

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

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Конспект лекций
для студентов всех специальностей и форм обучения БГУИР

Минск 2004

УДК 621.3.6 (075.8)
ББК 22.193 Я 73
Б 92

Рецензент:
зав. кафедрой ПОИТ БГУИР, канд. техн. наук,
доц. В.В.Бахтизин

Бусько В.Л.

Б 92 Основы алгоритмизации и программирования: Конспект лекций для студ. всех спец. и форм обуч. БГУИР / В.Л.Бусько, А.Г.Корбит, Т.М.Кривоносова. - Мн.: БГУИР, 2004. - 103 с.: ил.

ISBN 985-444-703-0

Конспект лекций включает темы, охватывающие основные конструкции языка Си. Необходимым дополнением к данной работе является лабораторный практикум [1], при выполнении заданий которого студенты получают навыки алгоритмизации и программирования, осваивают способы решения, в первую очередь, вычислительных задач на персональном компьютере.

УДК 621.3.6 (075.8)
ББК 22.193 Я 73

ISBN 985-444-703-0

© Бусько В.Л., Корбит А.Г.,
Кривоносова Т.М., 2004
© БГУИР, 2004

СОДЕРЖАНИЕ

1. Основные понятия и определения

- 1.1. Структура персональной ЭВМ
- 1.2. Размещение данных и программ в памяти ПЭВМ
- 1.3. Программные модули
- 1.4. Ошибки
- 1.5. Функциональная и модульная декомпозиции
- 1.6. Файловая система хранения информации
- 1.7. Операционная система

2. Понятие алгоритмов и способы их описания

- 2.1. Свойства алгоритмов
- 2.2. Способы описания алгоритмов
- 2.3. Основные символы схемы алгоритма
- 2.4. Пример линейного алгоритма

3. Базовые элементы 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 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

6. Обзор операций

- 6.1. Операции, выражения
- 6.2. Арифметические операции
- 6.3. Операция 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 Си

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

8. Синтаксис операторов языка Си

- 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. Операции над указателями (косвенная адресация)
- 11.2. Ссылка

12. Массивы

- 12.1. Одномерные массивы
- 12.2. Многомерные массивы
- 12.3. Операция *sizeof*
- 12.4. Применение указателей
- 12.5. Указатели на указатели

13. Работа с динамической памятью

- 13.1. Пример создания одномерного динамического массива
- 13.2. Пример создание двумерного динамического массива

14. 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

16. Классы памяти и области действия объектов

- 16.1. Автоматические переменные
- 16.2. Внешние переменные
- 16.3. 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 Си

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

Литература

- Приложение 1. **Таблицы символов ASCII**
- Приложение 2. **Операции языка Си**
- Приложение 3. **Возможности препроцессора**
- Приложение 4. **Некоторые возможности графической подсистемы**

1. Основные понятия и определения

1.1. Структура персональной ЭВМ

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

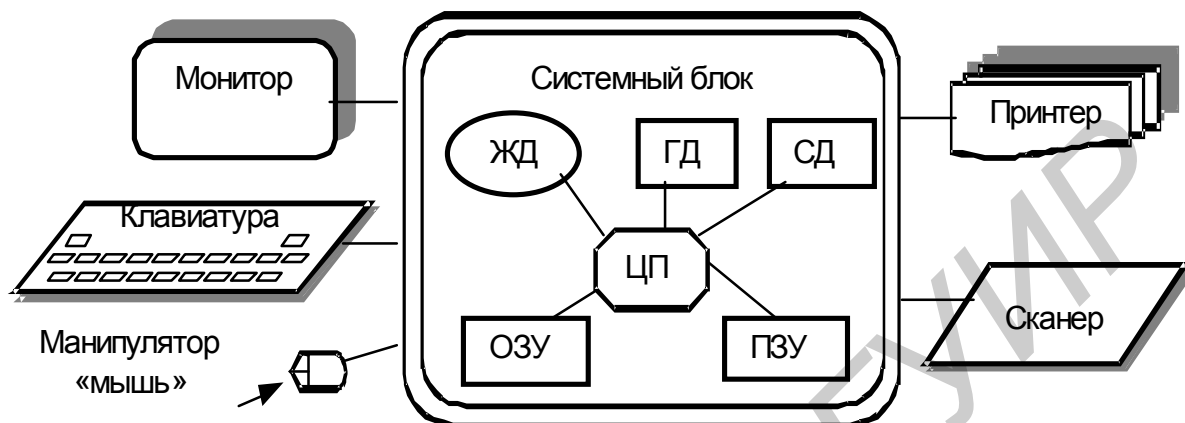


Рис.1. Схема ПЭВМ

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;
- **дискетод (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 информации - сканер.

1.2. 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 состоит из 8 двоичных позиций, т.е. в каждую позицию может быть записан либо 0, либо 1. Объем информации, который поме-

щается в одну двоичную позицию, называется **битом**. Объем информации, равный 8 битам, называется **байтом**.

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

При размещении данных производится их запись с помощью нулей и единиц - кодирование, при котором каждый символ заменяется последовательностью из 8 двоичных разрядов в соответствии со стандартной кодовой таблицей (ASCII). Например, D (код - 68) → 01000100; F (код - 70) → 00100110; 4 (код - 52) → 00110100.

При кодировании числа (коды) преобразуются в двоичное представление, например,

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

С увеличением числа количество разрядов для его представления в двоичной системе резко возрастает, поэтому для размещения большого числа выделяется несколько подряд расположенных байт. В этом случае адресом ячейки является адрес первого байта, один бит которого выделяется под знак числа.

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

Команда размещается в комбинированной ячейке следующим образом: в первом байте - код операции (КОП), которую необходимо выполнить над содержимым ячеек; в одной, двух или трех ячейках (операндах команды) по 2 (4) байта - адреса ячеек (A1, A2, A3), над которыми нужно выполнить указанную операцию. Номер первого байта называется адресом команды. Последовательность из этих команд называется **программой в машинных кодах** (рис. 2).

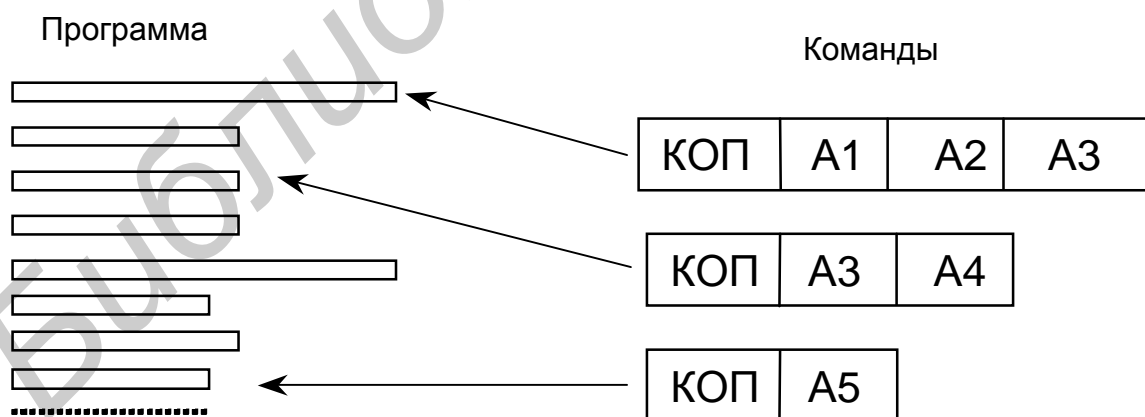


Рис. 2. Схема программы в машинных кодах

1.3. Программные модули

Программа записывается на **языке высокого уровня**, наиболее удобном для реализации алгоритма решения определенного класса задач. Исходный текст программы, введенный с помощью клавиатуры в память компьютера, - **исходный модуль** (в Си - расширение *.cpp).

Транслятор - программа, осуществляющая перевод текстов с одного языка на другой, т.е. с входного языка системы программирования на машинный язык ЭВМ. 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 выполняет созданную программу путем одновременного анализа и реализации предписанных действий, при использовании отсутствует разделение на две стадии - перевод и выполнение.

Большинство трансляторов языка Си - компиляторы.

Результат обработки исходного модуля компилятором - **объектный модуль** (расширение *.obj), это незавершенный вариант машинной программы, т.к., например, к нему должны быть присоединены модули стандартных библиотек. Здесь компилятор (*Compiler*) - вид транслятора, представляющий программу-переводчика 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 (расширение *.exe) – это программный модуль, представленный в форме, пригодной для выполнения.

1.4. Ошибки

Ошибки, допускаемые при написании программ, разделяются 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 - ошибки алгоритма и семантические, которые могут быть 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.

1.5. 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 чего производят декомпозицию сложного алгоритма поставленной задачи, т.е. разбивают его на более простые подзадачи, затем декомпозицию подзадач и т.д.

Основной прием - разбивка алгоритма на отдельные функции и/или модули, используя функциональную и/или модульную декомпозиции.

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

Алгоритм декомпозиции можно представить следующим образом:

- программу создать в виде последовательности 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 проверить, что:

- решение частных задач приводит к решению общей задачи;

- построенная декомпозиция позволяет получать команды, легко реализуемые на выбранном языке программирования.

Единица компиляции в языке Си - отдельный файл (модуль). **Модульная декомпозиция** - разбиение программы на отдельные файлы, каждый из которых решает конкретную задачу и облегчает процесс ее работы. Кроме того, код программы, разделенный на файлы, позволяет части этого кода использовать в других программах.

1.6. Файловая система хранения информации

Для размещения информации и программ на различных устройствах, 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, а информация отсутствует. Информация, помещенная в файл, получает имя этого файла.

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

Имя, которое присваивается файлу, может иметь тип, 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 общее имя. Если каталог вложен внутрь другого каталога, он является **подкаталогом**. Такая вложенность может быть многократной и тогда образуется иерархическая структура хранения данных.

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 разные файлы могут иметь одинаковые имена и быть расположены в разных каталогах (дисках), поэтому для точной идентификации (указания) файла необходимо кроме имени указывать путь к файлу, т.е. место на диске и цепочку подкаталогов, где он находится. Например:

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

Для работы с файлами обычно используют специальные программы, такие, как **FAR, WinCom** и **Проводник**.

1.7. Операционная система

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

Часть программ ОС предназначена для управления процессом выполнения задач. Группа программ так называемого администратора системы позволяет следить за работой пользователей в рамках системы. Важное место занимает блок программ, обеспечивающих обмен сообщениями между пользователями сети.

Удобства, предоставляемые пользователю, зависят от качества ОС, которые постоянно развиваются. В настоящее время наибольшее распространение имеют ОС WindowsXX и LinuxXX.

2. Понятие алгоритмов и способы их описания

Решение задачи на ЭВМ можно разбить на следующие этапы:

- математическая или информационная формулировка задачи;
- выбор метода (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.

Понятие алгоритма занимает центральное место в современной математике и программировании.

Алгоритмизация - сведение задачи к последовательным этапам действий, так чтобы результаты предыдущих действий использовались 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.

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

2.1. 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.

2.2. Способы описания алгоритмов

Наиболее распространенными способами описания алгоритмов являются словесное и графическое описания алгоритма.

Словесное описание алгоритма рассмотрим на конкретном примере: необходимо найти корни квадратного уравнения $ax^2+bx+c=0$ ($a \neq 0$):

1) вычислить $D = b^2 - 4ac$;

2) если $D < 0$, перейти к 4;

3) вычислить корни уравнения $x_1 = (-b + \sqrt{D}) / (2 \cdot a)$;

$x_2 = (-b - \sqrt{D}) / (2 \cdot a)$;

4) конец.

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

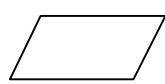

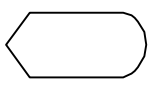
Графическое описание алгоритма - это представление алгоритма в виде схемы, состоящей из последовательности блоков (геометрических фигур), каждый из которых отображает содержание очередного шага алгоритма. Внутри фигур кратко записывают выполняемое действие. Такую схему называют блок-схемой алгоритма.

Правила изображения фигур сведены в единую систему документации (ГОСТ 19.701-90), по которой – это схема данных, отображающая путь данных при решении задачи и определяющая этапы их обработки.

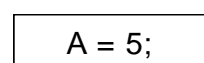
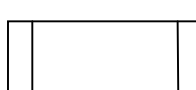
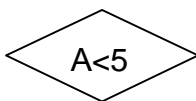
Схема содержит: *символы данных* (могут отображать тип носителя данных); *символы процесса*, который нужно выполнить над данными; *символы линий*, указывающих потоки данных между процессами и носителями данных; *специальные символы* (для удобства чтения схемы).

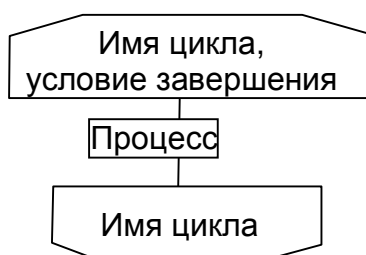
2.3. Основные символы схемы алгоритма

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, 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, из которых только один может быть активизирован после анализа условия, указанного внутри этого символа.






Граница цикла - начало и конец 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).
-  **Комментарии.**

2.4. Пример линейного алгоритма

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

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

при вещественном значении $x > 0$, целом значении m .

Разработка алгоритма обычно начинается с составления схемы, состоящей из оптимальной последовательности вычислений, при которой отсутствуют повторения. При написании алгоритма переменным обычно присваивают имена, фигурирующие в заданном выражении либо иллюстрирующие их смысл.

Для того чтобы не было длинных операторов, исходное выражение полезно разбить на более простые. 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.

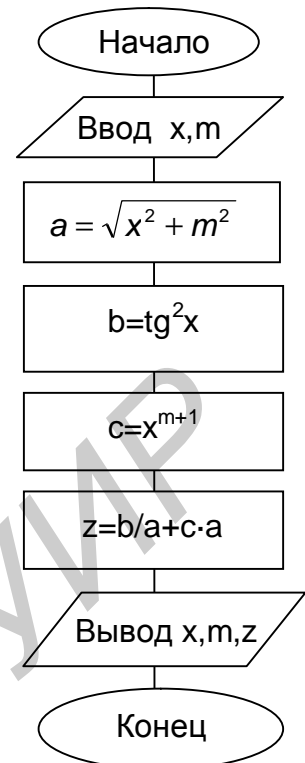


Рис. 3

3. Базовые элементы языка Си

В языке Си фундаментальным понятием является **инструкция** (**операция, оператор, функция**), которая представляет собой описание определенного набора действий над некоторыми объектами. Объектам, над которыми выполняются эти действия, вместо номеров ячеек в памяти принято давать имена (идентификаторы), а содержимое ячеек называть **переменными**, или **константами**, в зависимости от того, изменяется значение в процессе работы или нет.

