;
- ***жесткий магнитный диск*** (ЖД), получивший название винчестер;
- ***дисковод*** (ГД) для сменных, гибких магнитных дисков (дискет);
- ***CD-Rom*** (СД) – устройство чтения компакт-дисков.

В системный блок встроены электронные схемы, управляющие работой различных устройств, входящих в состав компьютера. К системному блоку подключаются дисплей (монитор) для отображения информации, клавиатура для ввода данных и команд, устройство для визуального управления - «мышь», печатающее устройство - принтер, устройство для считывания и ввода графической информации - сканер.

1.3. Классификация языков программирования

1. По степени абстракции от архитектуры компьютера:

- языки программирования низкого уровня - ***машинный язык*** (язык машинных кодов). Используя такой язык, программист должен полностью владеть архитектурой ЭВМ;
- языки программирования среднего уровня - ***язык мнемонических кодов*** (язык ассемблера – символьная форма машинного языка с возможностями языков высокого уровня). Используя такие языки, необходимо владеть архитектурой ЭВМ, а также иметь специальную программу-переводчика инструкций пользовательской программы на язык машинных кодов, называемую транслятор;
- языки высокого уровня, или ***алгоритмические языки*** (Паскаль, Алгол, Фортран, Си). Алгоритмические языки имеют свой алфавит и синтаксис, а также трансляторы со встроенной средой программирования, которая обладает развитыми средствами подготовки, редактирования, отладки и выполнения программ. Инструкции программы на алгоритмическом языке записываются в виде операторов.

2. По структурному строению программ:

- ***процедурно-ориентированные языки*** (Pascal, C и др.) – используется метод разбивки всей задачи на более простые подзадачи (процедуры или функции) и их независимая реализация. Достоинство - создание достаточно сложных программ.

- ***объектно-ориентированные языки*** (C++, Java и др.) – из области решаемой проблемы выделяются классы, объединяющие связанные каким-либо признаком данные и функции по их обработке. Далее создаются объекты данных классов, которые, взаимодействуя друг с другом, осуществляют решение поставленной задачи.

1.4. Размещение данных и программ в памяти ПЭВМ

Данные и программы во время работы ПЭВМ размещаются в оперативной памяти, которая представляет собой последовательность пронумерованных ячеек. По указанному номеру процессор находит нужную ячейку, поэтому номер ячейки называется ее адресом. Минимальная адресованная ячейка (согласно стандарту IBM), с точки зрения программиста, состоит из 8 двоичных позиций, т.е. в каждую позицию могут быть записаны либо 0, либо 1. Объем информации, который помещается в одну двоичную позицию, называется ***битом***. Объем информации, равный 8 битам, называется ***байтом***.

Таким образом, в одной ячейке из 8 двоичных разрядов помещается объем информации в один байт. Поэтому объем памяти принято оценивать количеством байт (2^{10} байт = 1024 байт = 1 Кб, 2^{10} Кб = 1048576 байт = 1 Мб).

Для помещения данных в такие ячейки производится их запись с помощью нулей и единиц (кодирование). При кодировании каждый символ, введенный с клавиатуры, заменяется последовательностью из 8 двоичных разрядов в соответствии со стандартной кодовой таблицей, т.е. символ занимает один байт. Например, в соответствии с таблицей кодов ASCII D $\rightarrow$ 01000100; F $\rightarrow$ 00100110; 4 $\rightarrow$ 00110100; ? $\rightarrow$ 0011110.

При кодировании числа преобразуются в двоичное представление. Например,

$2 = 1\cdot2^1 + 0\cdot2^0 = 10_2$; $5 = 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 = 101_2$; $256 = 1\cdot2^8 = 100000000_2$.

При работе с числами различают:

1) ***целые***: $\pm n$;

2) ***вещественные***:

- с фиксированной десятичной точкой: $\pm n.m$;

- с плавающей десятичной точкой (экспоненциальная форма): $\pm n.m$E$\pm p$, где n, m - целая и дробная части числа, p - порядок; $\pm$0.xxxE$\pm$p - нормализованный вид.

С увеличением числа количество разрядов для его представления в двоичной системе резко возрастает, поэтому для размещения большого числа выделяется несколько подряд расположенных однобайтных ячеек. В этом случае адресом такой расширенной ячейки является адрес первого байта. Один бит такой ячейки выделяется под знак числа. Числа, размещенные таким образом - целые.

Для хранения вещественных чисел их предварительно приводят к нормализованному виду. Например, $35{,}6 = 0.356\cdot10^{+2}$, где 0.356 – мантисса, +2 – порядок. После этого переводят порядок и мантиссу в двоичную систему. Такое чис-

ло запоминается в комбинированной ячейке, один байт которой содержит порядок, несколько других содержат мантиссу. Числа, размещенные таким образом - вещественные.

Программа – это последовательность ***команд*** (инструкций), которые помещаются в памяти и выполняются процессором в указанном порядке.

Команда размещается в комбинированной ячейке следующим образом. Первый байт содержит код операции (КОП) (например + или – или *), которую необходимо выполнить над содержимым ячеек памяти. В одной, двух или трех ячейках (операндах команды) по 2 или 4 байта содержатся адреса ячеек (А1, А2, А3), над которыми нужно выполнить указанную операцию. Номер первого байта команды называется ее адресом. Последовательность из этих команд называется ***программой в машинных кодах*** (рис. 2).

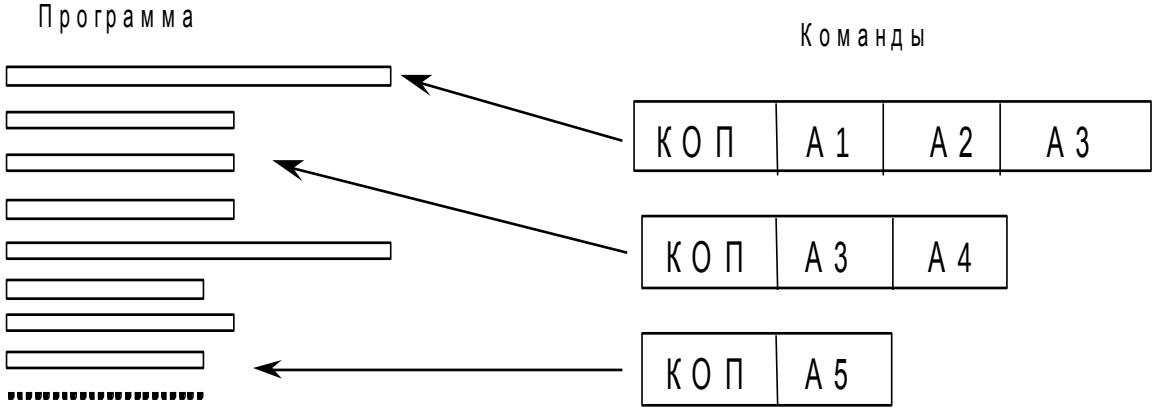


Рис. 2

1.5. Программные модули

Программист пишет программу на ***языке высокого уровня***, т.е. наиболее удобном для записи алгоритма решения определенного класса задач. Исходный текст программы, введенный с помощью клавиатуры в память компьютера - ***исходный модуль*** (sourse code, в языке Си имеет расширение *.cpp).

Транслятор - программа, осуществляющая перевод текстов с одного языка на другой, т.е. транслятор переводит программу с входного языка системы программирования на машинный язык ЭВМ, на которой функционирует данная система или будет функционировать разрабатываемая программа; либо на промежуточный язык программирования, уже реализованный или подлежащий реализации. Одной из разновидностей транслятора является ***компилятор***, обеспечивающий перевод программ с языка высокого уровня (приближенного к человеку) на язык более низкого уровня (близкий к ЭВМ), или машинозависимый язык.

Интерпретатор представляет собой программный продукт, выполняющий созданную программу путем одновременного анализа и реализации предписанных действий. При использовании интерпретатора отсутствует разделение на две стадии - перевод и выполнение.

Большинство трансляторов языка Си, с которыми мы будем работать - компиляторы.

Результат обработки исходного модуля компилятором - ***объектный модуль*** (object code, в языке Си имеет расширение *.obj). Он не может быть выполнен, т.е. это незавершенный вариант машинной программы, т.к., например, к нему должны быть присоединены модули стандартных библиотек. Здесь компилятор (compiler) - вид транслятора, представляющего программу-переводчик исходного модуля в язык машинных команд.

Исполняемый (абсолютный, загрузочный) модуль создает вторая специальная программа - «компоновщик». Ее еще называют редактором связей (Linker). Она и создает модуль, пригодный для выполнения на основе одного или нескольких объектных модулей.

Загрузочный модуль (Load module, расширение *.exe) – это программный модуль, представленный в форме, пригодной для загрузки его в память и выполнения.

1.6. Ошибки

Ошибки, допускаемые при написании программ, разделяют на синтаксические и логические.

Синтаксические ошибки - нарушение формальных правил написания программы на конкретном языке, обнаруживаются на этапе трансляции и могут быть легко исправлены.

Логические ошибки делят на ошибки алгоритма и семантические ошибки - могут быть найдены и исправлены только разработчиком программы.

Причина ошибки алгоритма - несоответствие построенного алгоритма ходу получения конечного результата сформулированной задачи.

Причина семантической ошибки - неправильное понимание смысла (семантики) операторов языка.

1.7. Функциональная и модульная декомпозиции

Для большинства задач алгоритмы их решения являются довольно большими и громоздкими. При программировании нужно стараться получить программу удобочитаемую, высокоэффективную и легко модифицируемую. Для этого производят декомпозицию сложного алгоритма поставленной задачи, т.е. его разбивка на отдельные более простые подзадачи, затем декомпозиция подзадач и т.д. Для этого используют приемы процедурного программирования.

Один из основных приемов - разбивка алгоритма на отдельные функции и/или модули, используя функциональную и/или модульную декомпозиции соответственно.

Функциональная декомпозиция - метод разбивки большой программы на отдельные функции, т.е. общий алгоритм - на отдельные шаги, которые потом и оформляют в виде отдельных функций.

Алгоритм декомпозиции можно представить следующим образом:

- программу делать как последовательность более мелких действий;
- каждую детализацию подробно описать;
- каждую детализацию представить в виде абстрактного оператора, который должен однозначно определять нужное действие, и в конечном итоге эти

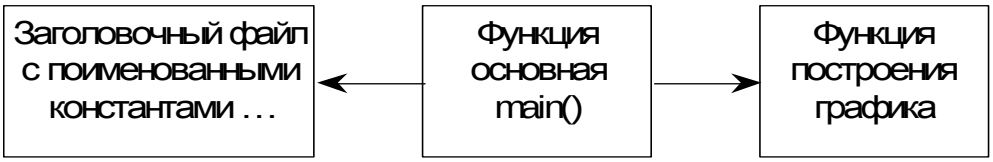
абстрактные действия заменятся на группы операторов выбранного языка программирования.

При этом надо помнить, что каждая детализация – это один из вариантов решения, и поэтому необходимо проверять, что

- решение частных задач приводит решению общей задачи;
- выбранная последовательность действий разумна;
- построенная декомпозиция позволяет получать команды, легко реализуемые на выбранном языке программирования.

Единицей компиляции в языке Си является отдельный файл (модуль). ***Модульная декомпозиция*** представляет собой разбиение программы на несколько отдельных файлов, каждый из которых решает отдельную конкретную задачу и, как правило, облегчает процесс ее работы. Кроме того, код программы, разделенный на отдельные файлы, позволяет части этого кода использовать в других программах.

Пример модульной декомпозиции кода Си:



1.8. Файловая система хранения информации

Для размещения информации и программ на различных устройствах компьютера, необходимых пользователю, была разработана концепция файлов.

Под ***файлом*** понимается поименованное на внешнем носителе место (запоминающее устройство, диск и т.п.), отведенное для размещения и (или) чтения некоторой информации. При этом файл может быть пустым, т.е. место отведено, поименовано, но информация отсутствует. Информация, помещенная в файл, получает имя этого файла, поэтому файлом часто называют эту размещаемую информацию.

За работу с файлами в компьютере отвечают специальные программы, набор которых называется ***файловой системой***, основные функции которой - предоставить пользователю удобные средства для работы с данными на всевозможных носителях.

Имя, которое присваивается файлу, может иметь тип, обычно называемый «расширение». Имя и тип разделяются точкой. При отсутствии типа точка необязательна.

Для более удобного размещения файлов введены каталоги.

Каталог (папка) – это группа файлов на одном носителе, имеющий свое имя. Если каталог вложен внутрь другого каталога, он является ***подкаталогом.*** Такая вложенность может быть многократной и тогда образуется иерархическая структура хранения данных.

Для удобства хранения внешним носителям присваиваются имена. Для дисков, например, имена обозначаются одной буквой - a:, b:, c:,…. При этом на одном винчестере для удобства размещения файлов может быть организовано несколько логических дисков с разными именами.

Маршрут (путь) файла. При сложной структуре хранения файлов может возникать такая ситуация, когда имеются разные файлы с одинаковым именем, но расположенные в разных каталогах или дисках. Для точной идентификации (указания) файла необходимо кроме имени указать на каком диске, и в какой цепочке подкаталогов он находится. Такая цепочка и называется путем к файлу. Например:

c:\bc31\doc\lec.doc или d:\work\prog.cpp.

Для работы с файлами обычно используют специальные программы, наибольшее распространение получили ***FAR***, ***WinCom*** и ***Проводник***.

1.9. Операционная система

Вся работа компьютера осуществляется под управлением специальных программ, называемых операционной системой (ОС). С точки зрения пользователя ОС представляет широкий набор системных команд, задавая которые можно потребовать от ПЭВМ выполнения многих полезных процедур и действий.

Часть программ ОС предназначена для управления процессом выполнения задач. Имеется группа программ, так называемого администратора системы, позволяющая следить за работой группы пользователей в рамках системы. Важное место занимает блок программ, обеспечивающих обмен сообщениями между пользователями сети, в том числе через интернет.

Удобства, предоставляемые пользователю, существенно зависят от качества ОС, которые по мере совершенствования компьютеров постоянно развиваются.

В настоящее время наибольшее распространение имеют ОС WindowsXX и LinuxXX.

2. Основные понятия и определения

2.1. Этапы решения задач на ЭВМ

Решение задачи на ЭВМ можно разбить на следующие этапы:

- математическая или информационная формулировка задачи;
- выбор метода (например, численного) решения поставленной задачи;
- построение алгоритма решения поставленной задачи;
- запись построенного алгоритма, т.е. написание текста программы;
- отладка программы - процесс обнаружения, локализации и устранения возможных ошибок;
- выполнение программы - получение требуемого результата.

2.2. Понятие алгоритма и способы его записи

Понятие алгоритма занимает центральное место в современной математике и программировании.

Алгоритмизация - сведение задачи к последовательным этапам действий так, что результаты предыдущих действий используются при выполнении следующих.

Числовой алгоритм - детально описанный способ преобразования числовых входных данных в выходные при помощи математических операций.

Существуют нечисловые алгоритмы, которые используются в экономике и технике, в различных научных исследованиях.

Тогда в общем: ***алгоритм*** - это строгая и четкая конечная система правил, определяющая последовательность действий над некоторыми объектами и после конечного числа шагов приводящая к достижению поставленной цели.

2.3. Свойства алгоритмов

Дискретность - значения новых величин (данных) вычисляются по определенным правилам из других величин с уже известными значениями.

Определенность (детерминированность) - каждое правило из системы однозначно, а данные однозначно связаны между собой, т.е. последовательность действий алгоритма строго и точно определена.

Результативность (конечность) - алгоритм решает поставленную задачу за конечное число шагов.

Массовость - алгоритм разрабатывается так, чтобы его можно было применить для целого класса задач, например алгоритм вычисления определенных интегралов с заданной точностью.

2.4. Способы описания алгоритмов

Существует несколько способов описания алгоритмов. Наиболее распространенными являются словесное и графическое описания алгоритма.

Словесное описание алгоритма рассмотрим на конкретном примере: пусть необходимо найти наибольший общий делитель для двух целых положительных чисел a и b.

1) Сравнить a и b. Если a<b, то положить d=a; m=b, иначе d=b и m=a.

2) Разделить m на d. Обозначить остаток от деления r.

3) Если d=0, то это искомое число. Закончить вычисления. Иначе перейти к пункту 4.

4) Заменить значение m значением d.

5) Заменить d значение значением r.

6) Перейти к пункту 2.

Здесь алгоритм описан с помощью естественного языка, а объекты обработки, являющиеся числами, обозначены буквами.

Например, рассмотрим алгоритм решения квадратного уравнения вида $a*x^2+b*x+c=0$:

1) вычислить D = b*b - 4 * a * c;

2) если D<0, то перейти к 4;

3) вычислить $X_1=(-b+\sqrt{D})/(2*a)$; $X_2=(-b-\sqrt{D})/(2*a)$;

4) конец.

2.5. Графическое описание алгоритма

Представление алгоритма в виде схемы, состоящей из последовательности блоков (геометрических фигур), каждый из которых отображает содержание очередного шага алгоритма, называется графическим описанием алгоритма. Внутри фигур кратко записывают действие, выполняемое в этом блоке. Такую схему называют блок-схемой алгоритма.

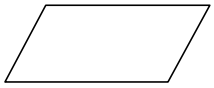
Правила изображения фигур сведены в единую систему программной документации (ГОСТ 19.701-90). По данному госту – это схема данных, которая отображает путь данных при решении задачи и определяет этапы их обработки.

Схема данных содержит:

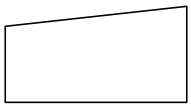
- символы данных (могут отображать тип носителя данных);
- символы процесса, который нужно выполнить над данными;
- символы линий, указывающих потоки данных между процессами и носителями данных;
- специальные символы, которые используют для удобства чтения схемы.

2.6. Основные символы схемы алгоритма

***Символы ввода-вывода данных**:*



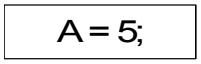
- данные ввода/вывода (носитель не определен);



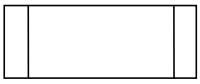
- ручной ввод данных с устройства любого типа, например, с клавиатуры;

- отображение данных в удобочитаемой форме на устройстве, например, дисплее.

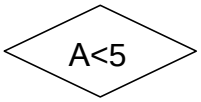
***Символы процесса**:*



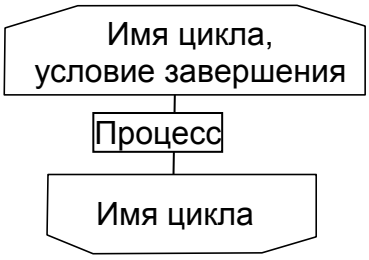
- процесс - отображение функции обработки данных, т.е. операции приводящей к изменению значения указанного объекта;



- предопределенный процесс - отображение группы операций, которые определены в другом месте, например, в подпрограмме;



- решение - отображение функции, имеющей один вход и ряд альтернативных выходов, их которых только один может быть активизирован после анализа условия: указанного внутри этого символа.



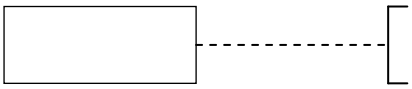
Граница цикла - отображение начала и конца цикла, или, наоборот, - условие завершения указывают в нижней границе.

Символы линий - отображают поток данных или управления. Линии - горизонтальные или вертикальные, имеющие только прямой угол перегиба. Стрелки - указатели направления не ставятся, если управление идет сверху-вниз или слева-направо.

Специальные символы

Соединитель - используется при обрыве линии и продолжении ее в другом месте (необходимо присвоить название).

Терминатор - вход из внешней среды или выход во внешнюю среду (начало или конец схемы программы).



Коментарий.

2.7. Пример простейшего линейного алгоритма

Наиболее часто в практике программирования требуется организовать расчет некоторого арифметического выражения при различных исходных данных. Например, такого:

$$z = \frac{tg^2 x}{\sqrt{x^2 + m^2}} + x^{(m+1)}\sqrt{x^2 + m^2} \quad ,$$

при x>0 - вещественное, m - целое.

Разработка алгоритма обычно начинается с составления схемы. Продумывается оптимальная последовательность вычислений, при которой, например, отсутствуют повторения. При написании алгоритма рекомендуется переменным присваивать те же имена, которые фигурируют в заданном арифметическом выражении, либо иллюстрируют их смысл.

Для того чтобы не было "длинных" операторов, исходное выражение полезно разбить на ряд более простых. В нашей задаче предлагается схема вычислений, представленная на рис. 3.

Она содержит ввод и вывод исходных данных, линейный вычислительный процесс, вывод полученного результата. Заметим, что выражение $\sqrt{x^2 + m^2}$ вычисляется только один раз. Введя дополнительные переменные a,b,c, мы разбили сложное выражение на ряд более простых выражений.