Таким образом, программа состоит из последовательности инструкций, оформленных в строгом соответствии с набором правил, составляющих **синтаксис языка Си**. Рассмотрим эти правила.

3.1. Алфавит языка

Каждому из множества значений, определяемых одним байтом (от 0 до 255), в таблице знакогенератора ЭВМ ставится в соответствие символ. По кодировке фирмы IBM символы с кодами от 0 до 127, образующие первую половину таблицы знакогенератора, построены по стандарту ASCII и одинаковы для всех компьютеров, вторая половина символов (коды 128 - 255) может отличаться и обычно используется для размещения символов национального алфавита, коды 176 - 223 отводятся под символы псевдографики и коды 240 - 255 – под специальные знаки (прил. 1).

Алфавит языка Си включает:

- буквы латинского алфавита и знак подчеркивания (код 95);
- арабские цифры от 0 до 9;
- специальные символы:

+ (плюс) – (минус) * (звездочка) / (дробная черта) = (равно) > (больше)
< (меньше) ; (точка с запятой) & (амперсанта) [] (квадратные скобки) { } (фигурные скобки) () (круглые скобки) _ (знак подчеркивания) (пробел) . (точка)
, (запятая) : (двоеточие) # (шарп) % (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 (ID) – это имя программного объекта (константы, переменной, метки, типа, функции, модуля и т.д.). В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания; первый символ ID – не цифра; пробелы внутри ID не допускаются.

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

При именовании объектов следует придерживаться общепринятых соглашений:

- ID переменной обычно пишется строчными буквами – *index*, а *Index* – это ID типа или функции, *INDEX* – константа;

- идентификатор должен нести смысл, поясняющий назначение объекта в программе, например, *birth_date* – день рождения, *sum* – сумма;

- если ID состоит из нескольких слов, как, например, *birth_date*, то принято либо разделять слова символом подчеркивания, либо писать каждое следующее слово с большой буквы – *BirthDate*.

В Си прописные и строчные буквы – различные символы.

Идентификаторы *Name*, *NAME*, *name* – различные объекты.

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

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, по количеству участвующих в них операндов.

3.5. Литералы (константы)

Когда в программе встречается некоторое число, например 21, то это число называется литералом, или литеральной константой. Константой, потому что мы не можем изменить его значение, и литералом, потому что оно буквально передает свое значение (от латинского *literal* – буквальный).

Константа является неадресуемой величиной, хотя реально она хранится в памяти машины, но нет никакого способа узнать ее адрес. Каждая константа 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, поскольку компилятор их игнорирует.

В Си комментарии ограничиваются парами символов */** и **/*, а в C++ 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, одна из которых обязательно 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) 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 структуру

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

Рассмотрим кратко основные части общей структуры программ.

Перед компиляцией программа обрабатывается препроцессором (прил. 3), который работает под управлением директив.

Препроцессорные директивы начинаются символом **#**, за которым следует наименование директивы, указывающее ее действие.

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, использующихся в программе. 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, если - в двойные кавычки (" "), то поиск 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, дисплей);

math.h - математические функции.

Второе основное назначение препроцессора – обработка макроопределений. Макроподстановка «определить» имеет общий вид

```
#define < ID > < строка >
```

Например: **#define** PI 3.1415927

- в ходе препроцессорной обработки программы идентификатор PI везде будет заменяться значением 3.1415927.

Рассмотрим пример, позволяющий понять 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 скобки () после ее идентификатора, в которые заключается список параметров. Если параметры отсутствуют, указывают атрибут *void* - отсутствие значения. Перед ID функции указывается тип возвращаемого ею результата, так как функция *main* ничего не возвращает - в качестве результата - *void*.

Код функции представляет собой набор инструкций, каждая из которых оканчивается символом «;». В нашем примере одна инструкция - функция *printf*, выполняющая 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: целый - *int* (integer), вещественный с одинарной точностью - *float* и символьный - *char* (character).

В свою очередь, данные целого типа могут быть короткими - *short*, длинными - *long* и беззнаковыми - *unsigned*, а вещественные - с удвоенной точностью - *double*.

Сложные типы данных – массивы, структуры - *struct*, объединения или смеси - *union*, перечисления - *enum*.

Данные целого и вещественного типов находятся в определенных диапазонах, т.к. занимают разный объем оперативной памяти, табл. 1.

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	$3,14 \cdot 10^{-38} \dots 3,14 \cdot 10^{38}$
double	8	$1,7 \cdot 10^{-308} \dots 1,7 \cdot 10^{308}$

4.3. Декларация объектов

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

- описание, не приводящее к выделению памяти;

- определение, при котором под объект выделяется объем памяти в соответствии с его типом; в этом случае объект можно инициализировать, т.е. задать его начальное значение.

Кроме констант, заданных 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 могут иметь 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> - информация об объекте: объем выделяемой памяти, вид представления и допустимые над ним действия (по умолчанию *int*).

Класс памяти и тип – атрибуты необязательные и при их отсутствии (но не одновременно) устанавливаются 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 чисел. Квалификаторы *short* и *long* указывают на различные размеры и определяют объем памяти, выделяемый под них (см. табл.1), например:

`short x; long x;`
`unsigned x = 8;` - декларация с инициализацией числом 8.

Атрибут *int* в этих случаях может быть опущен.

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

<i>int</i>	Знак	Значение числа															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	- номера бит

<i>unsigned int</i>	Значение числа															
	15															0

<i>long</i>	Знак	Значение числа														
	31	30														0

<i>unsigned long</i>	Значение числа															
	31															0

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

4.5. Данные символьного типа (*char*)

Символьная переменная занимает в памяти один байт. Закрепление конкретных символов за кодами производится кодовыми таблицами.

Для ПЭВМ наиболее распространена таблица кодов ASCII - *American Standard Code for Information Interchange* (прил. 1). Данные типа *char* рассматриваются компилятором как целые, поэтому возможно использование *signed char* (по умолчанию) - символы с кодами от -128 до +127 и *unsigned char* - символы с кодами от 0 до 255.

Примеры: `char res, simv1, simv2;`

`char let = 's';` - декларация с инициализацией символом s.

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

Данные вещественного типа в памяти занимают: *float* - 4 байта; *double* - 8 байт; *long double* (повышенная точность) - 10 байт. Для разме-

щения данных типа *float* обычно 8 бит выделено для представления порядка и знака и 24 бита под мантиссу, табл. 2.

Таблица 2

Тип	Точность (мантисса)	Порядок
<i>float</i>	7 цифр после запятой	± 38
<i>double</i>	15	± 308
<i>Long double</i>	19	± 4932

5. Константы в программах

Константы - объекты, **не подлежащие использованию в левой части оператора присваивания**, т.к. константа - неадресуемая величина. В языке Си константами являются:

- самоопределенные арифметические константы целого и вещественного типов, символьные и строковые данные;
- идентификаторы массивов и функций;
- элементы перечислений.

5.1. Целочисленные константы

Общий формат: $\pm n$ (+ 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)* - 273L (273l). 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	1000L	- десятичные;
0777	00033	01 l	- восьмеричные;
0x123	0X00ff	0xb8000l	- 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: $\pm n.m$, где n , m - целая и дробная части числа;

2) с плавающей десятичной точкой (экспоненциальная форма): $\pm n.mE\pm p$, где n , m - целая и дробная части числа, p - порядок; $\pm 0.xxxE\pm p$ - 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' (тип *char* → целое *int*).

Также используются специальные последовательности символов - управляющие (*escape*) последовательности, основные из них:

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

При присваивании символьной переменной они должны быть заключены в апострофы. Константа '\0', изображающая символ 0 (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 символов, например: `\` - обратный слеш; `'` - апостроф; `"` - кавычки.

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

5.4. Строковые константы

Строковая константа представляет собой последовательность символов кода ASCII, заключенную в кавычки (""). Во внутреннем представлении к строковым константам добавляется нулевой символ '\0', называемый нуль-терминатор, отмечающий конец строки. Кавычки не являются частью строки, а служат только для ее ограничения. Строка в языке Си представляет собой 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, который не является цифрой 0, на печать не выводится, в таблице кодов ASCII имеет код = 0.

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

6. Обзор операций

6.1. Операции, выражения

Выражения используются для вычисления значений (определенного типа) и состоят из операндов, операций и скобок. Каждый операнд может быть, в свою очередь, выражением.

Знак операции – это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются.

Операции делятся на унарные, бинарные и тернарные - по количеству участвующих в них операндов; выполняются в соответствии с приоритетами - для изменения порядка выполнения операций используются круглые скобки.

Большинство операций выполняется слева направо, например, $a+b+c \rightarrow (a+b)+c$. Исключение: унарные операции, операции присваивания и тернарная условная операция ($?:$) - справа налево.

Полный список операций приводится в прил. 2.

6.2. Арифметические операции

Арифметические операции - *бинарные*, их обозначения:

+ (сложение); - (вычитание); / (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, функции, элементы массивов, указатели, любые арифметические выражения.

Порядок выполнения операций:

- 1) выражения в круглых скобках;
- 2) функции (стандартные математические, функции пользователя);
- 3) операции * / (выполняются слева направо);
- 4) операции - + (слева направо).

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

Таким образом, т.к. операции *, /, % обладают высшим приоритетом над операциями +, -, при записи сложных выражений нужно использовать общепринятые математические правила: $x+y \cdot z - \frac{a}{b+c} \leftrightarrow x+y \cdot z - a/(b+c)$.

6.3. Операция присваивания

Формат операции присваивания:

$\langle ID \rangle = \langle \text{выражение} \rangle;$

Присваивание значения в языке Си в отличие от традиционной интерпретации рассматривается как *выражение*, имеющее значение левого операнда после присваивания. Таким образом, присваивание может вклю-

чать несколько операций присваивания, изменяя значения нескольких операндов, например:

`int i, j, k;`

`float x, y, z;`

...

`i = j = k = 0;` \leftrightarrow `k = 0, j = k, i = j;`

`x = i+(y = 3) - (z = 0);` \leftrightarrow `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 используются два вида сокращений записи операции присваивания:

а) вместо записи `v = v # e;`

где # - арифметическая операция (операция над битовым представлением операндов), рекомендуется использовать запись `v #= e;`

например, `i = i + 2;` \leftrightarrow `i += 2;` (**знаки операций без пробелов**);

б) вместо записи `x = x # 1;`

где # - символы, обозначающие операцию инкремента (+), либо декремента (-), x - целочисленная переменная (переменная-указатель), рекомендуется использовать запись:

`##x;` - префиксную или `x##;` - постфиксную.

Если эти операции используются в чистом виде, то различий между постфиксной и префиксной формами нет. Если же они используются в выражении, то в префиксной форме (`##x`) сначала значение x изменится на 1, а затем будет использовано в выражении; в постфиксной форме (`x##`) сначала значение используется в выражении, а затем изменяется на 1. Операции над указателями рассмотрим позже.

Пример 1:

`int i,j,k;`

`float x,y;`

...

`x* = y;`

`i+ = 2;`

`x/ = y+15;`

`--k`

`--k`

`j = i++;`

`j = ++i;`

Смысл записи

`x = x*y;`

`i = i+2;`

`x = x/(y+15);`

`k = k-1;`

`k = k-1;`

`j = i; i = i+1;`

`i = i+1; j = i;`

Пример 2:

`int n,a,b,c,d;`

`n = 2; a = b = c = 0;`

`a = ++n;`

`a+ = 2;`

`b = n++;`

`b- = 2;`

`c = --n;`

`c* = 2;`

`d = n--;`

`d% = 2;`

Значения

`n=3, a=3`

`a=5`

`b=3, n=4`

`b=1`

`n=3, c=3`

`c=6`

`d=3, n=2`

`d=1`

6.5. Преобразование типов операндов арифметических операций

В операциях могут участвовать операнды различных типов, в этом случае они преобразуются к общему типу в порядке увеличения их "размера памяти", т.е. объема памяти, необходимого для хранения их значений. Поэтому неявные преобразования всегда идут от "меньших" объектов к "большим". Схема выполнения преобразований операндов арифметических операций:

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

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*, что 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, т.к. при некорректном использовании операций присваивания могут возникнуть неконтролируемые ошибки. Так, при преобразовании *int* в *char* старший байт просто отбрасывается.

Пусть имеются значения: *float x*; *int i*; тогда $x=i$; и $i=x$; приводят к преобразованиям, причем *float* преобразуется в *int* отбрасыванием дробной части.

Тип *double* преобразуется во *float* округлением.

Длинное целое преобразуется в более короткое целое и *char* посредством отбрасывания лишних бит более высокого порядка.

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

6.6. Операция приведения типа

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

Вид записи операции: $(тип) выражение$;
ее результат - значение выражения, преобразованное к заданному типу.

Операция приведения типа вынуждает компилятор выполнить указанное преобразование, но ответственность за последствия возлагается

на программиста. Рекомендуется использовать эту операцию в исключительных случаях, например:

```
float x;
```

```
int n=6, k=4;
```

```
x=(n+k)/3;      → x=3, т.к. дробная часть будет отброшена;
```

```
x=(float)(n+k)/3; → x=3.333333 - использование операции приведения
```

типа позволяет избежать округления результата деления целочисленных операндов.

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:

<выражение 1> <знак операции> <выражение 2>

Общие правила:

- операндами могут быть любые базовые (скалярные) типы;

- значения *выражений* перед сравнением преобразуются к одному типу;

- результат операции отношения - значение 1, если отношение истинно, или 0 в противном случае (ложно). Следовательно, операция отношения может использоваться в любых арифметических выражениях.

6.8. Логические операции

Логические операции (в порядке убывания относительного приоритета) и их обозначения:

! - отрицание (логическое НЕТ);

&& - конъюнкция (логическое И);

|| - дизъюнкция (логическое ИЛИ).

Общий вид операции отрицания:

!*<выражение>*

Общий вид операций конъюнкции и дизъюнкции

<выражение 1> <операция> <выражение 2>

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 - *истина*, а нулевое - *ложь*, например:

!0 → 1

!5 → 0

x=10;

!((x=y)>0) → 0

Особенность операций конъюнкции и дизъюнкции – экономное последовательное вычисление выражений-операндов:

<выражение 1> <операция> <выражение 2>

- если *выражение 1* операции конъюнкция ложно, то результат операции ноль и *выражение 2* не вычисляется;

- если *выражение 1* операции дизъюнкция истинно, то результат операции единица и *выражение 2* не вычисляется.

Пример правильной записи двойного неравенства:

$0 < x < 100 \leftrightarrow (0 < x) \&\& (x < 100)$

6.9. Побитовые логические операции, операции над битами

В Си предусмотрен набор операций для работы с отдельными битами. Эти операции нельзя применять к переменным вещественного типа.

Операции 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.

Общий вид операции инвертирования:

~ <выражение>

Остальные операции над битами имеют вид

<выражение 1> <знак операции> <выражение 2>

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

$\sim 0xF0 \leftrightarrow x0F$

$0xFF \& 0x0F \leftrightarrow x0F$

$0xF0 | 0x11 \leftrightarrow xF1$

$0xF4 \wedge 0xF5 \leftrightarrow x01$

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

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

Операция (!) используется для включения бит $w = x ! u$, устанавливает в единицу те биты в x , которые =1 в u .

Необходимо отличать побитовые операции & и ! от логических операций && и ||, если $x=1$, $u=2$, то $x \& u$ равно нулю, а $x \&\& u$ равно 1.

$0x81 \ll 1 \leftrightarrow 0x02$

$0x81 \gg 1 \leftrightarrow 0x40$

Если *выражение 1* имеет тип *unsigned*, то при сдвиге вправо освобождающиеся разряды гарантированно заполняются нулями (логический сдвиг). Выражения типа *signed* могут, но необязательно, сдвигаться впра-

во с копированием знакового разряда (арифметический сдвиг). При сдвиге влево освобождающиеся разряды всегда заполняются нулями. Если *выражение 2* отрицательно либо больше длины *выражения 1* в битах, то результат операции сдвига не определен.

Унарная операция (~) дает дополнение к целому, т.е. каждый бит со значением 1 получает значение 0 и наоборот. Эта операция оказывается полезной в выражениях типа

$X \& (\sim)077,$

где последние 6 бит *X* маскируются нулем.

Операции сдвига << и >> осуществляют соответственно сдвиг вправо (влево) своего левого операнда на число позиций, задаваемых правым операндом, например, $x \ll 2$ сдвигает *x* влево на две позиции, заполняя освобождающиеся биты нулями (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*:

$x \ll 1 \leftrightarrow x * 2; \quad x \gg 1 \leftrightarrow x / 2$
 $x \ll 3 \leftrightarrow x * 8$

Подобное применение операций сдвига безопасно для беззнаковых и положительных значений *выражения 1*.

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 чётно!");`

Комбинирование операций над битами с арифметическими операциями часто позволяет упростить выражения.

6.10. Операция «,» (запятая)

Данная операция используется при организации строго гарантированной последовательности вычисления выражений (используется там, где по синтаксису допустима только одна операция, а нам необходимо разместить две и более, например, в операторе *for*). Форма записи:

выражение 1, ..., *выражение N*;

выражения 1, ..., *N* вычисляются последовательно и результатом операции становится значение *выражения N*, например:

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

получим последовательность вычислений: $i = 1, j = i = 1, i = 2, k = 6, n = 2 + 1 + 6,$ и в результате $m = n = 9.$

7. Обзор базовых инструкций языка Си

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 языка Си декларированы в файлах *math.h* и *stdlib.h*.

В большинстве приведенных здесь функций аргументы *x, y* и результат выполнения имеют тип *double*. Аргументы тригонометрических функций должны быть заданы в радианах (2π радиан = 3600), табл. 3.

Таблица 3

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)

7.3. Функции вывода данных

Для вывода информации на экран монитора в языке Си чаще всего используются функции *printf* и *puts*.

Формат функции форматного вывода на экран:

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 %) и специальные (управляющие) символы;

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

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

Функция *printf* выполняет 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:

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

Для типов *long* и *double* добавляется символ *l*, например, `%ld` - длинное целое, `%lf` – число вещественное с удвоенной точностью.

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

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

Получим: Только 5% предприятий не работало.

Управляют выводом специальные последовательности символов: `\n` - новая строка; `\t` - горизонтальная табуляция; `\b` - шаг назад; `\r` - возврат каретки; `\v` - вертикальная табуляция; `\\` - обратная косая; `'` - апостроф; `"` - кавычки; `\0` - нулевой символ (пусто).

В модификаторах формата функции *printf* после символа % можно указывать строку цифр, задающую минимальную ширину поля вывода, например: `%5d` (для целых), `%4.2f` (для вещественных - две цифры после запятой для поля, шириной 4 символа). Если указанной 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 (\n).

Функция **putchar** выдает на экран дисплея один символ без добавления символа '\n'.

7.4. Функции ввода информации

Функция, предназначенная для форматированного ввода исходной информации с клавиатуры:

scanf ("управляющая строка" , список объектов ввода);

в управляющей строке указываются **только модификаторы форматов**, количество, тип и порядок следования которых должны совпадать с количеством, типом и порядком следования вводимых объектов, иначе результат ввода непредсказуем.

Список объектов ввода представляет собой **адреса** переменных, разделенные запятыми, т.е. для ввода значения переменной перед ее идентификатором указывается символ &, обозначающий «взять адрес».

Если нужно ввести значение строковой переменной, то использовать символ & не нужно, т.к. строка - это массив символов, а ID массива эквивалентно адресу его первого элемента. 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 модификаторов форматов, что и **printf**, отличия - отсутствует формат %g, форматы %e,%f - эквивалентны.

Внимание! Функцией **scanf** (формат %s) строка вводится 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 Си, которое относится к любому переносу данных от источника к приемнику.

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, ввода **cin** - к клавиатуре. Для их работы необходимо подключить файл **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;
}

```

8. Синтаксис операторов языка Си

Операторы языка Си можно разделить на три группы: операторы-декларации (рассмотрены ранее); операторы преобразования объектов; операторы управления процессом выполнения алгоритма.

Программирование процесса преобразования объектов производится посредством записи выражений.

Простейший вид операторов - выражение, заканчивающееся символом «;» (точка с запятой).

Простые операторы: оператор присваивания (выполнение операций присваивания); оператор вызова функции (выполнение операции вызова функции); пустой оператор «;».

К **управляющим операторам** относятся: операторы условного и безусловного переходов, оператор выбора альтернатив (переключатель), операторы организации циклов и передачи управления (перехода).

Каждый из управляющих операторов имеет конкретную лексическую конструкцию, образуемую из ключевых слов языка Си, выражений и символов-разделителей.

Допускается 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 Си имеется две разновидности условных операторов: простой и полный. Синтаксис простого оператора:

if (выражение) оператор 1;

здесь *выражением*, как правило, является логическое или выражение отношения. Если *выражение* истинно (не ноль), то выполняется *оператор 1*, иначе он игнорируется; *оператор 1* - простой или составной (блок).

Примеры записи:

1) *if (x > 0) x = 0;*

- 2) `if (i != 1) j++, s = 1;` - используем операцию «запятая»;
 3) `if (i != 1) { j++; s = 1; }` - последовательность операций;
 4) `if (getch() != 27) {`
 `k=0;`
 `}` - если нажата клавиша, не "Esc".

5) `if (! x) exit (1);` \leftrightarrow `if (x == 0) exit (1);`

6) `if (i > 0)`
 `if (i < n) k++;` \leftrightarrow `if ((i > 0) && (i < n)) k++;`

Синтаксис полного оператора условного выполнения:

if (*выражение*) *оператор 1*;
 else *оператор 2*;

Если *выражение* не ноль (истина), то выполняется *оператор 1*, иначе - *оператор 2*; 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;`

Когда 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 (*выражение 1*) *оператор 1*;
 else if (*выражение 2*) *оператор 2*;
 else if (*выражение 3*) *оператор 3*;
 else *оператор 4*;

Если какое-то *выражение* оказывается истинным, то выполняется относящийся к нему *оператор* и этим вся цепочка заканчивается. Последняя часть с *else* - случай, когда ни одно из проверяемых условий не выполняется. Когда при этом не нужно предпринимать никаких явных действий, *else оператор 4*; может быть опущен, или его можно использовать для контроля, чтобы засечь "невозможное" условие.

Пример:

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

8.2. Условная операция «? :»

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

выражение 1 ? выражение 2 : выражение 3;

если *выражение 1* отлично от нуля (истинно), то результатом операции является *выражение 2*, в противном случае - *выражение 3*; каждый раз вычисляется только одно из выражений 2 или 3.

Для нахождения максимального значения из *a* и *b* (значение *z*) можно использовать оператор *if* :

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

Используя условную операцию, этот пример можно записать как

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

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

8.3. Оператор выбора альтернатив (переключатель)

Общий вид оператора:

```
switch (выражение) {
    case константа 1: оператор 1; break;
    case константа 2: оператор 2; break;
    ...
    case константа N: оператор N; break;
    default: оператор N+1; break; - может отсутствовать
}
```

Значение вычисленного *выражения* должно быть целого типа (символьного). Это значение (константа выбора) сравнивается со значениями *констант*, стоящих после *case*, и при совпадении с одной из них выполняется передача управления соответствующему *оператору*. В случае несовпадения значения *выражения* с одной из *констант* происходит переход на *default* либо при отсутствии *default* - к оператору, следующему за оператором *switch*. Оператор *break* (разрыв) выполняет выход из оператора *switch*; *break* может отсутствовать.

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*, использовали оператор *break*, поэтому результатом будет Случай 2.

Пример 2 (оператор *break* отсутствует):

```
void main(void)
{ 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:

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

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).

9.2. Оператор с предусловием *while*

Общий вид записи:

while (*выражение*) *код цикла*;

Если *выражение* - истинно (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* 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, что данный цикл всегда выполняется хотя бы один раз.

9.4. Оператор цикла с предусловием и коррекцией *for*

Общий вид оператора:

```
for (выражение 1; выражение 2; выражение 3) код цикла;
```

Цикл *for* эквивалентен последовательности инструкций:

```
выражение 1;
while (выражение 2)
{
    код цикла . . .
    выражение 3;
}
```

здесь *выражение 1* - инициализация счетчика (начальное значение), *выражение 2* - условие продолжения счета, *выражение 3* - 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 необязательно должно инициализировать переменную. Необходимо только помнить, что первое выражение вычисляется только один раз перед тем, как остальные части начнут выполняться.

```
for (printf(" Вводите 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 < Nk; n += delta) ... ;
```

параметры Nk, 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 после 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 или 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.

Программа с *goto* может быть написана без него за счет повторения некоторых проверок и введения дополнительных переменных.

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 сложная, а ошибки могут возникать в нескольких местах, то такая организация оказывается удобной.

10.2. Оператор *continue*

Этот оператор может использоваться во всех типах циклов, но не в операторах *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;  
}
```

10.3. Оператор *break*

Оператор ***break*** производит экстренный выход из самого внутреннего цикла или оператора *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 также 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.

11 . 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 Си 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;           - 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*.

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):

	4001	4003	4005	4007	4009	
4000	4002	4004	4006	4008	400A	
p	p1	p2				

$p = 4000, p1 = 4002 = (4000 + 1 * \text{sizeof}(*p)) \rightarrow 4000+2(int)$

$p2 = 4004 = (4000 + 2 * \text{sizeof}(*p)) \rightarrow 4000+2*2$

11.1. Операции над указателями (косвенная адресация)

Указатель может использоваться в выражениях вида

$p \# iv, \#\# p, p \#\#, p \# = iv,$

p - указатель, *iv* - целочисленное выражение, *#* - символ операций + или -.

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

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

При сравнении указателей могут использоваться операции отношения, наиболее важными 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 указатели 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`.

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 разыменования. Над ссылкой арифметические операции запрещены, т.к. ссылка декларируется следующим образом:

тип &ID = инициализатор;

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

```
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

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, \dots, c_n) , называемых его компонентами, причем каждая компонента имеет свой номер, который принято обозначать в виде индекса. Матрица A - это таблица чисел $(a_{ij}, i=1, \dots, m; j=1, \dots, n)$, 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. На-

пример, произведение матрицы на вектор $\vec{b} = A \cdot \vec{c}$, $b_i = \sum_{j=1}^n a_{ij} \cdot c_j$;

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

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.

12.1. 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 и т.д.);

размер – количество элементов в массиве.

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 данном массиве 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];
```

Декларация 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.

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

12.2. Многомерные массивы

Декларация многомерного массива в общем виде:

```
<тип> < ID >[размер 1][размер 2]...[размер N];
```

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 таком 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 одновременной инициализацией:

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

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

12.3. Операция `sizeof`

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

`sizeof (параметр);`

где *параметр* – тип или ID 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.

Наиболее часто операция `sizeof` применяется при динамическом распределении памяти.

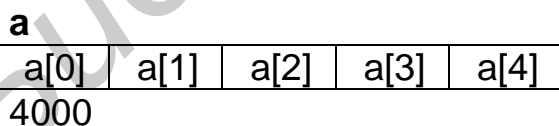
12.4. Применение указателей

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` и указатель `q`:

`int a[5], *q;`

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



Здесь приведено символическое изображение оперативной памяти, выделенной компилятором для объявленного целочисленного массива `a[5]`. Указатель `a` содержит адрес его начала в памяти, 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 памяти:

$\&a[0] \leftrightarrow \&>(*a) \leftrightarrow a$

2) обращение к первому элементу массива:

$*a \leftrightarrow a[0]$

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

`char s[256];` - размерность должна быть константой;

...

`int kol = sizeof(s)/sizeof(*s);` - количество элементов массива s .

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 элементу массива a можно присвоить адрес целочисленной переменной y , например $a[1]=\&y$; чтобы найти значение переменной y через данный элемент массива a , необходимо записать $*a[1]$.

12.5. 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 памяти, 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 двумерного массива – это указатель на массив указателей (переменная типа «указатель на указатель»: int **m;), поэтому выражение a[i][j] эквивалентно выражению *((*(m+i)+j).

Например, двумерный массив m(3×4) компилятор рассматривает как массив четырех указателей, каждый из которых указывает на начало массива со значениями размером по три элемента каждый.

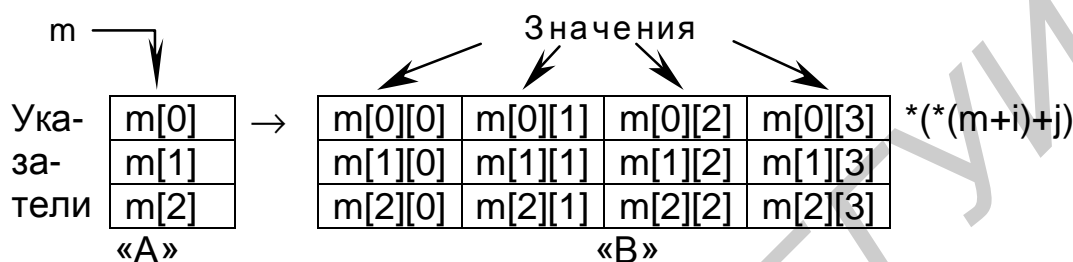


Рис. 4

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 получим и при следующем 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 *u* в 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.

13. Работа с динамической памятью

В языке Си 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 С++ введены две операции: захват памяти - ***new*** и освобождение захваченной ранее памяти - ***delete***.

Общий формат записи:

```

указатель = new тип (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 – 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 двумерного динамического массива

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

```
...  
int **m, n1, n2, i, j;  
puts(" Введите размеры массива (количество строк и столбцов: ");  
scanf("%d%d", &n1, &n2);  
m = new int * [n1]; // Захват памяти для указателей «А» (n1=3)  
for ( i=0; i<n1; i++) // Захват памяти для элементов «В» (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";
printf(" %d, %d .", strlen(S1), strlen(S2));
```

Результат выполнения данного участка программы: 6 , 5 .