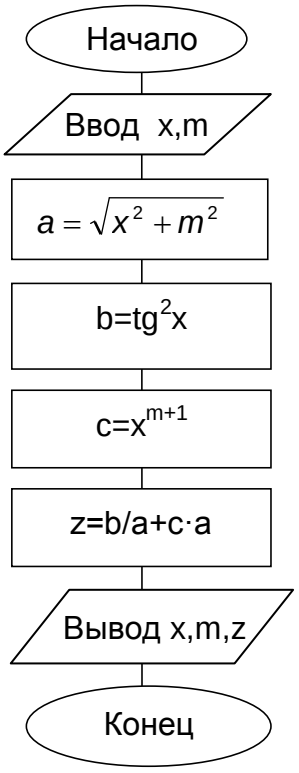


Рис. 3

2.8. Немного истории

Алгоритмический язык С был разработан в 1972 г. сотрудником фирмы AT&T Bell Laboratory Денисом Ритчи на базе языка B (автор К.Томпсон), который в свою очередь основывался на языке системного программирования BCPL. Первая версия языка была опубликована в книге авторов Д.Ритчи и Б.Кернигана и получила название стандарт K&R. Минимальная стандартная реализация, поддерживаемая любым компилятором, содержала всего 27 ключевых слов. Началось успешное развитие языка и чтобы избежать путаницы Американский институт стандартизации (American National Standart Institute) ввел в 1983 г. общий стандарт языка – ANSI стандарт.

Язык продолжает развиваться и в 1985 г. появляется язык C++, который в основном сохраняет все черты обычного С, но дополнен новыми существенными возможностями, которые позволили реализовать объектно-ориентированный стиль программирования.

Язык С отражает возможности современных компьютеров. Программы на С отличаются компактностью и быстротой исполнения. Структура С побуждает

программиста использовать в своей работе нисходящее программирование, структурное программирование, пошаговую разработку модулей.

Области применения языка С - системное программирование и прикладные задачи с жесткими требованиями по скорости и памяти.

3. Синтаксис языка Си

В языках программирования вместо номеров ячейкам памяти принято давать имена (идентификаторы), а содержимое ячеек называть ***переменными,*** или ***константами***, в зависимости от того, изменяется значение в процессе работы или нет.

В языке Си фундаментальным понятием является ***инструкция (операция, оператор, функция)***, которая представляет собой описание определенного набора действий. Таким образом, программа, написанная на языке Си, состоит из последовательности инструкций.

3.1. Алфавит языка

Каждому из множества значений, определяемых одним байтом, - от 0 до 255, - в таблице знакогенератора вычислительной машины ставится в соответствие символ. По системе кодировки фирмы IBM символы с кодами от 0 до 127, образующие первую половину таблицы знакогенератора, построены по стандарту ASCII и одинаковы на всех IBM-совместимых компьютерах. Вторая половина символов (коды 128 … 255) может отличаться на разных компьютерах. Обычно коды от 128 до 175 и от 224 до 239 используются для размещения символов национального алфавита, коды с 176 по 223 отводятся под символы псевдографики и коды с 240 по 255 – под специальные знаки (Приложение 1).

Алфавит языка Си включает:

- прописные и строчные буквы латинского алфавита, а также знак подчеркивания (код ASCII 95);
- арабские цифры от 0 до 9;
- специальные символы:

+(плюс) –(минус) *(звездочка) /(дробная черта) =(равно) >(больше) <(меньше) ;(точка с запятой) &(амперсант) [ ](квадратные скобки) { }(фигурные скобки) ()(круглые скобки) _(знак подчеркивания) (пробел) **.**(точка) **,**(запятая) :(двоеточие) #(номер) %(процент) ~(поразрядное отрицание) ?(знак вопроса) !(восклицательный знак) \(обратный слеш).

- пробельные (разделительные) символы: пробел, символы табуляции, перевода строки, возврата каретки, новая страница и новая строка.

3.2. Лексемы

Из символов алфавита формируются лексемы языка – минимальные значимые единицы текста в программе:

- идентификаторы;
- ключевые (зарезервированные) слова;
- знаки операций;
- константы;
- разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими, как разделители или знаки операций, а также комментариями.

3.3. Идентификаторы и ключевые слова

Идентификатор (в дальнейшем, для краткости - **ID**) – это имя программного объекта (константы, переменной, метки, типа, функции, модуля, поля в структуре). В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания; первым символом ID может быть буква или знак подчеркивания, но не цифра; пробелы внутри ID не допускаются.

Длина идентификатора определяется реализацией (версией) транслятора Си и редактора связей (компоновщика). Современная тенденция - снятие ограничений длины идентификатора.

При именовании объектов следует придерживаться общепринятых соглашений:

- ID переменной обычно пишется строчными буквами, например index (для сравнения: Index – это ID типа или функции, а INDEX – константа);
- идентификатор должен нести какой-либо смысл, поясняя назначение объекта в программе, например birth_date (день рождения) или sum (сумма);
- если ID состоит из нескольких слов, как, например birth_date, то принято либо разделять слова символом подчеркивания (birth_date), либо писать каждое следующее слово с большой буквы (birthDate).

Разделители идентификаторов объектов:

- пробелы;
- символы табуляции, перевода строки и страницы;
- комментарии (играют роль пробелов).

Наличие разделителей не влияет на работу программы.

В Си прописные и строчные буквы – различные символы. Идентификаторы Name, NAME, name – различные объекты.

Ключевые (зарезервированные) слова не могут быть использованы в качестве идентификаторов.

Ключевые слова Си:

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

3.4. Знаки операций

Знак операции – это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Операции делятся на унарные, бинарные и тернарные операции, по количеству участвующих в них операндов.

3.5. Литералы (константы)

Когда в программе встречается некоторое число, например 1, то это число называется литералом или литеральной константой. Константой, потому что мы не можем изменить его значение, и литералом, потому что буквально передает свое значение (от латинского literal – буквальный).

Литерал является неадресуемой величиной: хотя реально он, конечно, хранится в памяти машины, нет никакого способа узнать его адрес. Каждый литерал имеет определенный тип.

3.6. Комментарии

Еще один базовый элемент языка программирования – комментарий, – не является лексемой. Внутри комментария можно использовать любые допустимые на данном компьютере символы, а не только символы из алфавита языка программирования, поскольку компилятор комментарии игнорирует.

В Си комментарии ограничиваются парами символов /* и */, а в C++ был введен вариант комментария, который начинается символами // и заканчивается символом перехода на новую строку.

4. Базовые типы объектов

4.1. Простейшая программа

Программа написанная на языке Си состоит из одной или нескольких функций, причем одна функция обязательна имеет идентификатор (имя) main() – основная, главная. Ее назначение – управление всей работой программы (проекта). Данная функция, как правило, не имеет параметров и не возвращает результат, но наличие круглых скобок (как и для других функций без параметров) обязательно.

Общая структура программы на языке Си имеет вид:

<директивы препроцессора>
<определение типов пользователя – typedef>
<описание прототипов функций>
<определение глобальных переменных>
<функции>

В свою очередь, функции имеют такую структуру:

<класс памяти> <тип> < ID функции> (<объявление параметров>)
{ - начало функции
код функции
} - конец функции

Рассмотрим кратко основные части общей структуры программ.

Перед компиляцией программа на языке Си обрабатывается специальной программой – препроцессором, который работает под управлением директив.

Препроцессорные директивы начинаются с символа #, за которым следует наименование директивы, указывающее текущую операцию препроцессора.

Препроцессор решает ряд задач по предварительной обработке программы, основной из которых является «подключение» к программе так называемых

заголовочных файлов (обычных текстов) с декларацией стандартных библиотечных функций, которые используются в программе. Наименование такой директивы: *include* (подключить), а общий формат ее использования:

#include < ID_файла.h>

где «h» – расширение заголовочных файлов.

Если идентификатор файла заключен в угловые скобки (< >), то поиск данного файла производится в стандартной директории с этими файлами, если же ID файла заключено в двойные кавычки (" "), то поиск данного файла производится в текущей директории.

К наиболее часто используемым библиотекам относятся:
stdio.h - содержит стандартные функции файлового ввода-вывода;
conio.h - функции для работы с консолью (клавиатура, экран монитора);
math.h - математические функции.

Второе основное назначение препроцессора – это обработка макроопределений. Макроподстановка define (определить) имеет общий вид:

#define < ID > <строка>

Например: #define PI 3.1415927

В ходе препроцессорной обработки программы появление в тексте идентификатора PI везде заменяется значением 3.1415927.

Основные возможности препроцессора приведены в Приложении 3.

Рассмотрим небольшой пример, позволяющий понять самые простейшие приемы программирования на языке Си:

```
#include <stdio.h>
    void main(void)
    {                          // Начало функции main
        printf(" Высшая оценка знаний - 10 !");
    }                          // Окончание функции main
```

Отличительным признаком функции служат круглые скобки ( ) после идентификатора функции, в которые заключается список аргументов. Если аргументы отсутствуют, указывают атрибут *void* - отсутствие значения. Перед ID функции обычно указывается тип возвращаемого ею результата, так как функция main() ничего не возвращает - в качестве результата указывается void.

Код функции представляет собой набор инструкций, каждая из которых оканчивается символом «;». В нашем примере одна инструкция - функция *printf*() выполняет форматный вывод данных на экран, в данном случае, указанную фразу.

4.2. Основные типы данных

Данные в языке Си разделяются на две категории: простые (скалярные), будем их называть базовыми, и сложные (составные) типы данных.

Основные типы базовых данных: стандартный целый (int), вещественный с одинарной точностью (float) и символьный (char).

В свою очередь, данные целого типа могут быть короткими (short), длинными (long), и без знаковыми (unsigned), а вещественные - с удвоенной точностью (double).

Сложные типы – массивы, структуры (struct), объединения или смеси (union), перечисление (enum).

Данные целого и вещественного типов находятся в определенных числовых диапазонах так как занимают разный объем оперативной памяти:

Таблица 1

Тип данных	Объем памяти (байт)	Диапазон значений
char	1	-128 …127
int	2	-32768…32767
short	2(1)	-32768…32767(-128…127)
long	4	-2147483648…2147483647
unsigned int	4	0…65535
unsigned long	4	0…4294967295
float	4	$3{,}14*10^{-38}$…$3{,}14*10^{38}$
double	8	$1{,}7*10^{-308}$ $1{,}7*10^{308}$

4.3. Декларация (объявление) объектов

Все объекты, с которыми работает программа в Си необходимо декларировать, т.е. объявить компилятору об их присутствии в программе. При этом возможны две формы декларации:

- описание, не приводящее к выделению памяти;
- определение, при котором под объект будет выделен объем оперативной памяти, в соответствии с его типом; в этом случае объект можно сразу инициализировать, т.е. задать его начальное значение.

Кроме констант, которые можно задавать в исходном тексте, все объекты программы должны быть явно декларированы по следующему формату:

<атрибуты> <список ID *объектов>;*

элементы списка разделяются запятыми, а атрибуты - разделителями. Например: *int i,j,k; float a,b;*

Объекты программы в общем случае имеют следующие атрибуты:

<класс памяти> - характеристика способа размещения объектов в памяти (статическая, динамическая), определяет область видимости и время жизни переменной (по умолчанию - *auto*), данные атрибуты будут рассмотрены позже;

<тип> - характеристика механизма интерпретации данных, т.е. это совокупность информации о том, сколько объекту нужно выделить памяти, какой вид имеет представление информации и какие действия над ней допустимы (по умолчанию - *int*).

Класс памяти и тип – атрибуты необязательные и могут отсутствовать, тогда их значения установятся по умолчанию.

Примеры декларации простых объектов:

```
int i,j,k;
char r;
double gfd;
```

Рассмотрим основные базовые типы данных более подробно.

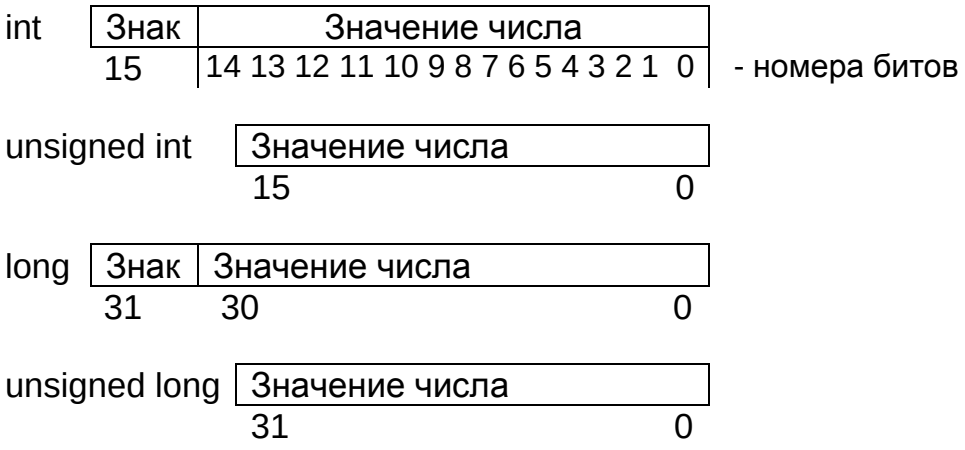
4.4. Данные целого типа (*int*)

Тип int - целое число, обычно соответствующее естественному размеру целых в используемой ЭВМ. Квалификаторы short и long, которые можно использовать с типом int, указывают на различные размеры целых, т.е. определяют размер памяти, выделяемый под переменные (табл.1).

Примеры: short int x;
long int x;
unsigned int x = 8; (декларация с одновременной инициализацией числом 8).

Атрибут int в таких ситуациях может быть опущен.

Атрибуты signed и unsigned показывают, как интерпретируется старший бит числа, как знак или как часть числа:



Если указан только атрибут int, это означает - short signed int.

4.5. Данные символьного типа (*char*)

Символьная переменная занимает в памяти 1 байт и представляется кодом от 0 до 255. Закрепление конкретных символов за кодами производится кодовыми таблицами.

Для персональных компьютеров наиболее распространена ASCII (American Standard Code for Information Interchenge) таблица кодов (Приложение 1). Данные типа char рассматриваются компилятором как "целые", поэтому возможно использование signed char (по умолчанию) - символы с кодами от -128 до +127 (т.е. только символы с кодами до 127) и unsigned char - символы с кодами от 0 до 255 (в том числе и русские).

Примеры:

char res, simv1, simv2;

char let = 's'; (декларация с одновременной инициализацией символом s).

4.6. Данные вещественного типа (*float, double*)

Данные вещественного типа в памяти занимают, соответственно, float - 4 байта; double - 8 байт; long double (повышенная точность) - 10 байт. Для размещения данных типа float обычно 8 бит выделено для представления порядка и знака и 24 бита под мантиссу.

Тип	Точность (мантисса)	Порядок
float	7 цифр после запятой	$\pm$ 38
double	15	$\pm$ 308
Long double	19	$\pm$ 4932

5. Константы в программах

Константы - объекты, не подлежащие использованию в левой части оператора присваивания, т.к. константа - является неадресуемой величиной и, хотя она хранится в памяти ЭВМ, нет никакого способа узнать ее адрес. В языке Си константами являются:

- самоопределенные арифметические, символьные и строковые данные;
- идентификаторы массивов и функций;
- элементы перечислений.

Арифметические константы могут быть целого или вещественного типов.

5.1. Целочисленные константы

Общий формат: ±*n* (+ обычно не ставится).

Десятичные константы - последовательность цифр 0...9, первая из которых не должна быть 0. Например, 22 и 273 - обычные целые константы, если нужно ввести длинную целую константу, то указывается признак *L*(*l*) - 273*L* (273*l*). Для такой константы будет отведено – 4 байта. Обычная целая константа, которая слишком длинна для типа *int* рассматривается как *long*.

Существует система обозначений для восьмеричных и шестнадцатеричных констант.

Восьмеричные константы - последовательность цифр от 0 до 7, первая из которых должна быть 0, например: 020 = 16-десятичное.

Шестнадцатеричные константы - последовательность цифр от 0 до 9 и букв от A до F (a...f), начинающаяся символами 0X (0x), например: 0X1F (0x1f) = 31-десятичное.

Восьмеричные и шестнадцатеричные константы могут также заканчиваться буквой L(l) - long, например, 020L или 0X20L.

Примеры целочисленных констант:

1992	13, 777	1000*L*	- десятичные;
0777	00033	01*l*	- восьмеричные;
0x123	0X00ff	0xb8000*l*	- шестнадцатеричные.

5.2. Константы вещественного типа

Данные константы размещаются в памяти по формату double, а во внешнем представлении могут иметь две формы:

1) с фиксированной десятичной точкой, формат записи: ±*n*.*m*, где *n*, *m* - целая и дробная части числа;

2) с плавающей десятичной точкой (экспоненциальная форма): ±*n*.*m*E±*p*, где *n*, *m* - целая и дробная части числа, *p* - порядок; ±0.xxxE±p - нормализованный вид, например, $1{,}25 \cdot 10^{-8} = 0.125E-8$.

Примеры констант с фиксированной и плавающей точками:

1.0 -3.125 100e-10 0.12537e+13

5.3. Символьные константы

Символьная константа - это символ, заключенный в одинарные кавычки: 'A', 'x' (занимает 1 байт). Тип *char* → целое *int*.

Так же используются специальные последовательности символов, это управляющие последовательности (escape-последовательности), основные их них:

\n - новая строка;
\t - горизонтальная табуляция;
\0 - нулевой символ (пусто).

При присваивании символьной переменной эти последовательности должны быть заключены в апострофы. Символьная константа '\0', изображающая символ 0 (нуль – пусто), часто записывается вместо целой константы 0, чтобы подчеркнуть символьную природу некоторого выражения.

Текстовые символы непосредственно вводятся с клавиатуры, а специальные и управляющие представляются в исходном тексте парами текстовых символов. Примеры представления специальных символов языка Си:

**** - обратная косая черта; **\'** - апостроф; **\"** - кавычки.

Примеры символьных констант: 'A', '9', '$', '\n', '\72'.

5.4. Строковые константы

Строковая константа представляет собой последовательность символов кода ASCII, заключенная в кавычки (”) . Во внутреннем представлении к строковым константам добавляется нулевой символ '\0', еще называемый нуль-терминатор, отмечающий конец строки. Кавычки не являются частью строки, а служат только для ее ограничения. Строка - это массив, состоящий из символов. Внутреннее представление константы "01234\0ABCDEF":

'0','1','2','3','4','\0','A','B','C','D','E','F','\0'

Примеры строковых констант:

"Система", "\n\t Аргумент \n", "Состояние \"WAIT\""

В конец строковой константы компилятор автоматически помещает нуль-символ. Нуль-символ - это не цифра 0, он на печать не выводится и в таблице кода ASCII имеет код 0.

Например, строка " " - пустая строка (нуль-строка).

6. Обзор операций

6.1. Операции, выражения

Операции языка Си предназначены для управления данными (более 40). Для их использования необходимо знать:

- синтаксис;
- приоритеты (15 уровней);
- порядок выполнения.

Выражения состоят из операндов, операций, скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев.

Операции, применяемые к одному операнду, - унарные, к двум операндам – бинарные, есть операция с тремя операндами - тернарная. Операции выпол-

няются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки.

Большинство операций выполняются слева направо, например, a+b+c → (a+b)+c. Исключение: унарные операции, операции присваивания и условная операция (?:) - справа налево.

Полный список операций в соответствии с их приоритетом приводится в Приложении 2.

Рассмотрим кратко основные операции языка Си.

6.2. Арифметические операции

Арифметические операции - бинарные. Перечень арифметических операций и их обозначений:

+ - сложение;
- - вычитание (либо унарная операция - изменение знака);
/ - деление (для int операндов - с отбрасыванием остатка);
* - умножение;
% - остаток от деления целочисленных операндов со знаком первого операнда (деление по модулю).

Операндами традиционных арифметических операций (+ - * /) могут быть константы, переменные, идентификаторы функций, элементы массивов, указатели, любые арифметические выражения.

Порядок выполнения операций:

- выражения в круглых скобках;
- функции (стандартные математические, функции пользователя);
- * / выполняются слева направо;
- + — слева направо.

Порядок выполнения операций можно определять круглыми скобками, тогда выражение в скобках выполняется в первую очередь (слева направо).

Унарные операции + и – обладают самым высоким приоритетом, определены только для целых и вещественных операндов, «+» носит только информационный характер, «–» меняет знак значения операнда на противоположный (не адресная операция).

Таким образом, так как операции *, /, % обладают высшим приоритетом над операциями +, -, при записи сложных выражений нужно использовать общепринятые математические правила:

x+y*z-a/b ⇔ x+(y*z)-(a/b)

6.3. Операции присваивания

Формат операции присваивания:

< ID > = <выражение>;

Присваивание значения в языке Си, в отличие от традиционной интерпретации рассматривается как *выражение*, имеющее значение левого операнда после присваивания. Таким образом, присваивание может включать несколько операций присваивания, изменяя значения нескольких операндов. Например:

```
int i, j, k;
float x, y, z;
    ...
```

i = j = k = 0; ⇔ k = 0; j = k; i = j;
x = i+(y = 3) - (z = 0); ⇔ z = 0; y = 3; x = i + y - z;

Внимание, левым операндом операции присваивания может быть только именованная либо косвенно адресуемая указателем переменная. Примеры недопустимых выражений:

а) присваивание константе: 2 = x+y;
б) присваивание функции: getch() = i;
в) присваивание результату операции: (i+1) = 2+y;

6.4. Сокращенная запись операции присваивания

В языке Си допускается две разновидности сокращений записи операции присваивания:

а) вместо записи:

v = v @ e;

где @ - арифметическая операция либо операция над битовым представлением операндов, рекомендуется использовать запись

v @ = e;

например, i = i + 2; ↔ i + = 2;

б) вместо записи (автоувеличение):

x = x # 1;

где # - символ + либо -, обозначающий операцию инкремента либо декремента, x - целочисленная переменная, переменная-указатель), рекомендуется использовать запись

префиксная: ##x; постфиксная: x##;

Если операции инкремента или декремента в коде программы используются в чистом виде, то различий в постфиксной и префиксной формами нет. Если же эти операции используются в выражении, то при использовании префиксной формы (##x), сначала значение x изменится на 1, а затем будет использовано в выражении. В постфиксной форме (x##), значение сначала используется в выражении, а затем изменяется на 1. Операции над указателями рассмотрим позже.

Примеры использования сокращений (фрагменты программ):

1)
```
int i,j,k;
float x,y;
   ...
x* = y;      ↔   x = x*y;
i+ = 2;      ↔   i = i+2;
x/ = y+15;   ↔   x = x/(y+15);
k--;         ↔   k = k-1;
--k;         ↔   k = k-1;
j = i++;     ↔   j = i;   i = i+1;
j = ++i;     ↔   i = i+1;   j = i;
```

2)
```
int n,a,b,c,d;
n = 2; a = b = c = 0;
a = ++n;    //  n=3, a=3
a+ = 2;     //  a=5
b = n++;    //  b=3, n=4
```

```
b- = 2;        //  b=1
c = --n;       //  n=3, c=3
c* = 2;        //  c=6
d = n--;       //  d=3, n=2
d% = 2;        //  d=1
```

Рекомендации использования сокращений обоснованы возможностью оптимизации программы. Например, схема выражения вида v@=e соответствует схеме выполнения многих машинных команд типа "регистр-память", а использование ##x и x## наличием в Си специальных процессорных команд инкремента и декремента.

6.5. Преобразование типов операндов арифметических операций

При выполнении операций могут встречаться операнды различных типов. В этом случае они преобразуются к общему типу в соответствии с небольшим набором правил.

Типы операндов преобразуются в порядке увеличения их "размера памяти", т.е. объема памяти, необходимого для хранения их значений. Поэтому можно говорить, что неявные преобразования всегда идут от "меньших" объектов к "большим". Схема выполнения преобразований операндов арифметических операций:

short, char $\rightarrow$ int $\rightarrow$ unsigned $\rightarrow$ long $\rightarrow$ double
float $\rightarrow$ double

Горизонтальные стрелки отмечают преобразования даже однотипных операндов перед выполнением операции, т.е. действуют следующие правила:

- значения типов char и short всегда преобразуются в int;
- если любой из операндов (*a* или *b*) имеет тип double, то второй преобразуется в double;
- если один из операндов long, то другой преобразуется в long.

***Внимание*:** результатом 1/3 будет «0», чтобы избежать такого рода ошибок необходимо явно изменять тип хотя бы одного операнда, т.е. записывать, например: 1. / 3.

Типы char и int могут свободно смешиваться в арифметических выражениях. Каждая переменная типа char автоматически преобразуется в int. Это обеспечивает значительную гибкость при проведении определенных преобразований символов.

При присваивании значение правой части преобразуется к типу левой, который и является типом результата. И здесь необходимо быть внимательным, так как при некорректном использовании операций присваивания могут возникнуть неконтролируемые ошибки. Так, при преобразовании int в char старший байт просто отбрасывается.

Если объявлены: float x; int i; то как x=i; так и i=x; приводят к преобразованиям. При этом float преобразуется в int отбрасыванием дробной части.

Тип double преобразуется во float округлением.

Длинное целое преобразуется в более короткое целое и переменные типа char посредством отбрасывания лишних битов более высокого порядка.

При передаче данных функциям также происходит преобразование типов: в частности, char становится int, a float - double.

6.6. Операция приведения типа

В любом выражении преобразование типов может быть осуществлено явно. Для этого достаточно перед любым выражением поставить в скобках идентификатор соответствующего типа.

Вид записи операции: *(тип) выражение*;

Ее результат - значение выражения, преобразованное к заданному типу представления.

Операция приведения типа вынуждает компилятор выполнить указанное преобразование, но ответственность за последствия возлагаются на программиста. Рекомендуется использовать эту операцию в исключительных случаях.

Например:

```
float x;
int n=6, k=4;
1) x=(n+k)/3;        - дробная часть будет отброшена
2) x=(float)(n+k)/3; - использование операции приведения типа здесь позволяет избежать округления результата деления целочисленных операндов.
```

Также использование операции преобразования типа может оказаться полезным, если нужно преобразовать фактический параметр к типу соответствующего формата параметра функции. Функции exp, log, sqrt из стандартной библиотеки математических функций рассчитаны на параметр типа double и дают результат типа double. Если нужно получить натуральный логарифм от значения переменной x типа float нужно написать: log ((double) x).

6.7. Операции сравнения

```
==   - равно или эквивалентно;
!=   - не равно;
<    - меньше;
<=   - меньше либо равно;
>    - больше;
>=   - больше либо равно.
```

Пары символов соответствующих операций разделять нельзя.

Общий вид операций отношений:

<выражение1> <знак_операции> <выражение2>

Общие правила:

- операндами могут быть любые базовые (скалярные) типы;
- значения операндов после вычисления перед сравнением преобразуются к одному типу;
- результат операции отношения - целое значение 1, если отношение истинно, или 0 в противном случае. Следовательно, операция отношения может использоваться в любых арифметических выражениях.

6.8. Логические операции

Перечень логических операций в порядке убывания относительного приоритета и их обозначения:

```
!    - отрицание (логическое НЕТ);
```

&& - коньюнкция (логическое И);

|| - дизьюнкция (логическое ИЛИ).

Общий вид операции отрицания:

!<выражение>

Общий вид операций коньюнкции и дизьюнкции

<выражение1> <знак_операции> <выражение2>

Например:

y>0 && x=7 → истина, если 1-е и 2-е выражения истинны;

e>0 || x=7 → истина, если хотя бы одно выражение истинно.

Ненулевое значение операнда трактуется как "истина", а нулевое - "ложь".

Например:

!0 → 1

!5 → 0

x=10;

!((x=y)>0) → 0

Особенность операций коньюнкции и дизьюнкции – экономное последовательное вычисление выражений-операндов:

<выражение1> <операция><выражение2>,

- если выражение1 операции коньюнкция ложно, то результат операции - ноль и выражение2 не вычисляется;

- если выражение1 операции дизьюнкция истинно, то результат операции - единица и выражение2 не вычисляется.

Таким образом, появляется возможность записью логического выражения задать условную последовательность вычисления выражений в направлении слева направо:

scanf("%d",&i) && test1(i) && test2(i) → нулевой результат одной из функций приведет к игнорированию вызова остальных;

search1(x) || search2(x) || search3(x) → только ненулевой результат одной из функций приведет к игнорированию вызова остальных.

Пример правильной записи двойного неравенства:

0<x<100 ↔ (0<x)&&(x<100)

6.9. Побитовые логические операции. Операции над битами

В СИ предусмотрен набор операций для работы с отдельными битами слов. Эти операции нельзя применять к переменным вещественного типа (float, double).

Перечень операций над битами и их обозначения:

~ - дополнение (унарная операция); инвертирование (одноместная операция);

& - побитовое И - конъюнкция;

| - побитовое включающее ИЛИ - дизъюнкция;

^ - побитовое исключающее ИЛИ - сложение по модулю 2;

>> - сдвиг вправо;

<< - сдвиг влево.

Пары символов (>>,<<) разделять нельзя.

Общий вид операции инвертирования:

~ <выражение>

Остальные операции над битами имеют вид:

<выражение1> <знак_операции> <выражение2>

Операндами операций над битами могут быть только выражения, приводимые к целому типу. Операции (~, &, |, ^) выполняются поразрядно над всеми битами операндов (знаковый разряд особо не выделяется):

~0xF0 ⇔ x0F
0xFF & 0x0F ⇔ x0F
0xF0 | 0x11 ⇔ xF1
0xF4 ^ 0xF5 ⇔ x01

Операция & часто используется для маскирования некоторого множества битов. Например, оператор w = n & 0177 передает в w семь младших битов n, полагая остальные равными нулю.

Операции сдвига выполняются также для всех разрядов с потерей выходящих за границы битов.

Операция (!) используется для включения битов w = x ! y, устанавливает в единицу те биты в x, которые =1 в y.

Необходимо отличать побитовые операции & и ! от логических операций && и ||, которые подразумевают вычисление значения истинности слева направо. Если x=1, y=2, то x & y равно нулю, а x && y равно 1.

0x81<<1 ⇔ 0x02
0x81>>1 ⇔ 0x40

Если выражение1 имеет тип unsigned, то при сдвиге вправо освобождающиеся разряды гарантированно заполняются нулями (логический сдвиг). Выражения типа signed могут, но не обязательно, сдвигаться вправо с копированием знакового разряда (арифметический сдвиг). При сдвиге влево освобождающиеся разряды всегда заполняются нулями. Если выражение2 отрицательно либо больше длины выражения1 в битах, то результат операции сдвига не.

Унарная операция (~) дает дополнение к целому. Это означает, что каждый бит со значением 1 получает значение 0 и наоборот. Эта операция обычно оказывается полезной в выражениях типа:

X & (~)077,

где последние 6 битов X маскируются нулем. Это выражение не зависит от длины слова и поэтому предпочтительнее, чем, например:

X & 0177700,

где предполагается, что X занимает 16 битов, такая переносимая форма не требует никаких дополнительных затрат.

Операции сдвига << и >> осуществляют соответственно сдвиг вправо (влево) своего левого операнда, на число битовых позиций, задаваемых правым операндом. Таким образом, x<<2 сдвигает x влево на две позиции, заполняя, освобождающиеся биты, нулями, что эквивалентно умножению на 4.

Операции сдвига вправо на k разрядов весьма эффективны для деления, а сдвиг влево - для умножения целых чисел на 2 в степени k:

x<<1 ↔ x*2
x>>1 ↔ x/2
x<<3 ↔ x*8

Подобное применение операций сдвига безопасно для беззнаковых и положительных значений выражения1.

Двуместные операции над битами (&, |, ^, <<, >>) могут использоваться в сокращенных формах записи операции присваивания:

```
int i,j,k;
        . . .
i |= j          ↔     i = i | j        - включение в поле i единиц из поля j;
i &= 0xFF       ↔     i = i & 0xFF     - выделение в поле i единиц по маске поля 0x00FF;
k ^= j          - выделение в поле k отличающихся разрядов в полях k и j;
i ^= i          - обнуление всех разрядов поля i .
```

Операции над битами реализуются, как правило, одной машинной командой и рекомендуются для использования во всех подходящих случаях.

В математическом смысле операнды логических операций над битами можно рассматривать как отображение некоторых множеств с размерностью не более разрядности операнда на значения {0,1}.

Пусть единица означает обладание элемента множества некоторым свойством, тогда очевидна теоретико-множественная интерпретация рассматриваемых операций:

~ - дополнение; **|** - объединение; **&** - пересечение.

Простейшее применение - проверка нечетности целого числа:

```
int i;
...
  if (i &1) printf(" Значение i четно!");
```

Комбинирование операций над битами с арифметическими операциями часто позволяет упростить выражения. Например, получение размера области в параграфах размером 16 байт для размещения объекта размером x байт:

```
(x + 15)>>4
```

Другие возможности оперирования над битами:

- использование структур с битовыми полями;
- доступ к битам как разрядам арифметических данных.

6.10. Операция , (запятая)

Данная операция используется при организации строго гарантированной последовательности вычисления выражений. Форма записи:

выражение1**,** …**,** выражениеN;

выражения1,…,N вычисляются гарантированно последовательно и результатом операции становится значение выражения N.

Пример:

```
m=(i=1, j=i++, k=6, n=i+j+k);
```

получим последовательность вычислений: i=1, j=i=1, i=2, k=6, n=2+1+6, и в результате m=n=9. Данный пример ничем не отличается от такого участка кода:

```
i =1; j = i; i++; k = 6; n = i+j+k; m = n;
```

Данная операция используется там, где по синтаксису допустима только одна операция, а нам необходимо разместить несколько последовательно выполняемых операций (см. оператор for).

При передаче последовательности вычислений в функцию в качестве параметра – их необходимо взять в скобки.

7. Обзор базовых инструкций языка С

7.1. Стандартная библиотека языка Си

В любой программе кроме операторов и операций используются средства библиотек, входящих в среду программирования, которые облегчают создание программ.

Часть библиотек - стандартизована и поставляется с компилятором.

В стандартную библиотеку входят функции, макросы, глобальные константы. Это файлы с расширением *.h, хранящиеся в папке **include**.

Рассмотрим наиболее часто используемые функции из стандартной библиотеки языка Си.

7.2. Стандартные математические функции

Математические функции алгоритмического языка Си декларированы в файлах <math.h> и <stdlib.h>. В последующих записях аргументы *x* и *y* имеют тип double; параметр *n* имеет тип int.

Аргументы тригонометрических функций должны быть заданы в радианах (2π радиан = 3600).

Большинство математических функций (приведенных здесь) возвращают значение (результат) типа double.

Математическая функция	ID функции в языке Си
$\sqrt{x}$	sqrt(x)
|x|	fabs(x)
e^x	exp(x)
x^y	pow(x,y)
ln(x)	log(x)
$lg_{10}(x)$	log10(x)
sin(x)	sin(x)
cos(x)	cos(x)
tg(x)	tan(x)
arcsin(x)	asin(x)
arccos(x)	acos(x)
arctg(x)	atan(x)
arctg(x / y)	atan2(x)
$sh(x)=0.5\ (e^x-e^{-x})$	sinh(x)
$ch(x)=0.5\ (e^x+e^{-x})$	cosh(x)
tgh(x)	tanh(x)
остаток от деления *x* на *y*	fmod(x,y)
наименьшее целое >=x	ceil(x)

наибольшее целое <=x	floor(x)

Единственной исключительной (контролируемой) ситуацией при выполнении арифметических операций является ***деление на нуль***, другие виды ошибочных ситуаций (переполнение, исчезновение порядка или потеря значимости) игнорируются.

7.3. Функции вывода данных на дисплей

Для вывода информации в языке Си чаще всего используются функции: printf() и puts().

Формат функции **printf()**:

printf((<управляющая строка>, список объектов вывода);

- в управляющей строке, заключенной в кавычки, записывают: поясняющий текст; список модификаторов форматов, указывающих компилятору способ вывода объектов (признаком модификатора формата является символ %);

- в списке объектов вывода указываются идентификаторы печатаемых объектов, разделенных запятыми: переменные, константы или выражения, вычисляемые перед выводом на печать.

Количество и порядок следования форматов должен совпадать с количеством и порядком следования печатаемых объектов.

Функция printf() выполняет форматированный вывод данных в соответствии с указанными форматами, поэтому формат может использоваться и для преобразования типов выводимых объектов.

Если признака модификации (%) нет, то вся информация выводится как комментарии.

Основные модификаторы формата:

%d (%i) - десятичное целое число;
%c - один символ;
%s - строка символов;
%f - число с плавающей точкой, десятичная запись;
%e - число с плавающей точкой, экспоненциальная запись;
%g - используется вместо f,e для исключения незначащих нулей;
%o - восьмеричное число без знака;
%x - шестнадцатеричное число без знака.

Для чисел long добавляется символ *l*, например, %ld - длинное целое, %lf – число вещественное с удвоенной точностью (double).

Если нужно напечатать сам символ %, то его нужно указать 2 раза.

printf("Только %d%% предприятий не работало. \n",5);

Получим: Только 5% предприятий не работало.

Так же используются специальные последовательности символов, это управляющие последовательности (escape-последовательности):

\n - новая строка;
\t - горизонтальная табуляция;
\b - шаг назад;
\r - возврат каретки;
\v - вертикальная табуляция;
\f - перевод формата (переход на новую строку);

\\ - обратная косая;
\' - апостроф;
\" - кавычки;
\0 - нулевой символ (пусто).

При присваивании символьной переменной эти последовательности должны быть заключены в апострофы. Например, можно записать: NEXTF='\n'; а затем вывести на печать переменную NEXTF.

В модификаторах формата функции printf() после символа % можно указывать строку цифр, задающую минимальную ширину поля вывода, например: %5d (для целых), %4.2f (для действительных - две цифры после запятой для поля, шириной 4 символа). Если этой ширины не хватает, происходит автоматическое расширение.

Можно использовать функцию printf() для нахождения кода ASCII некоторого символа.

```
printf(" %c  -  %d\n",'a','a');
```

получим десятичный код ASCII символа a: a - 65

Функция **puts()** выводит на экран дисплея строку символов, автоматически добавляя к ней символ перехода на начало новой строки.

Функция **putchar()** выдает на экран дисплея один символ без добавления символа “\n”.

7.4. Функции ввода информации

Функция **scanf()** предназначена для форматированного ввода исходной информации с клавиатуры

Общий вид этой функции:

```
scanf (<управляющая строка>, список адресов элементов ввода>);
```

Количество, тип и порядок следования форматов должен точно совпадать с количеством, типом и порядком следования вводимых объектов, иначе результат ввода непредсказуем.

Для нее также как и для printf() указывается управляющая строка, в которой указываются ***только модификаторы форматов*** (без пробелов), список аргументов. Но если функция printf() использует идентификаторы переменных, константы и выражения, то scanf() использует только указатели на переменные, то есть их адреса.

Таким образом, для ввода значения переменной перед ее идентификатором требуется указать символ &, обозначающий адрес переменной.

Если нужно ввести значение строковой переменной, то использовать символ & не нужно, так как строка - это массив символов, а ID массива эквивалентно адресу его нулевого элемента, т.е. его базовому адресу. Например:

```
int course;
float grant;
char name[20];
printf(" Укажите курс, стипендию, имя \n");
scanf("%d%f%s",&course, &grant,  name);
```

Вводить данные с клавиатуры можно как в одной строке через пробелы, так и в разных строках.

Функция scanf() использует практически тот же набор спецификаций преобразования (форматов), что и функция printf(), отличия - отсутствует формат %g, форматы %e,%f - эквивалентны.

Внимание, используя функцию scanf() для ввода строки по формату %s необходимо помнить, что ввод происходит только до первого пробела. Для ввода фраз, состоящих из слов используется функция:

gets(<ID строковой переменной>);

7.5. Ввод - вывод потоками

Поток - это абстрактное понятие расширенной версии языка С, которое относится к любому переносу данных от источника к приемнику.

Чтение данных из потока - это извлечение (extraction).

Вывод данных в поток - помещение или включение (insertion).

Для ввода-вывода используются две переопределенные операции побитового сдвига << , >>. Формат записи:

```
cout  <<  ID_переменной;
cin >>  ID_переменной;
```

Стандартный поток вывода ***cout*** - по умолчанию подключен к монитору.

Стандартный поток ввода ***cin*** - по умолчанию подключен к клавиатуре.

Для их работы необходимо подключить стандартную библиотеку **iostream.h**.

Пример:

```
#include<iostream.h>
#include<conio.h>
        void  main(void) {
                cout << " Hello! " << endl;    //  end line - переход на новую строку
                cout << " Input  i, j ";
                int  i, j, k;
                cin  >> i >> j ;
                k = i + j ;
                cout << " Sum  i , j = " << k << endl;
        }
```

7.6. Дополнительные функции

В дальнейшем мы будем часто пользоваться стандартными библиотечными функциями. Рассмотрим некоторые из них.

Функция void **clrscr**(void); - полностью очищает экран дисплея, переводя курсор в верхний левый угол. При работе в Visual C++ очистку экрана можно выполнить, используя функцию **system**("**cls**");

Функция int **kbhit**(void); - возвращает ненулевое целое значение при нажатии клавиши, в противном случае - нулевое.

8. Синтаксис операторов языка С

Операторы языка Си можно разделить на три группы:

- операторы-декларации (рассмотрены ранее);
- операторы преобразования объектов;

- операторы управления процессом выполнения алгоритма.