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 "с" 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. прил. 1). Символы национального (рус-

ского) алфавита - вторая половина кодов. Для преобразования символов из кодировки ANSI в кодировку ASCII можно использовать функцию **CharToOem** и функцию **OemToChar** - для обратного преобразования (находятся в файле **windows.h**). Приведем пример «русификации» текстов.

```
...
#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 языке Си 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 форме определения.

Описание функции заключается в приведении в начале 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 переменной 1, ..., тип переменной N*);

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

Описание прототипа дает возможность компилятору проверить соответствие типов и количества параметров при 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  
    return выражение;  
}
```

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

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 значения, она описывается как функция типа **void** и в данном случае оператор **return** не ставится.

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 в функции оператора.

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 Си, 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 Си 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, порядку следования и 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 Си 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 <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 Си 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 Си, 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 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 идентификатор этой функции:

```
переменная-указатель = ID_функции;
```

Например, имеется функция с прототипом `float f1(char, float);` тогда операция `p_f = f1;` установит указатель `p_f` 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.

При этом для удобства 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.

Программа для решения данной задачи может быть следующей:

```
...
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, имеющей тип, введенный с помощью операции *typedef* */

```

float sum(p_f fun, int n, float x) {
float s=0;
for(int i=1; i<=n; i++) s+=fun(x);
return s;
}
float f1(float r) { // Первое слагаемое
return (r/5.);
}
float f2(float r) { // Второе слагаемое
return (r/2.);
}

```

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

Передача в функцию одномерного массива:

```

void main (void) {
int vect [20];
... fun(vect); ...
}
void fun( int v [ ])
{ ... }

```

Передача в функцию двумерного массива:

```

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

```

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*, который определяет время существования (время жизни) переменной и область ее 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.

Классы памяти объектов в языке Си:

- 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*).

16.1. 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 коде функции, имеют атрибут класса памяти *auto*.

Принадлежность к этому классу можно подчеркнуть явно, 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 C++) неприме-

нима операция адресации &, а также это неприменимо для массивов, структур, объединений и переменных с плавающей точкой.

16.2. 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 Си ключевое слово *static* имеет разный смысл для локальных и глобальных объектов.

Переменные, описанные вне функции, являются внешними переменными. 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 собирает 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*.

Описание *extern* может быть опущено, если внешнее определение переменной находится в том же файле, но до ее использования в некоторой конкретной функции.

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

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;  
}
```

Описания функций подразумевают атрибут *extern* по умолчанию.

16.3. Область действия переменных

В языке Си 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 Си:

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

Файл исходного текста может включать любое количество определенных 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
    ...
}

```

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 внутреннее объявление объекта скрывает доступ к объекту, объявленному на более высоком уровне, например:

```

...
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;
}

```

В результате выполнения этой программы на экране получим:

```
Block 1 - 3
```


Block 2 - 2.5
Block 3 - а
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. Данные, входящие в эту группу, называют полями.

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

- структура – обозначение участка оперативной памяти, где располагаются конкретные значения данных; в дальнейшем – это структурная переменная, поля которой располагаются в смежных областях памяти;
- структура – правила формирования структурной переменной, которыми руководствуется компилятор при выделении ей места в памяти 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 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, Ф.И.О. и средний балл. Один из возможных вариантов:

```
struct Stud_type {  
    char Number[10];  
    char 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).

Так как одним из параметров групповой обработки структурных объектов является размер, не рекомендуется декларировать поле структуры указателем на объект переменной размерности, поскольку в данном случае многие операции со структурными данными будут некорректны.

17.3. Создание структурных переменных

Как уже отмечалось, само описание структуры не приводит к выделению под нее места в памяти. 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 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." };
```

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 *класс памяти*, данный атрибут можно определить только для всей структуры;

3) идентификаторы 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, 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.

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 к полям объявленного ранее шаблона:

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

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

```
s2 -> Number,        s2 -> Fio,        s2 -> S_b.
```

Участок программы, иллюстрирующий передачу структурных данных в функцию:

```
struct Spisok {  
    char Fio[20];  
    float S_Bal;  
};
```

```
void Out(Spisok );
```

```
// Описание прототипов
```

```
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 Введите 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 быть связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основному шаблону.

Например, в структуре *person*, содержащей Ф.И.О. и дату рождения, сделать дату рождения внутренней структурой *date* по отношению к структуре *person*. Шаблон такой конструкции будет выглядеть следующим образом:

```

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 следующим:

```
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 *typedef* упрощает определение структурных переменных, 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*.

17.5. Массивы структур

Структурный тип может быть использован для декларации массивов, элементами которых являются структурные переменные, например:

```

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

```

или

```

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

```

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

```

spisok[i].day,      *(spisok+i).day,      (spisok+i)->day.

```

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 памяти для структурных переменных с учетом выравнивания, 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 байт было четное.

17.7. Объединения

Объединение - это 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:

```
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, пусть поток сообщений по каналу связи 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]; };  
};
```

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));
}

```

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 - это 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 t_day, end, start; // Текущий день, начало и конец недели
    puts(" Введите день недели (от 1 до 7) : ");
    scanf("%d", &t_day);
    w_day = su;
    start = mo;
    end = w_day - t_day;
    printf("\n Понедельник - %d-й день недели, \
сейчас %d-й день. \n\
До конца недели %d дней (дня). ", start, t_day, end );
}
```

Результат работы программы:

Введите день недели (от 1 до 7) : 2

Понедельник - 1-й день недели, сейчас 2-й день.

До конца недели 5 дней (дня).

18. Файлы в языке Си

Файл – это набор данных, размещенный на внешнем носителе и рассматриваемый в процессе обработки 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, дополне-  
                           ние;  
    char fd;              - дескриптор файла, т.е. число, определяющее его  
                           номер;  
    unsigned char hold;   - переданный 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 те-  
                           кущее значение указателя внутри буфера в зави-  
                           симости 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 * имя_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 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;

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. Закрывание 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* имя_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 Си имеется возможность работы с временными файлами, которые нужны только в процессе работы программы. 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.

Блочный ввод-вывод

В функциях блочного 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.

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 так, 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 блоковой ввод-вывод *fread* и *fwrite*.

Рассмотрим наиболее распространенные функции, с помощью которых можно организовать работу с файлами:

1) *int fileno*(FILE *f) – возвращает значение дескриптора файла *f* - *fd* (число, определяющее номер файла);

2) *long filelength*(int fd) – возвращает длину файла, имеющего номер (дескриптор) *fd* в байтах;

3) *int chsize*(int fd, long pos) – выполняет изменение размера файла, имеющего номер *fd*, признак конца файла устанавливается после байта с номером *pos*;

4) *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;

5) *long ftell*(FILE *f) – возвращает значение указателя на текущую позицию файла (-1 – ошибка);

6) *int feof*(FILE *f) – возвращает ненулевое значение при правильной записи признака конца файла;

7) *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, size = sizeof(Sved);
    char kodR;
    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 Switch
} // 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;
}

```

Литература

1. Бусько В.Л., Корбит А.Г. и др. Программирование: Лаб. практикум для студ. 1-2-го курсов всех спец. БГУИР всех форм обучения. Ч.2. - Мн.: БГУИР, 2003. - 73 с.
2. Березин Б.И., Березин С.Б. Начальный курс С и С++. – М.: Диалог-МРТИ, 1999. - 288 с.
3. Демидович Е.М. Основы алгоритмизации и программирования. Язык Си. - Мн.: Бестпринт, 2001. – 440 с.
4. Касаткин А.И., Вольвачев А.Н. Профессиональное программирование на языке Си: От Turbo–С к Borland С++: Справ.пособие. – Мн.: Выш. шк., 1992. - 240 с.
5. Касаткин А.Н. Профессиональное программирование на языке Си. Управление ресурсами: Справ.пособие. - Мн.: Выш. шк. 1992
6. Керниган Б., Ритчи Д. Язык программирования Си. - М.: Финансы и статистика, 1992. - 271 с.
7. Климова Л.И. С++. Практическое программирование. - М.: Кудиц-Образ, 2001. – 587 с.
8. Павловская Т.А. С/С++. Программирование на языке высокого уровня. - СПб.: Питер, 2004. - 641 с.
9. Петзольд Ч. Программирование для Windows 95. – BHV.: Санкт-Петербург, 1997.
10. Подбельский В.В., Фомин С.С. Программирование на языке Си. - М.: Финансы и статистика, 2001.
11. Страуструп Б. Язык программирования С++. 2-е изд.: В 2 т. Киев: ДиаСофт, 1993.
12. Тимофеев В.В. Программирование в среде С++ Builder 5. - М.: БИНОМ, 2000.
13. Шилд Г. Программирование на Borland С++. - Мн.: ПОПУРРИ, 1999. – 800 с.
14. Юлин В.А., Булатова И.Р. Приглашение к Си. - Мн.: Выш.шк., 1990.
15. Синицын А.К. Конспект лекций по курсу «Программирование» для студентов 1-2-го курсов радиотехнических специальностей. - Мн.: БГУИР, 2001. - 75 с.

Таблицы символов ASCII

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	ё
131	Г	147	У	163	г	179	└	195	└	211	└	227	у	243	ё
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	і
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	о
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	α
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.

Опера-ция	Краткое описание	Использование	Порядок выполне-ния
Унарные операции			
.	Доступ к члену	<i>объект . член</i>	Слева направо
->	Доступ по указателю	<i>указатель -> член</i>	
[] ()	Индексирование Вызов функции	<i>переменная [выражение]</i> <i>ID(список)</i>	
++ -- sizeof ++ -- ~ ! - (+) * & ()	Постфиксный инкремент Постфиксный декремент Размер объекта (типа) Префиксный инкремент Префиксный декремент Побитовое НЕ Логическое НЕ Унарный минус (плюс) Раскрытие указателя Адрес Приведение типа	<i>lvalue++</i> <i>lvalue--</i> <i>sizeof(ID или тип)</i> <i>++lvalue</i> <i>--lvalue</i> <i>~выражение</i> <i>!выражение</i> <i>- (+)выражение</i> <i>*выражение</i> <i>&выражение</i> <i>(тип)выражение</i>	Справа налево
Бинарные и тернарная операции			
* / %	Умножение Деление Получение остатка	<i>выражение * выражение</i> <i>выражение / выражение</i> <i>выражение % выражение</i>	Слева направо
+ -	Сложение Вычитание	<i>выражение + выражение</i> <i>выражение - выражение</i>	
<< >>	Сдвиг влево Сдвиг вправо	<i>выражение << выражение</i> <i>выражение >> выражение</i>	
< <= > >=	Меньше Меньше или равно Больше Больше или равно	<i>выражение < выражение</i> <i>выражение <= выражение</i> <i>выражение > выражение</i> <i>выражение >= выражение</i>	
== !=	Равно Не равно	<i>выражение == выражение</i> <i>выражение != выражение</i>	
& ^ &&	Побитовое И Побитовое исключ. ИЛИ Побитовое ИЛИ Логическое И	<i>выражение & выражение</i> <i>выражение ^ выражение</i> <i>выражение выражение</i> <i>выражение && выражение</i>	

Опера-ция	Краткое описание	Использование	Порядок выполнения
	Логическое ИЛИ	<i>выражение выражение</i>	Слева направо
?:	Условная операция (тернарная)	<i>выражение ? выражение : выражение</i>	
=	Присваивание	<i>lvalue = выражение</i>	Справа налево
*=	Умножение с присваиванием	<i>lvalue *= выражение</i>	
/=	Деление с присваиванием	<i>lvalue /= выражение</i>	
%=	Остаток от деления с присваиванием	<i>lvalue %= выражение</i>	
+=	Сложение с присваиванием	<i>lvalue += выражение</i>	
-=	Вычитание с присваиванием	<i>lvalue -= выражение</i>	
<<=	Сдвиг влево с присваиванием	<i>lvalue <<= выражение</i>	
>>=	Сдвиг вправо с присваиванием	<i>lvalue >>= выражение</i>	
&=	Поразрядное И с присваиванием	<i>lvalue &= выражение</i>	
=	Поразрядное ИЛИ с присваиванием	<i>lvalue = выражение</i>	
^=	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием	<i>lvalue ^= выражение</i>	
,	Последовательное вычисление	<i>выражение, выражение</i>	Слева направо

Возможности препроцессора

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.

Признаком директивы препроцессора является символ **#**. При необходимости продолжения директивы в следующей строке текущую строку должен завершать символ ****.

Возможности препроцессора языка Си:

- 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 (ID):

```
#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.