Программирование процесса преобразования объектов программы производится посредством записи выражений. Выражение включает один или несколько операндов и символов операций. Любое выражение, заканчивающееся символом «;» (точка с запятой), является оператором.

Простейший вид операторов - операторы-выражения.

Простые операторы:

- оператор присваивания - выполнение операций присваивания;
- оператор вызова функции - выполнение операции вызова функции;
- пустой оператор «;».

Классы ***управляющих операторов*** в языке Си следующие:

- операторы условного и безусловного перехода;
- оператор выбора альтернатив (переключатель);
- операторы организации циклов;
- операторы передачи управления (перехода).

Каждый из управляющих операторов имеет конкретную лексическую конструкцию, образуемую из ключевых слов языка С, выражений и символов-разделителей: { } , : ( ) .

Операторы языка С записываются в свободном формате с использованием разделителей между ключевыми словами. Допустима вложенность операторов. В случае необходимости можно использовать составной оператор - ***блок***, состоящий из любой последовательности операторов, заключенных в фигурные скобки - { и }, после закрывающей скобки символ «;» не ставится.

8.1. Условные операторы

В языке С имеется две разновидности условных операторов: простой и полный. Синтаксис простого оператора условного выполнения:

if (выражение) оператор1;

здесь выражением, как правило, является логическое или выражение отношения. Если выражение в скобках не ноль, т.е. истинно, то выполняется оператор1, иначе он игнорируется. Оператор1 - простой или составной (блок).

Примеры записи:

```
if (x>0) x=0;
if (i!=1)   j++,  s=1;          - используем операцию «запятая»;
if (i!=1) { j++;  s=1; }        - последовательность операций;
        if (getch()!=27) {      - если нажата клавиша, не “Esc”.
                        k=0;
                        }
        if (i) exit(1);         ↔     if (i!=0) exit(1);
        if (i>0)
        if (i<n) k++;           ↔     if ((i>0)&&(i<n)) k++;
        if (1) i=0;             ↔     i=0;
```

Синтаксис полного оператора условного выполнения:

```
if (выражение) оператор1;
        else оператор2;
```

Если выражение в скобках не ноль (истина), то выполняется оператор1, иначе - оператор2. Операторы 1 и 2 могут быть простыми или составными.

Примеры записи:

```
if (x>0) j=k+10;
        else m=i+10;
```

Если есть вложенная последовательность операторов if-else, то else связывается с ближайшим предыдущим if, не содержащим else. Например:

```
if (n>0)
   if(a>b)  z=a;
        else   z=b;
```

Если необходимо связать фразу else с внешним if, то используем операторные скобки:

```
if(n>0)
        { if (a>b)  z=a;   }
   else   z=b;
```

В следующей цепочке операторов if-else-if выражения просматриваются последовательно:

```
if (выражение1)  оператор1;
   else if (выражение2)  оператор2;
        else if (выражение3)  оператор3;
                else   оператор4;
```

Если какое-то выражение оказывается истинным, то выполняется относящийся к нему оператор и этим вся цепочка заканчивается. Каждый оператор может быть либо отдельным оператором, либо группой операторов в фигурных скобках. Последняя часть с else имеет дело со случаем, когда ни одно из проверяемых условий не выполняется. Иногда при этом не нужно предпринимать никаких явных действий, в этом случае else оператор4; может быть опущен, или его можно использовать для контроля, чтобы засечь "невозможное" условие (экономия на проверке условий).

Пример:

```
if ( n < 0 ) printf ( "n отрицательное\n" );
        else if ( n==0 ) printf ( "n равно нулю\n );
                else prinf ( "n положительное\n );
```

8.2. Условная операция «? :»

Условная операция - тернарная, в ней участвуют три операнда. Формат написания условной операции следующий:

выражение 1 ? выражение 2 : выражение 3;

если выражение 1 отлично от нуля (Истинно), то результатом операции является выражение 2, в противном случае - результатом операции является выражения 3. Каждый раз вычисляется только одно из выражений 2 или 3.

Запишем оператор if, вычисляющий максимум из *a* и *b* и присваивающий его значение *z*.

```
if (a > b)  z=a;
        else  z=b;
```

Используя условную операцию, этот пример можно записать:

```
z = (a>b) ? a : b;
```

Условную операцию можно использовать также как и любое другое выражение. Если выражения 2 и 3 имеют разные типы, то тип результата определяется по правилам преобразования.

8.3. Оператор выбора альтернатив (переключатель)

Общий вид оператора:

```
switch (выражение) {
        case константа1: оператор1;  break;
        case константа2: оператор2;  break;
             ...
        case константаN: операторN;  break;
        default:  оператор(N+1);  break;        // может отсутствовать
        }
```

Значение вычисленного выражения должно быть целого типа (или символьного, поскольку он автоматически преобразуется в целый). Это целое используется для выбора одного из нескольких операторов, который нужно выполнить. Оператор, следующий за селектирующим выражением состоит из одного или более операторов, перед каждым из которых стоит конструкция:

```
case константное выражение:
```

Целочисленное выражение (константа выбора) после вычисления сравнивается со значениями констант и при совпадении с одной из них выполняется передача управления соответствующему оператору. В случае несовпадения значения выражения с одной из констант происходит переход на метку default, либо, при ее отсутствии, к оператору, следующему за оператором switch.

Управляющий оператор break (разрыв) позволяет организовать выход из оператора switch на первый выполняемый оператор, следующий после данной конструкции (оператор switch).

Пример 1 с использованием оператора break:

```
void main(void)
{   int i = 2;
        switch(i)      {
                case 1: puts ( "Случай 1. "); break;
                case 2: puts ( "Случай 2. "); break;
                case 3: puts ( "Случай 3. "); break;
                default: puts ( "Случай default. "); break;
              }
    }
```

Для того, чтобы выйти из оператора switch в любом месте использовали оператор break, поэтому результатом будет: Случай 2.

Пример 2 (оператор break отсутствует):

```
void main()
{   int i=2;
        switch(i)      {
                case 1: puts ( "Случай 1. ");
                case 2: puts ( "Случай 2. ");
                case 3: puts ( "Случай 3. ");
                default: puts ( "Случай default. ");
              }
```

```
}
```

Так как оператор разрыва отсутствует, результатом будет:

Случай 2.

Случай 3.

Случай default.

9. Составление циклических алгоритмов

9.1. Понятие цикла

Практически все алгоритмы решения задач содержат циклически повторяемые участки. Цикл это одно из фундаментальных понятий программирования. Под циклом понимается организованное повторение некоторой последовательности операторов.

Для организации циклов используются специальные операторы.

Любой цикл состоит из кода цикла, т.е. тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла.

Один проход цикла называется итерацией. Проверка условия выполняется на каждой итерации либо до кода цикла (с предусловием), либо после кода цикла (с постусловием).

Перечень разновидностей операторов цикла:

- оператор цикла с предусловием;
- оператор цикла с постусловием;
- оператор цикла с предусловием и коррекцией.

9.2. Оператор с предусловием while

Общий вид:

while (выражение) код_цикла;

Если выражение в скобках - истина (не равно 0), то выполняется код_цикла. Это повторяется до тех пор, пока выражение не примет значение 0 (ложь). В этом случае выполняется оператор, следующий за while. Если выражение в скобках - ложно (равно 0), то цикл не выполнится ни разу.

Код_цикла может включать любое количество управляющих операторов, связанных с конструкцией while, взятых в фигурные скобки (блок), если их более одного. Среди этих операторов могут быть continue - переход к следующей итерации цикла и break - выход из цикла.

Например, необходимо сосчитать количество символов в строке. Предполагается, что входной поток настроен на начало строки. Тогда подсчет символов выполняется следующим образом:

```
char ch;
int count=0;
while (( ch=getchar())!='\n')   count++;
```

Для выхода из цикла while при истинности выражения, как и для выхода из других циклов можно пользоваться оператором break.

Пример 1:

```
while (1) {                              // Организация бесконечного цикла
        ...
    if (kbhit()&&(getch()==27)) break;
// Если нажата клавиша (результат работы функции kbhit()>0) и код ее равен 27
// (код клавиши “Esc”), то выходим из цикла
        ...
    }
```

Пример 2:

```
    ...
    while (!kbhit());      // Выполнять до тех пор, пока не нажата клавиша
    ...
```

9.3. Оператор цикла с постусловием do - while

Общий вид записи:

do код_цикла ***while*** (выражение);

Код_цикла будет выполняться до тех пор, пока «выражение» истинно. Все, что говорилось выше справедливо и здесь, за исключением того, данный цикл всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз.

9.4. Оператор цикла с предусловием и коррекцией for

Общий вид оператора:

for (выражение1; выражение2; выражение3) код_цикла;

Цикл for эквивалентен последовательности инструкций:

```
выражение1;
while (выражение2)
{
    код_цикла ...
    выражение3;
}
```

здесь выражение1 - инициация счетчика (начальное значение), выражение2 - условие продолжения счета, выражение3 - увеличение счетчика. Выражения 1,2 и 3 могут отсутствовать (пустые выражения), но символы «;» опускать нельзя.

Например, для суммирования первых N натуральных чисел можно записать:

```
sum = 0;
for ( i=1; i<=N; i++) sum+=i;
```

Операция «запятая» чаще всего используется в операторе for. Она позволяет включать в его спецификацию несколько инициализирующих выражений. Предыдущий пример можно записать в виде:

```
for ( sum=0 , i=1; i<=N; sum+= i , i++) ;
```

Оператор for имеет следующие возможности:

- можно вести подсчет с помощью символов, а не только чисел:

```
for (ch = 'a'; ch<='z'; ch++) ... ;
```

- можно проверить выполнение некоторого произвольного условия:

```
for (n = 0; s[i]>='0' && s[i]<'9'; i++) ... ;
```

или:

```
for (n = 1; n*n*n <=216; n++) ... ;
```

Первое выражение не обязательно должно инициализировать переменную. Необходимо только помнить, что первое выражение вычисляется только один раз перед тем, как остальные части начнут выполняться.

```
for (printf(" вводить числа по порядку! \n"); num!=6;)
        scanf("%d", & num);
printf(" последнее число - это то, что нужно.\n");
```

В этом фрагменте первое сообщение выводится на печать один раз, а затем осуществляется прием вводимых чисел, пока не поступит число 6.

Параметры, входящие в выражения, находящиеся в спецификации цикла можно изменять при выполнении операций в коде цикла.

Например:

```
for (n = 1; n<1000; n += delta) ... ;
```

Параметр delta можно менять в процессе выполнения цикла.

Использование условных выражений позволяет во многих случаях значительно упростить программу. Например:

```
for (i=0;i<n;i++)
   printf("%6d%c",a[i],( (i%10==0) || (i==n-1) ) ? '\n' : ' ');
```

В этом цикле печатаются *n* элементов массива *a* по 10 в строке, разделяя каждый столбец одним пробелом и заканчивая каждую строку (включая последнюю) одним символом перевода строки. Символ перевода строки записывается поле каждого десятого и *n*-го элементов. За всеми остальными - пробел.

10. Операторы передачи управления

Формально к операторам передачи управления относятся:

- оператор безусловного перехода **goto**;
- оператор перехода к следующему шагу (итерации) цикла **continue**;
- выход из цикла, либо оператора switch - **break;**
- оператор возврата из функции **return**.

Рассмотрим их более подробно.

10.1. Оператор безусловного перехода goto

В языке Си предусмотрен оператор goto, хотя в большинстве случаев можно обойтись без него. Общий вид оператора:

goto < метка >;

Он предназначен для передачи управления на оператор, помеченный меткой. Метка представляет собой идентификатор, оформленный по всем правилам идентификации переменных с символом «двоеточие» после него, например, пустой помеченный оператор:

***m1*: ;**

Область действия метки - функция, где эта метка определена. В случае необходимости можно использовать блок.

Наиболее характерный случай использования оператора goto - выполнение прерывания (выхода) во вложенной структуре при возникновении грубых неисправимых ошибок во входных данных. И в этом случае необходимо, выйти

из двух (или более) циклов, где нельзя использовать непосредственно оператор break, т.к. он прерывает только самый внутренний цикл:

```
for (...)
    for (...)
{ ...
        if ( ошибка )  goto error;
}
        ...
error:  - операторы для устранения ошибки;
```

Если программа обработки ошибок нетривиальна и ошибки могут возникать в нескольких местах, то такая организация оказывается удобной.

Пример нахождения первого отрицательного числа в двумерном массиве:

```
for (i=0; i<N; i++)
    for(j=0; j<M; j++)
    {
        if (v[i][j]<0)  goto found;
            ...                     // Не найден
    }
found:  ...                         // Найден
```

Программа с goto может быть написана и без него, хотя, за счет повторения некоторых проверок и введения дополнительных переменных. Например, рассмотренная программа может быть записана следующим образом:

```
found = 0;
    for (i=0; i<N && !found; i++)
        for (j=0; j<M && !found; j++)
            found = v[i][j]<0;
            if (found)   ...                    // Найден
                else     ...                    // Не найден
```

10.2. Оператор continue

Этот оператор может использоваться во всех типах циклов, но не в операторах переключателя switch. Наличие оператора continue вызывает пропуск "оставшейся" части итерации и переход к началу следующей, т.е. досрочное завершение текущего шага и переход к следующему шагу.

В циклах while и do это означает непосредственный переход к проверочной части. В цикле for управление передается на шаг коррекции, т.е. модификации выражения 3.

Фрагмент программы обработки только положительных элементов массива *a*, отрицательные значения пропускаются:

```
for ( i = 0; i<n; i++)
{   if( a[i]<0)  continue;
   ...                          // обработка положительных элементов
 }
```

Оператор continue часто используется, когда последующая часть цикла оказывается слишком сложной, так что рассмотрение условия, обратного проверяемому, приводит к слишком высокому уровню вложенности программы.

10.3. Оператор break

Оператор ***break*** производит экстренный выход из самого внутреннего цикла или оператора-переключателя switch, к которому он принадлежит, и передает управление первому оператору, следующему за текущим оператором.

10.4. Оператор return

Оператор ***return*;** производит досрочный выход из текущей функции. Он, так же возвращает значение результата функции: ***return*** <выражение>;

В функциях, не возвращающих результат, он неявно присутствует после последнего оператора. Значение выражения при необходимости преобразуется к типу возвращаемого функцией значения.

Пример 1:

```
float estim(float *x, int n)  {
    int i;
    float y;
    if ((!x)||(!n) {
            error(x,n);
            return 0;
            }
   for (y=i=0; i<n; i++) y+=x[i];
      return y/n;
    }
```

Пример 2:

```
void error(void *x, int n)
{
        if (!x) printf("\nМассив не создан");
        if (!n) printf("\nМассив пустой");
}
```

11 . Указатели

11.1. Указатели

Указатель – это переменная, которая может содержать адрес некоторого объекта. Указатель объявляется следующим образом:

<тип> *< ID переменной-указателя>;

Например: int *a; double *f;   char *w;

С указателями связаны две унарные операции & и *.

Операция **&** означает «взять адрес» операнда. Операция ***** имеет смысл - «значение, расположенное по указанному адресу».

Обращение к объектам любого типа как операндам операций в языке С может проводиться:

- по имени (идентификатору - ID);
- по указателю (операция косвенной адресации):

указатель = &ID_объекта;

Пример 1:

```
int x,                  // переменная типа int
*y;                     // указатель на элемент данных типа int
y=&x;                   // y - адрес переменной x
*y=1;                   // косвенная адресация указателем поля x, т.е.
                        // по указанному адресу записать 1 → x=1;
```

Пример 2:

```
int i, j=8,k=5, *y;
  y=&i;
  *y=2;                 // i=2
   y=&j;
  *y+=i;                // j+=i  →  j=j+i  →  j=j+2=10
   y=&k;
   k+=*y;               // k+=k → k=k+k = 10
   (*y)++;              // k++   → k=k+1 = 10+1 = 11
```

При вычислении адресов объектов следует учитывать, что идентификаторы массивов и функций являются константными указателями. Такую константу можно присвоить переменной типа указатель, но нельзя подвергать преобразованиям, например:

```
int x[100], *y;
y = x;          // Правильно - присваивание константы переменной
x = y;          // Ошибка: в левой части - указатель-константа
```

Указателю-переменной можно присвоить значение другого указателя, либо выражения типа указатель с использованием, при необходимости, операции приведения типа. Приведение типа необязательно, если один из указателей имеет тип "void *".

```
int i,*x;
char *y;
x=&i;                   // x → поле объекта int
y=(char *)x;            // y → поле объекта char
y=(char *)&i;   // y → поле объекта char
```

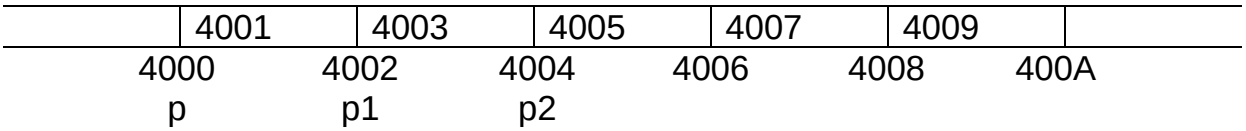
Значение указателя можно вывести на экран с помощью спецификации %p (pointer), результат выводится в шестнадцатеричном виде.

Рассмотрим фрагмент программы:

```
int a=5, *p, *p1, *p2;
p=&a;  p2=p1=p;
++p1;p2+=2;
      printf("a=%d, p=%d, p=%p, p1=%p, p2=%p.\n", a, p, p, p1, p2);
```

Результат выполнения: a=5, *p=5, p=FFC8, p1=FFCC, p2=FFD0.

Графически это выглядит так (адреса взяты символически):



p=4000, p1=4002=(4000+1*sizeof(*p)) -> 4000+2 (int)

p2=4004=(4000+2*sizeof(*p)) -> 4000+2*2

11.2. Операции над указателями (косвенная адресация)

Указатель может использоваться в выражениях вида

p # ie, ##p, p##, p# = ie,

где: p - указатель, ie - целочисленное выражение, # - символ операции '+' или '-'.

Значением таких выражений является увеличенное или уменьшенное значение указателя на величину ie*sizeof(*p). Следует помнить, что операции с указателями выполняются в единицах памяти того типа объекта, на который ссылается этот указатель.

Текущее значение указателя всегда ссылается на позицию некоторого объекта в памяти с учетом правил выравнивания для соответствующего типа данных. Таким образом, значение p#ie указывает на объект того же типа, расположенный в памяти со смещением на ie позиций.

При сравнении указателей могут использоваться отношения любого вида (">", ">=", "<", "<=", "==", "!="). Наиболее важными видами проверок являются отношения равенства или неравенства.

Отношения порядка имеют смысл только для указателей на последовательно размещенные объекты (элементы одного массива).

Разность двух указателей дает число объектов адресуемого ими типа в соответствующем диапазоне адресов. Очевидно, что уменьшаемый и вычитаемый указатель также должны соответствовать одному массиву, иначе результат операции не имеет практической ценности.

Любой указатель можно сравнивать со значением NULL, которое означает недействительный адрес. Значение NULL можно присваивать указателю как признак пустого указателя. NULL заменяется препроцессором на выражение (void *)0.

11.3. Ссылка

Ссылка - это не тип данных, а константный указатель, т.е. это объект, который указывает на положение другой переменной.

Ссылка - это константный указатель, который отличается от переменного указателя тем, что для ссылки не требуется специальной операции разименования. Над указателями возможны арифметические операции. Над ссылкой арифметические операции запрещены, т.к. ссылка декларируется следующим образом:

```
type &ID = инициализатор;
```

Инициализатор - это идентификатор объекта, на который в дальнейшем будет указывать ссылка. Пример:

```
int  a = 8;
int  &r = a;
```

Ссылка получила псевдоним объекта указанного в качестве инициализатора. В данном примере, одинаковыми будут следующие действия:

```
a++;
r++;
```

12. Массивы

12.1. Понятие массива

В математике для удобства записи различных операций часто используют индексированные переменные: векторы, матрицы, тензоры. Так, вектор $\vec{c}$ представляется набором чисел (c_1 c_2....c_n), называемых его компонентами, причем каждая компонента имеет свой номер, который принято обозначать в виде индекса. Матрица А – это таблица чисел (a_{ij}, i=1,..., m; j=1,..., n), i – номер строки, j – номер столбца. Операции над матрицами и векторами обычно имеют короткую запись, которая обозначает определенные, порой сложные действия над их индексными компонентами. Например, произведения двух векторов записывается как

$$\vec{c} \cdot \vec{b} = \sum_{i=1}^{n} c_i \, b_i \, .$$

Произведение матрицы на вектор

$$\vec{b} = A \cdot \vec{c}, \quad b_i = \sum_{j=1}^{n} a_{ij} \cdot c_j \, .$$

Произведение двух матриц

$$D = A \cdot G, \quad d_{ij} = \sum_{k=1}^{n} a_{ik} \cdot g_{kj} \, .$$

Введение индексированных переменных в языках программирования также позволяет значительно облегчить реализацию многих сложных алгоритмов, связанных с обработкой массивов однотипных данных.