Макроза́мещение

Макроза́мещение - обобщение лексемного замещения посредством параметризации строки директивы `define` в виде:

```
#define ID(параметр1,... ) строка
```

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
```

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 устраняет 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 ≠ 0;

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 заметной привязки к фрагментам текста изменить номер текущей строки и ID файла на новые значения («ID_файла» можно опустить).

НЕКОТОРЫЕ ВОЗМОЖНОСТИ ГРАФИЧЕСКОЙ ПОДСИСТЕМЫ

1. Основные понятия

В ОС Windows для создания программ с использованием графики существует интерфейс программирования приложений API (*Application programming interface*). Наряду с этой возможностью есть еще много средств программирования для Windows, одним из которых является язык C++, использующийся в основном в сочетании с библиотеками классов MFC (*Microsoft Foundation Classes*) или OWL (*Object Windows Library*).

Графическая информация в Windows обрабатывается в основном функциями GDI (*Graphics Device Interface*), представляющими собой унифицированный интерфейс устройств (средств) отображения.

Поскольку к ЭВМ может быть подключено множество различных устройств отображения, одной из основных задач GDI является поддержка аппаратно-независимой графики.

Все графические устройства отображения делятся на растровые, представляющие графические образы шаблоном точек (видеоадаптеры, матричные и лазерные принтеры), и векторные - отображающие графические образы с использованием линий (плоттеры).

Контекст устройства

Работа GDI базируется на понятии *контекст устройства* (*DC - device context*), абстрагирующего свойства реальных устройств, к которым, в первую очередь, относятся экран, принтер и битовый образ в памяти. Контекст - внутренний объект Windows, доступ к которому осуществляется с помощью функций API. Контекст идентифицируется описателем типа HDC (*handle DC*), который необходим практически каждой функции GDI.

Контекст сопоставляется системой с каждым изображаемым элементом (чаще всего окном) и может быть получен прикладной программой, после чего к нему можно обращаться с единым набором функций, причем поведение контекста будет одинаковым независимо от того, с каким устройством он связан.

Примитивы GDI

К основным типам графических объектов, которые часто называют *примитивами*, относятся:

- **линии** (прямые, прямоугольники, эллипсы, дуги, сплайны Безье, сложные кривые изображаются как ломаные линии, состоящие из коротких прямых), рисуются с использованием графического объекта **пера**;

- **закрашенные области**; если набор прямых и кривых линий ограничивает со всех сторон некоторую область, то она может быть закрашена с использованием **кисти**, выбранной в контексте устройства;

- **битовые шаблоны** - двумерный массив бит, соответствующий пикселям устройства отображения (базовый инструмент растровой графики); битовые образы используются для сложных изображений (значки, курсоры мыши, кнопки панели инструментов);

- **текст** - отличается от других объектов графики, т.к. типов текста много и структуры данных, используемые для описания шрифтов и получения информации о них, - самые большие среди других структур данных в Windows, поэтому поддержка текста часто наиболее сложная часть в системах компьютерной графики.

2. Пример 1 - вывод текста

Чтобы легче было разобраться в некоторых аспектах работы с графикой для Windows, рассмотрим программу, создающую окно, в котором выводится текст "Hello, Windows !". Большая часть программы является надстройкой и будет почти в каждой программе для Windows.

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                    PSTR szCmdLine, int iCmdShow) {
```

```
    static char szAppName[] = "Hello" ;
```

```
    HWND hwnd ;
```

```
    MSG msg ;
```

```
    WNDCLASSEX wndclass ;
```

```
    wndclass.cbSize = sizeof (wndclass) ;
```

```
    wndclass.style = CS_HREDRAW | CS_VREDRAW ;
```

```
    wndclass.lpfnWndProc = WndProc ;
```

```
    wndclass.cbClsExtra = 0 ;
```

```
    wndclass.cbWndExtra = 0 ;
```

```
    wndclass.hInstance = hInstance ;
```

```
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
```

```
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
```

```
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
```

```
    wndclass.lpszMenuName = NULL ;
```

```
    wndclass.lpszClassName = szAppName ;
```

```
    wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION) ;
```

```
    RegisterClassEx (&wndclass) ;
```

```
    hwnd = CreateWindow (szAppName, "First Example",
```

```
        WS_OVERLAPPEDWINDOW,
```

```
        CW_USEDEFAULT, CW_USEDEFAULT,
```

```
        CW_USEDEFAULT, CW_USEDEFAULT,
```

```
        NULL, NULL, hInstance, NULL) ;
```

```
    ShowWindow (hwnd, iCmdShow) ;
```

```
    UpdateWindow (hwnd) ;
```

```
    while (GetMessage (&msg, NULL, 0, 0)) {
```

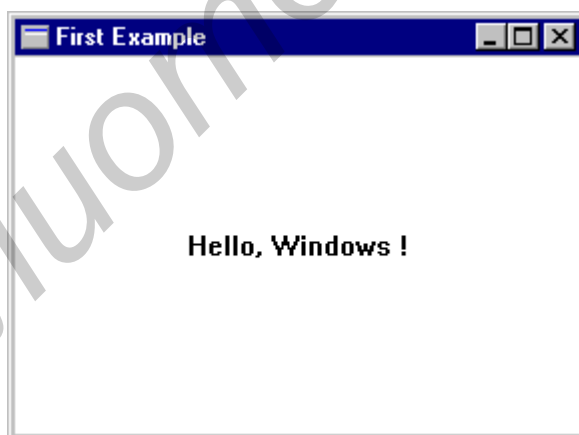
```
        TranslateMessage (&msg) ;
```

```

        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
    LPARAM lParam) {
    HDC hdc ;    PAINTSTRUCT ps ;    RECT rect ;
switch (iMsg) {
    case WM_PAINT :
        hdc = BeginPaint (hwnd, &ps) ;
        GetClientRect (hwnd, &rect) ;
        DrawText (hdc, "Hello, Windows !", -1, &rect,
            DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
        EndPaint (hwnd, &ps) ;
        return 0 ;
    case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
    }
return DefWindowProc (hwnd, iMsg, wParam, lParam);
}

```

В результате программы будет создано окно,



имеющее все свойства Windows: можно захватить мышью заголовок окна и перемещать его по экрану, изменить размеры, развернуть окно и увеличить его до размеров экрана, свернуть или завершить программу.

Функции исходного текста

Рассмотрим данную программу построчно. Функция *WinMain* - основная функция, аналог стандартной функции *main* языка Си; функция *WndProc* - так называемая оконная процедура, отвечающая за ввод и вывод информации.

В подключенном файле **windows.h** содержатся заголовочные файлы с объявлениями функций, структур и числовых констант. Стандартные функции, использующиеся в программе:

LoadIcon - загружает значок (*Icon*) для использования в программе;
LoadCursor - загружает курсор (*Cursor*) мыши;
GetStockObject - получает описатель кисти (*Stock*);
RegisterClassEx - регистрирует класс окна;
CreateWindow - создает окно на основе класса окна;
ShowWindow - выводит окно на экран;
UpdateWindow - заставляет окно перерисовать свое содержимое;
GetMessage - получает сообщение из очереди сообщений;
TranslateMessage - преобразует полученные сообщения;
DispatchMessage - отправляет сообщение оконной процедуре;
BeginPaint - инициирует начало рисования;
GetClientRect - получает размер рабочей области окна;
DrawText - выводит на экран строку текста;
EndPaint - прекращает рисование;
PostQuitMessage - вставляет сообщение "завершить" в очередь;
DefWindowProc - выполняет обработку сообщений по умолчанию.

Идентификаторы и типы данных

Для Windows используют соглашения по именованию переменных - идентификатор переменной составлен из букв или частей слов, отражающих ее смысл; обычно начинается со строчных букв или букв, которые отмечают тип данных переменной (префикс).

Префиксы некоторых переменных, использующихся в дальнейшем: **c** - символ; **by** - BYTE (беззнаковый символ); **n** - короткое целое; **i** - целое; **cx**, **cy** - целое (длины **x** и **y** **c** означает счет - count); **b** или **f** - BOOL (булево целое, **f** - флаг - flag); **w** - WORD (беззнаковое короткое целое); **l** - LONG (длинное целое); **dw** - DWORD (беззнаковое длинное целое); **fn** - функция; **s** - строка; **sz** - строка, завершаемая нулем (string terminated by zero); **h** - описатель (handle); **p** - указатель (pointer).

Идентификаторы, написанные прописными буквами, задаются в заголовочных файлах Windows. Двух- или трехбуквенный префикс, за которым следует символ подчеркивания, показывает основную категорию ее принадлежности, например: **CS** - опция стиля класса (Class Style); **IDI** - идентификационный номер иконки (ID Icon); **IDC** - идентификационный номер курсора; **WS** - стиль окна (windows style); **WM** - сообщение окна.

Аналогичен смысл новых типов данных, например, тип **UINT** - 32-разрядное беззнаковое целое (unsigned int), **PSTR** - указатель на строку символов (pointer string), т.е. *char**; **LONG** - длинное целое.

WndProc возвращает значение типа LRESULT - Long RESULT. Функция *WinMain* получает тип WINAPI (как и любая другая функция Windows), а функция *WndProc* получает тип CALLBACK; - эти идентификаторы явля-

ются ссылкой на особую последовательность вызовов функций, которая имеет место между ОС Windows и ее приложением.

В программе использованы структуры данных: **MSG** - структура сообщения (message); **WNDCLASSEX** - структура класса окна; **PAINTSTRUCT** - структура рисования; **RECT** - структура прямоугольника.

При обозначении переменных структуры пользуются именем самой структуры и строчными буквами, например, переменная *msg* - структура типа MSG; *wndclass* - структура типа WNDCLASSEX.

В программе используются идентификаторы, предназначенные для разных типов описателей (handles): **HINSTANCE** - описатель экземпляра (*instance*) самой программы; **HWND** - описатель окна (*handle to a window*); **HDC** - описатель контекста устройства.

Основная функция WinMain:

```
int WINAPI WinMain ( INSTANCE hInstance, HINSTANCE hPrevInstance,  
    PSTR szCmdLine, int iCmdShow);
```

использует последовательность вызовов WINAPI и возвращает ОС целое значение; параметры:

hInstance - описатель экземпляра - идентифицирует программу;

hPrevInstance - предыдущий (previous) экземпляр; если в данный момент не было загружено копий программы, *hPrevInstance* = 0 или NULL;

szCmdLine - указатель на строку, в которой содержатся любые параметры, переданные в программу из командной строки;

iCmdShow - число, показывающее, каким должно быть выведено на экран окно в начальный момент; обычно: SW_SHOWNORMAL (1) - вывод окна нормального размера, SW_SHOWMINNOACTIVE (7) - окно должно быть изначально свернутым (SW - показать окно - show window).

Регистрация класса окна

Окно всегда создается на основе класса окна, который идентифицирует оконную процедуру, выполняющую процесс обработки поступающих сообщений. Для регистрации класса окна необходимо определить структуру: **WNDCLASSEX** *wndclass*; поля которой описывают характеристики окон, создаваемых на основе класса окна:

cbSize - длина структуры;

wndclass.style = CS_HREDRAW | CS_VREDRAW; объединение стилей класса (CS); значения CS_VREDRAW (вертикальный) и CS_HREDRAW (горизонтальный) показывают, что все окна должны целиком перерисовываться при изменении размеров окна;

wndclass.lpfnWndProc = *WndProc*; устанавливает оконную процедуру;

wndclass.cbClsExtra = 0; *wndclass.cbWndExtra* = 0; резервируют дополнительное пространство, которое может быть использовано программой (не используется - 0, иначе - число байт резервируемой памяти);

wndclass.hInstance = *hInstance*; - описатель экземпляра программы;

wndclass.hIcon = **LoadIcon** (NULL, IDI_APPLICATION);

`wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION);`

устанавливают значки («иконки»), которые появляются на панели задач и в заголовке окна: для получения описателя стандартного значка первый параметр - NULL; при загрузке пользовательского значка - равен описателю экземпляра программы; значок IDI_APPLICATION - маленькое изображение окна; возвращается описатель значка (HICON - описатель значка - handle to an icon);

`wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);` загружает стандартный курсор IDC_ARROW и возвращает его описатель;

`wndclass.hbrBackground = GetStockObject (WHITE_BRUSH);` задает цвет фона рабочей области окон; возвращает описатель белой кисти;

`wndclass.lpszMenuName=NULL;` задает меню класса окна (меню нет);

`wndclass.lpszClassName = szAppName;` классу присваивается имя.

Префиксы обозначают: **LP (lp)** - длинный указатель (long pointer); **lpfn** - длинный указатель на функцию (long pointer to a function); **cb** - счетчик байт (counter of bytes); **hbr** - описатель кисти (handle to a brush).

После того как инициализированы все поля структуры, регистрируем класс окна, вызывая функцию: **RegisterClassEx** (&wndclass);

Создание окна

Функция **CreateWindow** создает окно, детализируя информацию о нем, которая передается функции в качестве параметров:

`hwnd = CreateWindow (szAppName, - имя класса окна;
"First Example", - заголовок окна;
WS_OVERLAPPEDWINDOW, - стиль окна;
CW_USEDEFAULT, - начальное положение по x
CW_USEDEFAULT, и по y;
CW_USEDEFAULT, - начальные размеры по x
CW_USEDEFAULT, и по y;
NULL, - описатель родительского окна;
NULL, - описатель меню окна;
hInstance, - описатель экземпляра программы;
NULL); - параметры создания.`

`szAppName` - содержит строку "Hello" - имя зарегистрированного класса окна, этот параметр связывает окно с его классом; созданное окно - обычное перекрывающееся окно с заголовком, системным меню, «иконками» для сворачивания, разворачивания и закрытия окна;

`WS_OVERLAPPEDWINDOW` - стандартный стиль окна;

`заголовок окна` - текст, который выводится в строке заголовка;

`начальное положение по x и по y` - начальные координаты верхнего левого угла окна относительно левого верхнего угла экрана; `CW_USEDEFAULT` - начальное положение для перекрывающегося окна (по умолчанию); аналогично - "`начальные размеры по x и по y`";

параметры создания - NULL, при необходимости этот параметр используется в качестве указателя на данные, к которым программа в дальнейшем могла бы обратиться.

Функция *CreateWindow* возвращает дескриптор созданного окна *hwnd* типа HWND, являющийся одним из важнейших дескрипторов, которыми оперирует программа для Windows.

Отображение окна

Отображение созданного окна на экране монитора:

ShowWindow (*hwnd*, *iCmdShow*);

hwnd - дескриптор окна; *iCmdShow* - начальный вид окна: SW_SHOW-NORMAL (1) - обычное окно, фон рабочей области закрашивается заданной кистью; SW_SHOWMINNOACTIVE (7) - окно не выводится, на панели задач появляются его имя и «иконка».

Функция **UpdateWindow** (*hwnd*); выполняет перерисовку рабочей области окна, посылая сообщение WM_PAINT в оконную процедуру.

Обработка сообщений

Программа получает информацию от пользователя с клавиатуры и при помощи мыши. Когда происходит ввод информации, она преобразуется в сообщение (*message*), которое помещается в очередь сообщений.

Программа извлекает сообщения из очереди, выполняя блок команд:

```
while (GetMessage (&msg, NULL, 0, 0)) {  
    TranslateMessage (&msg);  
    DispatchMessage (&msg);  
}
```

```
return msg.wParam;
```

Переменная *msg* - структура типа MSG, поля которой:

hwnd - дескриптор окна, для которого предназначено сообщение;

message - идентификатор сообщения;

wParam, *lParam* - параметры сообщения, смысл и значение которых зависят от особенностей сообщения;

time - время поступления сообщения в очередь;

pt - координаты курсора в момент помещения сообщения в очередь - структура типа POINT, имеющая поля: LONG x; LONG y.

Функция **GetMessage** (&msg, NULL, 0, 0); извлекает сообщение из очереди и передает ОС указатель на структуру *msg*; параметры NULL (0) показывают, что программа получает все сообщения от всех окон.

Если *message* равно любому значению, кроме WM_QUIT - функция **GetMessage** возвращает ненулевое значение; сообщение WM_QUIT прерывает цикл обработки сообщений.

Функция **TranslateMessage** (&msg); передает структуру *msg* в ОС для преобразования сообщения, а **DispatchMessage** (&msg); передает структуру *msg* для обработки сообщения оконной процедуре, после чего сообщение возвращается в ОС, которая все еще обслуживает вызов функции

DispatchMessage. Когда ОС возвращает управление в программу к следующему за вызовом *DispatchMessage* коду, цикл обработки сообщений в очередной раз возобновляет работу.

Оконная процедура

Оконная процедура определяет, что выводится в рабочую область окна и как окну реагировать на пользовательский ввод:

LRESULT CALLBACK **WndProc** (hwnd, iMsg, wParam, lParam);

ее параметры идентичны первым четырем полям структуры MSG.

Для определения, какое сообщение получила оконная процедура и как его обрабатывать, обычно используют оператор *switch*. Если оконная процедура обрабатывает сообщение, возвращается значение 0.

В функции *WndProc* обрабатываются два сообщения *WM_PAINT* и *WM_DESTROY*, функция *DefWindowProc* обрабатывает (по умолчанию) все сообщения, не обработанные оконной процедурой.

Сообщение **WM_PAINT** - указывает программе, что часть или вся рабочая область окна недействительна, и ее следует перерисовать.

При первом создании окна недействительна вся рабочая зона и сообщение *WM_PAINT* заставляет оконную процедуру рисовать в рабочей области, что происходит также и при изменении размера окна.

Обработка сообщения *WM_PAINT* почти всегда начинается вызовом:

hdc = **BeginPaint** (hwnd, &ps); и заканчивается: **EndPaint** (hwnd, &ps);

ps - указатель на структуру типа PAINTSTRUCT, содержащую информацию, необходимую для рисования.

Функция *BeginPaint* обновляет фон рабочей области, заданной кистью (рабочая область становится действительной) и возвращает **описатель контекста устройства**, описывающий физическое устройство отображения информации (дисплей) и его драйвер. Функция *EndPaint* освобождает описатель контекста устройства.

Для определения размера рабочей области окна

GetClientRect (hwnd, &rect);

rect - структура типа RECT (прямоугольник - rectangle), содержащая поля типа LONG (*left* - левое, *top* - верх, *right* - правое, *bottom* - низ), определяющие размеры рабочей области окна; *left*, *top* устанавливаются в 0; поля *right*, *bottom* - ширина и высота рабочей области в пикселях.

Структура *rect* используется в функции отображения текста

DrawText (hdc, "Hello, Windows !", -1, &rect,
DT_SINGLELINE | DT_CENTER | DT_VCENTER);

второй параметр - отображаемый текст; третий параметр (-1) указывает, что текст заканчивается нулевым символом; последний параметр - набор флагов, показывающих, что текст выводится в одну строку, по центру внутри прямоугольной области размером *rect*. В результате в центре рабочей области созданного окна отображается строка "Hello, Windows !".

Сообщение **WM_DESTROY** вызывается, если щелкнуть кнопку закрытия окна (выбрать Close из системного меню программы, нажать <Alt>+<F4>), после чего функция

PostQuitMessage (0);

ставит сообщение **WM_QUIT** в очередь, при получении которого функция **GetMessage** возвращает 0, что заставляет **WinMain** прервать цикл обработки сообщений и выйти в систему, закончив программу.

3. Получение описателя контекста устройства

В простейшем случае описатель контекста (контекст) может быть получен при обработке сообщения **WM_PAINT**, как было рассмотрено раньше, а также с помощью функций:

HDC **GetDC**(*hwnd*); - возвращает контекст клиентской области окна (без заголовка, рамки и пр.).

HDC **GetWindowDC**(*hwnd*); - возвращает контекст всего окна.

Поведение функций зависит от некоторых установок стиля оконного класса: **CS_CLASSDC** - использовать единственный разделяемый контекст для всех окон данного класса; **CS_OWNDC** - использовать собственный контекст для каждого окна; **CS_PARENTDC** - использовать контекст и регион отсечения родительского окна и т.д.

Функции

HDC **CreateDC**(LPCTSTR *lpszDriver*, LPCTSTR *lpszDevice*, LPCTSTR *lpszOutput*, CONST DEVMODE* *lpInitData*);

HDC **CreateCompatibleDC**(HDC *hPrimDC*);

создают новый контекст, связанный с указанным устройством (**CreateDC**) или совместимым с известным контекстом (**CreateCompatibleDC**). Во втором случае если образцовый контекст не задан, то создается контекст в памяти, совместимый с текущими установками экрана; возвращают описатель контекста или NULL - в случае ошибки. Параметры:

lpszDriver – может быть "DISPLAY" для контекстов, связанных с экраном, и NULL для всех других устройств;

lpszDevice – логическое имя устройства в системе;

lpszOutput – имя устройства в файловой системе (Win32 - NULL);

lpInitData – указатель на структуру DEVMODE с иницилирующими данными для устройства, NULL - настройки по умолчанию;

hPrimDC – образцовый контекст, с которым будет совместим создаваемый, если NULL - экран с текущими настройками.

По окончании работы с контекстом он должен быть освобожден (закрыт). Для контекстов, полученных с помощью функций **Get...**, используется функция **int ReleaseDC**(*hwnd*, *hdc*), освобождающая общие и оконные контексты; для контекстов, созданных с помощью **Create...**, - функция **int DeleteDC**(*hdc*); обе функции возвращают 1 при успешном завершении, 0 - ошибка.

4. Основные инструменты графической подсистемы

В Windows за формирование изображения отвечают *инструменты*, а функции рисования задают их поведение. Инструменты - системные объекты, с которыми может работать прикладная программа.

К основным инструментам относятся:

- **перо** (Pen) – для отображения контурных примитивов;
- **кисть** (Brush) – для заполнения внутренних областей;
- **шрифт** (Font) – для отображения символов и строк;
- **битовая карта** (Bitmap) – "готовые" растровые изображения.

Инструмент идентифицируется его описателем и создается соответствующей функцией вида **Create...**, которая возвращает этот описатель (NULL - признак ошибки). Количество создаваемых инструментов не ограничивается, но в любом контексте одновременно может быть активным только один инструмент каждого типа. Инструменты различного типа между собой взаимно независимы. Перед удалением инструмент следует деактивировать, выбрав активным другой инструмент того же типа (сохраненный предыдущий).

Инструмент Pen

Для отображения контурных примитивов используется **перо**, выбранное в контексте и определяющее цвет, ширину и стиль линии (сплошное - solid, точечное - dotted, пунктирное - dashed).

По умолчанию устанавливается черное перо - одно из трех **стандартных** перьев, рисующих сплошные линии толщиной в один пиксель, выбранного цвета: BLACK_PEN - черное перо, WHITE_PEN - белое перо и NULL_PEN - пустое (нерисующее) перо.

Определив переменную (описатель пера - handle to a pen) **hPen**; получить описатель стандартного белого пера:

```
hPen = GetStockObject (WHITE_PEN);
```

сделать это перо текущим: **SelectObject** (hdc, hPen); после чего все линии будут использовать его до тех пор, пока не выберем другое перо в контекст устройства или пока не освободим контекст устройства.

Все вышесказанное можно совместить в одной инструкции:

```
hPen = SelectObject (hdc, GetStockObject (WHITE_PEN));
```

если это первый вызов, **hPen** получает описатель уже выбранного черного (по умолчанию) пера; текущим становится белое перо; а вернуться к предыдущему черному перу: **SelectObject** (hdc, hPen);

Для **создания пера** используется функция

```
hPen = CreatePen (iPenStyle, iWidth, rgbColor);
```

iPenStyle - стиль линии: PS_SOLID - сплошная, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT - штриховая, пунктирная и штрихпунктирная; PS_NULL - нерисующее (пустое) перо; PS_INSIDEFRAME - внутренняя обводка, в замкнутом контуре автоматически отступает внутрь в соответствии с толщиной линии;

iWidth для стилей PS_SOLID, PS_NULL и PS_INSIDEFRAME - ширина пера (*iWidth* = 0 - перо шириной в один пиксель);

rgbColor - цвет пера; для перьев всех стилей, кроме PS_INSIDEFRAME, преобразуется в ближайший чистый цвет, PS_INSIDEFRAME позволяет использовать полутона при ширине больше 1.

Функция `hPen = CreatePenIndirect (&logpen);` создает перо на основе структуры (логическое перо - logical pen) **LOGPEN logpen**; содержащей поля: *lpenStyle* - стиль пера; *lpenWidth* - ширина пера в логических единицах измерения; *lpenColor* - цвет пера.

Получить информацию о существующем пере:

GetObject (hPen, sizeof(LOGPEN), &logpen);

Функции `CreatePen` и `CreatePenIndirect` не требуют описателя контекста устройства и создают перья, никак не связанные с контекстом устройства до тех пор, пока не вызвать функцию `SelectObject`.

Рассмотрим пример работы с перьями - красное шириной 3 и черное точечное. Определим переменные для описателей перьев:

`static HPEN hPen1, hPen2;`

При обработке сообщения **WM_CREATE** создадим перья:

`hPen1 = CreatePen (PS_SOLID, 3, RGB (255, 0, 0));`

`hPen2 = CreatePen (PS_DOT, 0, 0);`

При обработке сообщения **WM_PAINT** выберем одно из них в контекст устройства и можем рисовать:

`SelectObject (hdc, hPen2);` [функции рисования]

`SelectObject (hdc, hPen1);` [другие функции рисования]

В процессе обработки сообщения **WM_DESTROY** удалим их:

`DeleteObject (hPen1); DeleteObject (hPen2);`

Инструмент Brush

Объект *кисть* - это битовый образ, который размножается в горизонтальном и вертикальном направлении при закрашивании области.

Имеется шесть **стандартных** (Stock) кистей: WHITE_BRUSH - белая, LTGRAY_BRUSH - светло-серая, GRAY_BRUSH - серая, DKGRAY_BRUSH - темно-серая, BLACK_BRUSH - черная и NULL_BRUSH, HOLLOW_BRUSH - пустые кисти.

Выбор стандартной кисти в контекст устройства аналогичен выбору пера: **HBRUSH hBrush**;

`hBrush = GetStockObject (GRAY_BRUSH);`

`SelectObject (hdc, hBrush);`

теперь внутренняя область фигур будет закрашиваться серым.

Нарисовать фигуру без рамки:

`SelectObject (hdc, GetStockObject (NULL_PEN));`

Нарисовать контур фигуры без закрашивания внутренней области:

`SelectObject (hdc, GetStockObject (NULL_BRUSH));`

Создать сплошную (Solid) логическую кисть:

`hBrush = CreateSolidBrush (rgbColor);`

Создать штриховую (Hatch) кисть, состоящую из горизонтальных, вертикальных или диагональных линий:

hBrush = **CreateHatchBrush** (iHatchStyle, rgbColor);

iHatchStyle - стиль штриховки: HS_HORIZONTAL, HS_VERTICAL, HS_BDIAGONAL - диагональная слева направо вверх; HS_FDIAGONAL - диагональная слева направо вниз; HS_CROSS - прямая сетка; HS_DIAGCROSS - диагональная сетка; *rgbColor* - цвет штриховых линий.

Промежутки между штриховыми линиями закрашиваются в соответствии с режимом и цветом фона.

Можно **создавать кисти**, основанные на битовых шаблонах, используя функцию hBrush = **CreatePatternBrush** (hBitmap);

Функция, включающая три рассмотренные выше функции:

hBrush = **CreateBrushIndirect** (&logbrush);

logbrush - структура типа LOGBRUSH (логическая кисть - logical brush), содержащая поля: *lbStyle* – стиль кисти: BS_SOLID - сплошная; BS_HOLLOW, BS_NULL - пустая (невидимая); BS_HATCHED - штрихованная; BS_PATTERN - задается битовой картой и др.; *lbColor* – цвет кисти, для пустой или шаблонной кисти игнорируется.

Получить описатель кисти: **SelectObject** (hdc, hBrush);

Удалить созданную кисть: **DeleteObject** (hBrush);

Получить информацию о кисти:

GetObject (hBrush, sizeof (LOGBRUSH), &logbrush);

5. Режимы фона и рисования

Если выбранные перо или кисть не сплошные, то они не воздействуют на фоновые промежутки, например, между штрихами, эти промежутки заполняются фоновым цветом, для работы с которым служат функции:

COLORREF **SetBkColor**(hdc, crColor);

COLORREF **GetBkColor**(hdc);

устанавливают новый или получают текущий фоновый цвета; признак ошибки - значение CLR_INVALID; **SetBkColor** возвращает значение предыдущего цвета. Функции:

int **SetBkMode** (hdc, iBkMode); int **GetBkMode** (hdc);

устанавливают новый или определяют текущий режим фона; 0 - признак ошибки; int *iBkMode* - режим фона, по умолчанию белый (OPAQUE) - сначала рисуется фон, затем передний план; режим TRANSPARENT - отменяет заполнение пустот, цвет фона игнорируется.

По умолчанию в контексте режим рисования R2_COPYPEN - простой перенос цвета пера в приемник.

Определить текущий режим рисования: *iDrawMode* = **GetROP2** (hdc);

Установить новый режим рисования:

int WINAPI **SetROP2** (hdc, fnDrawMode);

fnDrawMode - режим рисования, некоторые значения которого представлены ниже:

Режим рисования	Формула	Цвет пикселя
R2_COPYPEN	P	Соответствует цвету пера
R2_BLACK	0	Черный
R2_WHITE	1	Белый
R2_NOP	D	Не меняется - перо ничего не рисует
R2_NOT	~D	Инвертирование цвета подложки, т.е. цвета пикселя до рисования
R2_NOTCOPYPEN	~P	Инвертирование цвета пера
R2_NOTMASKPEN	~(P&D)	Инверсия предыдущего значения
R2_MERGE PEN	P D	Комбинация компонентов цветов
R2_NOTMERGEPEN	~(P D)	Инверсия предыдущего значения
R2_XORPEN	P^D	Исключающее ИЛИ

Цвет пера обозначается буквой P, цвет подложки – D.

6. Инструмент Font

Все символы **шрифта** формируются в соответствии с зарегистрированным в системе шрифтом. Физический шрифт – файл (образ в памяти) с описанием начертаний всех известных в данном шрифте символов. Логический шрифт – объект GDI, характеризуемый как физическим шрифтом, так и его конкретными характеристиками. Он же является и инструментом, отвечающим за формирование символов.