В языке Си для этой цели имеется сложный тип переменных – ***массив***, представляющий собой упорядоченную конечную совокупность элементов одного типа. Число элементов массива называют его размером. Каждый элемент массива определяется идентификатором массива и своим порядковым номером - индексом. Индекс – целое число, по которому производится доступ к элементу массива. Индексов может быть несколько. В этом случае массив называют многомерным, а количество индексов одного элемента массива является его размерностью.

12.2. Одномерные массивы

Индексы у одномерных массивов в языке Си начинаются с 0, а в программе одномерный массив объявляется следующим образом:

<тип> < ID_массива>[размер]={список начальных значений};

где: тип – базовый тип элементов (целый, вещественный, символьный);
размер – количество элементов массива.

Список начальных значений используется при необходимости инициализировать данные при объявлении, он может отсутствовать.

Размер массива может задаваться константой или константным выражением. Нельзя задавать массив переменного размера. Для этого существует отдельный механизм – динамическое выделение памяти.

Пример объявления массива целого типа: int a[5];

В массиве «а» первый элемент: a[0], второй – a[1], … пятый - a[4].

Обращение к элементу массива в программе на языке Си осуществляется в традиционном для многих других языков стиле - записи операции обращения по индексу, например:

```
a[0]=1;
a[i]++;
a[3]=a[i]+a[i+1];
```

Пример объявления массива целого типа с инициализацией значений:

```
int a[5]={2, 4, 6, 8, 10};
```

Если в группе {…} список значений короче, то оставшимся элементам присваивается 0.

Внимание. В языке С с целью повышения быстродействия программы отсутствует механизм контроля границ изменения индексов массивов. При необходимости такой механизм должен быть запрограммирован явно.

12.3. Многомерные массивы

Как уже отмечалось, в языке Си кроме одномерных массивов возможна работа с многомерными массивами. Объявление многомерного массива:

```
<тип> < ID >[размер1][размер2]…[размерN]={{список начальных значений},
                                          {список начальных значений},…};
```

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке Си размещаются в памяти компьютера в последовательности столбцов.

Например, элементы двухмерного массива b[3][2] размещаются в памяти компьютера в следующем порядке:

```
b[0][0], b[0][1], b[1][0], b[1][1], b[2][0], b[2][1].
```

Следующий пример иллюстрирует определение массива целого типа, состоящего из трех строк и четырех столбцов, с одновременным присвоением его элементам (инициализацией) начальных значений:

```
int a[3][4] = {{1,2,0,0},{9,-2,4,1},{-7,0,0,0}};
```

Если в какой-то группе { } список значений короче, то оставшимся элементам присваивается 0.

12.4. Операция sizeof

Данная операция позволяет определить размер объекта по ID или типу, результатом является размер памяти в байтах (тип результата int). Формат записи:

sizeof(параметр);

где: «параметр» – тип или идентификатор объекта (не ID функции).

Если указан идентификатор сложного объекта (массив, структура, объединение), то получаем размер всего сложного объекта. Например:

```
sizeof(int)   →    размер памяти – 2 байта,
int b[5];
sizeof(b)     →    размер памяти – 10 байт.
```

Одно из полезных применений операции sizeof – определение реального количества элементов массива с объявленной размерностью:

```
char s[256];
int kol = sizeof(s)/sizeof(*s);          // количество элементов массива s
```

Наиболее часто операция sizeof применяется при динамическом распределении памяти:

```
float *x;                  // Указатель массива
int n;                     // Количество элементов массива
x=calloc(n,sizeof(*x));    // Захват и очистка памяти
```

12.5. Применение указателей

Идентификатор массива – это адрес памяти, начиная с которого он расположен, т.е. адрес его первого элемента. Работа с массивами тесно связана с применением указателей.

Пусть объявлены массив *a* из 5 целочисленных элементов и указатель *q* на целочисленные переменные:

```
int a[5], *q;
```

ID массива *a* является константным указателем на его начало.

a

a[0]	a[1]	a[2]	a[3]	a[4]

4000

Здесь приведено символическое изображение оперативной памяти, выделенной компилятором для объявленного целочисленного массива a[5]. Указатель ***a*** содержит адрес его начала в оперативной памяти (ОП), т.е. «символический адрес»=4000 (a=4000).

Если выполнена операция: q=a; - присваивание константы переменной, т.е. q=4000 (аналог: q=&a[0]), то с учетом адресной арифметики выражения a[i] и *(q+i) приводят к одинаковым результатам – обращению к i-му элементу массива.

Идентификаторы a и q - указатели, очевидно, что выражения a[i] и *(a+i) эквивалентны. Отсюда следует, что операция обращения к элементу массива по индексу применима и при его именовании переменной-указателем. Таким образом, для любых указателей можно использовать две эквивалентные формы выражений для доступа к элементам массива: q[i] и *(q+i). Первая форма удобнее для читаемости текста, вторая - эффективнее по быстродействию программы.

Например, для получения значения 4-го элемента массива можно написать a[3] или *(a+3), результат будет один и тот же.

Очевидна эквивалентность выражений:

1) получение адреса начала массива в ОП:

&a[0] ↔ &(*a) ↔ a

2) обращение к первому элементу массива:

*a ↔ a[0]

Последнее объясняет правильность выражения для получения количества элементов массива с объявленной размерностью:

```
type x[100]; // Размерность должна быть константой
    ...
int n = sizeof(x) / sizeof(*x);
```

Указатели, как и переменные любого другого типа, могут объединяться в массивы.

Объявление массива указателей на целые числа имеет вид:

```
int *a[10], y;
```

Теперь каждому из элементов массива можно присвоить адрес целочисленной переменной *y*, например: a[1]=&y;

Чтобы теперь найти значение переменной *y* через данный элемент массива *a*, необходимо записать *a[1].

12.6. Указатели на указатели

В языке Си можно описать переменную типа «указатель на указатель». Это ячейка оперативной памяти, в которой будет храниться адрес указателя на какую либо переменную. Признак такого типа данных – повторение символа «*» перед идентификатором переменной. Количество символов «*» определяет уровень вложенности указателей друг в друга. При объявлении указателей на указатели возможна их одновременная инициализация. Например:

```
int a=5;
int *p1=&a;
int **pp1=&p1;
int ***ppp1=&pp1;
```

Теперь присвоим целочисленной переменной *a* новое значение, например 10. Одинаковое присваивание произведут следующие операции:

a=10; *p1=10; **pp1=10; ***ppp1=10;

Для доступа к области ОП, отведенной под переменную ***a*** можно использовать и индексы. Справедливы следующие аналоги, если мы работаем с многомерными массивами:

*p1 <-> p1[0] **pp1 <-> pp1[0][0] ***ppp1 <-> ppp1[0][0][0]

Отметим, что идентификатор двухмерного массива – это указатель на массив указателей (переменная типа указатель на указатель: int **m;), поэтому выражение a[i][j] эквивалентно выражению *(*(m+i)+j).

Например, двухмерный массив m[3][4]; компилятор рассматривает как массив четырех указателей, каждый из которых указывает на начало массива со значениями размером по три элемента каждый.

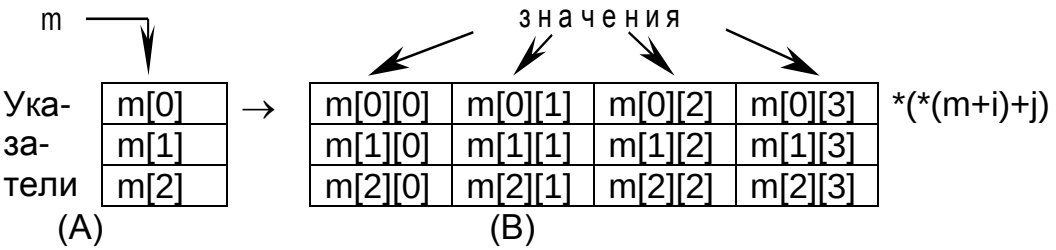


Рис. 4

Очевидна и схема размещения такого массива в памяти - последовательное (друг за другом) размещение "строк" - одномерных массивов со значениями. Аналогичным образом можно установить соответствие между указателями и массивами с произвольным числом измерений:

float name[][][][]; ↔ float ****name;

Пример программы конструирования массива массивов:

```
#include <stdio.h>
int x0[4]={ 1, 2, 3, 4};
int x1[4]={11,12,13, 14};                    // Декларация и инициализация
```

```
        int x2[4]={21,22,23, 24};                  //  массивов целых чисел
        int *y[3]={x0,x1,x2};                      // Создание массива указателей
void main(void)
        {      int i,j;
               for (i=0; i<3; i++)
                 {  printf("\n %2d)",i);
                  for (j=0; j<4; j++)   printf(" %2d",y[i][j]);
                   }
           }
```

Результаты работы программы:

```
 0)  1  2  3 4
 1) 11 12 13 14
 2) 21 22 23 24
```

Такие же результаты будут получены и при таком объявлении массива:

```
int y[3][4]={
        { 1, 2, 3, 4},
        {11,12,13,14},              //  Декларация и инициализация
        {21,22,23,24},              // массива массивов целых чисел
        };
```

В последнем случае массив указателей на массивы создается компилятором. Здесь собственно данные массива располагаются в памяти последовательно по строкам, что является основанием для объявления массива *y* в следующем виде:

```
int z[3][4]={  1, 2, 3, 4,
               11,12,13,14,              //  Декларация и инициализация
               21,22,23,24};             // массива массивов целых чисел
```

Замена скобочного выражения z[3][4] на z[12] здесь не допускается, так как массив указателей в данном случае создан не будет.

12.7. Адресная функция

Использование многомерных массивов в языке Си связано с расходами оперативной памяти на массивы указателей.

Можно избежать таких расходов, если ввести адресную функцию для доступа к элементам одномерного массива (цепочка одномерных массивов со значениями) по значениям индексов многомерного массива. Например, адресная функция для следующего трехмерного массива x(1..n1, 1..n2, 1..n3) выглядит следующим образом:

$$L(i, j, k)=n1*n2*(i-1) + n2*(j-1)+k,$$

где индексы принадлежат диапазонам: i=1..n1, j=1..n2, k=1..n3.

Для размещения такого массива требуется область оперативной памяти размером n1*n2*n3*(число байт отводимых компилятором для одного элемента массива в зависимости от типа данных). Рассматривая такую область как одномерный массив *y* размером n1*n2*n3 можно установить соответствие элемента массива x элементу памяти для его размещения:

$$x(i, j, k)=y(L(i, j, k));$$

Так как массивы являются данными сложного (составного) типа, операции над ними желательно выполнять, используя стандартные библиотечные функции, например, для объявленных массивов: type x[100], y[100]; операция присваивания:

memcpy(x,y,sizeof(x)); - содержимое массива y присвоить содержимому массива x.

13. Работа с динамической памятью

В языке С размерность массива при объявлении должна задаваться константным выражением. При необходимости работы с массивами переменной размерности вместо массива достаточно объявить указатель требуемого типа и присвоить ему адрес свободной области памяти (захватить память). После обработки массива занятую память надо освободить. Библиотечные функции работы с памятью описаны в файле **alloc.h**.

Пример создания динамического массива:

```
float *x;
int    n;
printf("\nРазмерность - "); scanf(" %d",&n);
if ((x = calloc(n, sizeof(*x)))==NULL)  {                  // Захват памяти
        printf("\n Предел размерности “);
                    exit(1);
              }
 else {
   printf("\n Массив создан !");
                    ...
               for (i=0; i<n; i++)
                     printf("\n%f",x[i]);
                    ...
             free(x);                                      // Освобождение памяти
        }
```

В C++ введены две операции: захват памяти - ***new***, освобождение, захваченной ранее памяти - ***delete***.

Общий формат записи:

```
указатель = new  type  (значение);
        . . .
delete  указатель;
```

Например:

```
int  *a;
        a = new int (8);
```

В данном случае создана целочисленная динамическая переменная, на которую установлен указатель a и которой присвоено начальное значение 8. После работы с ней освобождаем память:

```
        . . .
        delete a;
```

Операции ***new*** и ***delete*** для массивов:

указатель = new тип [количество] ;

Результат операции – адрес начала области памяти для размещения данных, указанного количества и типа. При нехватке памяти – результат NULL.

Операция delete:

delete [ ] указатель;

13.1. Пример создания одномерного динамического массива:

Массив объявляем указателем.

```
...
double *x;
int   i, n;
...
puts(" Введите размер массива: ");
scanf("%d", &n);
x = new double [n] ;
if (x == NULL) {
puts(" Предел размерности ! ");
return;
}
for (i=0; i<n; i++)                    // Ввод элементов массива
        scanf("%lf", &x[i]);
...
delete [ ]x;                           // Освобождение памяти
...
```

13.2. Пример создания двухмерного динамического массива:

Напомним, что ID двухмерного массива - указатель на указатель (рис. 4):

```
...
int  **m, n1, n2;
puts(" Введите размеры массива (количество строк и столбцов: ");
scanf("%d%d", &n1, &n2);
m = new  int * [n1];                   // Захват памяти для указателей - A (n1=3)
for ( int i=0; i<n1; i++)              // Захват памяти для элементов - B (n2=4)
        m[i] = new  int [n2];
                . . .
for ( i=0; i<n1; i++)
        for ( j=0; j<n2; j++)
                m[i] [j] = i+j;        //  *(*(m+i)+j) = i+j;
                . . .
for ( i=0; i<n1; i++)                  // Освобождение памяти
        delete [ ] m[i];
delete [ ] m;
                . . .
```

14. Строки в языке Си

В языке Си отдельного типа данных «строки символов» нет. Работа со строками реализована путем использования одномерных массивов типа char, т.е. строка символов – это одномерный массив типа char, заканчивающийся нулевым байтом.

Нулевой байт – это байт, каждый бит которого равен нулю, при этом для нулевого байта определена символьная константа ´\0´ (признак окончания строки, или нуль-терминатор). Поэтому, если строка должна содержать k символов, то в описании массива необходимо указать k+1 элемент.

Например, char a[7]; - означает, что строка может содержать шесть символов, а последний байт отведен под нулевой.

Строковая константа – это набор символов, заключенных в двойные кавычки. Например:

```
char S[ ]=“Работа со строками”;
```

В конце строковой константы явно указывать символ ´\0´ не нужно.

При работе со строками удобно пользоваться указателями, например:

```
char *x;
    x = "БГУИР";
    x = (i>0)? "положительное":(i<0)? "отрицательное":"нулевое";
```

Напомним, что для ввода строк обычно используются две стандартные функции:

scanf() - вводит значения для строковых переменных спецификатором ввода **%s** до появления первого символа “пробел” (символ «&» перед ID строковых данных указывать не надо);

gets() - ввод строки с пробелами внутри этой строки завершается нажатием клавиши ENTER.

Обе функции автоматически ставят в конец строки нулевой байт.

Вывод строк производится функциями printf() или puts() до первого нулевого байта (‘\0’):

printf() не переводит курсор после вывода на начало новой строки;

puts() автоматически переводит курсор после вывода строковой информации в начало новой строки.

Например:

```
char Str[30];
printf(“ Введите строку без пробелов : \n”);
scanf(“%s”,Str);
```

или

```
puts(“ Введите строку ”);
gets(Str);
```

Остальные операции над строками выполняются с использованием стандартных библиотечных функций, описание прототипов которых находятся в файле **string.h**. Рассмотрим наиболее часто используемые функции.

Функция int **strlen**(char *S) возвращает длину строки (количество символов в строке), при этом завершающий нулевой байт не учитывается.

Пример:

```
char *S1=”Минск!\0”, S2[]=”БГУИР-Ура!”;
```

```
printf(“ %d, %d .”, strlen(S1), strlen(S2));
```

Результат выполнения данного участка программы:

6 , 10 .

Функция int **strcpy**(char *S1, char *S2) - копирует содержимое строки S2 в строку S1.

Функция **strcat**(char *S1, char *S2) - присоединяет строку S2 к строке S1 и помещает ее в массив, где находилась строка S1, при этом строка S2 не изменяется. Нулевой байт, который завершал строку S1, заменяется первым символом строки S2.

Функция int **strcmp**(char *S1, char *S2) сравнивает строки S1 и S2 и возвращает значение <0, если S1<S2; >0, если S1>S2; =0, если строки равны, т.е. содержат одно и то же число одинаковых символов.

Функции преобразования строки S в число:

- целое: int **atoi**(char *S);
- длинное целое: long **atol**(char *S);
- действительное: double **atof**(char *S);

при ошибке данные функции возвращают значение 0.

Функции преобразования числа V в строку S:

- целое: **itoa**(int V, char *S, int kod);
- длинное целое: **ltoa**(long V, char *S, int kod); 2≤ kod ≤36, для отрицательных чисел kod=10.

Пример функции del_c(), в которой удаляется символ "c" из строки s каждый раз, когда он встречается.

```
void del_c( char s[ ], int c)
{  int i,j;
        for( i=j=0; s[i] != '\0'; i++)
             if( s[i]!=c)  s[j++]=s[i];
        s[j]='\0';
}
```

14.1. Русификация под Visual

При работе в консольном приложении Visual ввод-вывод выполняется в кодировке ASCII, которая является международной только в первой половине кодов (от 0 до 127, см. Приложение 1). Символы национального (русского) алфавита - вторая половина кодов. Для выполнения этого можно использовать функцию **CharToOem**() для преобразования символов из кодировки ANSI в кодировку ASCII и функцию **OemToChar**() для обратного преобразования, находящиеся в библиотеке **windows.h**. Приведем пример их использования.

```
        ...
#include<windows.h>
        char bufRus[256];
        char* Rus(const char*);          // Описание прототипа
void main(void)
 {  int a=2;
        float r=5.5;
         char s[]="Минск !", s1[256];
          printf("\n %s ",Rus(s));
```

```
  printf("\n Vvedi string ");
  gets(s1);
  printf("\n %s ",s1);
  printf(Rus("\n Значение a = %d r = %f\n"), a, r);
 }
char* Rus(const char *text)              // Функция преобразования символов
 {
 CharToOem(text, bufRus);
 return bufRus;
 }
```

15. Функции пользователя

В отличие от других языков программирования высокого уровня в языке «С» нет разделения на подпрограммы-процедуры, подпрограммы-функции, здесь вся программа строится только из функций.

В языке «С» любая подпрограмма – функция, представляющая собой отдельный программный модуль, к которому можно обратиться в любой момент и (в случае необходимости) передавать через параметры некоторые исходные данные и который (в случае необходимости) способен возвращать один или несколько результатов своей работы.

15.1. Декларация функции

Как объект языка Си, функцию необходимо объявить. Объявление функции пользователя, т.е. ее декларация, выполняется в двух формах – в форме описания и в форме определения.

Описание функции заключается в приведении вначале программного файла ее прототипа. Прототип функции сообщает компилятору о том, что далее в тексте программы будет приведено ее полное определение (полный ее текст): в текущем или другом файле исходного текста либо находится в библиотеке.

В стандарте языка используется следующий способ декларации функций:

тип_результата ID*_функции(тип переменной1, …, тип переменной N);*

Заметим, что идентификаторы переменных в круглых скобках прототипа указывать не обязательно, так как компилятор языка их не обрабатывает.

Описание прототипа дает возможность компилятору проверить соответствие типов и количества параметров при фактическом вызове этой функции.

Пример описания функции fun со списком параметров:

float fun(int, float, int, int);

Полное определение функции имеет следующий вид:

тип_результата ID*_функции(список параметров)*
{
код функции
}

Тип результата определяет тип выражения, значение которого возвращается в точку ее вызова при помощи оператора ***return <выражение>***.

Если тип функции не указан, то по умолчанию предполагается тип ***int***.

Список параметров состоит из перечня типов и идентификаторов параметров, разделенных запятыми.

Функция может не иметь параметров, но круглые скобки необходимы в любом случае.

Если функция не возвращает никакого значения, она должна быть описана как функция типа ***void*** (пустая).

В данном случае оператор ***return*** можно не ставить.

В функции может быть несколько операторов ***return***, но может и не быть ни одного. В таких случаях возврат в вызывающую программу происходит после выполнения последнего оператора в функции.

Пример функции, определяющей наименьшее значение из двух целочисленных переменных:

```
int  min (int x, int y)
{
        return (x<y)? x : y;
    }
```

Все функции, возвращающие значение, должны использоваться в правой части выражений языка С, иначе возвращаемый результат будет утерян.

Если у функции отсутствует список параметров, то при декларации такой функции желательно в круглых скобках также указать ключевое слово void. Например, *void main(void)*.

В языке С каждая функция – это отдельный блок программы, вход в который возможен только через вызов данной функции.

Наличие определения функции делает излишним запись ее описания в остатке файла исходного текста.

15.2. Вызов функции

Вызов функции имеет следующий формат:

ID *_функции (список_аргументов);*

где в качестве аргументов можно использовать константы, переменные, выражения (их значения перед вызовом функции будут определены компилятором).

Аргументы списка вызова должны полностью совпадать со списком параметров вызываемой функции по количеству, по порядку следования и по типам соответствующих им параметров.

Связь между функциями осуществляется через аргументы и возвращаемые функциями значения. Ее можно осуществить также через внешние, глобальные переменные.

Функции могут располагаться в исходном файле в любом порядке. А сама исходная программа может размещаться в нескольких файлах.

В языке С аргументы при стандартном вызове функции передаются по значению, т.е. в стеке выделяется место для формальных параметров функции и в это выделенное место при ее вызове заносятся значения фактических аргументов. Затем функция использует и может изменять эти значения в стеке.

При выходе из функции измененные значения теряются. Вызванная функция не может изменить значения переменных, указанных как фактические аргументы при обращении к данной функции.

В случае необходимости функцию можно использовать для изменения передаваемых ей аргументов. В этом случае в качестве аргумента необходимо в вызываемую функцию передавать не значение переменной, а ее адрес. А для обращения к значению аргумента-оригинала использовать операцию «*».

Пример функции, в которой меняются местами значения аргументов x и y:

```
void  zam (int *x, int *y)   {
        int t;
        t = *x;
        *x = *y;
        *y = t;
        }
```

Участок программы с обращением к данной функции:

```
void zam (int*, int*);
void main (void)  {
```

```
        int a=2, b=3;
        …
        printf(" a = %d , b = %d\n", a, b);
        zam (&a, &b);
        printf(" a = %d , b = %d\n", a, b);
        …
}
```

При таком способе передачи аргументов в вызываемую функцию, их значения будут изменены, т.е. на экран монитора будет выведено:

a = 2 , b=3
a = 3 , b=2

15.3. Операция typedef

Любому типу данных, как стандартному, так и определенному пользователем, можно задать новое имя с помощью операции **typedef**:

typedef <тип> <новое_имя>;

Введенный таким образом новый тип используется аналогично стандартным типам, например, введя пользовательские типы:

```
typedef unsigned int UINT;
typedef char M_s[100];
```

декларации идентификаторов введенных типов имеют вид:

```
UINT i, j;        →    две переменные типа unsigned int
M_s str[10];      →    массив из 10 строк по 100 символов
```

15.4. Указатели на функции

В языке С идентификатор функции является константным указателем на начало функции в оперативной памяти и не может быть значением переменной. Но имеется возможность декларировать указатели на функции, с которыми можно обращаться как с переменными (например, можно создать массив, элементами которого будут указатели на функции).

Рассмотрим методику работы с указателями на функции.

1. Как и любой объект языка С, указатель на функции необходимо декларировать. Формат объявления указателя на функции следующий:

тип (*переменная-указатель)(список параметров);

т.е. декларируется указатель, который можно устанавливать на функции, возвращающие результат указанного типа и которые имеют указанный список параметров. Наличие первых круглых скобок обязательно, так как без них – это декларация функции, которая возвращает указатель на результат своей работы указанного типа и имеет указанный список параметров.

Например, объявление вида: **float (* p_f)(char, float);** говорит о том, что декларируется указатель p_f, который можно устанавливать на функции, возвращающие вещественный результат и имеющие два параметра: первый – символьного типа, а второй – вещественного типа.

2. Идентификатор функции является константным указателем, поэтому для того, чтобы установить переменную-указатель на конкретную функцию, достаточно ей присвоить ее идентификатор:

переменная-указатель = ID*_функции;*

Например, имеется функция с прототипом: **float f1(char, float);** тогда операция **p_f = f1**; установит указатель p_1 на данную функцию.

3. Вызов функции после установки на нее указателя выглядит так:

*(*переменная-указатель)(список аргументов);*

или

переменная-указатель (список аргументов);

После таких действий кроме стандартного обращения к функции:

ID*_функции(список аргументов);*

появляется еще два способа вызова функции:

*(*переменная-указатель)(список аргументов);*

или

переменная-указатель (список аргументов);

Последнее справедливо, так как p_f также является адресом начала функции в оперативной памяти.

Для нашего примера к функции f1 можно обратиться следующими способами:

```
f1('z', 1.5);        // Обращение к функции по ID
(* p_f)('z', 1.5);   // Обращение к функции по указателю
p_f('z', 1.5);       // Обращение к функции по ID указателя
```

4. Пусть имеется вторая функция с прототипом: **float f2(char, float);** тогда переустановив указатель p_f на эту функцию: **p_f = f2**; имеем опять три способа ее вызова:

```
f2('z', 1.5);        // по ID функции
(* p_f)('z', 1.5);   // по указателю на функцию
p_f('z', 1.5);       // по ID указателя на функцию
```

Основное назначение указателей на функции – это обеспечение возможности передачи идентификаторов функций в качестве параметров в функцию, которая реализует некоторый вычислительный процесс, используя формальное имя вызываемой функции.

Пример: написать функцию вычисления суммы sum,обозначив слагаемое формальной функцией fun(x), а при вызове функции суммирования передавать через параметр реальное имя функции, в которой запрограммирован явный вид этого слагаемого. Например, пусть требуется вычислить две суммы:

$$S_1 = \sum_{i=1}^{2n} \frac{x}{5} \quad \text{и} \quad S_2 = \sum_{i=1}^{n} \frac{x}{2} \ .$$

Поместим слагаемые этих сумм в пользовательские функции f1 и f2, соответственно.

При этом для более удобной работы воспользуемся операцией typedef, введем пользовательский тип данных: указатель на функции, который можно устанавливать на функции, возвращающие результат, указанного типа, и имеющие указанный список параметров:

typedef тип_результата (переменная-указатель)(параметры);*

Тогда в списке параметров функции суммирования достаточно указывать фактические ID функций данного типа.

Текст программы для решения данной задачи может быть следующим:

```
. . .
// Декларация пользовательского типа: указатель на функции,
// возвращающие float результат и имеющие один float параметр
        typedef float (*p_f)(float);
        float sum(p_f  fun, int, float);              // Декларации прототипов функций
        float f1(float);
        float f2(float);
void main(void) {
        float x, s1, s2;
        int n;
        puts(" Введите кол-во слагаемых n и значение x: ");
        scanf("%d%f", &n, %x);
        s1=sum(f1, 2*n, x);
        s2=sum(f2, n, x);
        printf("\n\t N = %d , X = %f", n, x);
        printf("\n\t Сумма 1 = %f\n\t Сумма 2 = %f", s1, s2);
        }
// Функция вычисления суммы, первый параметр которой – формальное имя
// функции, введенного с помощью typedef типа
        float sum(p_f  fun, int n, float x)  {
                float s=0;
                for(int i=1; i<=n; i++)  s+=fun(x);
                return s;
        }
// Первое слагаемое
        float f1(float r) {
                return (r/5.);
                }
// Второе слагаемое
        float f2(float r)  {
                return (r/2.);
                }
```

В заключение рассмотрим оптимальную передачу в функции одномерных и двухмерных массивов.

***Передача в функцию одномерного массива**:*

```
        void main (void)
        {
                int vect [20];
```

```
        …
        fun(vect);
        …
 }
void   fun( int v [ ])
        {       …       }
```

Передача в функцию двухмерного массива:

```
void main(void)
{
  int mass [ 2 ][ 3 ]={{1,2,3}, {4,5,6}};
        …
   fun (mas);
        …
}
void fun( int m [ ][3])
        {       …       }
```

15.5. Параметры командной строки функции main

Функция main, с которой начинается выполнение программы, может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию main используются два параметра, первый - служит для передачи числа передаваемых строк, второй - для передачи самих строк. Общепринятые (но не обязательные) идентификаторы этих параметров ***argc*** и ***argv***. Параметр argc имеет тип int, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой функции. Параметр argv это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ пробел, то при записи его в командную строку оно должно быть заключено в кавычки.

Функция main может иметь третий параметр, который принято называть ***argp***, и который служит для передачи в функцию main параметров операционной системы (ОС), в которой выполняется программа.

Заголовок функции main с параметрами имеет вид:

```
int main (int argc, char *argv[], char *argp[])
```

Если, например, командная строка программы имеет вид:

```
a:\>cprog working 'C program' 1
```

то аргументы argc, argv, argp представляются в памяти следующим образом:

```
argc  [ 4  ]
argv  [   ]   →     [   ]  →     [a:\cprog.exe\0]
[   ]         →     [working\0]
[   ]         →     [C program\0]
[   ]         →     [1\0]
[NULL]
argp  [   ]   →     [   ]  →     [path = a:\;C:\\0]
```

```
[   ]          →     [lib = d:\lib\0]
[   ]          →     [include = d:\include\0]
[   ]          →     [conspec = c:\command.com\]
[NULL]
```

ОС поддерживает передачу значений для параметров argc, argv, argp, а на пользователе лежит ответственность за передачу и использование фактических аргументов функции main.

Следующий пример представляет программу печати фактических аргументов, передаваемых в функцию main из ОС и параметров ОС.

```
int main ( int argc, char *argv[], char *argp[]) {
  int i = 0;
  printf ("\n Имя программы %s", argv[0]);
  for( i=1; i>=argc; i++)
       printf ("\n Аргумент %d  =  %s", argv[i]);
  printf ("\n Параметры ОС: ");
  while  (*argp) {
       printf ("\n %s",*argp);
       argp++;
       }
  return 0;
}
```

Функция main может вызываться рекурсивно из любой функции.

15.6. Функции с переменным числом параметров

Иногда нельзя перечислить типы и число всех возможных параметров функции. В этих случаях список параметров представляется многоточием (...), которое отключает механизм проверки типов. Наличие многоточия говорит компилятору, что у функции может быть произвольное число параметров неизвестных заранее типов. Многоточие употребляется в двух форматах:

```
void varParFun(param_list, ...);
void varParFun(...);
```

Первый формат предоставляет объявления для части параметров. В этом случае проверка типов для объявленных параметров производится, а для оставшихся фактических аргументов – нет. Запятая после объявления известных параметров необязательна.

Примером вынужденного использования многоточия служит функция printf() стандартной библиотеки С. Ее первый параметр является С-строкой:

```
int printf (const char* ...);
```

Это гарантирует, что при любом вызове printf() ей будет передан первый аргумент типа const char*. Содержание такой строки, называемой *форматной*, определяет, необходимы ли дополнительные аргументы при вызове. При наличии в строке формата метасимволов, начинающихся с символа %, функция ждет присутствия этих аргументов. Например, вызов

```
printf (“hello, world\n”);
```

имеет один строковый параметр. Но

printf (“hello, %s\n”, userName);

имеет два параметра. Символ “%” говорит о наличии второго параметра, а буква s, следующая за ним, определяет его тип - в данном случае символьную строку.

Большинство функций с многоточием в объявлении получают информацию о типах и количестве фактических параметров по значению явно объявленного параметра. Следовательно, первый формат многоточия употребляется чаще.

Следующие объявления неэквивалентны:

```
void f();
void f(...);
```

В первом случае f() объявлена как функция без параметров, во втором – как имеющая нуль или более параметров. Вызовы:

```
f (someValue);
f (cnt, a, b, c);
```

корректны только для второго объявления. Вызов

```
f();
```

применим к любой из двух функций.

Все аргументы, заданные в вызове функции, размещаются в стеке. Количество формальных параметров, объявленных для функции, определяется числом аргументов, которые берутся из стека и присваиваются формальным параметрам. Программист отвечает за правильность выбора дополнительных аргументов из стека и определение числа аргументов, находящихся в стеке.

Для обеспечения удобного способа доступа к аргументам функции с переменным числом параметров имеются три макроопределения (макросы) va_start, va_arg, va_end, находящиеся в заголовочном файле stdarg.h. Эти макросы указывают на то, что функция, разработанная пользователем, имеет некоторое число обязательных аргументов, за которыми следует переменное число необязательных аргументов. Обязательные аргументы доступны через свои ID как при вызове обычной функции. Для извлечения необязательных аргументов используются макросы va_start, va_arg, va_end в следующем порядке.

Макрос va_start предназначен для установки аргумента arg_ptr на начало списка необязательных параметров и имеет вид функции с двумя параметрами:

```
void va_start(arg_ptr,prav_param);
```

Параметр prav_param должен быть последним обязательным параметром вызываемой функции, а указатель arg_prt должен быть объявлен с предопределением в списке переменных типа va_list в виде:

```
va_list arg_ptr;
```

Макрос va_start должен быть использован до первого использования макроса va_arg.

Макрокоманда va_arg обеспечивает доступ к текущему параметру вызываемой функции и тоже имеет вид функции с двумя параметрами

```
type_arg va_arg(arg_ptr,type);
```

Эта макрокоманда извлекает значение типа type по адресу, заданному указателем arg_ptr, увеличивает значение указателя arg_ptr на длину использованного параметра (длина type) и таким образом параметр arg_ptr будет указывать на следующий параметр вызываемой функции. Макрокоманда va_arg используется столько раз, сколько необходимо для извлечения всех параметров вызываемой функции.

Макрос va_end используется по окончании обработки всех параметров функции и устанавливает указатель списка необязательных параметров на ноль (NULL).

Рассмотрим применение этих макросов для обработки параметров функции вычисляющей среднее значение произвольной последовательности целых чисел. Поскольку функция имеет переменное число параметров, будем считать концом списка значение равное -1. Поскольку в списке должен быть хотя бы один элемент, у функции будет один обязательный параметр.

Пример:

```
#include <stdarg.h>
  int main(void)   {
        int n;
        int sred_znach(int,...);
        n=sred_znach(2,3,4,-1);          // Вызов с четырьмя параметрами
        printf("n=%d",n);
        n=sred_znach(5,6,7,8,9,-1);      // Вызов с шестью параметрами
        printf("n=%d",n);
    return (0);
  }

  int sred_znach(int x,...)
  {
    int i=0,  j=0,  sum=0;
    va_list uk_arg;
    va_start(uk_arg,x);
// Установка указателя uk_arg на первый необязятельный параметр
    if (x != -1) sum=x;              // Проверка на пустоту списка
    else return (0);
    j++;
    while ( (i = va_arg(uk_arg,int)) != -1)
// Выборка очередного параметра и проверка на конец списка
    {
      sum+=i;
      j++;
    }
    va_end(uk_arg);                  // Закрытие списка параметров
    return (sum / j);
  }
```

16. Классы памяти и области действия объектов

Напомним, что все объекты перед их использованием должны быть декларированы. Одним из атрибутов в декларации объекта является - класс памяти.

Класс памяти переменной определяет время ее существования (время жизни) и область видимости (действия).

Имеется три основных места, где объявляются переменные: внутри функции, при определении параметров функции и вне функции. Эти переменные называются соответственно локальными (внутренними) переменными, формальными параметрами и глобальными (внешними) переменными.

16.1. Классы памяти объектов в языке С:

- динамическая память, которая выделяется при вызове функции и освобождается при выходе из нее (атрибуты: auto - автоматический; register - регистровый);

- статическая память, которая распределяется на этапе трансляции и заполняется по умолчанию нулями (атрибуты: внешний – extern, статический – static).

16.2. Автоматические переменные

Переменные, декларированные внутри функций, являются внутренними и называются *локальными* переменными. Никакая другая функция не имеет прямого доступа к ним. Такие объекты существуют временно на этапе активности функции.

Каждая локальная переменная существует только в блоке кода, в котором она объявлена, и уничтожается при выходе из него. Эти переменные называют автоматическими и располагаются в стековой области памяти.

По умолчанию, локальные объекты, объявленные в теле функции, имеют атрибут класса памяти auto.

Принадлежность к этому классу можно также подчеркнуть явно с помощью ключевого слова auto. Например:

```
void main(void)      {
  auto int max, lin;
  ...              }
```

так поступают, если хотят показать, что определение переменной не нужно искать вне функции.

Регистровая память (атрибут **register**) - объекты целого типа и символы рекомендуется разместить не в оперативной памяти, а в регистрах общего назначения (процессора), а при нехватке регистров - в стековой памяти (размер объекта не должен превышать разрядности регистра), для других типов компилятор может использовать другие способы размещения, а может просто проигнорировать данную рекомендацию.

Регистровая память позволяет увеличить быстродействие программы, но к размещаемым в ней объектам в языке Си (но не С++) не применима операция адресации &, а также это неприменимо для массивов, структур, объединений и переменных с плавающей точкой.

16.3. Внешние переменные

Объекты, размещаемые в статической памяти, декларируются с атрибутом **static** и могут иметь любой атрибут области действия. Глобальные объекты всегда являются статическими. Атрибут static, использованный при описании глобального объекта, предписывает ограничение его области применимости только в пределах остатка текущего файла. Значения локальных статических объектов сохраняются при повторном вызове функции. Таким образом, в языке

С ключевое слово static имеет разный смысл для локальных и глобальных объектов.

Переменная, описанная вне функции, является внешней переменной. Их еще называют глобальными переменными.

Так как внешние переменные доступны всюду, их можно использовать вместо списка аргументов для передачи значений между функциями.

Внешние переменные существуют постоянно. Они сохраняют свои значения и после того, как функции, присвоившие им эти значения, завершат свою работу.

При отсутствии явной инициализации для внешних и статических переменных гарантируется их обнуление, автоматические и регистровые переменные имеют неопределенные начальные значения («мусор»).

Внешняя переменная должна быть определена вне всех функций. При этом ей выделяется фактическое место в памяти. Такая переменная должна быть описана в каждой функции, которая собирается ее использовать. Это можно сделать либо явным описанием extern, либо по контексту.

Чтобы функция могла использовать внешнюю переменную, ей нужно сообщить идентификатор этой переменной. Один из способов - включить в функцию описание extern.

В определенных ситуациях описание extern может быть опущено: если внешнее определение переменной находится в том же файле, раньше ее использования в некоторой конкретной функции, то не обязательно включать описание extern для этой переменной в саму функцию. Фактически, обычная практика заключается в помещении определений всех переменных в начале исходного файла и последующем опусканием всех описаний extern.

Включение ключевого слова extern позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле.

Важно различать описание внешней переменной и ее определение. Описание указывает свойство переменной, ее размер, тип и т. д.; определение же вызывает еще и отведение памяти. Например, если вне какой-либо функции появляются строчки:

```
int sp;
double val[20];
```

то они определяют внешние переменные sp и val, вызывают отведение памяти для них и служат в качестве описания для остальной части этого исходного файла. В то же время строчки:

```
extern int sp;
extern double val [];
```

описывают в остальной части этого исходного файла переменную sp как int, а vol как массив типа double, но не создают переменных и не отводят им места в памяти.

Во всех файлах, составляющих исходную программу, должно содержаться только одно определение внешней переменной. Другие файлы могут содержать описание extern для доступа к ней.

Любая инициализация внешней переменной проводится только в декларации. В декларации должны указываться размеры массивов, а в описании extern этого можно не делать.

Например, в файле 1:

```
int sp = 0;
double val [20];
...
```

в файле 2:

```
extern int sp;
extern double val [];
                ...
```

В Си есть возможность с помощью #include иметь во всей программе только одну копию описаний extern и вставлять ее в каждый файл во время его компиляции.

Если переменная с таким же идентификатором как внешняя декларирована в функции без указания extern, то тем самым она становится внутренней в данной функции.

Не стоит злоупотреблять внешними переменными. Такой стиль программирования приводит к программам, связи данных внутри которых не вполне очевидны. Переменные при этом могут изменяться неожиданным образом. Программу становится трудно модифицировать.

Файл 1:

```
int x, y;
char str[ ] = “Rezult = ”;

void main(void)
{
 . . .
}

void fun1(void)
{
 y = 15;
 cout << str << y;
}
```

Файл 2:

```
extern int x, y;
extern char str[ ];
     int r, q;
void fun2(void)
{
  x = y / 5;
  cout << str << x;
}

void fun3(void)
{
 int z= x + y;
 cout << str << z;
}
```

16.4. Область действия переменных

В языке С нет ключевого слова, указывающего область действия объекта. Область действия определяется местоположением декларации объекта в файле исходного текста программы (любой объект полностью описывается в одном операторе).

Напомним общую структуру исходного текста программ на языке С:

```
<директивы препроцессора>
<описание глобальных объектов>
    <заголовок функции>
    {
    <описание локальных объектов>
    <список инструкций>
```

```
}
```

Файл исходного текста может включать любое количество определений функций и/или глобальных данных.