Для **создания** логического шрифта используется функция

HFONT **CreateFont** (nHeight, nWidth, nEscapement, nOrientation, fnWeight, fdwItalic, fdwUnderline, fdwStrikeOut, fdwCharSet, fdwOutputPrecision, fdwClipPrecision, fdwQuality, fdwPitchAndFamily, lpszFace);

возвращающая описатель созданного инструмента, параметры которой:

nHeight – основной размер (высота) шрифта: положительное значение определяет высоту знакоместа, отрицательное – высоту шрифта (после смены знака), нулевое – размер по умолчанию;

nWidth – приблизительная ширина шрифта, 0 - стандартная для выбранного основного размера;

nEscapement – направление вывода строки символов (угол между базовой линией и горизонтальной осью);

nOrientation – ориентация отдельного символа (исчисление аналогично предыдущему параметру);

fnWeight – толщина символов в условных единицах, например: 0 (FW_DONTCARE) – по умолчанию, 400 (FW_NORMAL, FW_REGULAR) – шрифт стандартной толщины, 700 (FW_BOLD) – выделенный шрифт и т.д.;

fdwItalic, *fdwUnderline*, *fdwStrikeOut* – флаги, указывающие, является ли шрифт наклонным, подчеркнутым или перечеркнутым;

fdwCharSet – тип символьного набора: DEFAULT_CHARSET, ANSI_CHARSET, OEM_CHARSET и т.д., (национальные наборы символов);

fdwOutputPrecision – точность вывода символов, фактически предпочтение типа шрифта при наличии альтернативного выбора: OUT_DEFAULT_PRECIS, OUT_DEVICE_PRECIS и т.д.;

fdwClipPrecision – точность отсечения; большинство значений не используются, поведение по умолчанию – CLIP_DEFAULT_PRECIS;

fdwQuality – качество вывода, т.е. степень предпочтительности визуального качества по сравнению с точностью воспроизведения прочих параметров: DEFAULT_QUALITY, DRAFT_QUALITY, PROOF_QUALITY;

fdwPitchAndFamily – комбинация параметров при неопределенном имени шрифта: питч (шаг символов) – 2 младших бита: DEFAULT_PITCH, FIXED_PITCH, VARIABLE_PITCH; "семейство" шрифта – 4 старших бита: FF_DONTCARE - по умолчанию, FF_MODERN - моноширинные шрифты с засечками и без, FF_ROMAN - "книжные" с засечками, FF_SWISS - без засечек, FF_SCRIPT - "рукописные" и курсивные;

lpszFace – имя шрифта, обычно совпадает с именем файла; NULL - система подбирает шрифт, наиболее отвечающий заданным требованиям.

Функция **CreateFontIndirect** (lpGFont); использует в качестве аргумента структуру с полями аналогичного назначения.

Параметры шрифта не включают цвет отображающего инструмента. Управление цветом выводимого текста осуществляется функциями

COLORREF **SetTextColor** (hdc, crColor); COLORREF **GetTextColor** (hdc);

Базовой функцией вывода символа является

BOOL **TextOut** (hdc, nXStart, nYStart, lpString, cbString);

Позицией символа считается верхний левый угол его знакоместа.

7. Системы координат и единицы измерения

Все координаты и размеры в GDI исчисляются в логических единицах. Величина логической единицы и соответствие ее физическим единицам, а также направления отсчета координат составляют режим отображения: MM_TEXT – логическая единица соответствует пикселю, направление координатных осей – вправо и вниз (по умолчанию); MM_LOENGLISH, MM_HIENGLISH – 0,01 и 0,001 дюйма вправо и вверх; MM_LOMETRIC, MM_HIMETRIC – соответственно 0,1 и 0,01 миллиметра вправо и вверх; MM_ISOTROPIC – единица и отсчет устанавливаются дополнительно; MM_ANISOTROPIC – то же, но единица измерения может быть задана различной по вертикальной и горизонтальной осям.

Для изменения и получения режима отображения служат функции

int **SetMapMode** (hdc, fnMapMode); int **GetMapMode** (hdc);

коды возврата и *fnMapMode* принимают перечисленные выше значения.

Режимы MM_ISOTROPIC и MM_ANISOTROPIC требуют использования дополнительных функций для установки разрешающей способности логической системы координат. Данный параметр задается отдельно для окна, к которому относится контекст и поля вывода (viewport) в целом.

BOOL **SetWindowExtEx** (hdc, nXExtent, nYExtent, lpSize);

BOOL **SetViewportExtEx** (hdc, nXExtent, nYExtent, lpSize);

- функции установки соотношения вертикального и горизонтального масштабов логических единиц относительно физических для окна и поля вывода, эффективны только для режимов MM_ISOTROPIC и MM_ANISOTROPIC; параметры *nXExtent*, *nYExtent* - вертикальный и горизонтальный размеры; *lpSize* – указатель на структуру SIZE (поля cx и cy типа LONG), в которую записываются предыдущие значения, может быть NULL; возвращают признак выполнения и структуру *lpSize* (или NULL).

Направление вертикальной оси экранной системы координат сверху вниз - противоположно математическому (чертежному).

Значения разрешений поля вывода и окна, устанавливаемых в некоторых режимах (Windows NT - 800×600): MM_LOMETRIC - 800×600, 3200×2400; MM_HIMETRIC - 800×600, 32000×24000; MM_LOENGLISH - 800×600, 1260×945; MM_TEXT - 1×1, 1×1; MM_ANISOTROPIC - изначально 1×1, 1×1; MM_ISOTROPIC - изначально 800×600, 3200×2400.

Разрешение поля вывода задается либо единичным, либо в соответствии с текущим видеорежимом, отрицательное значение разворачивает вертикальную ось в привычном направлении.

Изменить точку начала отсчета (0,0) можно функциями **SetWindowOrgEx** и **SetViewportOrgExt**.

Текущие значения могут быть получены с помощью функций **Get...**

Логические координаты точки (размеры) в данном контексте могут быть пересчитаны в физические и обратно с помощью функций
BOOL **LPtoDP** (hdc, lpPoints, nCount); BOOL **DPtoLP** (hdc, lpPoints, nCount);

8. Рисование линий и кривых

Функция **LineTo** (hdc, xEnd, yEnd); - рисует отрезок прямой из текущего положения пера до точки (*xEnd*, *yEnd*), которая не включается в отрезок.

Для рисования отрезка из точки (*xStart*, *yStart*) в (*xEnd*, *yEnd*) необходимо сначала использовать функцию **MoveToEx** (hdc, xStart, yStart, &pt); где *pt* - структура типа POINT, определяющая предыдущую позицию, а затем **LineTo** (hdc, xEnd, yEnd); в результате будет нарисован требуемый отрезок и текущее положение пера установится в точку (*xEnd*, *yEnd*).

Текущее положение пера можно определить с помощью функции **GetCurrentPositionEx** (hdc, &pt);

Если необходимо соединить отрезками массив точек *pt* размером *cPoint*, используется функция **Polyline** (hdc, pt, cPoint);

Например, определим массив из 5 точек (10 значений), описывающих контур прямоугольника (первая и последняя точки совпадают):

POINT pt [5] = { 100, 100, 200, 100, 200, 200, 100, 200, 100, 100 };

Для отображения этого прямоугольника используем функцию **Polyline** (hdc, pt, 5);

Функция **Polyline** не учитывает и не изменяет текущее положение пера, а **PolylineTo** использует текущее положение пера для начальной точки

и устанавливает его в конец последнего отрезка. Предыдущий пример можно записать иначе:

```
MoveToEx (hdc, pt[0].x, pt[0].y, NULL); PolylineTo (hdc, pt + 1, 4);
```

В **примере 2** рассмотрено использование этой функции.

Для рисования дуги эллипса используется функция:

```
Arc (hdc, x1, y1, x2, y2, xStart, yStart, xEnd, yEnd);
```

(*x1, y1*) - левый верхний угол, (*x2, y2*) - правый нижний; (*xStart, yStart*) - начало дуги; (*xEnd, yEnd*) - конец дуги.

Для рисования сплайнов Безье используются функции

```
PolyBezier (hdc, pt, iCount); PolyBezierTo (hdc, pt, iCount);
```

Пример 2 - изображение графика функции sin

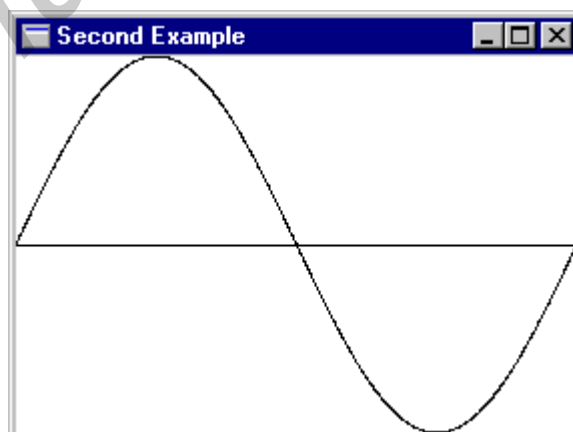
```
#include <windows.h>
#include <math.h>
#define NUM 1000
#define TWOPI (2 * 3.14159)
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
    PSTR szCmdLine, int iCmdShow) {
    static char szAppName[] = "Sin" ;
    HWND hwnd ; MSG msg ;
    WNDCLASSEX wndclass ;
    wndclass.cbSize = sizeof (wndclass) ;
    wndclass.style = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ; wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = szAppName ;
    wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION) ;
    RegisterClassEx (&wndclass) ;
    hwnd = CreateWindow (szAppName, "Second Example",
        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL) ;
    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg) ; DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}
```

```

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg,
    WPARAM wParam, LPARAM lParam) {
    static int cxClient, cyClient ;
    HDC      hdc ;          int      i ;
    PAINTSTRUCT ps ;      POINT  pt [NUM] ;
    switch (iMsg) {
        case WM_SIZE:
            cxClient = LOWORD (lParam); cyClient = HIWORD (lParam);
            return 0 ;
        case WM_PAINT:
            hdc = BeginPaint (hwnd, &ps) ;
            MoveToEx (hdc, 0,      cyClient / 2, NULL) ;
            LineTo  (hdc, cxClient, cyClient / 2) ;
            for (i = 0 ; i < NUM ; i++) {
                pt[i].x = i * cxClient / NUM ;
                pt[i].y = (int) (cyClient / 2 * (1 - sin (TWOPI * i / NUM))) ;
            }
            Polyline (hdc, pt, NUM) ;
            return 0 ;
        case WM_DESTROY:
            PostQuitMessage (0) ;
            return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

```

Программа содержит массив из 1000 структур. В цикле элемент x структуры растёт от 0 до $cxClient$, а y - определяет значение синуса и масштабируется до размеров клиентской области окна. Вся кривая отображается функцией *Polyline*. Результаты работы программы:



Рисование замкнутых фигур

Контур замкнутой фигуры рисуется текущим пером, а фигура закрашивается текущей кистью (по умолчанию WHITE_BRUSH).

Функции рисования прямоугольника **Rectangle** (hdc, x1, y1, x2, y2); и эллипса **Ellipse** (hdc, x1, y1, x2, y2); имеют параметры (x1,y1) - координаты левого верхнего угла, (x2,y2) - правого нижнего угла.

Фигура, отображаемая функцией Ellipse (с ограничивающим прямоугольником), показана на рисунке.

Для рисования прямоугольника с округленными углами: **RoundRect** (hdc, x1, y1, x2, y2, xEllipse, yEllipse); имеет дополнительные параметры: для рисования скругленных углов используется эллипс, шириной *xEllipse*, высотой *yEllipse*. Скругленные углы могут быть получены по формулам

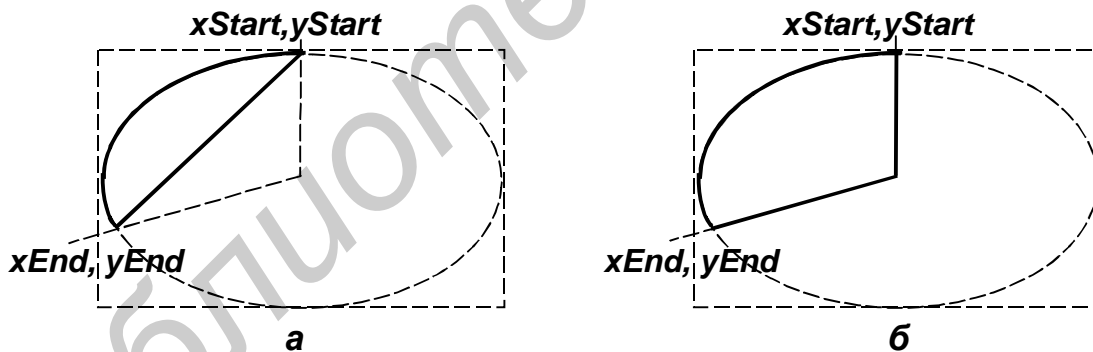
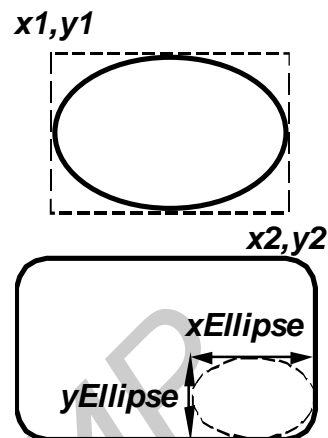
$$xEllipse = (x2 - x1)/4; yEllipse = (y2 - y1)/4;$$

Функции **Chord** (сегмент эллипса) и **Pie** (сектор эллипса) имеют одинаковые параметры:

Chord (hdc, x1, y1, x2, y2, xStart, yStart, xEnd, yEnd);

Pie (hdc, x1, y1, x2, y2, xStart, yStart, xEnd, yEnd);

при рисовании используют воображаемую линию для соединения начала дуги (xStart,yStart) с центром эллипса, в точке пересечения линии с ограничивающим прямоугольником начинается рисование; аналогично для конца дуги (xEnd, yEnd). В функции **Chord** соединяются конечные точки дуги; в **Pie** - начальная и конечная точки дуги с центром эллипса. Фигуры, отображаемые функциями **Chord** и **Pie**, приведены на рисунке.



Фигуры, нарисованные с использованием:
а - функции **Chord**; б - функции **Pie**

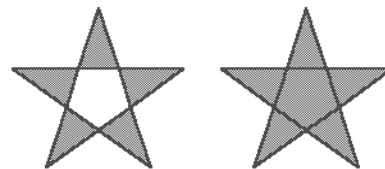
Для рисования многоугольника используется функция: **Polygon** (hdc, pt, iCount); в которой, если последняя точка в массиве не совпадает с первой, добавляется соединяющая их линия.

Режим закрашивания устанавливается функцией

SetPolyFillMode (hdc, iMode);

iMode - режим закрашивания; по умолчанию устанавливается ALTERNATE (попеременный) - закрашиваются только фрагменты внутренней области многоугольника, полученные соединением линий с нечетными номерами (1,3,5...), другие фрагменты внутренней области не закрашиваются; WINDING (сквозной) - закрашиваются все внутренние области, например,

слева нарисована звезда в режиме ALTERNATE, справа - в режиме WINDING.