Параметры функции являются локальными объектами и должны отличаться по идентификаторам от используемых в коде функции глобальных объектов. Локальные объекты, описанные в коде функции, имеют приоритет перед объектами, описанными вне функции, например:

```
        . . .
    int n;              // глобальное n
void main (void)  {
    int i;
     ...
    f1(i);
     ...
    f2(n);              // локальное n
    }
 f1(int i)  {
    ...
    i=n;                // глобальное n
    ...
    }
  f2(int n)  {
    int i;
     ...
     i=n;               // локальное n
     ...
    }
```

В любом месте файла исходного текста можно ссылаться на глобальные объекты, определенные ниже в остатке текущего файла или в других файлах. Для этого необходимо описать тип объекта и добавить к описанию ключевое слово **extern**. Описания функций подразумевают атрибут extern по умолчанию. Разрешается опускать длину внешних одномерных массивов, но операция sizeof к таким массивам становится бессмысленной.

Следует учитывать, что любая декларация объекта действует только на остаток файла исходного текста.

В C++ допускается в разных блоках программы использовать один и тот же идентификатор объекта, тогда более внутреннее объявление объекта скрывает доступ к объекту на более высоком уровне декларации.

```
void main(void)   {
    int i = 3;
    cout << "\n Block 1 - " << i ;
    {
        float i = 2.5;
        cout << "\n Block 2 - " << i ;
        {
            char i = 'a';
            cout << "\n Block 3 - " << i ;
```

```
            }
        }
        cout << "\n New Block 1  - " <<  ++i ;
        getch();
        }
```

В результате выполнения этой программы, на экране получим:

```
Block 1 - 3
Block 2 - 2.5
Block 3 - a
New Block 1  - 4
```

17. Структуры, объединения, перечисления

17.1. Структуры

Структура это составной объект языка Си, представляющий собой совокупность логически связанных данных различного типа, объединенных в группу под одним идентификатором (ID). Данные, входящие в эту группу называют полями.

Термин «структура» в языке Си соответствует двум разным по смыслу понятиям:

- структура – это обозначение участка оперативной памяти, где располагаются конкретные значения данных. В дальнейшем – это структурная переменная, поля которой располагаются в смежных областях ОП.
- структура – это правила формирования структурной переменной, которыми руководствуется компилятор при выделении ей места в ОП и организации доступа к ее полям.

Определение объектов типа структуры производится за два шага:

- декларация структурного типа данных, не приводящая к выделению участка памяти;
- определение структурных переменных с выделением для них памяти.

17.2. Декларация структурного типа данных

Структурный тип данных задается в виде шаблона, общий формат описания которого следующий:

```
struct  ID структурного типа {
                описание полей
            };
```

Атрибут «ID структурного типа», т.е. ее идентификатор является необязательным и может отсутствовать.

Описание полей производится обычным способом: указываются типы и идентификаторы.

Пример определения структурного типа. Необходимо создать шаблон, описывающий информацию о студенте: номер группы, ФИО и средний балл. Один из возможных вариантов:

```
struct Stud_type  {
    char Number[10];
    char Fio[40];
```

```
    double S_b;
    };
```

Поля одного типа при описании можно объединять в одну группу:

```
struct Stud_type  {
    char Number[10], Fio[40];
    double S_b;
    };
```

Интерпретация объекта типа Stud_type:

Number	Fio	S_b
10	40	8

длина в байтах

Структурный тип данных удобно применять для групповой обработки логически связанных объектов. Параметрами таких операций являются адрес и размер структуры.

Примеры групповых операций:

- захват и освобождение памяти для объекта;
- запись и чтение данных, хранящихся на внешних носителях как физические и/или логические записи с известной структурой (при работе с файлами).

Т.к. одним из параметров групповой обработки структурных объектов является размер, не рекомендуется декларировать поле структуры указателем на объект переменной размерности, т.к. в данном случае многие операции со структурными данными будут не корректны, например

```
    struct Stud_type  {
        char *Number, *fio;
        double S_b;
        };
```

в данном случае, вводя строки Number и fio различной длины, размеры объектов будут также различны.

17.3. Создание структурных переменных

Как уже отмечалось само описание структуры не приводит к выделению под нее места в ОП. Теперь необходимо создать нужное количество переменных с приведенной структурой и сделать это можно двумя способами.

Способ 1. В любом месте программы для декларации структурных переменных, массивов, функций и т.д., используется объявленный в шаблоне структурный тип, например:

```
struct Stud_type  student;      - структурная переменная;
Stud_type  Stud[100];           - массив структур
Stud_type  *p_stud;             - указатель на структуру
Stud_type* Fun(Stud_type);      - прототип функции с параметром структурного типа, возвращающей указатель на объект структурного типа.
```

Способ 2: в шаблоне структуры между закрывающейся фигурной скобкой и символом «;» указывают через запятые идентификаторы структурных данных.

Для нашего примера, используя, можно записать:

```
struct Stud_type  {
    char Number[10], Fio[40];
```

```
    double S_b;
    } student, Stud[100], *p_stud;
```

Если дальше в программе не понадобится вводить новые данные объявленного структурного типа, «Stud_type» можно не указывать.

При декларации структурных переменных возможна их одновременная инициализация

Например:

```
  struct Stud_type      {
    char Number[10], Fio[40];
    double S_b;
    } student = {“123456”, “Иванов И.И.”, 6.53 };
```

или:

```
    Stud_Type  stud1 = {“123456”, “Иванов И.И.” };
```

Отметим, что если список инициализации будет короче, то оставшиеся поля структурной переменной будут заполнены нулями.

Некоторые особенности:

1) поля структуры, как правило, имеют разный тип кроме функций, файлов и самой этой структуры;

2) поля не могут иметь атрибут, указывающий «класс памяти», данный атрибут можно определить только для всей структуры;

3) идентификаторы (ID), как самой структуры, так и ее полей могут совпадать с ID других объектов программы, т.к. шаблон структуры обладает собственным пространством имен;

4) при наличии в программе функций пользователя шаблон структуры рекомендуется поместить глобально перед определениями всех функций и в этом случае он будет доступен всем функциям.

Элементы структур в общем случае размещаются в памяти последовательно с учетом выравнивания начальных адресов.

Выравнивание - установка значения адреса, кратного некоторой величине, определяемой особенностями адресации данных на аппаратном уровне. Часто выравнивание не обязательно, но при этом скорость обращению к объекту может снижаться.

Обращение к полям структур производится при помощи составных имен, которые образуются двумя способами:

а) использованием операции принадлежности (**.**) в виде:

```
    ID_структуры . ID_поля
```

или

```
    (*указатель_структуры).ID_поля
```

б) при помощи операции косвенной адресации (**->**) в виде:

```
    указатель_структуры -> ID_поля
```

или

```
    (&ID_структуры) ->ID_поля
```

Примеры обращения для объявленного ранее шаблона:

1). Stud_Type s1, s2;

Обратиться к полям элемента объявленной структуры s1:

s1. Number, s1. Fio, s1. S_b;

2). Stud_Type *s1, *s2;

Обратиться к полям элемента объявленной структуры s1:

s1 -> Number, s1 -> Fio, s1 -> S_b;

Так как структурная переменная является сложным объектом, операции, например, присваивания для таких переменных необходимо производить при помощи стандартных функций.

Пусть необходимо содержимое структурной переменной s2 присвоить s1:

```
Stud_Type  s1, s2;
```

Идентификатор структурной переменной является константным указателем на начало размещения данного объекта в ОП, поэтому, в выражении s1 = s2; допущена ошибка, т.к. в левой части - константа. Правильное присваивание:

```
memcpy(&s1,&s2,sizeof(Stud_Type));
```

или

```
Stud_Type  *s1, s2;
    s1 = s2;
```

Участок программы, иллюстрирующий передачу структурных данных в функцию:

```
struct Spisok {
        char Fio[20];
        float S_Bal;
        };
//  Описание прототипов
    void Out(Spisok );
    void Vvod(int, Spisok *);
void main(void)  {
    Spisok Stud[50], *sved;
        . . .
        for(i=0;i<N;i++)
            Vvod(i, &Stud[i]);
        puts("\n Spisok Students");
        for(i=0;i<N;i++)
            Out(Stud[i]);
        . . .
        }

void Out(Spisok dan)   {
    printf("\n %20s  %4.2f",dan.Fio, dan.S_Bal);
    }

 void Vvod(int nom, Spisok *sved)  {
    printf("\n Введи сведения %d : ",nom+1);
    fflush(stdin);
    puts("\n ФИО      - ");
    gets(sved->Fio);
    puts("\n Средний балл   - ");
    scanf("%f", &sved->S_Bal);
    }
```

17.4. Вложенные структуры

Структуры могут быть вложенными, т.е. поле структуры может связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основному шаблону.

Например, в структуре person, содержащей сведения - Ф.И.О., дата рождения, сделать дату рождения внутренней структурой date по отношению к структуре person. Тогда шаблон такой конструкции будет выглядеть так:

```
struct date   {
        int day, month, year;
        };
 struct person   {
        char fio[40];
        struct date f1;
        } ;
```

Объявляем переменную и указатель на переменные такой структуры:

```
        struct person  a, *p;
```

Инициализируем указатель *p* адресом переменной *a*:

```
                p = &a;
```

Тогда, обращение к полям структурной переменной *a* будет выглядеть следующим образом:

```
a .fio          a.f1.day          a.f1.month          a.f1.year
```

или

```
p->fio          p->f1.day         p->f1.month         p->f1.year
```

Можно в качестве связи с вложенной структурой использовать указатель на нее:

```
struct date  {
        int day, month, year;
        };
 struct person  {
        char fio[40];
        struct date *f1;
        } ;
```

Тогда обращение к полям будет следующим:

```
a .fio          a.f1->day         a.f1->month         a.f1->year
```

или

```
p->fio          p->f1->day        p->f1->month        p->f1->year
```

Использование средства **typedef** упрощает определение структурных переменных, так как отпадает необходимость при их декларации указывать ключевое слово stuct. Например:

```
typedef struct person {
        char fio[40];
        int day, month, year;
        } W ;
```

здесь *W* - созданный пользователем тип данных - «структура с указанными полями» и для нашего примера:

W t1, t2; - декларация двух переменных типа *W*, а это значит, что можно на такие переменные устанавливать указатели и использовать косвенную адресацию.

17.5. Массивы структур

Структурный тип "struct ID_структуры" как правило используют для декларации массивов, элементами которых являются структурные переменные. Это позволяет создавать программы, оперирующие с "примитивными базами данных". Например:

```
struct person spisok[100];              // spisok - массив структур
```

Или можно записать так:

```
struct person {
        char fio[40];
        int day, month, year;
        }  spisok[100];
```

В данном случае обращение к полю, например, day *i*-той записи может быть выполнено одним из следующих способов:

spisok[i]**.**day=22; *(spisok+i)**.**day=22; (spisok+i)**->**day=22;

17.6. Размещение структурных переменных в памяти

При анализе размеров структурных переменных иногда число байт, выделенных компилятором под структурную переменную оказывается больше, чем сумма байт ее полей. Это связано с тем, что компилятор выделяет участок ОП для структурных переменных с учетом выравнивания граней, добавляя между полями пустые байты по следующим правилам:

- структурные переменные, являющиеся элементами массива начинаются на границе слова, т.е. с четного адреса;
- любое поле структурной переменной начинается на границе слова, т.е. с четного адреса и имеет четное смещение по отношению к началу переменной;
- при необходимости в конец переменной добавляется пустой байт, чтобы общее число байтов было четное.

17. 7. Объединения

Объединение - поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента.

Объединенный тип данных декларируется подобно структурному типу:

```
   union ID_объединения {
                описание полей
                        };
```

Пример описания объединенного типа:

```
    union word {
            int nom;
            char str[20];
               };
```

Пример объявления объектов объединенного типа:

```
union word *p_w,  mas_w[100];
```

Объединения применяют для экономии памяти в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных.

Например, поток сообщений по каналу связи пусть содержит сообщения трех видов:

```
struct m1 {
        char code;
        float data[100];    };
struct m2 {
        char code;
        int mode;    };
struct m3 {
        char code, note[80];    };
```

Элемент code - признак вида сообщения. Удобно описать буфер для хранения сообщений в виде

```
struct m123 {
        char code;
        union {
                float data[100];
                int mode;
                char note[80];    };
       };
```

Практически все вышесказанное для структур имеет место и для объединений.

Декларация данных типа union, создание переменных этого типа и обращение к полям объединений производится аналогично структурам.

Пример использования переменных типа union:

```
        . . .
typedef union q {
                int a;
                float b;
                char s[5];
                } W;
void main(void) {
        W  s, *p = &s;
        s.a = 4;
        printf("\n Integer a = %d, Sizeof(s.a) = %d", s.a, sizeof(s.a));
        p -> b = 1.5;
        printf("\n Float b = %f, Sizeof(s.b) = %d", s.b, sizeof(s.b));
        strcpy(p->s, "Minsk");
        printf("\n String a = %s, Sizeof(s.s) = %d", s.s, sizeof(s.s));
        printf("\n Sizeof(s) = %d", sizeof(s));
        getch();
        }
```

Результат работы программы:

```
Integer a = 4, Sizeof(s.a) = 2
Float b = 1.500000, Sizeof(s.b) = 4
String a = Minsk, Sizeof(s.s) = 5
```

Sizeof(s) = 5

17.8. Перечисления

Перечисления - средство создания типа данных посредством задания ограниченного множества значений.

Определение перечислимого типа данных имеет вид

enum ID_перечислимого_типа {
список_значений
};

Значения данных перечислимого типа указываются идентификаторами. Например:

```
enum marks {
    zero, two, three, four, five
        };
```

Транслятор последовательно присваивает идентификаторам списка значений целочисленные величины 0,1,..., . При необходимости можно явно задать значение идентификатора, тогда очередные элементы списка будут получать последующие возрастающие значения. Например:

```
enum level {
        low=100, medium=500, high=1000, limit
        };
```

Примеры объявления переменных перечислимого типа:

```
enum marks Est;
enum level state;
```

Переменная типа marks может принимать только значения из множества {zero, two, three, four, five}.

Основные операции с данными перечислимого типа:

- присваивание переменных и констант одного типа;
- сравнение для выявления равенства либо неравенства.

Практическое назначение перечисления - определение множества различающихся символических констант целого типа.

Пример использования переменных перечислимого типа:

```
    . . .
typedef enum {
    mo=1, tu, we, th, fr, sa, su
    } days;
void main(void) {
    days w_day;                 // Переменная перечислимого типа
    int cur_day, _end, _start;
// Текущий день недели, начало и конец недели, соответственно
    clrscr();
    puts(“ Введите день недели (от 1 до 7) : ”);
    scanf(“%d”, &cur_day);
    w_day = su;
    _start = mo;
    _end = w_day - cur_day;
    printf(“\n Понедельник - %d день недели, \
```

```
            сейчас %d - й день и \n\
            до конца недели %d дней (дня)", _start, cur_day, _end );
        getch();
        }
```

Результат работы программы:

```
Введите день недели (от 1 до 7) : 5
Понедельник - 1 день недели, сейчас 5 - й день и
до конца недели 2 дней (дня)
```

18. Файлы в языке С

Файл – это набор данных, размещенный на внешнем носителе и рассматриваемый в процессе обработки и пересылке как единое целое. В файлах размещаются данные, предназначенные для длительного хранения.

Различают два вида файлов: текстовые и бинарные. Текстовые файлы представляют собой последовательность ASCII символов и могут быть просмотрены и отредактированы с помощью любого текстового редактора.

Бинарные (двоичные) файлы представляют собой последовательность данных, структура которых определяется программно.

В языке Си имеется большой набор функций для работы с файлами, большинство которых находятся в библиотеках **stdio.h** и **io.h**.

18.1. Открытие файла

Каждому файлу присваивается внутреннее логическое имя, используемое в дальнейшем при обращении к нему. Логическое имя (идентификатор файла) - это указатель на файл, т.е. на область памяти, где содержится вся необходимая информация о файле. Формат объявления указателя на файл следующий:

FILE *указатель на файл;

FILE - идентификатор структурного типа, описанный в стандартной библиотеке stdio.h и содержащий следующую информацию:

type struct {

short level;	- число оставшихся в буфере непрочитанных байт; обычный размер буфера - 512 байт; как только level=0, в буфер из файла читается следующий блок данных;
unsigned flags;	- флаг статуса файла - чтение, запись, дополнение;
char fd;	- дескриптор файла, т.е. число, определяющее его номер;
unsigned char hold;	- непереданный символ, т.е. ungetc-символ;
short bsize;	- размер внутреннего промежуточного буфера;
unsigned char buffer;	- значение указателя для доступа внутри буфера, т.е. задает начало буфера, начало строки или текущее значение указателя внутри буфера в зависимости от режима буферизации;
unsigned char *curp;	- текущее значение указателя для доступа внутри буфера, т.е. задает текущую позицию в буфере для обмена с программой;

unsigned istemp; - флаг временного файла;
short token; - флаг при работе с файлом;
} **FILE**;

Прежде, чем начать работать с файлом, т.е. получить возможность чтения или записи информации в файл, его нужно открыть для доступа. Для этого обычно используется функция

FILE* **fopen**(char * ID_файла, char *режим);

она берет внешнее представление - физическое имя файла на носителе (дискета, винчестер) и ставит ему в соответствие логическое имя.

Физическое имя, т.е. имя файла и путь к нему задается первым параметром - строкой, например, “a:Mas_dat.dat” - файл с именем Mas_dat.dat, находящийся на дискете, “d:\\work\\Sved.txt” - файл с именем Sved.txt, находящийся на винчестере, в каталоге work.

Внимание, обратный слеш (\), как специальный символ в строке записывается дважды!

При успешном открытии функция fopen() возвращает указатель на файл (в дальнейшем - указатель файла). При ошибке возвращается **NULL**. Данная ситуация обычно возникает, когда неверно указывается путь к открываемому файлу. Например, если в дисплейном классе нашего университета, указать путь, запрещенный для записи (обычно, разрешенным является d:\work\).

Второй параметр - строка, в которой задается режим доступа к файлу:

w - файл открывается для записи; если файла с заданным именем нет, то он будет создан; если такой файл существует, то перед открытием прежняя информация уничтожается;

r - файл открывается только для чтения; если такого файла нет, то возникает ошибка;

a - файл открывается для добавления в его конец новой информации;

r+ - файл открывается для редактирования данных - возможны и запись, и чтение информации;

w+ - то же, что и для r+;

a+ - то же, что и для a, только запись можно выполнять в любое место файла; доступно и чтение файла;

t - файл открывается в текстовом режиме; используются поля r, w, a, r+, w+, a+;

b - файл открывается в двоичном режиме.

Текстовый режим отличается от двоичного тем, что при открытии файла как текстового пара символов «перевод строки», «возврат каретки» заменяется на один символ: «перевод строки» для всех функций записи данных в файл, а для всех функций вывода символ «перевод строки» теперь заменяется на два символа: «перевод строки», «возврат каретки».

По умолчанию файл открывается в текстовом режиме.

Пример: FILE *f; - объявляется указатель на файл f;

f = fopen ("d:\\work\\Dat_sp.cpp", "w"); - открывается для записи файл с логическим именем f, имеющим физическое имя Dat_sp.cpp, находящийся на диске d, в каталоге work.

или более кратко:

FILE *f = fopen ("d:\\work\\Dat_sp.cpp", "w");

18.2. Закрытие файла

После работы с файлом доступ к нему необходимо закрыть. Это выполняет функция int **fclose**(указатель файла). Например, из предыдущего примера файл закрывается так: fclose (f);

Для закрытия нескольких файлов введена функция, объявленная следующим образом: void **fcloseall**(void);

Если требуется изменить режим доступа к файлу, то для этого сначала необходимо закрыть данный файл, а затем вновь его открыть, но с другими правами доступа. Для этого используют стандартную функцию:

FILE* **freopen** (char *ID_файла, char *режим, FILE *указатель_файла);

Эта функция сначала закрывает файл, объявленный «указателем_файла» (как это делает функция fopen), а затем открывает файл с «именем_файла» и правами доступа «режим».

В языке С имеется возможность работы с временными файлами, которые нужны только в процессе работы программы и которые надо удалить после выполнения части вычислений. В этом случае используется функция:

FILE* **tmpfile** (void);

которая создает на диске временный файл с правами доступа «w+b», после завершения работы программы или после закрытия временного файла он автоматически удаляется.

18.3. Запись - чтение информации

Все действия по чтению-записи данных в файл можно разделить на три группы:

- операции посимвольного ввода-вывода;
- операции построчного ввода-вывода;
- операции ввода-вывода по блокам.

Рассмотрим основные функции, применяемые в каждой из указанных трех групп операций.

Посимвольный ввод-вывод

В функциях посимвольного ввода-вывода происходит прием одного символа из файла или передача одного символа в файл:

int **fgetc**(FILE *f) - считывает и возвращает символ из файла f;
int **fputc**(int ch, FILE *f) - записывает в файл f код ch символа.