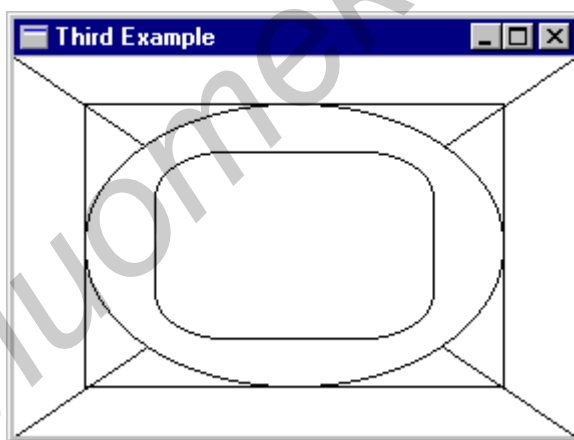


Пример 3 - отображение линий

Если в примере 2 заменить *case WM_PAINT* (убрать лишние переменные), получим программу (Third Example), в которой рисуются прямоугольник, эллипс, прямоугольник с округленными углами и два отрезка.

```
...
case WM_PAINT:
hdc = BeginPaint (hwnd, &ps) ;
Rectangle (hdc, cxClient / 8, cyClient / 8, 7*cxClient / 8, 7*cyClient / 8) ;
MoveToEx (hdc, 0, 0, NULL) ;           LineTo (hdc, cxClient, cyClient) ;
MoveToEx (hdc, 0, cyClient, NULL) ;     LineTo (hdc, cxClient, 0) ;
Ellipse (hdc, cxClient / 8, cyClient / 8, 7*cxClient / 8, 7*cyClient / 8) ;
RoundRect (hdc, cxClient / 4, cyClient / 4, 3*cxClient / 4, 3*cyClient / 4,
           cxClient / 4, cyClient / 4) ;
EndPaint (hwnd, &ps) ;
return 0 ;
...
```

Программа показывает, как закрашиваются области, поэтому под эллипсом отрезки не видны.



9. Управление областями вывода и отсечением

Стандартно графический вывод отсекается по границам окна, с которым связан контекст, и удаляются части, перекрытые другими окнами, т.е. границы области вывода могут иметь достаточно сложную форму.

В Windows используются функции, работающие с прямоугольными областями, использующими структуры типа RECT (прямоугольник) и произвольными областями - регионами (regions).

Простейшим средством, задающим границу области вывода, является **прямоугольник**.

Функция **FillRect** (hdc, &rect, hBrush); закрашивает прямоугольник заданной кистью (не включая правую и нижнюю координаты).

Функция **FrameRect** (hdc, &rect, hBrush); использует кисть для рисования прямоугольной рамки, но не закрашивает внутреннюю область.

Функция **InvertRect** (hdc, &rect); инвертирует все пиксели.

Легко позволяют манипулировать со структур RECT функции:

- установка всех полей структуры RECT в заданные значения:

SetRect (&rect, xLeft, yTop, xRight, yBottom);

- перемещение прямоугольника на заданное число координат вдоль осей x и y: **OffsetRect** (&rect, x, y);

- увеличение или уменьшение размеров прямоугольника:

InflateRect (&rect, x, y);

- установка полей структуры прямоугольника в ноль:

SetRectEmpty (&rect);

- копирование одного прямоугольника в другой:

CopyRect (&DestRect, &SrcRect);

- пересечение двух прямоугольников:

IntersectRect (&DestRect, &SrcRect1, &SrcRect2);

- объединение двух прямоугольников:

UnionRect (&DestRect, &SrcRect1, &SrcRect2);

- определение, является ли прямоугольник пустым:

bEmpty = **IsRectEmpty** (&rect);

- определение, содержится ли точка внутри прямоугольника:

blnRect = **PtInRect** (&rect, point);

Унифицированным средством, задающим границу области вывода, является **регион**, который может иметь как прямоугольную, так и многоугольную, эллиптическую формы или их сочетание. Регион является объектом, идентифицируемым его описателем HRGN.

Произвольный регион создается универсальной функцией:

ExtCreateRegion (const XFORM *lpXform, DWORD nDataSize, const RGNDATA lpRgnData);

структура XFORM описывает преобразование региона в экранные координаты (NULL - координаты считаются идентичными); регион описывается структурой RGNDATA.

Простейший тип региона - прямоугольник может быть создан с помощью функций:

hRgn = **CreateRectRgn** (xLeft, yTop, xRight, yBottom);

hRgn = **CreateRectRgnIndirect** (&rect);

Для создания эллиптических регионов: **Rect** замените на **Elliptic**.

Функция **CreateRoundRectRgn** строит прямоугольный регион со скругленными углами.

Создание многоугольного региона:

hRgn = **CreatePolygonRgn** (&point, iCount, iPolyFillMode);

Регион из множества многоугольников - **CreatePolyPolygonRgn**.

Два региона могут быть объединены в один функцией

iRgnType = **CombineRgn** (*hDestRgn*, *hSrcRgn1*, *hSrcRgn2*, *iCombine*);
комбинирует два исходных региона (*hSrcRgn1* и *hSrcRgn2*) и строит - *hDestRgn*; все описатели регионов до вызова функции должны быть действительными, регион *hDestRgn* - уничтожается; *iCombine* задает правило объединения: RGN_AND - область пересечения исходных регионов; RGN_OR - объединение исходных регионов; RGN_XOR - объединение исходных регионов за исключением области пересечения и т.д.

Возвращаемое функцией значение *iRgnType*: NULLREGION - регион пуст; SIMPLEREGION - простые прямоугольник, эллипс или многоугольник; COMPLEXREGION - комбинация прямоугольников, эллипсов или многоугольников; ERROR - ошибка.

Полученный описатель региона можно использовать в функциях **FillRgn** (*hdc*, *hRgn*, *hBrush*);

FrameRgn (*hdc*, *hRgn*, *hBrush*, *xFrame*, *yFrame*); **InvertRgn** (*hdc*, *hRgn*);
параметры *xFrame* и *yFrame* - логические ширина и высота рамки, которая будет нарисована вокруг региона.

Функция **PaintRgn** (*hdc*, *hRgn*); закрашивает внутреннюю область региона текущей кистью.

Прямоугольники и регионы отсечения

Прямоугольники и регионы могут принимать участие в отсечении. Функция **InvalidateRect** делает недействительным прямоугольную область дисплея и генерирует сообщение WM_PAINT. Ее можно использовать, например, для обновления рабочей области:

InvalidateRect (*hwnd*, NULL, TRUE);

Получить координаты недействительного прямоугольника: **GetUpdateRect**; сделать действительным прямоугольник в рабочей области: **ValidateRect**.

Получая сообщение WM_PAINT, координаты недействительного прямоугольника доступны из полей структуры PAINTSTRUCT, заполняемой при вызове функции **BeginPaint**. Этот недействительный прямоугольник также определяет регион отсечения.

Для регионов **InvalidateRgn** (*hwnd*, *hRgn*, *bErase*);

ValidateRgn (*hwnd*, *hRgn*);

Пути

Путь - это набор прямых и кривых линий, хранящийся в GDI. Пути и регионы очень похожи, можно конвертировать путь в регион и использовать его для отсечения. Для определения пути используется функция

BeginPath (*hdc*);

после чего любая рисуемая линия будет запоминаться как часть пути.

Пути могут состоять из связанных линий, созданных функциями **LineTo**, **PolylineTo** и **BezierTo**, рисующими линии из текущего положения пера. Изменяя текущее положение пера, создаем подпуть в рамках пути.

Таким образом, путь может состоять из одного или нескольких подпутей, каждый подпуть - это серия связанных линий.

Подпуть в рамках пути может быть открыт или закрыт. Подпуть закрыт, если в нем первая точка первой связанной линии и последняя точка последней - совпадают.

Функция **CloseFigure** - закрывает подпуть, добавляя при необходимости прямую линию. Любой последующий вызов функции рисования начинается новый подпуть. Определение пути завершается функцией

EndPath (hdc);

Уничтожают определение пути функции

StrokePath (hdc); **FillPath** (hdc); **StrokeAndFillPath** (hdc);

hRgn = **PathToRegion** (hdc); **SelectClipPath** (hdc, iCombine);

StrokePath рисует путь текущим пером; другие функции закрывают все открытые пути прямыми линиями; **FillPath** закрашивает путь текущей кистью, текущим режимом закрашивания многоугольников; **StrokeAndFillPath** выполняет оба указанных действия.

Можно преобразовать путь в регион или использовать путь как область отсечения. Параметр *iCombine* - показывает, как путь должен комбинироваться с текущим регионом отсечения.

Пути являются более гибкими структурами по сравнению с регионами, т.к. могут состоять из любых линий.

10. Растровая графика

Все рассмотренные выше функции базировались на вычерчивании графических примитивов определенными инструментами по заданным командам, т.е. по векторному принципу. Растровая графика предусматривает доступ к изображению на уровне образующих его точек.

Для большинства устройств отображения первичен растровый принцип формирования изображения. Некоторые контексты поддерживают не все функции растровой графики. Информацию о совместимости может предоставить функция **GetDeviceCaps**.

Простейшим способом получения произвольных изображений является доступ к его точкам. Функции

COLORREF **SetPixel** (hdc, nX, nY, crColor);

BOOL **SetPixelV** (hdc, nX, nY, crColor);

COLORREF **GetPixel** (hdc, nX, nY);

выполняют изменение состояния (цвета) одной логической точки и получение текущего состояния; **SetPixelV** приводит значение цвета к ближайшему представимому в данном контексте; возвращаемое значение - текущее состояние точки (COLORREF), либо признак успешности выполнения (BOOL); *nX*, *nY* – логические координаты точки; *crColor* - новое значение цвета точки.

Более эффективные функции манипулируют не отдельными точками, а массивами точек – фрагментами изображений и битовыми образами.

Битовый образ (bitmap) – двумерный массив числовых значений, характеризующий состояние точек некоторой области.

В простейшем случае битовый образ описывается структурой **BITMAP**, содержащей поля: *bmType* - тип образа (0); *bmWidth*, *bmHeight* - значения ширины и высоты прямоугольной области в пикселях; *bmWidthBytes* - размер образа одной строки изображения в байтах; *bmPlanes* - количество цветовых планов (плоскостей), т.е. компонент, задающих цвет; *bmBitsPixel* - количество бит для кодирования цвета точки; *bmBits* - указатель на двумерный массив данных, каждая строка которого соответствует одной строке изображения.

Используются монохромный и цветной типы образов, при монохромном - одноцветный план и один бит на точку, единичное значение бита задает для точки цвет переднего плана, нулевое - заднего.

Битовые образы идентифицируются их описателями **HBITMAP**. Различают совместимые и контекстно-независимые объекты **BITMAP**.

Для создания объекта **BITMAP** с указанными характеристиками:

HBITMAP CreateBitmap (int nWidth, int nHeight, UINT cPlanes, UINT cBitsPerPel, const void* lpvBits);

HBITMAP CreateBitmapIndirect (const **BITMAP*** lpBitmap);

возвращают описатель объекта или **NULL** в случае ошибки.

Для создания объекта **BITMAP** совместимого типа для заданного контекста с заданными размерами:

HBITMAP CreateCompatibleBitmap (hdc, int nWidth, int nHeight);

в зависимости от контекста он может быть цветным или монохромным (если в контексте заданы данные раздела **DIB** - контекстно-независимым).

Для доступа к содержимому битового образа предусмотрены функции **SetDIBits** и **GetDIBits**, работающие построчно.

Если объект **BITMAP** связывается с контекстом функцией **SelectObject**, все изменения в контексте будут отображаться и в битовом образе.

Перенос прямоугольного фрагмента изображения из контекста-источника в контекст-приемник (с трансформацией и дополнительными операциями) выполняют функции:

BOOL BitBlt (HDC hDstDC, nDstX, nDstY, nDstWidth, nDstHeight, HDC hSrcDC, nSrcX, nSrcY, dwRop);

BOOL StretchBlt (HDC hDstDC, nDstX, nDstY, nDstWidth, nDstHeight, HDC hSrcDC, nSrcX, nSrcY, nSrcWidth, nSrcHeight, dwRop);

BOOL MaskBlt (HDC hDstDC, nDstX, nDstY, nDstWidth, nDstHeight, HDC hSrcDC, nSrcX, nSrcY, **HBITMAP** hbmMask, nMaskX, nMaskY, dwRop);

BOOL PlgBlt (HDC hDstDC, const **POINT*** lpDstVertices, HDC hSrcDC, nSrcX, nSrcY, **HBITMAP** hbmMask, nMaskX, nMaskY, dwRop);

Функция **StretchBlt** изменяет масштаб изображения фрагмента; **MaskBlt** позволяет маскировать часть изображения; **PlgBlt** осуществляет перенос в непрямоугольную область приемника с соответствующим иска-

жением; возвращаемое значение - признак успешности выполнения; параметры:

hSrcDC, hDstDC – контексты источника и приемника данных;

nSrcX, nSrcY, nDstX, nDstY – координаты фрагмента;

nSrcWidth, nSrcHeight, nDstWidth, nDstHeight – размеры фрагментов;

hbmMask – битовый образ маски, монохромного типа;

nMaskX, nMaskY – точка привязки в образе маски;

lpDstVertices – массив структур, задающих вершины параллелограмма, образующего фрагмент-приемник;

dwRop – дополнительная операция, применяемая к фрагменту при переносе: SRCCOPY - простое копирование, SRCAND - комбинация цветов источника и приемника по И, SRCPAINT - комбинация по ИЛИ, DSTINVERT - инверсия фрагмента-приемника, BLACKNESS, WHITENESS - заполнение фрагмента-приемника цветом соответственно 0 и 1 физической палитры, и др. Для успешного применения этих функций требуется, чтобы оба контекста относились к одному устройству или идентичным.

Библиотека БГУИР

Учебное издание

Бусько Виталий Леонидович,
Корбит Анатолий Григорьевич,
Кривоносова Татьяна Михайловна

ОСНОВЫ АЛГОРИТИМЗАЦИИ И ПРОГРАММИРОВАНИЯ

Конспект лекций

для студентов всех специальностей и форм обучения БГУИР

Редактор Е.Н. Батурчик
Компьютерная верстка М.В. Шишло

Подписано в печать 21.09.2004.	Формат 60x84 1/16.	Бумага офсетная.
Печать ризографическая.	Гарнитура «Ариал».	Усл. печ. л.
Уч.-изд.л. 4,7.	Тираж 550 экз.	Заказ 362.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия на осуществление издательской деятельности №02330/0056964 от 01.04.2004.
Лицензия на осуществление полиграфической деятельности №02330/0133108 от 30.04.2004.
220013, Минск, П. Бровки, 6