Построчный ввод-вывод

В функциях построчного ввода-вывода происходит перенос из файла, или в файл строк символов:

int **fgets** (char *S, int m, FILE *f) - чтение из файла f в строку S m байт;
int **fputs** (char *S, FILE *f) - запись в файл f строки S до тех пор, пока не встретится '\0', который в файл не переносится и на символ '\n' не заменяется.

Блоковый ввод-вывод

В функциях блокового ввода-вывода работа происходит с целыми блоками информации:

int **fread** (void *p, int size, int n, FILE *f) - считывает n блоков по size байт каждый из файла f в область памяти с указателем p (необходимо заранее отвести память под считываемый блок);

int **fwrite** (void *p, int size, int n, FILE *f) - записывает n блоков по size байт каждый из области памяти с указателем p в файл f.

Форматированный ввод-вывод производится функциями:

int **fscanf** (FILE *f, char *формат, список адресов объектов) - считывает из файла f информацию для объектов в соответствии с указанными форматами;

int **fprintf** (FILE *f, char *формат, список объектов) - записывает в файл f объекты, указанные в списке в соответствии с форматами.

Данные функции аналогичны функциям **scanf**() и **printf**(), рассмотренным раньше, только добавлен параметр – указатель на файл.

18.4. Текстовые файлы

Для работы с текстовыми файлами удобнее всего пользоваться функциями fprintf(), fscanf(), fgets() и fputs().

Создание текстовых результирующих файлов обычно необходимо для оформления отчетов по лабораторным и курсовым работам.

Рассмотрим пример создания текстового файла:

```
#include<stdio.h>
    void main(void)  {
          FILE *f1;
          int a=2, b=3;
          If(!(f1=fopen(“d:\\work\\f_rez.txt”,”w+t”)))
          {
                puts(“Файл не создан!”);
                return;
          }
          fprintf(f1,”  Файл результатов \n”);
          fprintf(f1,” %d плюс %d = %d\n”,a,b,a+b);
      fclose(f1);
     }
```

Просмотрев содержимое файла, можно убедиться, что данные в нем располагаются точно как на экране при использовании функции printf().

18.5. Бинарные файлы

Бинарные (двоичные) файлы обычно используются для организации баз данных, состоящих, как правило, из объектов структурного типа. При чтении-записи бинарных файлов удобнее всего пользоваться функциями, выполняемыми блоковый ввод-вывод **fread**() и **fwrite**().

Рассмотрим наиболее распространенные функции с помощью которых можно организовать работу с файлами:

int **fileno**(FILE *f) – возвращает значение дескриптора файла f - fd (число, определяющее номер файла);

long **filelength**(int fd) – возвращает длину файла, имеющего номер (дескриптор) fd в байтах;

int **chsize**(int fd, long pos)	– выполняет изменение размера файла, имеющего номер fd, признак конца файла устанавливается после байта с номером pos;
int **fseek**(FILE *f, long size, int kod)	– выполняет смещение указателя файла f на size байт в направлении признака kod: 0 - от начала файла; 1 - от текущей позиции указателя; 2 - от конца файла;
long **ftell**(FILE *f)	– возвращает значение указателя на текущую позицию файла (-1 – ошибка);
int **feof**(FILE *f)	– возвращает ненулевое значение при правильной записи признака конца файла;
int **fgetpos**(FILE *f, long *pos)	– определяет значение текущей позиции pos файла f, возвращает 0 при успешном завершении.

Пример программы работы с файлом структур:

```
. . .
struct Sved {
        char Fam[30];
        float S_Bal;
        } zap,zapt;
char Spis[]="c:\\bc31\\work\\Sp.dat";
FILE *F_zap;
FILE* Open_file(char *, char *);
void main (void) {
int i, j, kodR, size = sizeof(Sved);
while(1) {
puts("Создание - 1\nПросмотр - 2\nДобавление - 3\nВыход - 0");
switch(kodR = getch())
{
case '1': case '3':
        if(kodR==1) F_zap = Open_file (Spis,"w+");
                else F_zap = Open_file (Spis,"a+");
        while(2) {
                cout << "\n Fam "; cin >> zap.Fam;
                if((zap.Fam[0])=='0')  break;
                cout << "\n Средний балл: ";
                cin >> zap.S_Bal;
        fwrite(&zap,1,size,F_zap);
        }
  fclose(F_zap);
 break;
case '2':   F_zap = Open_file (Spis,"r+"); int nom=1;
while(2)  {
        if(!fread(&zap,size, 1, F_zap)) break;
        printf(" %2d: %20s %5.2f\n", nom++, zap.Fam, zap.S_Bal);
        }
    fclose(F_zap);
```

```
        break;
        case '0':   return;             // exit(0);
        }       //      Конец While(1)
    }           //      Конец Switc
}               //      Конец программы

FILE* Open_file(char *file, char *kod)
        {
        FILE *f;
         if(!(f = fopen(file, kod)))
                {
                puts("Файл не создан!");
                getch();
                exit(1);
                }
        else return f;
        }
```

Список рекомендуемой литературы

1. Бусько В.Л., Корбит А.Г. и др. Программирование. Лабораторный практикум для студентов 1-2-го курсов всех специальностей БГУИР всех форм обучения. Часть 2. Основы программирования на алгоритмическом языке С.

2. Аксенкин М.А., Целобенок О.Н. Язык С. - Мн.: Універсітэцкае, 1995. – 302 с.

3. Березин Б.И., Березин С.Б. Начальный курс С и С++. – М.: Диалог-МРТИ,1999. - 288 с.

4. Берри В., Микинз Б. Язык СИ: введение для программистов. - М.: Финансы и статистика,1988.

5. Больски М.Н. Язык программирования СИ. Справочник. - М.: Радио и связь, 1988.

6. Демидович Е.М. Основы алгоритмизации и программирования. Язык СИ. - Мн.: Бестпринт, 2001. – 440 с.

7. Касаткин А.И., Вольвачев А.Н. Профессиональное программирование на языке Си: От Turbo–C к Borland C++: Справочное пособие – Мн.: Вышэйшая школа,1992. - 240 с.

8. Касаткин А.Н. Профессиональное программирование на языке СИ. Управление ресурсами. Справочное пособие. Мн.: Высш. школа.1992

9. Керниган Б., Ритчи Д. Язык программирования Си. - М.: Финансы и статистика,1992. - 271 с.

10. Климова Л.И. С++. Практическое программирование. - М.: Кудиц-Образ, 2001. – 587 с.

11. Котлинская Г.П., Галиновский О.И. Программирование на языке СИ. - Мн.: Выш.шк., 1991. – 155 с.

12. Подбельский В.В., Фомин С.С. Программирование на языке Си. М.: Финансы и статистика. 2001.

13. Романовская Л.М., Русс Т.В., Свитковский С.Г. Программирование в среде СИ для ПЭВМ ЕС. - М.: Финансы и статистика, 1992.

14. Страуструп Б. Язык программирования С++. 2-е изд.: В 2 т. Киев: Диа-Софт,1993.

15. Тимофеев В.В. Программирование в среде С++ Builder 5. - М.: БИНОМ, 2000.

16. Уингер Р. Язык Турбо СИ. - М.: Мир, 1991.

17. Уэйт М., Прама С., Мартин Д. Язык СИ. Руководство для начинающих. - М.: Мир, 1988.

18. Фьюэр А. Задачи по языку СИ. - М.: Финансы и статистика, 1985.

19. Хэнкок Л., Кригер М. Введение в программирование на языке СИ. - М.: Радио и связь,1986.

20. Шилд Г. Программирование на Borland C++. - Мн.: ПОПУРРИ, 1999. – 800 с.

21. Юлин В.А., Булатова И.Р. Приглашение к СИ. - Мн.: Высш.шк., 1990.

Список используемой литературы

1. Синицын А.К. Конспект лекций по курсу «Программирование» для студентов 1-2-го курсов радиотехнических специальностей. - Мн.: БГУИР, 2001. - 75с.: ил. 10.

Стандартная часть таблицы символов (ASCII)

КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С
0		16	►	32		48	0	64	@	80	P	96	`	112	p
1	☺	17	◄	33	!	49	1	65	A	81	Q	97	a	113	q
2	☻	18	↕	34	"	50	2	66	B	82	R	98	b	114	r
3	♥	19	‼	35	#	51	3	67	C	83	S	99	c	115	s
4	♦	20	¶	36	$	52	4	68	D	84	T	100	d	116	t
5	♣	21	§	37	%	53	5	69	E	85	U	101	e	117	u
6	♠	22	▬	38	&	54	6	70	F	86	V	102	f	118	v
7	•	23	↨	39	'	55	7	71	G	87	W	103	g	119	w
8	◘	24	↑	40	(	56	8	72	H	88	X	104	h	120	x
9	○	25	↓	41	)	57	9	73	I	89	Y	105	i	121	y
10	◙	26	→	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂	27	←	43	+	59	;	75	K	91	[	107	k	123	{
12	♀	28	∟	44	,	60	<	76	L	92	\	108	l	124	|
13	♪	29	↔	45	-	61	=	77	M	93	]	109	m	125	}
14	♫	30	▲	46	.	62	>	78	N	94	^	110	n	126	~
15	☼	31	▼	47	/	63	?	79	O	95	_	111	o	127	⌂

Некоторые из вышеперечисленных символов имеют особый смысл. Так, например, символ с кодом 9 обозначает символ горизонтальной табуляции, символ с кодом 10 – символ перевода строки, символ с кодом 13 – символ возврата каретки.

Дополнительная часть таблицы символов

КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С
128	А	144	Р	160	а	176	▒	192	└	208	╨	224	р	240	Ё
129	Б	145	С	161	б	177	▒	193	┴	209	╤	225	с	241	ё
130	В	146	Т	162	в	178	▓	194	┬	210	╥	226	т	242	Є
131	Г	147	У	163	г	179	│	195	├	211	╙	227	у	243	є
132	Д	148	Ф	164	д	180	┤	196	─	212	╘	228	ф	244	Ї
133	Е	149	Х	165	е	181	╡	197	┼	213	╒	229	х	245	ї
134	Ж	150	Ц	166	ж	182	╢	198	╞	214	╓	230	ц	246	Ў
135	З	151	Ч	167	з	183	╖	199	╟	215	╫	231	ч	247	ў
136	И	152	Ш	168	и	184	╕	200	╚	216	╪	232	ш	248	°
137	Й	153	Щ	169	й	185	╣	201	╔	217	┘	233	щ	249	∙
138	К	154	Ъ	170	к	186	║	202	╩	218	┌	234	ъ	250	·
139	Л	155	Ы	171	л	187	╗	203	╦	219	█	235	ы	251	√
140	М	156	Ь	172	м	188	╝	204	╠	220	▄	236	ь	252	№
141	Н	157	Э	173	н	189	╜	205	═	221	▌	237	э	253	¤
142	О	158	Ю	174	о	190	╛	206	╬	222	▐	238	ю	254	■
143	П	159	Я	175	п	191	┐	207	╧	223	▀	239	я	255	

В таблицах обозначение **КС** означает "код символа", а **С** – "символ".

Операции языка Си

Операции приведены в порядке убывания приоритета. Операции с разными приоритетами разделены чертой.

Операция	Краткое описание	Использование	Порядок выполнения
Унарные операции			
.	Доступ к члену	*объект* **.** *член*	Слева направо
->	Доступ к члену по указателю	*указатель* -> *член*	
[]	Индексирование	*переменная*[*выражение*]	
()	Вызов функции	ID(список_выр.)	
++	Постфиксный инкремент	*lvalue*++	Справа налево
--	Постфиксный декремент	*lvalue*--	
sizeof	Размер объекта или типа	sizeof(*выражение или тип*)	
++	Префиксный инкремент	++*lvalue*	
--	Префиксный декремент	--*lvalue*	
~	Побитовое НЕ	~*выражение*	
!	Логическое НЕ	!*выражение*	
-	Унарный минус	-*выражение*	
+	Унарный плюс	+*выражение*	
*****	Раскрытие указателя	**выражение*	
&	Адрес	&*выражение*	
()	Приведение типа	(*тип*)*выражение*	
Бинарные и тернарная операции			
*****	Умножение	*выражение* * *выражение*	Слева направо
/	Деление	*выражение* / *выражение*	
%	Получение остатка	*выражение* % *выражение*	
+	Сложение	*выражение* + *выражение*	
-	Вычитание	*выражение* – *выражение*	
<<	Сдвиг влево	*выражение* << *выражение*	
>>	Сдвиг вправо	*выражение* >> *выражение*	
<	Меньше	*выражение* < *выражение*	
<=	Меньше или равно	*выражение* <= *выражение*	
>	Больше	*выражение* > *выражение*	
>=	Больше или равно	*выражение* >= *выражение*	
==	Равно	*выражение* == *выражение*	
!=	Не равно	*выражение* != *выражение*	
&	Побитовое И	*выражение* & *выражение*	
^	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	*выражение* ^ *выражение*	
|	Побитовое ИЛИ	*выражение* | *выражение*	
&&	Логическое И	*выражение* && *выражение*	
||	Логическое ИЛИ	*выражение* || *выражение*	
?:	Условная операция **(тернарная)**	*выражение* ? *выражение* : *выражение*	

Операция	Краткое описание	Использование	Порядок выполнения
=	Присваивание	*lvalue* = ***выражение***	Справа налево
***=**	Умножение с присваиванием	*lvalue* *= ***выражение***	
/=	Деление с присваиванием	*lvalue* /= ***выражение***	
%=	Остаток от деления с присваиванием	*lvalue* %= ***выражение***	
+=	Сложение с присваиванием	*lvalue* += ***выражение***	
-=	Вычитание с присваиванием	*lvalue* -= ***выражение***	
<<=	Сдвиг влево с присваиванием	*lvalue* <<= ***выражение***	
>>=	Сдвиг вправо с присваиванием	*lvalue* >>= ***выражение***	
&=	Поразрядное И с присваиванием	*lvalue* &= ***выражение***	
|=	Поразрядное ИЛИ с присваиванием	*lvalue* |= ***выражение***	
^=	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием	*lvalue* ^= ***выражение***	
,	Последовательное вычисление	***выражение***, ***выражение***	Слева направо

Возможности препроцессора и его вызов

Препроцессор, как мы уже знаем, это программа предварительной обработки исходного текста программы перед этапом компиляции. Чаще всего препроцессор автоматически вызывается на этапе компиляции, если в исходном тексте обнаружена хотя бы одна его директива.

Напомним, что признаком директивы препроцессора является символ # (обычно в начале строки). При необходимости продолжения директивы в следующей строке, текущую строку должен завершать символ '\'.

Возможности препроцессора языка С:

- лексемное замещение идентификаторов;
- макрозамещение;
- включение файлов исходного текста;
- условная компиляция;
- изменение нумерации строк и текущего имени файла.

Директивы лексемного замещения идентификаторов

Директива определения значения идентификатора:

```
#define   ID   строка
```

В результате каждое вхождение в исходный текст элемента «ID» заменяется на значение элемента «строка»:

```
#define L_bufs 2048
#define binary int
#define WAIT fflush(stdin); getch()
#define BEEP sound(800);\
                delay(100);\
                nosound()
```

Лексемное замещение весьма удобно для сокращения записи повторяющихся фрагментов теста и определения символических констант:

```
#define YES 1
#define NO  2
#define ESC 27
#define Enter 30
```

которые могут быть в дальнейшем использованы:

```
if (x==ESC) break;
BEEP;
return(YES);
```

Директива отмены

```
#undef  ID
```

Далее по исходному тексту можно назначить новое значение такого идентификатора.

Макрозамещение

Макрозамещение - обобщение лексемного замещения посредством параметризации строки директивы define в виде:

#define ID(параметр1,...) строка

здесь между элементом «ID» и открывающей скобкой пробелы не допускаются.

Такой вариант директивы define иногда называют макроопределением. Элемент «строка» обычно содержит параметры, которые будут заменены препроцессором фактическими аргументами так называемой макрокоманды, записываемой в формате:

ID(аргумент1,...)

Пример макроопределения и макрокоманд:

```
#define P(X) printf("\n%s",X)
. . .
char *x;
P(x);                          // Использование макроопределения P(X)
P(" НАЧАЛО ОПТИМИЗАЦИИ");
printf("\n%s",x);              // Эквивалентные операторы
printf("\n%s"," НАЧАЛО ОПТИМИЗАЦИИ");
```

В строке макроопределений идентификаторы параметров сложных выражений рекомендуется заключать в круглые скобки:

```
#define MAX(A,B) ((A)>(B)? (A):(B))
#define ABS(X)    ((X)<0? –(X):(X))
```

Потребность в круглых скобках возникает при опасности искажения смысла вложенных выражений из-за действия правил приоритета операций. Пример искажения смысла операций:

```
#define BP(X) X*X
        . . .
int x,y,z;
x=BP(y+z);        ↔    x=y+z*y+z;  ↔    x=y+(z*y)+z;
```

Очевидно, что ошибки будут и при следующих вариантах:

```
#define BP(X) (X*X)
#define BP(X) (X)*(X)
```

Безопасный вариант:

```
#define BP(X) ((X)*(X))
```

Иногда источником ошибок может быть символ «точка с запятой» в конце строки макроопределения:

```
#define BP(X) ((X)*(X));
. . .
int x,y,z;
x=BP(z)-BP(y);    ↔    y=((z)*(z));        -((y)*(y));
```

Макроопределение отменяется директивой ***undef***.

Идентификаторы макроопределений обычно составляют из прописных букв латинского алфавита. Это позволяет отличать макрокоманды от вызова функций.

Макрокоманда внешне синтаксически эквивалентна операции вызова функции, но смысл их различен. Функция в программе имеется в одном экземпляре, но на ее вызов тратится время для подготовки параметров и передачи управления. Каждая макрокоманда замещается соответствующей частью макроопределения, но потерь на передачу управления нет.

Подключение файлов исходного текста

Напомним, что имеются два варианта запроса включения в текущий файл содержимого другого файла. Директива:

#include < ID_файла>

вводит содержимое файла из стандартного каталога (обычно - \include\), а директива:

#include " ID_файла"

организует последовательный поиск в текущем, системном и стандартном каталогах. Например:

```
#include <alloc.h>          // Средства распределения памяти
#include <dos.h>            // Обращения к функциям ОС
#include "a:\prs\head.h"    // Включение файла пользователя
```

В стандартных каталогах системы программирования помещены операторы описания внутренних функций языка С, системных переменных среды исполнения, структур данных файловой системы, строк диагностических сообщений, определения констант и т.п.

Рекомендуется описания системных объектов включать из стандартных каталогов и размещать их в начале файла исходного текста программы. Системные объекты в результате получают атрибут области действия "глобальный", что устранит неоднозначность их описания.

Условная компиляция

Директивы условной компиляции и реализуемые правила включения исходного текста:

а) условное включение (аналог работы оператора if):

```
#if<предикат_условия>
        ТЕКСТ_1
#endif
```

б) альтернативное включение (аналог if-else):

```
#if<предикат_условия>
        ТЕКСТ_1
#else
        ТЕКСТ_2
#endif
```

Виды предикатов условий:

константное_выражение → истина, если его значение не равно нулю;

def ID → истина, если «ID» был определен ранее оператором #define;

ndef ID → истина, если «ID» не был определен оператором #define.

Константное_выражение отделяется от ключевого слова if разделителем, а def и ndef - нет.

Пример:

```
#ifdef DEBUG
    print_state();
#endif
```

Элементы исходного текста "ТЕКСТ_1" или "ТЕКСТ_2" могут содержать любые директивы препроцессора.

Примеры:

```
#ifndef EOF
#define EOF -1
#endif
#if UNIT==CON
#include "conproc.c"
#else
#include "outproc.c"
#endif
```

Изменение нумерации строк и идентификатора файла

По умолчанию диагностические сообщения компилятора привязываются к номеру строки и ID файла исходного текста.

Директива

#line номер_строки ID_файла

позволяет с целью более приметной привязки к фрагментам текста изменить номер текущей строки и ID файла на новые значения («ID_файла» можно опустить). В системе программирования TURBO-C оператор line игнорируется.

Св. план 2004, поз. 27

Учебное издание

Бусько Виталий Леонидович,
Корбит Анатолий Григорьевич,
Кривоносова Татьяна Михайловна

Конспект лекций по курсу

ОСНОВЫ АЛГОРИТИМЗАЦИИ И ПРОГРАММИРОВАНИЯ

для студентов всех специальностей и всех форм обучения

Редактор Е.Н.Батурчик
Компъютерная верстка Т.В.Шестакова

Подписано в печать	Формат 60x84 1/16.	Бумага офсетная
Печать ризографическая	Гарнитура «Ариал»	Усл. печ. л.
Уч.-изд.л.	Тираж 500 экз.	Заказ .

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия ЛП № 156 от 05.02.2001.
Лицензия ЛВ № 509 от 03.08.2001.
220013, Минск, П.Бровки, 6.